PRODUCT DOCUMENTATION

Pivotal™ Greenplum Database®

Version 4.3

Reference Guide

Rev: A17

Notice

Copyright

Revised April 2016 (4.3.8.1)

Contents

Chapter 1

Preface

This guide provides reference information for Greenplum Database.

- *About This Guide*
- *Document Conventions*
- *Getting Support*

About This Guide

This guide provides reference information for a Greenplum Database system. This guide is intended for system and database administrators responsible for managing a Greenplum Database system.

This guide assumes knowledge of Linux/UNIX system administration, database management systems, database administration, and structured query language (SQL).

Because Greenplum Database is based on PostgreSQL 8.2.15, this guide assumes some familiarity with PostgreSQL. References to *PostgreSQL documentation* are provided throughout this guide for features that are similar to those in Greenplum Database.

This guide contains the following reference documentation:

- *SQL Command Reference*
- *SQL 2008 Optional Feature Compliance*
- *Greenplum Environment Variables*
- *System Catalog Reference*
- *The gp_toolkit Administrative Schema*
- *Greenplum Database Data Types*
- *Character Set Support*
- *Server Configuration Parameters*
- *Greenplum MapReduce Specification*
- *Greenplum PostGIS Extension*
- *Greenplum PL/R Language Extension*
- *Greenplum Fuzzy String Match Extension*
- *Summary of Greenplum Features*

About the Greenplum Database Documentation Set

The Greenplum Database 4.3 server documentation set consists of the following guides.

Table 1: Greenplum Database server documentation set

Guide Name	Description
Greenplum Database Administrator Guide	Information for administering the Greenplum Database system and managing databases. It covers topics such as Greenplum Database architecture and concepts and everyday system administration tasks such as configuring the server, monitoring system activity, enabling high-availability, backing up and restoring databases, and expanding the system. Database administration topics include configuring access control, creating databases and database objects, loading data into databases, writing queries, managing workloads, and monitoring and troubleshooting performance.
Greenplum Database Reference Guide	Reference information for Greenplum Database systems: SQL commands, system catalogs, environment variables, character set support, datatypes, the Greenplum MapReduce specification, postGIS extension, server parameters, the gp_toolkit administrative schema, and SQL 2008 support.
Greenplum Database Utility Guide	Reference information for command-line utilities, client programs, and Oracle compatibility functions.
Greenplum Database Installation Guide	Information and instructions for installing and initializing a Greenplum Database system.

Document Conventions

The following conventions are used throughout the Greenplum Database documentation to help you identify certain types of information.

- *Command Syntax Conventions*

Command Syntax Conventions

Table 2: Command Syntax Conventions

Text Convention	Usage	Examples
`{ }`	Within command syntax, curly braces group related command options. Do not type the curly braces.	`FROM { 'filename' | STDIN }`
`[ ]`	Within command syntax, square brackets denote optional arguments. Do not type the brackets.	`TRUNCATE [ TABLE ] name`
`...`	Within command syntax, an ellipsis denotes repetition of a command, variable, or option. Do not type the ellipsis.	`DROP TABLE name [, ...]`
`|`	Within command syntax, the pipe symbol denotes an "OR" relationship. Do not type the pipe symbol.	`VACUUM [ FULL | FREEZE ]`

`$ system_command`
`# root_system_command`
`=> gpdb_command`
`=# su_gpdb_command`
	Denotes a command prompt - do not type the prompt symbol. `$` and `#` denote terminal command prompts. `=>` and `=#` denote Greenplum Database interactive program command prompts (`psql` or `gpssh`, for example).	`$ createdb mydatabase`
		`# chown gpadmin -R /datadir`
		`=> SELECT * FROM mytable;`
		`=# SELECT * FROM pg_database;`

Getting Support

Pivotal/Greenplum support, product, and licensing information can be obtained as follows.

Product information and Technical Support

For technical support, documentation, release notes, software updates, or for information about Pivotal products, licensing, and services, go to *www.gopivotal.com*.

Additionally, you can still obtain product and support information from the EMC Support Site at: *http://support.emc.com*

Chapter 2

SQL Command Reference

The following SQL commands are available in Greenplum Database:

- *ABORT*
- *ALTER AGGREGATE*
- *ALTER CONVERSION*
- *ALTER DATABASE*
- *ALTER DOMAIN*
- *ALTER EXTERNAL TABLE*
- *ALTER FILESPACE*
- `ALTER FOREIGN DATA WRAPPER`*
- `ALTER FOREIGN TABLE`*
- *ALTER FUNCTION*
- *ALTER GROUP*
- *ALTER INDEX*
- *ALTER LANGUAGE*
- *ALTER OPERATOR*
- *ALTER OPERATOR CLASS*
- *ALTER PROTOCOL*
- *ALTER RESOURCE QUEUE*
- *ALTER ROLE*
- *ALTER SCHEMA*
- *ALTER SEQUENCE*
- `ALTER SERVER`*
- *ALTER TABLE*
- *ALTER TABLESPACE*
- *ALTER TYPE*
- *ALTER USER*
- `ALTER USER MAPPING`*
- *ANALYZE*
- *BEGIN*
- *CHECKPOINT*
- *CLOSE*
- *CLUSTER*
- *COMMENT*
- *COMMIT*
- *COPY*
- *CREATE AGGREGATE*
- *CREATE CAST*
- *CREATE CONVERSION*
- *CREATE DATABASE*
- *CREATE DOMAIN*

- *CREATE EXTERNAL TABLE*
- `CREATE FOREIGN DATA WRAPPER`*
- `CREATE FOREIGN TABLE`*
- *CREATE FUNCTION*
- *CREATE GROUP*
- *CREATE INDEX*
- *CREATE LANGUAGE*
- *CREATE OPERATOR*
- *CREATE OPERATOR CLASS*
- *CREATE RESOURCE QUEUE*
- *CREATE ROLE*
- *CREATE RULE*
- *CREATE SCHEMA*
- *CREATE SEQUENCE*
- `CREATE SERVER`*
- *CREATE TABLE*
- *CREATE TABLE AS*
- *CREATE TABLESPACE*
- *CREATE TYPE*
- *CREATE USER*
- `CREATE USER MAPPING`*
- *CREATE VIEW*
- *DEALLOCATE*
- *DECLARE*
- *DELETE*
- *DROP AGGREGATE*
- *DROP CAST*
- *DROP CONVERSION*
- *DROP DATABASE*
- *DROP DOMAIN*
- *DROP EXTERNAL TABLE*
- *DROP FILESPACE*
- `DROP FOREIGN DATA WRAPPER`*
- `DROP FOREIGN TABLE`*
- *DROP FUNCTION*
- *DROP GROUP*
- *DROP INDEX*
- *DROP LANGUAGE*
- *DROP OPERATOR*
- *DROP OPERATOR CLASS*
- *DROP OWNED*
- *DROP RESOURCE QUEUE*
- *DROP ROLE*
- *DROP RULE*
- *DROP SCHEMA*
- *DROP SEQUENCE*

- `DROP SERVER`*
- *DROP TABLE*
- *DROP TABLESPACE*
- *DROP TYPE*
- *DROP USER*
- `DROP USER MAPPING`*
- *DROP VIEW*
- *END*
- *EXECUTE*
- *EXPLAIN*
- *FETCH*
- *GRANT*
- *INSERT*
- *LOAD*
- *LOCK*
- *MOVE*
- *PREPARE*
- *REASSIGN OWNED*
- *REINDEX*
- *RELEASE SAVEPOINT*
- *RESET*
- *REVOKE*
- *ROLLBACK*
- *ROLLBACK TO SAVEPOINT*
- *SAVEPOINT*
- *SELECT*
- *SELECT INTO*
- *SET*
- *SET ROLE*
- *SET SESSION AUTHORIZATION*
- *SET TRANSACTION*
- *SHOW*
- *START TRANSACTION*
- *TRUNCATE*
- *UPDATE*
- *VACUUM*
- *VALUES*

* *Not implemented in 4.3*

SQL Syntax Summary

ABORT

Aborts the current transaction.

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

See *ABORT* for more information.

ALTER AGGREGATE

Changes the definition of an aggregate function

```
ALTER AGGREGATE name ( type [ , ... ] ) RENAME TO new_name

ALTER AGGREGATE name ( type [ , ... ] ) OWNER TO new_owner

ALTER AGGREGATE name ( type [ , ... ] ) SET SCHEMA new_schema
```

See *ALTER AGGREGATE* for more information.

ALTER CONVERSION

Changes the definition of a conversion.

```
ALTER CONVERSION name RENAME TO newname

ALTER CONVERSION name OWNER TO newowner
```

See *ALTER CONVERSION* for more information.

ALTER DATABASE

Changes the attributes of a database.

```
ALTER DATABASE name [ WITH CONNECTION LIMIT connlimit ]

ALTER DATABASE name SET parameter { TO | = } { value | DEFAULT }

ALTER DATABASE name RESET parameter

ALTER DATABASE name RENAME TO newname

ALTER DATABASE name OWNER TO new_owner
```

See *ALTER DATABASE* for more information.

ALTER DOMAIN

Changes the definition of a domain.

```
ALTER DOMAIN name { SET DEFAULT expression | DROP DEFAULT }

ALTER DOMAIN name { SET | DROP } NOT NULL

ALTER DOMAIN name ADD domain_constraint

ALTER DOMAIN name DROP CONSTRAINT constraint_name [RESTRICT | CASCADE]
```

```
ALTER DOMAIN name OWNER TO new_owner

ALTER DOMAIN name SET SCHEMA new_schema
```

See *ALTER DOMAIN* for more information.

ALTER EXTERNAL TABLE

Changes the definition of an external table.

```
ALTER EXTERNAL TABLE name RENAME [COLUMN] column TO new_column

ALTER EXTERNAL TABLE name RENAME TO new_name

ALTER EXTERNAL TABLE name SET SCHEMA new_schema

ALTER EXTERNAL TABLE name action [, ... ]
```

See *ALTER EXTERNAL TABLE* for more information.

ALTER FILESPACE

Changes the definition of a filespace.

```
ALTER FILESPACE name RENAME TO newname

ALTER FILESPACE name OWNER TO newowner
```

See *ALTER FILESPACE* for more information.

ALTER FUNCTION

Changes the definition of a function.

```
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   action [, ... ] [RESTRICT]

ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   RENAME TO new_name

ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   OWNER TO new_owner

ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   SET SCHEMA new_schema
```

See *ALTER FUNCTION* for more information.

ALTER GROUP

Changes a role name or membership.

```
ALTER GROUP groupname ADD USER username [, ... ]

ALTER GROUP groupname DROP USER username [, ... ]

ALTER GROUP groupname RENAME TO newname
```

See *ALTER GROUP* for more information.

ALTER INDEX

Changes the definition of an index.

```
ALTER INDEX name RENAME TO new_name

ALTER INDEX name SET TABLESPACE tablespace_name

ALTER INDEX name SET ( FILLFACTOR = value )

ALTER INDEX name RESET ( FILLFACTOR )
```

See *ALTER INDEX* for more information.

ALTER LANGUAGE

Changes the name of a procedural language.

```
ALTER LANGUAGE name RENAME TO newname
```

See *ALTER LANGUAGE* for more information.

ALTER OPERATOR

Changes the definition of an operator.

```
ALTER OPERATOR name ( {lefttype | NONE} , {righttype | NONE} )
   OWNER TO newowner
```

See *ALTER OPERATOR* for more information.

ALTER OPERATOR CLASS

Changes the definition of an operator class.

```
ALTER OPERATOR CLASS name USING index_method RENAME TO newname

ALTER OPERATOR CLASS name USING index_method OWNER TO newowner
```

See *ALTER OPERATOR CLASS* for more information.

ALTER PROTOCOL

Changes the definition of a protocol.

```
ALTER PROTOCOL name RENAME TO newname

ALTER PROTOCOL name OWNER TO newowner
```

See *ALTER PROTOCOL* for more information.

ALTER RESOURCE QUEUE

Changes the limits of a resource queue.

```
ALTER RESOURCE QUEUE name WITH ( queue_attribute=value [, ... ] )
```

See *ALTER RESOURCE QUEUE* for more information.

ALTER ROLE

Changes a database role (user or group).

```
ALTER ROLE name RENAME TO newname

ALTER ROLE name SET config_parameter {TO | =} {value | DEFAULT}

ALTER ROLE name RESET config_parameter

ALTER ROLE name RESOURCE QUEUE {queue_name | NONE}

ALTER ROLE name [ [WITH] option [ ... ] ]
```

See *ALTER ROLE* for more information.

ALTER SCHEMA

Changes the definition of a schema.

```
ALTER SCHEMA name RENAME TO newname

ALTER SCHEMA name OWNER TO newowner
```

See *ALTER SCHEMA* for more information.

ALTER SEQUENCE

Changes the definition of a sequence generator.

```
ALTER SEQUENCE name [INCREMENT [ BY ] increment]
     [MINVALUE minvalue | NO MINVALUE]
     [MAXVALUE maxvalue | NO MAXVALUE]
     [RESTART [ WITH ] start]
     [CACHE cache] [[ NO ] CYCLE]
     [OWNED BY {table.column | NONE}]

ALTER SEQUENCE name SET SCHEMA new_schema
```

See *ALTER SEQUENCE* for more information.

ALTER TABLE

Changes the definition of a table.

```
ALTER TABLE [ONLY] name RENAME [COLUMN] column TO new_column

ALTER TABLE name RENAME TO new_name

ALTER TABLE name SET SCHEMA new_schema

ALTER TABLE [ONLY] name SET
     DISTRIBUTED BY (column, [ ... ] )
   | DISTRIBUTED RANDOMLY
   | WITH (REORGANIZE=true|false)

ALTER TABLE [ONLY] name action [, ... ]

ALTER TABLE name
   [ ALTER PARTITION { partition_name | FOR (RANK(number))
   | FOR (value) } partition_action [...] ]
   partition_action
```

See *ALTER TABLE* for more information.

ALTER TABLESPACE

Changes the definition of a tablespace.

```
ALTER TABLESPACE name RENAME TO newname

ALTER TABLESPACE name OWNER TO newowner
```

See *ALTER TABLESPACE* for more information.

ALTER TYPE

Changes the definition of a data type.

```
ALTER TYPE name
   OWNER TO new_owner | SET SCHEMA new_schema
```

See *ALTER TYPE* for more information.

ALTER USER

Changes the definition of a database role (user).

```
ALTER USER name RENAME TO newname

ALTER USER name SET config_parameter {TO | =} {value | DEFAULT}

ALTER USER name RESET config_parameter

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

See *ALTER USER* for more information.

ANALYZE

Collects statistics about a database.

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

See *ANALYZE* for more information.

BEGIN

Starts a transaction block.

```
BEGIN [WORK | TRANSACTION] [transaction_mode]
      [READ ONLY | READ WRITE]
```

See *BEGIN* for more information.

CHECKPOINT

Forces a transaction log checkpoint.

```
CHECKPOINT
```

See *CHECKPOINT* for more information.

CLOSE

Closes a cursor.

```
CLOSE cursor_name
```

See *CLOSE* for more information.

CLUSTER

Physically reorders a heap storage table on disk according to an index. Not a recommended operation in Greenplum Database.

```
CLUSTER indexname ON tablename

CLUSTER tablename

CLUSTER
```

See *CLUSTER* for more information.

COMMENT

Defines or change the comment of an object.

```
COMMENT ON
{ TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE agg_name (agg_type [, ...]) |
  CAST (sourcetype AS targettype) |
  CONSTRAINT constraint_name ON table_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  FILESPACE object_name |
  FUNCTION func_name ([[argmode] [argname] argtype [, ...]]) |
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  OPERATOR op (leftoperand_type, rightoperand_type) |
  OPERATOR CLASS object_name USING index_method |
  [PROCEDURAL] LANGUAGE object_name |
  RESOURCE QUEUE object_name |
  ROLE object_name |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  TABLESPACE object_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name }
IS 'text'
```

See *COMMENT* for more information.

COMMIT

Commits the current transaction.

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

See *COMMIT* for more information.

COPY

Copies data between a file and a table.

```
COPY table [(column [, ...])] FROM {'file' | STDIN}
     [ [WITH]
       [OIDS]
       [HEADER]
       [DELIMITER [ AS ] 'delimiter']
       [NULL [ AS ] 'null string']
       [ESCAPE [ AS ] 'escape' | 'OFF']
       [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
       [CSV [QUOTE [ AS ] 'quote']
            [FORCE NOT NULL column [, ...]]
       [FILL MISSING FIELDS]
       [[LOG ERRORS [INTO error_table] [KEEP]
       SEGMENT REJECT LIMIT count [ROWS | PERCENT] ]

COPY {table [(column [, ...])] | (query)} TO {'file' | STDOUT}
      [ [WITH]
        [OIDS]
        [HEADER]
        [DELIMITER [ AS ] 'delimiter']
        [NULL [ AS ] 'null string']
        [ESCAPE [ AS ] 'escape' | 'OFF']
        [CSV [QUOTE [ AS ] 'quote']
             [FORCE QUOTE column [, ...]] ]
      [IGNORE EXTERNAL PARTITIONS ]
```

See *COPY* for more information.

CREATE AGGREGATE

Defines a new aggregate function.

```
CREATE [ORDERED] AGGREGATE name (input_data_type [ , ... ])
      ( SFUNC = sfunc,
        STYPE = state_data_type
        [, PREFUNC = prefunc]
        [, FINALFUNC = ffunc]
        [, INITCOND = initial_condition]
        [, SORTOP = sort_operator] )
```

See *CREATE AGGREGATE* for more information.

CREATE CAST

Defines a new cast.

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

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

See *CREATE CAST* for more information.

CREATE CONVERSION

Defines a new encoding conversion.

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

See *CREATE CONVERSION* for more information.

CREATE DATABASE

Creates a new database.

```
CREATE DATABASE name [ [WITH] [OWNER [=] dbowner]
                     [TEMPLATE [=] template]
                     [ENCODING [=] encoding]
                     [TABLESPACE [=] tablespace]
                     [CONNECTION LIMIT [=] connlimit ] ]
```

See *CREATE DATABASE* for more information.

CREATE DOMAIN

Defines a new domain.

```
CREATE DOMAIN name [AS] data_type [DEFAULT expression]
       [CONSTRAINT constraint_name
       | NOT NULL | NULL
       | CHECK (expression) [...]]
```

See *CREATE DOMAIN* for more information.

CREATE EXTERNAL TABLE

Defines a new external table.

```
CREATE [READABLE] EXTERNAL TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
      LOCATION ('file://seghost[:port]/path/file' [, ...])
        | ('gpfdist://filehost[:port]/file_pattern[#transform]' [, ...]
        | ('gpfdists://filehost[:port]/file_pattern[#transform]'
            [, ...])
        | ('gphdfs://hdfs_host[:port]/path/file')
        | ('s3://S3_endpoint/bucket_name/[S3_prefix]
            [config=config_file]')
      FORMAT 'TEXT'
            [( [HEADER]
               [DELIMITER [AS] 'delimiter' | 'OFF']
               [NULL [AS] 'null string']
               [ESCAPE [AS] 'escape' | 'OFF']
               [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
               [FILL MISSING FIELDS] )]
           | 'CSV'
            [( [HEADER]
               [QUOTE [AS] 'quote']
               [DELIMITER [AS] 'delimiter']
               [NULL [AS] 'null string']
               [FORCE NOT NULL column [, ...]]
               [ESCAPE [AS] 'escape']
               [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
               [FILL MISSING FIELDS] )]
           | 'AVRO'
           | 'PARQUET'

           | 'CUSTOM' (Formatter=<formatter specifications>)
      [ ENCODING 'encoding' ]
      [ [LOG ERRORS [INTO error_table]] SEGMENT REJECT LIMIT count
        [ROWS | PERCENT] ]

CREATE [READABLE] EXTERNAL WEB TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
      LOCATION ('http://webhost[:port]/path/file' [, ...])
    | EXECUTE 'command' [ON ALL
```

```
                          | MASTER
                          | number_of_segments
                          | HOST ['segment_hostname']
                          | SEGMENT segment_id ]
      FORMAT 'TEXT'
            [( [HEADER]
               [DELIMITER [AS] 'delimiter' | 'OFF']
               [NULL [AS] 'null string']
               [ESCAPE [AS] 'escape' | 'OFF']
               [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
               [FILL MISSING FIELDS] )]
           | 'CSV'
            [( [HEADER]
               [QUOTE [AS] 'quote']
               [DELIMITER [AS] 'delimiter']
               [NULL [AS] 'null string']
               [FORCE NOT NULL column [, ...]]
               [ESCAPE [AS] 'escape']
               [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
               [FILL MISSING FIELDS] )]
           | 'CUSTOM' (Formatter=<formatter specifications>)
     [ ENCODING 'encoding' ]
     [ [LOG ERRORS [INTO error_table]] SEGMENT REJECT LIMIT count
       [ROWS | PERCENT] ]

CREATE WRITABLE EXTERNAL TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
     LOCATION('gpfdist://outputhost[:port]/filename[#transform]'
      | ('gpfdists://outputhost[:port]/file_pattern[#transform]'
          [, ...])
      | ('gphdfs://hdfs_host[:port]/path')
      FORMAT 'TEXT'
               [( [DELIMITER [AS] 'delimiter']
               [NULL [AS] 'null string']
               [ESCAPE [AS] 'escape' | 'OFF'] )]
          | 'CSV'
               [([QUOTE [AS] 'quote']
               [DELIMITER [AS] 'delimiter']
               [NULL [AS] 'null string']
               [FORCE QUOTE column [, ...]] ]
               [ESCAPE [AS] 'escape'] )]
           | 'AVRO'
           | 'PARQUET'

           | 'CUSTOM' (Formatter=<formatter specifications>)
    [ ENCODING 'write_encoding' ]
    [ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]

CREATE WRITABLE EXTERNAL WEB TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
    EXECUTE 'command' [ON ALL]
    FORMAT 'TEXT'
               [( [DELIMITER [AS] 'delimiter']
               [NULL [AS] 'null string']
               [ESCAPE [AS] 'escape' | 'OFF'] )]
          | 'CSV'
               [([QUOTE [AS] 'quote']
               [DELIMITER [AS] 'delimiter']
               [NULL [AS] 'null string']
               [FORCE QUOTE column [, ...]] ]
               [ESCAPE [AS] 'escape'] )]
           | 'CUSTOM' (Formatter=<formatter specifications>)
    [ ENCODING 'write_encoding' ]
    [ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]
```

See *CREATE EXTERNAL TABLE* for more information.

CREATE FUNCTION

Defines a new function.

```
CREATE [OR REPLACE] FUNCTION name
    ( [ [argmode] [argname] argtype [, ...] ] )
      [ RETURNS { [ SETOF ] rettype
        | TABLE ([{ argname argtype | LIKE other table }
          [, ...]])
        } ]
    { LANGUAGE langname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER
    | AS 'definition'
    | AS 'obj_file', 'link_symbol' } ...
    [ WITH ({ DESCRIBE = describe_function
           } [, ...] ) ]
```

See *CREATE FUNCTION* for more information.

CREATE GROUP

Defines a new database role.

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

See *CREATE GROUP* for more information.

CREATE INDEX

Defines a new index.

```
CREATE [UNIQUE] INDEX name ON table
       [USING btree|bitmap|gist]
       ( {column | (expression)} [opclass] [, ...] )
       [ WITH ( FILLFACTOR = value ) ]
       [TABLESPACE tablespace]
       [WHERE predicate]
```

See *CREATE INDEX* for more information.

CREATE LANGUAGE

Defines a new procedural language.

```
CREATE [PROCEDURAL] LANGUAGE name

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

See *CREATE LANGUAGE* for more information.

CREATE OPERATOR

Defines a new operator.

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

```
       [, LTCMP = less_than_op] [, GTCMP = greater_than_op] )
```

See *CREATE OPERATOR* for more information.

CREATE OPERATOR CLASS

Defines a new operator class.

```
CREATE OPERATOR CLASS name [DEFAULT] FOR TYPE data_type
  USING index_method AS
  {
  OPERATOR strategy_number op_name [(op_type, op_type)] [RECHECK]
  | FUNCTION support_number funcname (argument_type [, ...] )
  | STORAGE storage_type
  } [, ... ]
```

See *CREATE OPERATOR CLASS* for more information.

CREATE RESOURCE QUEUE

Defines a new resource queue.

```
CREATE RESOURCE QUEUE name WITH (queue_attribute=value [, ... ])
```

See *CREATE RESOURCE QUEUE* for more information.

CREATE ROLE

Defines a new database role (user or group).

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

See *CREATE ROLE* for more information.

CREATE RULE

Defines a new rewrite rule.

```
CREATE [OR REPLACE] RULE name AS ON event
  TO table [WHERE condition]
  DO [ALSO | INSTEAD] { NOTHING | command | (command; command
  ...) }
```

See *CREATE RULE* for more information.

CREATE SCHEMA

Defines a new schema.

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

CREATE SCHEMA AUTHORIZATION rolename [schema_element [ ... ]]
```

See *CREATE SCHEMA* for more information.

CREATE SEQUENCE

Defines a new sequence generator.

```
CREATE [TEMPORARY | TEMP] SEQUENCE name
       [INCREMENT [BY] value]
```

```
        [MINVALUE minvalue | NO MINVALUE]
        [MAXVALUE maxvalue | NO MAXVALUE]
        [START [ WITH ] start]
        [CACHE cache]
        [[NO] CYCLE]
        [OWNED BY { table.column | NONE }]
```

See *CREATE SEQUENCE* for more information.

CREATE TABLE

Defines a new table.

```
CREATE [[GLOBAL | LOCAL] {TEMPORARY | TEMP}] TABLE table_name (
[ { column_name data_type [ DEFAULT default_expr ]
   [column_constraint [ ... ]
[ ENCODING ( storage_directive [,...] ) ]
]
   | table_constraint
   | LIKE other_table [{INCLUDING | EXCLUDING}
                      {DEFAULTS | CONSTRAINTS}] ...}
   [, ... ] ]
   )
   [ INHERITS ( parent_table [, ... ] ) ]
   [ WITH ( storage_parameter=value [, ... ] )
   [ ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP} ]
   [ TABLESPACE tablespace ]
   [ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]
   [ PARTITION BY partition_type (column)
       [ SUBPARTITION BY partition_type (column) ]
          [ SUBPARTITION TEMPLATE ( template_spec ) ]
       [...]
    ( partition_spec )
        | [ SUBPARTITION BY partition_type (column) ]
          [...]
    ( partition_spec
      [ ( subpartition_spec
           [(...)]
         ) ]
    )
```

See *CREATE TABLE* for more information.

CREATE TABLE AS

Defines a new table from the results of a query.

```
CREATE [ [GLOBAL | LOCAL] {TEMPORARY | TEMP} ] TABLE table_name
   [(column_name [, ...] )]
   [ WITH ( storage_parameter=value [, ... ] ) ]
   [ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP}]
   [TABLESPACE tablespace]
   AS query
   [DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY]
```

See *CREATE TABLE AS* for more information.

CREATE TABLESPACE

Defines a new tablespace.

```
CREATE TABLESPACE tablespace_name [OWNER username]
       FILESPACE filespace_name
```

See *CREATE TABLESPACE* for more information.

CREATE TYPE

Defines a new data type.

```
CREATE TYPE name AS ( attribute_name data_type [, ... ] )

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

CREATE TYPE name
```

See *CREATE TYPE* for more information.

CREATE USER

Defines a new database role with the `LOGIN` privilege by default.

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

See *CREATE USER* for more information.

CREATE VIEW

Defines a new view.

```
CREATE [OR REPLACE] [TEMP | TEMPORARY] VIEW name
       [ ( column_name [, ...] ) ]
       AS query
```

See *CREATE VIEW* for more information.

DEALLOCATE

Deallocates a prepared statement.

```
DEALLOCATE [PREPARE] name
```

See *DEALLOCATE* for more information.

DECLARE

Defines a cursor.

```
DECLARE name [BINARY] [INSENSITIVE] [NO SCROLL] CURSOR
     [{WITH | WITHOUT} HOLD]
     FOR query [FOR READ ONLY]
```

See *DECLARE* for more information.

DELETE

Deletes rows from a table.

```
DELETE FROM [ONLY] table [[AS] alias]
      [USING usinglist]
      [WHERE condition | WHERE CURRENT OF cursor_name ]
```

See *DELETE* for more information.

DROP AGGREGATE

Removes an aggregate function.

```
DROP AGGREGATE [IF EXISTS] name ( type [, ...] ) [CASCADE | RESTRICT]
```

See *DROP AGGREGATE* for more information.

DROP CAST

Removes a cast.

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

See *DROP CAST* for more information.

DROP CONVERSION

Removes a conversion.

```
DROP CONVERSION [IF EXISTS] name [CASCADE | RESTRICT]
```

See *DROP CONVERSION* for more information.

DROP DATABASE

Removes a database.

```
DROP DATABASE [IF EXISTS] name
```

See *DROP DATABASE* for more information.

DROP DOMAIN

Removes a domain.

```
DROP DOMAIN [IF EXISTS] name [, ...]  [CASCADE | RESTRICT]
```

See *DROP DOMAIN* for more information.

DROP EXTERNAL TABLE

Removes an external table definition.

```
DROP EXTERNAL [WEB] TABLE [IF EXISTS] name [CASCADE | RESTRICT]
```

See *DROP EXTERNAL TABLE* for more information.

DROP FILESPACE

Removes a filespace.

```
DROP FILESPACE [IF EXISTS] filespacename
```

See *DROP FILESPACE* for more information.

DROP FUNCTION

Removes a function.

```
DROP FUNCTION [IF EXISTS] name ( [ [argmode] [argname] argtype
    [, ...] ] ) [CASCADE | RESTRICT]
```

See *DROP FUNCTION* for more information.

DROP GROUP

Removes a database role.

```
DROP GROUP [IF EXISTS] name [, ...]
```

See *DROP GROUP* for more information.

DROP INDEX

Removes an index.

```
DROP INDEX [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

See *DROP INDEX* for more information.

DROP LANGUAGE

Removes a procedural language.

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

See *DROP LANGUAGE* for more information.

DROP OPERATOR

Removes an operator.

```
DROP OPERATOR [IF EXISTS] name ( {lefttype | NONE} ,
    {righttype | NONE} ) [CASCADE | RESTRICT]
```

See *DROP OPERATOR* for more information.

DROP OPERATOR CLASS

Removes an operator class.

```
DROP OPERATOR CLASS [IF EXISTS] name USING index_method [CASCADE | RESTRICT]
```

See *DROP OPERATOR CLASS* for more information.

DROP OWNED

Removes database objects owned by a database role.

```
DROP OWNED BY name [, ...] [CASCADE | RESTRICT]
```

See *DROP OWNED* for more information.

DROP RESOURCE QUEUE

Removes a resource queue.

```
DROP RESOURCE QUEUE queue_name
```

See *DROP RESOURCE QUEUE* for more information.

DROP ROLE

Removes a database role.

```
DROP ROLE [IF EXISTS] name [, ...]
```

See *DROP ROLE* for more information.

DROP RULE

Removes a rewrite rule.

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

See *DROP RULE* for more information.

DROP SCHEMA

Removes a schema.

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

See *DROP SCHEMA* for more information.

DROP SEQUENCE

Removes a sequence.

```
DROP SEQUENCE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

See *DROP SEQUENCE* for more information.

DROP TABLE

Removes a table.

```
DROP TABLE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

See *DROP TABLE* for more information.

DROP TABLESPACE

Removes a tablespace.

```
DROP TABLESPACE [IF EXISTS] tablespacename
```

See *DROP TABLESPACE* for more information.

DROP TYPE

Removes a data type.

```
DROP TYPE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

See *DROP TYPE* for more information.

DROP USER

Removes a database role.

```
DROP USER [IF EXISTS] name [, ...]
```

See *DROP USER* for more information.

DROP VIEW

Removes a view.

```
DROP VIEW [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

See *DROP VIEW* for more information.

END

Commits the current transaction.

```
END [WORK | TRANSACTION]
```

See *END* for more information.

EXECUTE

Executes a prepared SQL statement.

```
EXECUTE name [ (parameter [, ...] ) ]
```

See *EXECUTE* for more information.

EXPLAIN

Shows the query plan of a statement.

```
EXPLAIN [ANALYZE] [VERBOSE] statement
```

See *EXPLAIN* for more information.

FETCH

Retrieves rows from a query using a cursor.

```
FETCH [ forward_direction { FROM | IN } ] cursorname
```

See *FETCH* for more information.

GRANT

Defines access privileges.

```
GRANT { {SELECT | INSERT | UPDATE | DELETE | REFERENCES |
TRIGGER | TRUNCATE } [,...] | ALL [PRIVILEGES] }
    ON [TABLE] tablename [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { {USAGE | SELECT | UPDATE} [,...] | ALL [PRIVILEGES] }
    ON SEQUENCE sequencename [, ...]
    TO { rolename | PUBLIC } [, ...] [WITH GRANT OPTION]

GRANT { {CREATE | CONNECT | TEMPORARY | TEMP} [,...] | ALL
[PRIVILEGES] }
    ON DATABASE dbname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { EXECUTE | ALL [PRIVILEGES] }
    ON FUNCTION funcname ( [ [argmode] [argname] argtype [, ...]
] ) [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { USAGE | ALL [PRIVILEGES] }
    ON LANGUAGE langname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { {CREATE | USAGE} [,...] | ALL [PRIVILEGES] }
    ON SCHEMA schemaname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { CREATE | ALL [PRIVILEGES] }
    ON TABLESPACE tablespacename [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT parent_role [, ...]
    TO member_role [, ...] [WITH ADMIN OPTION]

GRANT { SELECT | INSERT | ALL [PRIVILEGES] }
    ON PROTOCOL protocolname
    TO username
```

See *GRANT* for more information.

INSERT

Creates new rows in a table.

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

See *INSERT* for more information.

LOAD

Loads or reloads a shared library file.

```
LOAD 'filename'
```

See *LOAD* for more information.

LOCK

Locks a table.

```
LOCK [TABLE] name [, ...] [IN lockmode MODE] [NOWAIT]
```

See *LOCK* for more information.

MOVE

Positions a cursor.

```
MOVE [ forward_direction {FROM | IN} ] cursorname
```

See *MOVE* for more information.

PREPARE

Prepare a statement for execution.

```
PREPARE name [ (datatype [, ...] ) ] AS statement
```

See *PREPARE* for more information.

REASSIGN OWNED

Changes the ownership of database objects owned by a database role.

```
REASSIGN OWNED BY old_role [, ...] TO new_role
```

See *REASSIGN OWNED* for more information.

REINDEX

Rebuilds indexes.

```
REINDEX {INDEX | TABLE | DATABASE | SYSTEM} name
```

See *REINDEX* for more information.

RELEASE SAVEPOINT

Destroys a previously defined savepoint.

```
RELEASE [SAVEPOINT] savepoint_name
```

See *RELEASE SAVEPOINT* for more information.

RESET

Restores the value of a system configuration parameter to the default value.

```
RESET configuration_parameter

RESET ALL
```

See *RESET* for more information.

REVOKE

Removes access privileges.

```
REVOKE [GRANT OPTION FOR] { {SELECT | INSERT | UPDATE | DELETE
       | REFERENCES | TRIGGER | TRUNCATE } [,...] | ALL [PRIVILEGES] }
       ON [TABLE] tablename [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { {USAGE | SELECT | UPDATE} [,...]
       | ALL [PRIVILEGES] }
       ON SEQUENCE sequencename [, ...]
       FROM { rolename | PUBLIC } [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { {CREATE | CONNECT
       | TEMPORARY | TEMP} [,...] | ALL [PRIVILEGES] }
       ON DATABASE dbname [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] {EXECUTE | ALL [PRIVILEGES]}
       ON FUNCTION funcname ( [[argmode] [argname] argtype
                               [, ...]] ) [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] {USAGE | ALL [PRIVILEGES]}
       ON LANGUAGE langname [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [ CASCADE | RESTRICT ]

REVOKE [GRANT OPTION FOR] { {CREATE | USAGE} [,...]
       | ALL [PRIVILEGES] }
       ON SCHEMA schemaname [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { CREATE | ALL [PRIVILEGES] }
       ON TABLESPACE tablespacename [, ...]
       FROM { rolename | PUBLIC } [, ...]
       [CASCADE | RESTRICT]

REVOKE [ADMIN OPTION FOR] parent_role [, ...]
       FROM member_role [, ...]
       [CASCADE | RESTRICT]
```

See *REVOKE* for more information.

ROLLBACK

Aborts the current transaction.

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

See *ROLLBACK* for more information.

ROLLBACK TO SAVEPOINT

Rolls back the current transaction to a savepoint.

```
ROLLBACK [WORK | TRANSACTION] TO [SAVEPOINT] savepoint_name
```

See *ROLLBACK TO SAVEPOINT* for more information.

SAVEPOINT

Defines a new savepoint within the current transaction.

```
SAVEPOINT savepoint_name
```

See *SAVEPOINT* for more information.

SELECT

Retrieves rows from a table or view.

```
SELECT [ALL | DISTINCT [ON (expression [, ...])]]
  * | expression [[AS] output_name] [, ...]
  [FROM from_item [, ...]]
  [WHERE condition]
  [GROUP BY grouping_element [, ...]]
  [HAVING condition [, ...]]
  [WINDOW window_name AS (window_specification)]
  [{UNION | INTERSECT | EXCEPT} [ALL] select]
  [ORDER BY expression [ASC | DESC | USING operator] [, ...]]
  [LIMIT {count | ALL}]
  [OFFSET start]
  [FOR {UPDATE | SHARE} [OF table_name [, ...]] [NOWAIT] [...]]
```

See *SELECT* for more information.

SELECT INTO

Defines a new table from the results of a query.

```
SELECT [ALL | DISTINCT [ON ( expression [, ...] )]]
    * | expression [AS output_name] [, ...]
    INTO [TEMPORARY | TEMP] [TABLE] new_table
    [FROM from_item [, ...]]
    [WHERE condition]
    [GROUP BY expression [, ...]]
    [HAVING condition [, ...]]
    [{UNION | INTERSECT | EXCEPT} [ALL] select]
    [ORDER BY expression [ASC | DESC | USING operator] [, ...]]
    [LIMIT {count | ALL}]
    [OFFSET start]
    [FOR {UPDATE | SHARE} [OF table_name [, ...]] [NOWAIT]
    [...]]
```

See *SELECT INTO* for more information.

SET

Changes the value of a Greenplum Database configuration parameter.

```
SET [SESSION | LOCAL] configuration_parameter {TO | =} value |
    'value' | DEFAULT}

SET [SESSION | LOCAL] TIME ZONE {timezone | LOCAL | DEFAULT}
```

See *SET* for more information.

SET ROLE

Sets the current role identifier of the current session.

```
SET [SESSION | LOCAL] ROLE rolename

SET [SESSION | LOCAL] ROLE NONE

RESET ROLE
```

See *SET ROLE* for more information.

SET SESSION AUTHORIZATION

Sets the session role identifier and the current role identifier of the current session.

```
SET [SESSION | LOCAL] SESSION AUTHORIZATION rolename

SET [SESSION | LOCAL] SESSION AUTHORIZATION DEFAULT

RESET SESSION AUTHORIZATION
```

See *SET SESSION AUTHORIZATION* for more information.

SET TRANSACTION

Sets the characteristics of the current transaction.

```
SET TRANSACTION [transaction_mode] [READ ONLY | READ WRITE]

SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode
     [READ ONLY | READ WRITE]
```

See *SET TRANSACTION* for more information.

SHOW

Shows the value of a system configuration parameter.

```
SHOW configuration_parameter

SHOW ALL
```

See *SHOW* for more information.

START TRANSACTION

Starts a transaction block.

```
START TRANSACTION [SERIALIZABLE | READ COMMITTED | READ UNCOMMITTED]
                  [READ WRITE | READ ONLY]
```

See *START TRANSACTION* for more information.

TRUNCATE

Empties a table of all rows.

```
TRUNCATE [TABLE] name [, ...] [CASCADE | RESTRICT]
```

See *TRUNCATE* for more information.

UPDATE

Updates rows of a table.

```
UPDATE [ONLY] table [[AS] alias]
   SET {column = {expression | DEFAULT} |
   (column [, ...]) = ({expression | DEFAULT} [, ...])} [, ...]
   [FROM fromlist]
   [WHERE condition | WHERE CURRENT OF cursor_name ]
```

See *UPDATE* for more information.

VACUUM

Garbage-collects and optionally analyzes a database.

```
VACUUM [FULL] [FREEZE] [VERBOSE] [table]

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

See *VACUUM* for more information.

VALUES

Computes a set of rows.

```
VALUES ( expression [, ...] ) [, ...]
   [ORDER BY sort_expression [ASC | DESC | USING operator] [, ...]]
   [LIMIT {count | ALL}] [OFFSET start]
```

See *VALUES* for more information.

ABORT

Aborts the current transaction.

Synopsis

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

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 standard SQL command *`ROLLBACK`*, and is present only for historical reasons.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

Notes

Use `COMMIT` to successfully terminate a transaction.

Issuing `ABORT` when not inside a transaction does no harm, but it will provoke a warning message.

Compatibility

This command is a Greenplum Database extension present for historical reasons. `ROLLBACK` is the equivalent standard SQL command.

See Also

`BEGIN`, *`COMMIT`*, *`ROLLBACK`*

ALTER AGGREGATE

Changes the definition of an aggregate function

Synopsis

```
ALTER AGGREGATE name ( type [ , ... ] ) RENAME TO new_name

ALTER AGGREGATE name ( type [ , ... ] ) OWNER TO new_owner

ALTER AGGREGATE name ( type [ , ... ] ) SET SCHEMA new_schema
```

Description

`ALTER AGGREGATE` changes the definition of an aggregate function.

You must own the aggregate function to use `ALTER AGGREGATE`. To change the schema of an aggregate function, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the aggregate function's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the aggregate function. However, a superuser can alter ownership of any aggregate function anyway.)

Parameters

name

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

type

An input data type on which the aggregate function operates. To reference a zero-argument aggregate function, write * in place of the list of input data types.

new_name

The new name of the aggregate function.

new_owner

The new owner of the aggregate function.

new_schema

The new schema for the aggregate function.

Examples

To rename the aggregate function `myavg` for type `integer` to `my_average`:

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

To change the owner of the aggregate function `myavg` for type `integer` to `joe`:

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

To move the aggregate function `myavg` for type `integer` into schema `myschema`:

```
ALTER AGGREGATE myavg(integer) SET SCHEMA myschema;
```

Compatibility

There is no `ALTER AGGREGATE` statement in the SQL standard.

See Also

CREATE AGGREGATE, *DROP AGGREGATE*

ALTER CONVERSION

Changes the definition of a conversion.

Synopsis

```
ALTER CONVERSION name RENAME TO newname

ALTER CONVERSION name OWNER TO newowner
```

Description

`ALTER CONVERSION` changes the definition of a conversion.

You must own the conversion to use `ALTER CONVERSION`. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the conversion's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the conversion. However, a superuser can alter ownership of any conversion anyway.)

Parameters

name

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

newname

The new name of the conversion.

newowner

The new owner of the conversion.

Examples

To rename the conversion `iso_8859_1_to_utf8` to `latin1_to_unicode`:

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO
latin1_to_unicode;
```

To change the owner of the conversion `iso_8859_1_to_utf8` to `joe`:

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

Compatibility

There is no `ALTER CONVERSION` statement in the SQL standard.

See Also

CREATE CONVERSION, *DROP CONVERSION*

ALTER DATABASE

Changes the attributes of a database.

Synopsis

```
ALTER DATABASE name [ WITH CONNECTION LIMIT connlimit ]

ALTER DATABASE name SET parameter { TO | = } { value | DEFAULT }

ALTER DATABASE name RESET parameter

ALTER DATABASE name RENAME TO newname

ALTER DATABASE name OWNER TO new_owner
```

Description

`ALTER DATABASE` changes the attributes of a database.

The first form changes the allowed connection limit for a database. Only the database owner or a superuser can change this setting.

The second and third forms change the session default for a configuration parameter for a Greenplum database. Whenever a new session is subsequently started in that database, the specified value becomes the session default value. The database-specific default overrides whatever setting is present in the server configuration file (`postgresql.conf`). Only the database owner or a superuser can change the session defaults for a database. Certain parameters cannot be set this way, or can only be set by a superuser.

The fourth form changes the name of the database. Only the database owner or a superuser can rename a database; non-superuser owners must also have the `CREATEDB` privilege. You cannot rename the current database. Connect to a different database first.

The fifth form changes the owner of the database. To alter the owner, you must own the database and also be a direct or indirect member of the new owning role, and you must have the `CREATEDB` privilege. (Note that superusers have all these privileges automatically.)

Parameters

name

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

connlimit

The maximum number of concurrent connections possible. The default of -1 means there is no limitation.

parameter value

Set this database's session default for the specified configuration parameter to the given value. If value is `DEFAULT` or, equivalently, `RESET` is used, the database-specific setting is removed, so the system-wide default setting will be inherited in new sessions. Use `RESET ALL` to clear all database-specific settings. See *Server Configuration Parameters* for information about server parameters. for information about all user-settable configuration parameters.

newname

The new name of the database.

new_owner

The new owner of the database.

Notes

It is also possible to set a configuration parameter session default for a specific role (user) rather than to a database. Role-specific settings override database-specific ones if there is a conflict. See `ALTER ROLE`.

Examples

To set the default schema search path for the `mydatabase` database:

```
ALTER DATABASE mydatabase SET search_path TO myschema,
public, pg_catalog;
```

Compatibility

The `ALTER DATABASE` statement is a Greenplum Database extension.

See Also

`CREATE DATABASE`, *`DROP DATABASE`*, *`SET`*

ALTER DOMAIN

Changes the definition of a domain.

Synopsis

```
ALTER DOMAIN name { SET DEFAULT expression | DROP DEFAULT }

ALTER DOMAIN name { SET | DROP } NOT NULL

ALTER DOMAIN name ADD domain_constraint

ALTER DOMAIN name DROP CONSTRAINT constraint_name [RESTRICT | CASCADE]

ALTER DOMAIN name OWNER TO new_owner

ALTER DOMAIN name SET SCHEMA new_schema
```

Description

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

- **SET/DROP DEFAULT** — These forms set or remove the default value for a domain. Note that defaults only apply to subsequent `INSERT` commands. They do not affect rows already in a table using the domain.
- **SET/DROP NOT NULL** — These forms change whether a domain is marked to allow `NULL` values or to reject `NULL` values. You may only `SET NOT NULL` when the columns using the domain contain no null values.
- **ADD *domain_constraint*** — This form adds a new constraint to a domain using the same syntax as `CREATE DOMAIN`. This will only succeed if all columns using the domain satisfy the new constraint.
- **DROP CONSTRAINT** — This form drops constraints on a domain.
- **OWNER** — This form changes the owner of the domain to the specified user.
- **SET SCHEMA** — This form changes the schema of the domain. Any constraints associated with the domain are moved into the new schema as well.

You must own the domain to use `ALTER DOMAIN`. To change the schema of a domain, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the domain's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the domain. However, a superuser can alter ownership of any domain anyway.)

Parameters

name

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

domain_constraint

New domain constraint for the domain.

constraint_name

Name of an existing constraint to drop.

CASCADE

Automatically drop objects that depend on the constraint.

RESTRICT

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

new_owner

The user name of the new owner of the domain.

new_schema

The new schema for the domain.

Examples

To add a NOT NULL constraint to a domain:

```
ALTER DOMAIN zipcode SET NOT NULL;
```

To remove a NOT NULL constraint from a domain:

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

To add a check constraint to a domain:

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK
(char_length(VALUE) = 5);
```

To remove a check constraint from a domain:

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

To move the domain into a different schema:

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

Compatibility

ALTER DOMAIN conforms to the SQL standard, except for the OWNER and SET SCHEMA variants, which are Greenplum Database extensions.

See Also

CREATE DOMAIN, *DROP DOMAIN*

ALTER EXTERNAL TABLE

Changes the definition of an external table.

Synopsis

```
ALTER EXTERNAL TABLE name RENAME [COLUMN] column TO new_column

ALTER EXTERNAL TABLE name RENAME TO new_name

ALTER EXTERNAL TABLE name SET SCHEMA new_schema

ALTER EXTERNAL TABLE name action [, ... ]
```

where *action* is one of:

```
  ADD [COLUMN] column_name type
  DROP [COLUMN] column
  ALTER [COLUMN] column
  TYPE type [USING expression]
  OWNER TO new_owner
```

Description

`ALTER EXTERNAL TABLE` changes the definition of an existing external table. There are several subforms:

- **ADD COLUMN** — Adds a new column to the external table definition.
- **DROP COLUMN** — Drops a column from the external table definition. Note that if you drop readable external table columns, it only changes the table definition in Greenplum Database. External data files are not changed.
- **ALTER COLUMN TYPE** — Changes the data type of a column of a table. The optional `USING` clause specifies how to compute the new column value from the old. If omitted, the default conversion is the same as an assignment cast from old data type to new. A `USING` clause must be provided if there is no implicit or assignment cast from the old to new type.
- **OWNER** — Changes the owner of the external table to the specified user.
- **RENAME** — Changes the name of an external table or the name of an individual column in the table. There is no effect on the external data.
- **SET SCHEMA** — Moves the external table into another schema.

You must own the external table to use `ALTER EXTERNAL TABLE`. To change the schema of an external table, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the external table's schema. A superuser has these privileges automatically.

In this release, `ALTER EXTERNAL TABLE` cannot modify the external table type, the data format, or the location of the external data. To modify this information, you must drop and recreate the external table definition.

Parameters

name

The name (possibly schema-qualified) of an existing external table definition to alter.

column

Name of a new or existing column.

new_column

New name for an existing column.

new_name

New name for the external table.

type

Data type of the new column, or new data type for an existing column.

new_owner

The role name of the new owner of the external table.

new_schema

The name of the schema to which the external table will be moved.

Examples

Add a new column to an external table definition:

```
ALTER EXTERNAL TABLE ext_expenses ADD COLUMN manager text;
```

Change the name of an external table:

```
ALTER EXTERNAL TABLE ext_data RENAME TO ext_sales_data;
```

Change the owner of an external table:

```
ALTER EXTERNAL TABLE ext_data OWNER TO jojo;
```

Change the schema of an external table:

```
ALTER EXTERNAL TABLE ext_leads SET SCHEMA marketing;
```

Compatibility

`ALTER EXTERNAL TABLE` is a Greenplum Database extension. There is no `ALTER EXTERNAL TABLE` statement in the SQL standard or regular PostgreSQL.

See Also

CREATE EXTERNAL TABLE, *DROP EXTERNAL TABLE*

ALTER FILESPACE

Changes the definition of a filespace.

Synopsis

```
ALTER FILESPACE name RENAME TO newname

ALTER FILESPACE name OWNER TO newowner
```

Description

`ALTER FILESPACE` changes the definition of a filespace.

You must own the filespace to use `ALTER FILESPACE`. To alter the owner, you must also be a direct or indirect member of the new owning role (note that superusers have these privileges automatically).

Parameters

name

The name of an existing filespace.

newname

The new name of the filespace. The new name cannot begin with *pg_* or *gp_* (reserved for system filespaces).

newowner

The new owner of the filespace.

Examples

Rename filespace `myfs` to `fast_ssd`:

```
ALTER FILESPACE myfs RENAME TO fast_ssd;
```

Change the owner of tablespace `myfs`:

```
ALTER FILESPACE myfs OWNER TO dba;
```

Compatibility

There is no `ALTER FILESPACE` statement in the SQL standard or in PostgreSQL.

See Also

DROP FILESPACE, `gpfilespace` in the *Greenplum Database Utility Guide*

ALTER FUNCTION

Changes the definition of a function.

Synopsis

```
ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   action [, ... ] [RESTRICT]

ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   RENAME TO new_name

ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   OWNER TO new_owner

ALTER FUNCTION name ( [ [argmode] [argname] argtype [, ...] ] )
   SET SCHEMA new_schema
```

where *action* is one of:

```
{CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT}
{IMMUTABLE | STABLE | VOLATILE}
{[EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER}
```

Description

`ALTER FUNCTION` changes the definition of a function.

You must own the function to use `ALTER FUNCTION`. To change a function's schema, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the function's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the function. However, a superuser can alter ownership of any function anyway.)

Parameters

name

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

argmode

The mode of an argument: either `IN`, `OUT`, or `INOUT`. If omitted, the default is `IN`. Note that `ALTER FUNCTION` does not actually pay any attention to `OUT` arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the `IN` and `INOUT` arguments.

argname

The name of an argument. Note that `ALTER FUNCTION` does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any.

new_name

The new name of the function.

new_owner

The new owner of the function. Note that if the function is marked `SECURITY DEFINER`, it will subsequently execute as the new owner.

new_schema

The new schema for the function.

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

`CALLED ON NULL INPUT` changes the function so that it will be invoked when some or all of its arguments are null. `RETURNS NULL ON NULL INPUT` or `STRICT` changes the function so that it is not invoked if any of its arguments are null; instead, a null result is assumed automatically. See `CREATE FUNCTION` for more information.

IMMUTABLE
STABLE
VOLATILE

Change the volatility of the function to the specified setting. See `CREATE FUNCTION` for details.

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

Change whether the function is a security definer or not. The key word `EXTERNAL` is ignored for SQL conformance. See `CREATE FUNCTION` for more information about this capability.

RESTRICT

Ignored for conformance with the SQL standard.

Notes

Greenplum Database has limitations on the use of functions defined as `STABLE` or `VOLATILE`. See *`CREATE FUNCTION`* for more information.

Examples

To rename the function `sqrt` for type `integer` to `square_root`:

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

To change the owner of the function `sqrt` for type `integer` to `joe`:

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

To change the *schema* of the function `sqrt` for type `integer` to `math`:

```
ALTER FUNCTION sqrt(integer) SET SCHEMA math;
```

Compatibility

This statement is partially compatible with the `ALTER FUNCTION` statement in the SQL standard. The standard allows more properties of a function to be modified, but does not provide the ability to rename a function, make a function a security definer, or change the owner, schema, or volatility of a function. The standard also requires the `RESTRICT` key word, which is optional in Greenplum Database.

See Also

`CREATE FUNCTION`, *`DROP FUNCTION`*

ALTER GROUP

Changes a role name or membership.

Synopsis

```
ALTER GROUP groupname ADD USER username [, ... ]

ALTER GROUP groupname DROP USER username [, ... ]

ALTER GROUP groupname RENAME TO newname
```

Description

`ALTER GROUP` is an obsolete command, though still accepted for backwards compatibility. Groups (and users) have been superseded by the more general concept of roles. See *`ALTER ROLE`* for more information.

Parameters

groupname

The name of the group (role) to modify.

username

Users (roles) that are to be added to or removed from the group. The users (roles) must already exist.

newname

The new name of the group (role).

Examples

To add users to a group:

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

To remove a user from a group:

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

Compatibility

There is no `ALTER GROUP` statement in the SQL standard.

See Also

`ALTER ROLE`, *`GRANT`*, *`REVOKE`*

ALTER INDEX

Changes the definition of an index.

Synopsis

```
ALTER INDEX name RENAME TO new_name

ALTER INDEX name SET TABLESPACE tablespace_name

ALTER INDEX name SET ( FILLFACTOR = value )

ALTER INDEX name RESET ( FILLFACTOR )
```

Description

`ALTER INDEX` changes the definition of an existing index. There are several subforms:

- **RENAME** — Changes the name of the index. There is no effect on the stored data.
- **SET TABLESPACE** — Changes the index's tablespace to the specified tablespace and moves the data file(s) associated with the index to the new tablespace. See also `CREATE TABLESPACE`.
- **SET FILLFACTOR** — Changes the index-method-specific storage parameters for the index. The built-in index methods all accept a single parameter: `FILLFACTOR`. The fillfactor for an index is a percentage that determines how full the index method will try to pack index pages. Index contents will not be modified immediately by this command. Use `REINDEX` to rebuild the index to get the desired effects.
- **RESET FILLFACTOR** — Resets `FILLFACTOR` to the default. As with `SET`, a `REINDEX` may be needed to update the index entirely.

Parameters

name

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

new_name

New name for the index.

tablespace_name

The tablespace to which the index will be moved.

FILLFACTOR

The fillfactor for an index is a percentage that determines how full the index method will try to pack index pages. For B-trees, leaf pages are filled to this percentage during initial index build, and also when extending the index at the right (largest key values). If pages subsequently become completely full, they will be split, leading to gradual degradation in the index's efficiency.

B-trees use a default fillfactor of 90, but any value from 10 to 100 can be selected. If the table is static then fillfactor 100 is best to minimize the index's physical size, but for heavily updated tables a smaller fillfactor is better to minimize the need for page splits. The other index methods use fillfactor in different but roughly analogous ways; the default fillfactor varies between methods.

Notes

These operations are also possible using `ALTER TABLE`.

Changing any part of a system catalog index is not permitted.

Examples

To rename an existing index:

```
ALTER INDEX distributors RENAME TO suppliers;
```

To move an index to a different tablespace:

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

To change an index's fill factor (assuming that the index method supports it):

```
ALTER INDEX distributors SET (fillfactor = 75);
REINDEX INDEX distributors;
```

Compatibility

`ALTER INDEX` is a Greenplum Database extension.

See Also

CREATE INDEX, *REINDEX*, *ALTER TABLE*

ALTER LANGUAGE

Changes the name of a procedural language.

Synopsis

```
ALTER LANGUAGE name RENAME TO newname
```

Description

`ALTER LANGUAGE` changes the name of a procedural language. Only a superuser can rename languages.

Parameters

name

Name of a language.

newname

The new name of the language.

Compatibility

There is no `ALTER LANGUAGE` statement in the SQL standard.

See Also

CREATE LANGUAGE, *DROP LANGUAGE*

ALTER OPERATOR

Changes the definition of an operator.

Synopsis

```
ALTER OPERATOR name ( {lefttype | NONE} , {righttype | NONE} )
   OWNER TO newowner
```

Description

`ALTER OPERATOR` changes the definition of an operator. The only currently available functionality is to change the owner of the operator.

You must own the operator to use `ALTER OPERATOR`. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the operator's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the operator. However, a superuser can alter ownership of any operator anyway.)

Parameters

name

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

lefttype

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

righttype

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

newowner

The new owner of the operator.

Examples

Change the owner of a custom operator `a @@ b` for type `text`:

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

Compatibility

There is no `ALTEROPERATOR` statement in the SQL standard.

See Also

CREATE OPERATOR, *DROP OPERATOR*

ALTER OPERATOR CLASS

Changes the definition of an operator class.

Synopsis

```
ALTER OPERATOR CLASS name USING index_method RENAME TO newname

ALTER OPERATOR CLASS name USING index_method OWNER TO newowner
```

Description

`ALTER OPERATOR CLASS` changes the definition of an operator class.

You must own the operator class to use `ALTER OPERATOR CLASS`. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the operator class's schema. (These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the operator class. However, a superuser can alter ownership of any operator class anyway.)

Parameters

name

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

index_method

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

newname

The new name of the operator class.

newowner

The new owner of the operator class

Compatibility

There is no `ALTER OPERATOR CLASS` statement in the SQL standard.

See Also

CREATE OPERATOR CLASS, *DROP OPERATOR CLASS*

ALTER PROTOCOL

Changes the definition of a protocol.

Synopsis

```
ALTER PROTOCOL name RENAME TO newname

ALTER PROTOCOL name OWNER TO newowner
```

Description

`ALTER PROTOCOL` changes the definition of a protocol. Only the protocol name or owner can be altered.

You must own the protocol to use `ALTER PROTOCOL`. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on schema of the conversion.

These restrictions are in place to ensure that altering the owner only makes changes that could by made by dropping and recreating the protocol. Note that a superuser can alter ownership of any protocol.

Parameters

name

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

newname

The new name of the protocol.

newowner

The new owner of the protocol.

Examples

To rename the conversion `GPDBauth` to `GPDB_authentication`:

```
ALTER PROTOCOL GPDBauth RENAME TO GPDB_authentication;
```

To change the owner of the conversion `GPDB_authentication` to `joe`:

```
ALTER PROTOCOL GPDB_authentication OWNER TO joe;
```

Compatibility

There is no `ALTER PROTOCOL` statement in the SQL standard.

ALTER RESOURCE QUEUE

Changes the limits of a resource queue.

Synopsis

```
ALTER RESOURCE QUEUE name WITH ( queue_attribute=value [, ... ] )
```

where *queue_attribute* is:

```
   ACTIVE_STATEMENTS=integer
   MEMORY_LIMIT='memory_units'
   MAX_COST=float
   COST_OVERCOMMIT={TRUE|FALSE}
   MIN_COST=float
   PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX}
```

```
ALTER RESOURCE QUEUE name WITHOUT ( queue_attribute [, ... ] )
```

where *queue_attribute* is:

```
   ACTIVE_STATEMENTS
   MEMORY_LIMIT
   MAX_COST
   COST_OVERCOMMIT
   MIN_COST
```

Note: A resource queue must have either an `ACTIVE_STATEMENTS` or a `MAX_COST` value. Do not remove both these `queue_attributes` from a resource queue.

Description

`ALTER RESOURCE QUEUE` changes the limits of a resource queue. Only a superuser can alter a resource queue. A resource queue must have either an `ACTIVE_STATEMENTS` or a `MAX_COST` value (or it can have both). You can also set or reset priority for a resource queue to control the relative share of available CPU resources used by queries associated with the queue, or memory limit of a resource queue to control the amount of memory that all queries submitted through the queue can consume on a segment host.

`ALTER RESOURCE QUEUE WITHOUT` removes the specified limits on a resource that were previously set. A resource queue must have either an `ACTIVE_STATEMENTS` or a `MAX_COST` value. Do not remove both these `queue_attributes` from a resource queue.

Parameters

name

The name of the resource queue whose limits are to be altered.

ACTIVE_STATEMENTS *integer*

The number of active statements submitted from users in this resource queue allowed on the system at any one time. The value for `ACTIVE_STATEMENTS` should be an integer greater than 0. To reset `ACTIVE_STATEMENTS` to have no limit, enter a value of `-1`.

MEMORY_LIMIT '*memory_units*'

Sets the total memory quota for all statements submitted from users in this resource queue. Memory units can be specified in kB, MB or GB. The minimum memory quota for a resource queue is 10MB. There is no maximum; however the upper boundary at query execution time is limited by the physical memory of a segment host. The default value is no limit (`-1`).

MAX_COST *float*

The total query optimizer cost of statements submitted from users in this resource queue allowed on the system at any one time. The value for `MAX_COST` is specified as a floating point number (for example 100.0) or can also be specified as an exponent (for example 1e+2). To reset `MAX_COST` to have no limit, enter a value of `-1.0`.

COST_OVERCOMMIT *boolean*

If a resource queue is limited based on query cost, then the administrator can allow cost overcommit (`COST_OVERCOMMIT=TRUE`, the default). This means that a query that exceeds the allowed cost threshold will be allowed to run but only when the system is idle. If `COST_OVERCOMMIT=FALSE` is specified, queries that exceed the cost limit will always be rejected and never allowed to run.

MIN_COST *float*

Queries with a cost under this limit will not be queued and run immediately. Cost is measured in units of disk page fetches; 1.0 equals one sequential disk page read. The value for `MIN_COST` is specified as a floating point number (for example 100.0) or can also be specified as an exponent (for example 1e+2). To reset `MIN_COST` to have no limit, enter a value of `-1.0`.

PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX}

Sets the priority of queries associated with a resource queue. Queries or statements in queues with higher priority levels will receive a larger share of available CPU resources in case of contention. Queries in low-priority queues may be delayed while higher priority queries are executed.

Notes

Use *`CREATE ROLE`* or *`ALTER ROLE`* to add a role (user) to a resource queue.

Examples

Change the active query limit for a resource queue:

```
ALTER RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=20);
```

Change the memory limit for a resource queue:

```
ALTER RESOURCE QUEUE myqueue WITH (MEMORY_LIMIT='2GB');
```

Reset the maximum and minimum query cost limit for a resource queue to no limit:

```
ALTER RESOURCE QUEUE myqueue WITH (MAX_COST=-1.0, MIN_COST=
-1.0);
```

Reset the query cost limit for a resource queue to 3^{10} (or 30000000000.0) and do not allow overcommit:

```
ALTER RESOURCE QUEUE myqueue WITH (MAX_COST=3e+10,
COST_OVERCOMMIT=FALSE);
```

Reset the priority of queries associated with a resource queue to the minimum level:

```
ALTER RESOURCE QUEUE myqueue WITH (PRIORITY=MIN);
```

Remove the `MAX_COST` and `MEMORY_LIMIT` limits from a resource queue:

```
ALTER RESOURCE QUEUE myqueue WITHOUT (MAX_COST, MEMORY_LIMIT);
```

Compatibility

The `ALTER RESOURCE QUEUE` statement is a Greenplum Database extension. This command does not exist in standard PostgreSQL.

See Also

CREATE RESOURCE QUEUE, *DROP RESOURCE QUEUE*, *CREATE ROLE*, *ALTER ROLE*

ALTER ROLE

Changes a database role (user or group).

Synopsis

```
ALTER ROLE name RENAME TO newname

ALTER ROLE name SET config_parameter {TO | =} {value | DEFAULT}

ALTER ROLE name RESET config_parameter

ALTER ROLE name RESOURCE QUEUE {queue_name | NONE}

ALTER ROLE name [ [WITH] option [ ... ] ]
```

where *option* can be:

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEEXTTABLE | NOCREATEEXTTABLE
      [ ( attribute='value'[, ...] ) ]
           where attributes and value are:
           type='readable'|'writable'
           protocol='gpfdist'|'http'
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT connlimit
    | [ENCRYPTED | UNENCRYPTED] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | [ DENY deny_point ]
    | [ DENY BETWEEN deny_point AND deny_point]
    | [ DROP DENY FOR deny_point ]
```

Description

`ALTER ROLE` changes the attributes of a Greenplum Database role. There are several variants of this command:

- **RENAME** — Changes the name of the role. Database superusers can rename any role. Roles having `CREATEROLE` privilege can rename non-superuser roles. The current session user cannot be renamed (connect as a different user to rename a role). Because MD5-encrypted passwords use the role name as cryptographic salt, renaming a role clears its password if the password is MD5-encrypted.
- **SET | RESET** — changes a role's session default for a specified configuration parameter. Whenever the role subsequently starts a new session, the specified value becomes the session default, overriding whatever setting is present in server configuration file (`postgresql.conf`). For a role without `LOGIN` privilege, session defaults have no effect. Ordinary roles can change their own session defaults. Superusers can change anyone's session defaults. Roles having `CREATEROLE` privilege can change defaults for non-superuser roles. See the *Greenplum Database Administrator Guide* for information about all user-settable configuration parameters.
- **RESOURCE QUEUE** — Assigns the role to a workload management resource queue. The role would then be subject to the limits assigned to the resource queue when issuing queries. Specify `NONE` to assign the role to the default resource queue. A role can only belong to one resource queue. For a role without `LOGIN` privilege, resource queues have no effect. See *CREATE RESOURCE QUEUE* for more information.
- **WITH *option*** — Changes many of the role attributes that can be specified in *CREATE ROLE*. Attributes not mentioned in the command retain their previous settings. Database superusers can change any of

these settings for any role. Roles having `CREATEROLE` privilege can change any of these settings, but only for non-superuser roles. Ordinary roles can only change their own password.

Parameters

name

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

newname

The new name of the role.

config_parameter=value

Set this role's session default for the specified configuration parameter to the given value. If value is `DEFAULT` or if `RESET` is used, the role-specific variable setting is removed, so the role will inherit the system-wide default setting in new sessions. Use `RESET ALL` to clear all role-specific settings. See *`SET`* and *Server Configuration Parameters* for information about user-settable configuration parameters.

queue_name

The name of the resource queue to which the user-level role is to be assigned. Only roles with `LOGIN` privilege can be assigned to a resource queue. To unassign a role from a resource queue and put it in the default resource queue, specify `NONE`. A role can only belong to one resource queue.

SUPERUSER | NOSUPERUSER
CREATEDB | NOCREATEDB
CREATEROLE | NOCREATEROLE
CREATEEXTTABLE | NOCREATEEXTTABLE [(attribute='value')]

If `CREATEEXTTABLE` is specified, the role being defined is allowed to create external tables. The default `type` is `readable` and the default `protocol` is `gpfdist` if not specified. `NOCREATEEXTTABLE` (the default) denies the role the ability to create external tables. Note that external tables that use the `file` or `execute` protocols can only be created by superusers.

INHERIT | NOINHERIT
LOGIN | NOLOGIN
CONNECTION LIMIT connlimit
PASSWORD password
ENCRYPTED | UNENCRYPTED
VALID UNTIL 'timestamp'

These clauses alter role attributes originally set by *`CREATE ROLE`*.

DENY deny_point
DENY BETWEEN deny_point AND deny_point

The `DENY` and `DENY BETWEEN` keywords set time-based constraints that are enforced at login. `DENY`sets a day or a day and time to deny access. `DENY BETWEEN` sets an interval during which access is denied. Both use the parameter *deny_point* that has following format:

```
DAY day [ TIME 'time' ]
```

The two parts of the `deny_point` parameter use the following formats:

For day:

```
{'Sunday' | 'Monday' | 'Tuesday' |'Wednesday' | 'Thursday' | 'Friday' |
'Saturday' | 0-6 }
```

For `time:`

{ 00-23 : 00-59 | 01-12 : 00-59 { AM | PM }}

The `DENY BETWEEN` clause uses two *deny_point* parameters.

```
DENY BETWEEN deny_point AND deny_point
```

For more information about time-based constraints and examples, see "Managing Roles and Privileges" in the *Greenplum Database Administrator Guide*.

DROP DENY FOR deny_point

The `DROP DENY FOR` clause removes a time-based constraint from the role. It uses the *deny_point* parameter described above.

For more information about time-based constraints and examples, see "Managing Roles and Privileges" in the *Greenplum Database Administrator Guide*.

Notes

Use *GRANT* and *REVOKE* for adding and removing role memberships.

Caution must be exercised when specifying an unencrypted password with this command. The password will be transmitted to the server in clear text, and it might also be logged in the client's command history or the server log. The `psql` command-line client contains a meta-command `\password` that can be used to safely change a role's password.

It is also possible to tie a session default to a specific database rather than to a role. Role-specific settings override database-specific ones if there is a conflict. See *ALTER DATABASE*.

Examples

Change the password for a role:

```
ALTER ROLE daria WITH PASSWORD 'passwd123';
```

Change a password expiration date:

```
ALTER ROLE scott VALID UNTIL 'May 4 12:00:00 2015 +1';
```

Make a password valid forever:

```
ALTER ROLE luke VALID UNTIL 'infinity';
```

Give a role the ability to create other roles and new databases:

```
ALTER ROLE joelle CREATEROLE CREATEDB;
```

Give a role a non-default setting of the `maintenance_work_mem` parameter:

```
ALTER ROLE admin SET maintenance_work_mem = 100000;
```

Assign a role to a resource queue:

```
ALTER ROLE sammy RESOURCE QUEUE poweruser;
```

Give a role permission to create writable external tables:

```
ALTER ROLE load CREATEEXTTABLE (type='writable');
```

Alter a role so it does not allow login access on Sundays:

```
ALTER ROLE user3 DENY DAY 'Sunday';
```

Alter a role to remove the constraint that does not allow login access on Sundays:

```
ALTER ROLE user3 DROP DENY FOR DAY 'Sunday';
```

Compatibility

The `ALTER ROLE` statement is a Greenplum Database extension.

See Also

CREATE ROLE, *DROP ROLE*, *SET*, *CREATE RESOURCE QUEUE*, *GRANT*, *REVOKE*

ALTER SCHEMA

Changes the definition of a schema.

Synopsis

```
ALTER SCHEMA name RENAME TO newname

ALTER SCHEMA name OWNER TO newowner
```

Description

`ALTER SCHEMA` changes the definition of a schema.

You must own the schema to use `ALTER SCHEMA`. To rename a schema you must also have the `CREATE` privilege for the database. To alter the owner, you must also be a direct or indirect member of the new owning role, and you must have the `CREATE` privilege for the database. Note that superusers have all these privileges automatically.

Parameters

name

The name of an existing schema.

newname

The new name of the schema. The new name cannot begin with `pg_`, as such names are reserved for system schemas.

newowner

The new owner of the schema.

Compatibility

There is no `ALTER SCHEMA` statement in the SQL standard.

See Also

CREATE SCHEMA, *DROP SCHEMA*

ALTER SEQUENCE

Changes the definition of a sequence generator.

Synopsis

```
ALTER SEQUENCE name [INCREMENT [ BY ] increment]
     [MINVALUE minvalue | NO MINVALUE]
     [MAXVALUE maxvalue | NO MAXVALUE]
     [RESTART [ WITH ] start]
     [CACHE cache] [[ NO ] CYCLE]
     [OWNED BY {table.column | NONE}]

ALTER SEQUENCE name SET SCHEMA new_schema
```

Description

`ALTER SEQUENCE` changes the parameters of an existing sequence generator. Any parameters not specifically set in the `ALTER SEQUENCE` command retain their prior settings.

You must own the sequence to use `ALTER SEQUENCE`. To change a sequence's schema, you must also have `CREATE` privilege on the new schema. Note that superusers have all these privileges automatically.

Parameters

name

The name (optionally schema-qualified) of a sequence to be altered.

increment

The clause `INCREMENT BY increment` is optional. A positive value will make an ascending sequence, a negative one a descending sequence. If unspecified, the old increment value will be maintained.

minvalue
NO MINVALUE

The optional clause `MINVALUE minvalue` determines the minimum value a sequence can generate. If `NO MINVALUE` is specified, the defaults of 1 and -263-1 for ascending and descending sequences, respectively, will be used. If neither option is specified, the current minimum value will be maintained.

maxvalue
NO MAXVALUE

The optional clause `MAXVALUE maxvalue` determines the maximum value for the sequence. If `NO MAXVALUE` is specified, the defaults are 263-1 and -1 for ascending and descending sequences, respectively, will be used. If neither option is specified, the current maximum value will be maintained.

start

The optional clause `RESTART WITH` start changes the current value of the sequence.

cache

The clause `CACHE cache` 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). If unspecified, the old cache value will be maintained.

CYCLE

The optional `CYCLE` key word may be used to enable the sequence to wrap around when the *maxvalue* or *minvalue* has been reached by an ascending or descending sequence. If the limit is reached, the next number generated will be the respective *minvalue* or *maxvalue*.

NO CYCLE

If the optional `NO CYCLE` key word is specified, any calls to `nextval` after the sequence has reached its maximum value will return an error. If neither `CYCLE` or `NO CYCLE` are specified, the old cycle behavior will be maintained.

OWNED BY *table.column*
OWNED BY NONE

The `OWNED BY` option causes the sequence to be associated with a specific table column, such that if that column (or its whole table) is dropped, the sequence will be automatically dropped as well. If specified, this association replaces any previously specified association for the sequence. The specified table must have the same owner and be in the same schema as the sequence. Specifying `OWNED BY NONE` removes any existing table column association.

new_schema

The new schema for the sequence.

Notes

To avoid blocking of concurrent transactions that obtain numbers from the same sequence, `ALTER SEQUENCE`'s effects on the sequence generation parameters are never rolled back; those changes take effect immediately and are not reversible. However, the `OWNED BY` and `SET SCHEMA` clauses are ordinary catalog updates and can be rolled back.

`ALTER SEQUENCE` will not immediately affect `nextval` results in sessions, other than the current one, that have preallocated (cached) sequence values. They will use up all cached values prior to noticing the changed sequence generation parameters. The current session will be affected immediately.

Some variants of *`ALTER TABLE`* can be used with sequences as well. For example, to rename a sequence use `ALTER TABLE RENAME`.

Examples

Restart a sequence called `serial` at `105`:

```
ALTER SEQUENCE serial RESTART WITH 105;
```

Compatibility

`ALTER SEQUENCE` conforms to the SQL standard, except for the `OWNED BY` and `SET SCHEMA` clauses, which are Greenplum Database extensions.

See Also

`CREATE SEQUENCE`, *`DROP SEQUENCE`*, *`ALTER TABLE`*

ALTER TABLE

Changes the definition of a table.

Synopsis

```
ALTER TABLE [ONLY] name RENAME [COLUMN] column TO new_column

ALTER TABLE name RENAME TO new_name

ALTER TABLE name SET SCHEMA new_schema

ALTER TABLE [ONLY] name SET
     DISTRIBUTED BY (column, [ ... ] )
   | DISTRIBUTED RANDOMLY
   | WITH (REORGANIZE=true|false)

ALTER TABLE [ONLY] name action [, ... ]

ALTER TABLE name
   [ ALTER PARTITION { partition_name | FOR (RANK(number))
   | FOR (value) } partition_action [...] ]
   partition_action
```

where *action* is one of:

```
  ADD [COLUMN] column_name type
      [column_constraint [ ... ]]
  DROP [COLUMN] column [RESTRICT | CASCADE]
  ALTER [COLUMN] column TYPE type [USING expression]
  ALTER [COLUMN] column SET DEFAULT expression
  ALTER [COLUMN] column DROP DEFAULT
  ALTER [COLUMN] column { SET | DROP } NOT NULL
  ALTER [COLUMN] column SET STATISTICS integer
  ADD table_constraint
  DROP CONSTRAINT constraint_name [RESTRICT | CASCADE]
  DISABLE TRIGGER [trigger_name | ALL | USER]
  ENABLE TRIGGER [trigger_name | ALL | USER]
  CLUSTER ON index_name
  SET WITHOUT CLUSTER
  SET WITHOUT OIDS
  SET (FILLFACTOR = value)
  RESET (FILLFACTOR)
  INHERIT parent_table
  NO INHERIT parent_table
  OWNER TO new_owner
  SET TABLESPACE new_tablespace
```

where *partition_action* is one of:

```
  ALTER DEFAULT PARTITION
  DROP DEFAULT PARTITION [IF EXISTS]
  DROP PARTITION [IF EXISTS] { partition_name |
      FOR (RANK(number)) | FOR (value) } [CASCADE]
  TRUNCATE DEFAULT PARTITION
  TRUNCATE PARTITION { partition_name | FOR (RANK(number)) |
      FOR (value) }
  RENAME DEFAULT PARTITION TO new_partition_name
  RENAME PARTITION { partition_name | FOR (RANK(number)) |
      FOR (value) } TO new_partition_name
  ADD DEFAULT PARTITION name [ ( subpartition_spec ) ]
  ADD PARTITION [partition_name] partition_element
     [ ( subpartition_spec ) ]
  EXCHANGE PARTITION { partition_name | FOR (RANK(number)) |
```

```
      FOR (value) } WITH TABLE table_name
       [ WITH | WITHOUT VALIDATION ]
  EXCHANGE DEFAULT PARTITION WITH TABLE table_name
   [ WITH | WITHOUT VALIDATION ]
  SET SUBPARTITION TEMPLATE (subpartition_spec)
  SPLIT DEFAULT PARTITION
    {  AT (list_value)
     | START([datatype] range_value) [INCLUSIVE | EXCLUSIVE]
        END([datatype] range_value) [INCLUSIVE | EXCLUSIVE] }
    [ INTO ( PARTITION new_partition_name,
             PARTITION default_partition_name ) ]
  SPLIT PARTITION { partition_name | FOR (RANK(number)) |
     FOR (value) } AT (value)
    [ INTO (PARTITION partition_name, PARTITION partition_name)]
```

where *partition_element* is:

```
    VALUES (list_value [,...] )
  | START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
     [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
  | END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
[ TABLESPACE tablespace ]
```

where *subpartition_spec* is:

```
subpartition_element [, ...]
```

and *subpartition_element* is:

```
   DEFAULT SUBPARTITION subpartition_name
  | [SUBPARTITION subpartition_name] VALUES (list_value [,...] )
  | [SUBPARTITION subpartition_name]
     START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
     [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
     [ EVERY ( [number | datatype] 'interval_value') ]
  | [SUBPARTITION subpartition_name]
     END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
     [ EVERY ( [number | datatype] 'interval_value') ]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
[ TABLESPACE tablespace ]
```

where *storage_parameter* is:

```
   APPENDONLY={TRUE|FALSE}
   BLOCKSIZE={8192-2097152}
   ORIENTATION={COLUMN|ROW}
   COMPRESSTYPE={ZLIB|QUICKLZ|RLE_TYPE|NONE}
   COMPRESSLEVEL={0-9}
   FILLFACTOR={10-100}
   OIDS[=TRUE|FALSE]
```

Description

`ALTER TABLE` changes the definition of an existing table. There are several subforms:

- **ADD COLUMN** — Adds a new column to the table, using the same syntax as `CREATE TABLE`.
- **DROP COLUMN** — Drops a column from a table. Note that if you drop table columns that are being used as the Greenplum Database distribution key, the distribution policy for the table will be changed to `DISTRIBUTED RANDOMLY`. Indexes and table constraints involving the column will be automatically dropped as well. You will need to say `CASCADE` if anything outside the table depends on the column (such as views).
- **ALTER COLUMN TYPE** — Changes the data type of a column of a table. Note that you cannot alter column data types that are being used as distribution or partitioning keys. Indexes and simple

table constraints involving the column will be automatically converted to use the new column type by reparsing the originally supplied expression. The optional `USING` clause specifies how to compute the new column value from the old. If omitted, the default conversion is the same as an assignment cast from old data type to new. A `USING` clause must be provided if there is no implicit or assignment cast from old to new type.

- **SET/DROP DEFAULT** — Sets or removes the default value for a column. The default values only apply to subsequent `INSERT` commands. They do not cause rows already in the table to change. Defaults may also be created for views, in which case they are inserted into statements on the view before the view's `ON INSERT` rule is applied.
- **SET/DROP NOT NULL** — Changes whether a column is marked to allow null values or to reject null values. You can only use `SET NOT NULL` when the column contains no null values.
- **SET STATISTICS** — Sets the per-column statistics-gathering target for subsequent `ANALYZE` operations. The target can be set in the range 0 to 1000, or set to -1 to revert to using the system default statistics target (`default_statistics_target`).
- **ADD *table_constraint*** — Adds a new constraint to a table (not just a partition) using the same syntax as `CREATE TABLE`.
- **DROP CONSTRAINT** — Drops the specified constraint on a table.
- **DISABLE/ENABLE TRIGGER** — Disables or enables trigger(s) belonging to the table. A disabled trigger is still known to the system, but is not executed when its triggering event occurs. For a deferred trigger, the enable status is checked when the event occurs, not when the trigger function is actually executed. One may disable or enable a single trigger specified by name, or all triggers on the table, or only user-created triggers. Disabling or enabling constraint triggers requires superuser privileges.

  Note: triggers are not supported in Greenplum Database. Triggers in general have very limited functionality due to the parallelism of Greenplum Database.
- **CLUSTER/SET WITHOUT CLUSTER** — Selects or removes the default index for future `CLUSTER` operations. It does not actually re-cluster the table. Note that `CLUSTER` is not the recommended way to physically reorder a table in Greenplum Database because it takes so long. It is better to recreate the table with *`CREATE TABLE AS`* and order it by the index column(s).
- **SET WITHOUT OIDS** — Removes the OID system column from the table. Note that there is no variant of `ALTER TABLE` that allows OIDs to be restored to a table once they have been removed.
- **SET (FILLFACTOR = *value*) / RESET (FILLFACTOR)** — Changes the fillfactor for the table. The fillfactor for a table is a percentage between 10 and 100. 100 (complete packing) is the default. When a smaller fillfactor is specified, `INSERT` operations pack table pages only to the indicated percentage; the remaining space on each page is reserved for updating rows on that page. This gives `UPDATE` a chance to place the updated copy of a row on the same page as the original, which is more efficient than placing it on a different page. For a table whose entries are never updated, complete packing is the best choice, but in heavily updated tables smaller fillfactors are appropriate. Note that the table contents will not be modified immediately by this command. You will need to rewrite the table to get the desired effects.
- **SET DISTRIBUTED** —Changes the distribution policy of a table. Changes to a hash distribution policy will cause the table data to be physically redistributed on disk, which can be resource intensive.
- **INHERIT *parent_table* / NO INHERIT *parent_table*** — Adds or removes the target table as a child of the specified parent table. Queries against the parent will include records of its child table. To be added as a child, the target table must already contain all the same columns as the parent (it could have additional columns, too). The columns must have matching data types, and if they have `NOTNULL` constraints in the parent then they must also have `NOT NULL` constraints in the child. There must also be matching child-table constraints for all `CHECK` constraints of the parent.
- **OWNER** — Changes the owner of the table, sequence, or view to the specified user.
- **SET TABLESPACE** — Changes the table's tablespace to the specified tablespace and moves the data file(s) associated with the table to the new tablespace. Indexes on the table, if any, are not moved; but they can be moved separately with additional `SET TABLESPACE` commands. See also `CREATE TABLESPACE`. If changing the tablespace of a partitioned table, all child table partitions will also be moved to the new tablespace.

- **RENAME** — Changes the name of a table (or an index, sequence, or view) or the name of an individual column in a table. There is no effect on the stored data. Note that Greenplum Database distribution key columns cannot be renamed.
- **SET SCHEMA** — Moves the table into another schema. Associated indexes, constraints, and sequences owned by table columns are moved as well.
- **ALTER PARTITION | DROP PARTITION | RENAME PARTITION | TRUNCATE PARTITION | ADD PARTITION | SPLIT PARTITION | EXCHANGE PARTITION | SET SUBPARTITION TEMPLATE** — Changes the structure of a partitioned table. In most cases, you must go through the parent table to alter one of its child table partitions.

  Note: If you add a partition to a table that has subpartition encodings, the new partition inherits the storage directives for the subpartitions. For more information about the precedence of compression settings, see "Using Compression" in the Greenplum Database Administrator Guide.

You must own the table to use `ALTER TABLE`. To change the schema of a table, you must also have `CREATE` privilege on the new schema. To add the table as a new child of a parent table, you must own the parent table as well. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the table's schema. A superuser has these privileges automatically.

Note: Memory usage increases significantly when a table has many partitions, if a table has compression, or if the blocksize for a table is large. If the number of relations associated with the table is large, this condition can force an operation on the table to use more memory. For example, if the table is a CO table and has a large number of columns, each column is a relation. An operation like `ALTER TABLE ALTER COLUMN` opens all the columns in the table allocates associated buffers. If a CO table has 40 columns and 100 partitions, and the columns are compressed and the blocksize is 2 MB (with a system factor of 3), the system attempts to allocate 24 GB, that is (40 ×100) × (2 ×3) MB or 24 GB.

Parameters

ONLY

Only perform the operation on the table name specified. If the `ONLY` keyword is not used, the operation will be performed on the named table and any child table partitions associated with that table.

name

The name (possibly schema-qualified) of an existing table to alter. If `ONLY` is specified, only that table is altered. If `ONLY` is not specified, the table and all its descendant tables (if any) are updated.

Note: Constraints can only be added to an entire table, not to a partition. Because of that restriction, the *name* parameter can only contain a table name, not a partition name.

column

Name of a new or existing column. Note that Greenplum Database distribution key columns must be treated with special care. Altering or dropping these columns can change the distribution policy for the table.

new_column

New name for an existing column.

new_name

New name for the table.

type

Data type of the new column, or new data type for an existing column. If changing the data type of a Greenplum distribution key column, you are only allowed to change it to a compatible type (for example, `text` to `varchar` is OK, but `text` to `int` is not).

table_constraint

New table constraint for the table. Note that foreign key constraints are currently not supported in Greenplum Database. Also a table is only allowed one unique constraint and the uniqueness must be within the Greenplum Database distribution key.

constraint_name

Name of an existing constraint to drop.

CASCADE

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

RESTRICT

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

trigger_name

Name of a single trigger to disable or enable. Note that Greenplum Database does not support triggers.

ALL

Disable or enable all triggers belonging to the table including constraint related triggers. This requires superuser privilege.

USER

Disable or enable all user-created triggers belonging to the table.

index_name

The index name on which the table should be marked for clustering. Note that `CLUSTER` is not the recommended way to physically reorder a table in Greenplum Database because it takes so long. It is better to recreate the table with *`CREATE TABLE AS`* and order it by the index column(s).

FILLFACTOR

Set the fillfactor percentage for a table.

value

The new value for the `FILLFACTOR` parameter, which is a percentage between 10 and 100. 100 is the default.

DISTRIBUTED BY (*column*) | DISTRIBUTED RANDOMLY

Specifies the distribution policy for a table. Changing a hash distribution policy will cause the table data to be physically redistributed on disk, which can be resource intensive. If you declare the same hash distribution policy or change from hash to random distribution, data will not be redistributed unless you declare `SET WITH (REORGANIZE=true)`.

REORGANIZE=true|false

Use `REORGANIZE=true` when the hash distribution policy has not changed or when you have changed from a hash to a random distribution, and you want to redistribute the data anyways.

parent_table

A parent table to associate or de-associate with this table.

new_owner

The role name of the new owner of the table.

new_tablespace

The name of the tablespace to which the table will be moved.

new_schema

The name of the schema to which the table will be moved.

parent_table_name

When altering a partitioned table, the name of the top-level parent table.

ALTER [DEFAULT] PARTITION

If altering a partition deeper than the first level of partitions, the `ALTER PARTITION` clause is used to specify which subpartition in the hierarchy you want to alter.

DROP [DEFAULT] PARTITION

Drops the specified partition. If the partition has subpartitions, the subpartitions are automatically dropped as well.

TRUNCATE [DEFAULT] PARTITION

Truncates the specified partition. If the partition has subpartitions, the subpartitions are automatically truncated as well.

RENAME [DEFAULT] PARTITION

Changes the partition name of a partition (not the relation name). Partitioned tables are created using the naming convention: `<`*parentname*`>_<`*level*`>_prt_<`*partition_name*`>`.

ADD DEFAULT PARTITION

Adds a default partition to an existing partition design. When data does not match to an existing partition, it is inserted into the default partition. Partition designs that do not have a default partition will reject incoming rows that do not match to an existing partition. Default partitions must be given a name.

ADD PARTITION

partition_element - Using the existing partition type of the table (range or list), defines the boundaries of new partition you are adding.

name - A name for this new partition.

VALUES - For list partitions, defines the value(s) that the partition will contain.

START - For range partitions, defines the starting range value for the partition. By default, start values are `INCLUSIVE`. For example, if you declared a start date of '`2008-01-01`', then the partition would contain all dates greater than or equal to '`2008-01-01`'. Typically the data type of the `START` expression is the same type as the partition key column. If that is not the case, then you must explicitly cast to the intended data type.

END - For range partitions, defines the ending range value for the partition. By default, end values are `EXCLUSIVE`. For example, if you declared an end date of '`2008-02-01`', then the partition would contain all dates less than but not equal to '`2008-02-01`'. Typically the data type of the `END` expression is the same type as the partition key column. If that is not the case, then you must explicitly cast to the intended data type.

WITH - Sets the table storage options for a partition. For example, you may want older partitions to be append-optimized tables and newer partitions to be regular heap tables. See *`CREATE TABLE`* for a description of the storage options.

TABLESPACE - The name of the tablespace in which the partition is to be created.

subpartition_spec - Only allowed on partition designs that were created without a subpartition template. Declares a subpartition specification for the new partition you are adding. If the partitioned table was originally defined using a subpartition template, then the template will be used to generate the subpartitions automatically.

EXCHANGE [DEFAULT] PARTITION

Exchanges another table into the partition hierarchy into the place of an existing partition. In a multi-level partition design, you can only exchange the lowest level partitions (those that contain data).

The Greenplum Database server configuration parameter `gp_enable_exchange_default_partition` controls availability of the `EXCHANGE DEFAULT`

PARTITION clause. The default value for the parameter is off. The clause is not available and Greenplum Database returns an error if the clause is specified in an ALTER TABLE command.

For information about the parameter, see *Server Configuration Parameters*.

> **Warning:** Before you exchange the default partition, you must ensure the data in the table to be exchanged, the new default partition, is valid for the default partition. For example, the data in the new default partition must not contain data that would be valid in other leaf child partitions of the partitioned table. Otherwise, queries against the partitioned table with the exchanged default partition that are executed by the Pivotal Query Optimizer might return incorrect results.

WITH TABLE *table_name* - The name of the table you are swapping into the partition design. You can exchange a table where the table data is stored in the database. For example, the table is created with the CREATE TABLE command.

With the EXCHANGE PARTITION clause, you can also exchange a readable external table (created with the CREATE EXTERNAL TABLE command) into the partition hierarchy in the place of an existing leaf child partition. If you specify a readable external table, you must also specify the WITHOUT VALIDATION clause to skip table validation against the CHECK constraint of the partition you are exchanging.

Exchanging a leaf child partition with an external table is not supported in these cases:

- The partitioned table is created with the SUBPARITION clause or if a partition has a subpartition.
- The partitioned table contains a column with a check constraint or a NOT NULL constraint.

WITH | WITHOUT VALIDATION - Validates that the data in the table matches the CHECK constraint of the partition you are exchanging. The default is to validate the data against the CHECK constraint.

> **Warning:** If you specify the WITHOUT VALIDATION clause, you must ensure that the data in table that you are exchanging for an existing child leaf partition is valid against the CHECK constraints on the partition. Otherwise, queries against the partitioned table might return incorrect results.

SET SUBPARTITION TEMPLATE

Modifies the subpartition template for an existing partition. After a new subpartition template is set, all new partitions added will have the new subpartition design (existing partitions are not modified).

SPLIT DEFAULT PARTITION

Splits a default partition. In a multi-level partition design, you can only split the lowest level default partitions (those that contain data). Splitting a default partition creates a new partition containing the values specified and leaves the default partition containing any values that do not match to an existing partition.

AT - For list partitioned tables, specifies a single list value that should be used as the criteria for the split.

START - For range partitioned tables, specifies a starting value for the new partition.

END - For range partitioned tables, specifies an ending value for the new partition.

INTO - Allows you to specify a name for the new partition. When using the INTO clause to split a default partition, the second partition name specified should always be that of the existing default partition. If you do not know the name of the default partition, you can look it up using the *pg_partitions* view.

SPLIT PARTITION

Splits an existing partition into two partitions. In a multi-level partition design, you can only split the lowest level partitions (those that contain data).

AT - Specifies a single value that should be used as the criteria for the split. The partition will be divided into two new partitions with the split value specified being the starting range for the *latter* partition.

INTO - Allows you to specify names for the two new partitions created by the split.

partition_name

The given name of a partition.

FOR (RANK(number))

For range partitions, the rank of the partition in the range.

FOR ('*value*')

Specifies a partition by declaring a value that falls within the partition boundary specification. If the value declared with `FOR` matches to both a partition and one of its subpartitions (for example, if the value is a date and the table is partitioned by month and then by day), then `FOR` will operate on the first level where a match is found (for example, the monthly partition). If your intent is to operate on a subpartition, you must declare so as follows: `ALTER TABLE name ALTER PARTITION FOR ('2008-10-01') DROP PARTITION FOR ('2008-10-01');`

Notes

The table name specified in the `ALTER TABLE` command cannot be the following table names:

- The name of a partition within a table.
- The name of a table specified in the `LOG ERRORS INTO` clause of the `CREATE EXTERNAL TABLE` command.

Take special care when altering or dropping columns that are part of the Greenplum Database distribution key as this can change the distribution policy for the table.

Greenplum Database does not currently support foreign key constraints. For a unique constraint to be enforced in Greenplum Database, the table must be hash-distributed (not `DISTRIBUTED RANDOMLY`), and all of the distribution key columns must be the same as the initial columns of the unique constraint columns.

Adding a `CHECK` or `NOT NULL` constraint requires scanning the table to verify that existing rows meet the constraint.

When a column is added with `ADD COLUMN`, all existing rows in the table are initialized with the column's default value (`NULL` if no `DEFAULT` clause is specified). Adding a column with a non-null default or changing the type of an existing column will require the entire table to be rewritten. This may take a significant amount of time for a large table; and it will temporarily require double the disk space.

You can specify multiple changes in a single `ALTER TABLE` command, which will be done in a single pass over the table.

The `DROP COLUMN` form does not physically remove the column, but simply makes it invisible to SQL operations. Subsequent insert and update operations in the table will store a null value for the column. Thus, dropping a column is quick but it will not immediately reduce the on-disk size of your table, as the space occupied by the dropped column is not reclaimed. The space will be reclaimed over time as existing rows are updated.

The fact that `ALTER TYPE` requires rewriting the whole table is sometimes an advantage, because the rewriting process eliminates any dead space in the table. For example, to reclaim the space occupied by a dropped column immediately, the fastest way is: `ALTER TABLE table ALTER COLUMN anycol TYPE sametype;` where *anycol* is any remaining table column and *sametype* is the same type that column already has. This results in no semantically-visible change in the table, but the command forces rewriting, which gets rid of no-longer-useful data.

If a table is partitioned or has any descendant tables, it is not permitted to add, rename, or change the type of a column in the parent table without doing the same to the descendants. This ensures that the descendants always have columns matching the parent.

To see the structure of a partitioned table, you can use the view *pg_partitions*. This view can help identify the particular partitions you may want to alter.

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

The `TRIGGER`, `CLUSTER`, `OWNER`, and `TABLESPACE` actions never recurse to descendant tables; that is, they always act as though `ONLY` were specified. Adding a constraint can recurse only for `CHECK` constraints.

These `ALTER PARTITION` operations are supported if no data is changed on a partitioned table that contains a leaf child partition that has been exchanged to use an external table Otherwise, an error is returned.

- Adding or dropping a column.
- Changing the data type of column.

These `ALTER PARTITION` operations are not supported for a partitioned table that contains a leaf child partition that has been exchanged to use an external table:

- Setting a subpartition template.
- Altering the partition properties.
- Creating a default partition.
- Setting a distribution policy.
- Setting or dropping a `NOT NULL` constraint of column.
- Adding or dropping constraints.
- Splitting an external partition.

Changing any part of a system catalog table is not permitted.

Examples

Add a column to a table:

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

Rename an existing column:

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

Rename an existing table:

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

Add a not-null constraint to a column:

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

Add a check constraint to a table:

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

Move a table to a different schema:

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

Add a new partition to a partitioned table:

```
ALTER TABLE sales ADD PARTITION
            START (date '2009-02-01') INCLUSIVE
            END (date '2009-03-01') EXCLUSIVE;
```

Add a default partition to an existing partition design:

```
ALTER TABLE sales ADD DEFAULT PARTITION other;
```

Rename a partition:

```
ALTER TABLE sales RENAME PARTITION FOR ('2008-01-01') TO
jan08;
```

Drop the first (oldest) partition in a range sequence:

```
ALTER TABLE sales DROP PARTITION FOR (RANK(1));
```

Exchange a table into your partition design:

```
ALTER TABLE sales EXCHANGE PARTITION FOR ('2008-01-01') WITH
TABLE jan08;
```

Split the default partition (where the existing default partition's name is `other`) to add a new monthly partition for January 2009:

```
ALTER TABLE sales SPLIT DEFAULT PARTITION
START ('2009-01-01') INCLUSIVE
END ('2009-02-01') EXCLUSIVE
INTO (PARTITION jan09, PARTITION other);
```

Split a monthly partition into two with the first partition containing dates January 1-15 and the second partition containing dates January 16-31:

```
ALTER TABLE sales SPLIT PARTITION FOR ('2008-01-01')
AT ('2008-01-16')
INTO (PARTITION jan081to15, PARTITION jan0816to31);
```

Compatibility

The `ADD`, `DROP`, and `SET DEFAULT` forms conform with the SQL standard. The other forms are Greenplum Database extensions of the SQL standard. Also, the ability to specify more than one manipulation in a single `ALTER TABLE` command is an extension.

`ALTER TABLE DROP COLUMN` can be used to drop the only column of a table, leaving a zero-column table. This is an extension of SQL, which disallows zero-column tables.

See Also

CREATE TABLE, *DROP TABLE*

ALTER TABLESPACE

Changes the definition of a tablespace.

Synopsis

```
ALTER TABLESPACE name RENAME TO newname

ALTER TABLESPACE name OWNER TO newowner
```

Description

`ALTER TABLESPACE` changes the definition of a tablespace.

You must own the tablespace to use `ALTER TABLESPACE`. To alter the owner, you must also be a direct or indirect member of the new owning role. (Note that superusers have these privileges automatically.)

Parameters

name

The name of an existing tablespace.

newname

The new name of the tablespace. The new name cannot begin with *pg_* or *gp_* (reserved for system tablespaces).

newowner

The new owner of the tablespace.

Examples

Rename tablespace `index_space` to `fast_raid`:

```
ALTER TABLESPACE index_space RENAME TO fast_raid;
```

Change the owner of tablespace `index_space`:

```
ALTER TABLESPACE index_space OWNER TO mary;
```

Compatibility

There is no `ALTER TABLESPACE` statement in the SQL standard.

See Also

`CREATE TABLESPACE`, *`DROP TABLESPACE`*

ALTER TYPE

Changes the definition of a data type.

Synopsis

```
ALTER TYPE name
   OWNER TO new_owner | SET SCHEMA new_schema
```

Description

`ALTER TYPE` changes the definition of an existing type. You can change the owner and the schema of a type.

You must own the type to use `ALTER TYPE`. To change the schema of a type, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the type's schema. (These restrictions enforce that altering the owner does not do anything that could be done by dropping and recreating the type. However, a superuser can alter ownership of any type.)

Parameters

name

The name (optionally schema-qualified) of an existing type to alter.

new_owner

The user name of the new owner of the type.

new_schema

The new schema for the type.

Examples

To change the owner of the user-defined type `email` to `joe`:

```
ALTER TYPE email OWNER TO joe;
```

To change the schema of the user-defined type `email` to `customers`:

```
ALTER TYPE email SET SCHEMA customers;
```

Compatibility

There is no `ALTER TYPE` statement in the SQL standard.

See Also

CREATE TYPE, *DROP TYPE*

ALTER USER

Changes the definition of a database role (user).

Synopsis

```
ALTER USER name RENAME TO newname

ALTER USER name SET config_parameter {TO | =} {value | DEFAULT}

ALTER USER name RESET config_parameter

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

where *option* can be:

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
```

Description

`ALTER USER` is a deprecated command but is still accepted for historical reasons. It is an alias for `ALTER ROLE`. See *ALTER ROLE* for more information.

Compatibility

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

See Also

ALTER ROLE

ANALYZE

Collects statistics about a database.

Synopsis

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

Description

`ANALYZE` collects statistics about the contents of tables in the database, and stores the results in the system table *pg_statistic*. Subsequently, Greenplum Database uses these statistics to help determine the most efficient execution plans for queries.

With no parameter, `ANALYZE` collects statistics for every table in the current database. You can specify a table name to collect statistics for a single table. You can specify a set of column names, in which case the statistics only for those columns are collected.

`ANALYZE` does not collect statistics on external tables.

> **Important:** If you intend to execute queries on partitioned tables with the Pivotal Query Optimizer enabled, you must collect statistics on the root partition of the partitioned table with the `ANALYZE ROOTPARTITION` command. For information about the Pivotal Query Optimizer, see "Querying Data" in the *Greenplum Database Administrator Guide*.

> **Note:** You can also use the Greenplum Database utility `analyzedb` to update table statistics. The `analyzedb` utility can update statistics for multiple tables concurrently. The utility can also check table statistics and update statistics only if the statistics are not current or do not exist. For information about the utility, see the *Greenplum Database Utility Guide*.

Parameters

ROOTPARTITION [ALL]

Collect statistics only on the root partition of partitioned tables. When you specify `ROOTPARTITION`, you must specify either `ALL` or the name of a partitioned table.

If you specify `ALL` with `ROOTPARTITION`, Greenplum Database collects statistics for the root partition of all partitioned tables in the database. If there are no partitioned tables in the database, a message stating that there are no partitioned tables is returned. For tables that are not partitioned tables, statistics are not collected.

If you specify a table name with `ROOTPARTITION` and the table is not a partitioned table, no statistics are collected for the table and a warning message is returned.

The `ROOTPARTITION` clause is not valid with `VACUUM ANALYZE`. The command `VACUUM ANALYZE ROOTPARTITION` returns an error.

The cost of refreshing the root level statistics is comparable to analyzing one leaf partition.

For the partitioned table *sales_curr_yr*, this example command collects statistics only on the root partition of the partitioned table. `ANALYZE ROOTPARTITION sales_curr_yr;`

This example `ANALYZE` command collects statistics on the root partition of all the partitioned tables in the database.

```
ANALYZE ROOTPARTITION ALL;
```

VERBOSE

Enables display of progress messages. Enables display of progress messages. When specified, `ANALYZE` emits this information

- The table that is being processed.
- The query that is executed to generate the sample table.
- The column for which statistics is being computed.
- The queries that are issued to collect the different statistics for a single column.
- The statistics that are generated.

table

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

column

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

Notes

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

`ANALYZE` requires `SHARE UPDATE EXCLUSIVE` lock on the target table. This lock conflicts with these locks: `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE`, `ACCESS EXCLUSIVE`.

For a partitioned table, specifying which portion of the table to analyze, the root partition or subpartitions (leaf child tables) can be useful if the partitioned table has large number of partitions that have been analyzed and only a few leaf child tables have changed.

- When you run `ANALYZE` on the root partitioned table, statistics are collected for all the leaf child tables (the lowest-level tables in the hierarchy of child tables created by Greenplum Database for use by the partitioned table).
- When you run `ANALYZE` on a leaf child table, statistics are collected only for that leaf child table. When you run `ANALYZE` on a child table that is not a leaf child table, statistics are not collected.

  For example, you can create a partitioned table with partitions for the years 2000 to 2010 and subpartitions for each month in each year. If you run `ANALYZE` on the child table for the year 2005 no statistics are collected. If you run `ANALYZE` on the leaf child table for March of 2005, statistics are collected only for that leaf child table.

  > **Note:** When you create a partitioned table with the `CREATE TABLE` command, Greenplum Database creates the table that you specify (the root partition or parent table), and also creates a hierarchy of tables based on the partition hierarchy that you specified (the child tables). Partitioned tables, child tables and their inheritance level relationships are tracked in the system view *pg_partitions*.

For a partitioned table that contains a leaf child partition that has been exchanged to use an external table, `ANALYZE` does not collect statistics for the external table partition:

- If `ANALYZE [ROOTPARTITION]` is run, external table partitions are not analyzed and root table statistics do not include external table partition.
- If `ANALYZE` is run on an external table partition, the partition is not analyzed.
- If the `VERBOSE` clause is specified, an informational message is displayed: `skipping external table`.

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

For large tables, ANALYZE takes a random sample of the table contents, rather than examining every row. This allows even very large tables to be analyzed in a small amount of time. Note, however, that the statistics are only approximate, and will change slightly each time ANALYZE is run, even if the actual table contents did not change. This may result in small changes in the planner's estimated costs shown by EXPLAIN. In rare situations, this non-determinism will cause the query optimizer to choose a different query plan between runs of ANALYZE. To avoid this, raise the amount of statistics collected by ANALYZE by adjusting the *default_statistics_target* configuration parameter, or on a column-by-column basis by setting the per-column statistics target with ALTER TABLE ... ALTER COLUMN ... SET STATISTICS (see ALTER TABLE). The target value sets the maximum number of entries in the most-common-value list and the maximum number of bins in the histogram. The default target value is 10, but this can be adjusted up or down to trade off accuracy of planner estimates against the time taken for ANALYZE and the amount of space occupied in pg_statistic. In particular, setting the statistics target to zero disables collection of statistics for that column. It may be useful to do that for columns that are never used as part of the WHERE, GROUP BY, or ORDER BY clauses of queries, since the planner will have no use for statistics on such columns.

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

When Greenplum Database performs an ANALYZE operation to collect statistics for a table and detects that all the sampled table data pages are empty (do not contain valid data), Greenplum Database displays a message that a VACUUM FULL operation should be performed. If the sampled pages are empty, the table statistics will be inaccurate. Pages become empty after a large number of changes to the table, for example deleting a large number of rows. A VACUUM FULL operation removes the empty pages and allows an ANALYZE operation to collect accurate statistics.

Examples

Collect statistics for the table mytable:

```
ANALYZE mytable;
```

Compatibility

There is no ANALYZE statement in the SQL standard.

See Also

ALTER TABLE, *EXPLAIN*, *VACUUM*, analyzedb utility in the *Greenplum Database Utility Guide*.

BEGIN

Starts a transaction block.

Synopsis

```
BEGIN [WORK | TRANSACTION] [transaction_mode]
      [READ ONLY | READ WRITE]
```

where *transaction_mode* is one of:

```
ISOLATION LEVEL | {SERIALIZABLE | READ COMMITTED | READ UNCOMMITTED}
```

Description

`BEGIN` initiates a transaction block, that is, all statements after a `BEGIN` command will be executed in a single transaction until an explicit `COMMIT` or `ROLLBACK` is given. By default (without `BEGIN`), Greenplum Database executes transactions in autocommit mode, that is, each 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).

Statements are executed more quickly in a transaction block, because transaction start/commit requires significant CPU and disk activity. Execution of multiple statements inside a transaction is also useful to ensure consistency when making several related changes: other sessions will be unable to see the intermediate states wherein not all the related updates have been done.

If the isolation level or read/write mode is specified, the new transaction has those characteristics, as if *SET TRANSACTION* was executed.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

SERIALIZABLE
READ COMMITTED
READ UNCOMMITTED

The SQL standard defines four transaction isolation levels: `READ COMMITTED`, `READ UNCOMMITTED`, `SERIALIZABLE`, and `REPEATABLE READ`. The default behavior is that a statement can only see rows committed before it began (`READ COMMITTED`). In Greenplum Database `READ UNCOMMITTED` is treated the same as `READ COMMITTED`. `REPEATABLE READ` is not supported; use `SERIALIZABLE` if this behavior is required. `SERIALIZABLE` is the strictest transaction isolation. This level emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. Applications using this level must be prepared to retry transactions due to serialization failures.

READ WRITE
READ ONLY

Determines whether the transaction is read/write or read-only. Read/write is the default. When a transaction is read-only, the following SQL commands are disallowed: `INSERT`, `UPDATE`, `DELETE`, and `COPY FROM` if the table they would write to is not a temporary table; all `CREATE`, `ALTER`, and `DROP` commands; `GRANT`, `REVOKE`, `TRUNCATE`; and `EXPLAIN ANALYZE` and `EXECUTE` if the command they would execute is among those listed.

Notes

START TRANSACTION has the same functionality as `BEGIN`.

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

Issuing `BEGIN` when already inside a transaction block will provoke a warning message. The state of the transaction is not affected. To nest transactions within a transaction block, use savepoints (see `SAVEPOINT`).

Examples

To begin a transaction block:

```
BEGIN;
```

To begin a transaction block with the serializable isolation level:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Compatibility

`BEGIN` is a Greenplum Database language extension. It is equivalent to the SQL-standard command *START TRANSACTION*.

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

See Also

COMMIT, *ROLLBACK*, *START TRANSACTION*, *SAVEPOINT*

CHECKPOINT

Forces a transaction log checkpoint.

Synopsis

```
CHECKPOINT
```

Description

Write-Ahead Logging (WAL) puts a checkpoint in the transaction log every so often. The automatic checkpoint interval is set per Greenplum Database segment instance by the server configuration parameters *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.

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

Compatibility

The `CHECKPOINT` command is a Greenplum Database language extension.

CLOSE

Closes a cursor.

Synopsis

```
CLOSE cursor_name
```

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.

Every non-holdable open cursor is implicitly closed when a transaction is terminated by `COMMIT` or `ROLLBACK`. A holdable cursor is implicitly closed if the transaction that created it aborts via `ROLLBACK`. If the creating transaction successfully commits, the holdable cursor remains open until an explicit `CLOSE` is executed, or the client disconnects.

Parameters

cursor_name

The name of an open cursor to close.

Notes

Greenplum Database does not have an explicit `OPEN` cursor statement. A cursor is considered open when it is declared. Use the `DECLARE` statement to declare (and open) a cursor.

You can see all available cursors by querying the `pg_cursors` system view.

Examples

Close the cursor `portala`:

```
CLOSE portala;
```

Compatibility

`CLOSE` is fully conforming with the SQL standard.

See Also

DECLARE, *FETCH*, *MOVE*

CLUSTER

Physically reorders a heap storage table on disk according to an index. Not a recommended operation in Greenplum Database.

Synopsis

```
CLUSTER indexname ON tablename

CLUSTER tablename

CLUSTER
```

Description

`CLUSTER` orders a heap storage table based on an index. `CLUSTER` is not supported on append-optmized storage tables. Clustering an index means that the records are physically ordered on disk according to the index information. If the records you need are distributed randomly on disk, then the database has to seek across the disk to get the records requested. If those records are stored more closely together, then the fetching from disk is more sequential. A good example for a clustered index is on a date column where the data is ordered sequentially by date. A query against a specific date range will result in an ordered fetch from the disk, which leverages faster sequential access.

Clustering is a one-time operation: when the table is subsequently updated, the changes are not clustered. That is, no attempt is made to store new or updated rows according to their index order. If one wishes, one can periodically recluster by issuing the command again.

When a table is clustered using this command, Greenplum Database remembers on which index it was clustered. The form `CLUSTER` *tablename* reclusters the table on the same index that it was clustered before. `CLUSTER` without any parameter reclusters all previously clustered tables in the current database that the calling user owns, or all tables if called by a superuser. This form of `CLUSTER` cannot be executed inside a transaction block.

When a table is being clustered, an `ACCESS EXCLUSIVE` lock is acquired on it. This prevents any other database operations (both reads and writes) from operating on the table until the `CLUSTER` is finished.

Parameters

indexname

The name of an index.

tablename

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

Notes

In cases where you are accessing single rows randomly within a table, the actual order of the data in the 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`. 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 table page for the first row that matches, all other rows that match are probably already on the same table page, and so you save disk accesses and speed up the query.

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

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

There is another way to cluster data. The `CLUSTER` command reorders the original table by scanning it using the index you specify. This can be slow on large tables because the rows are fetched from the table in index order, and if the table is disordered, the entries are on random pages, so there is one disk page retrieved for every row moved. (Greenplum Database has a cache, but the majority of a big table will not fit in the cache.) The other way to cluster a table is to use a statement such as:

```
CREATE TABLE newtable AS SELECT * FROM table ORDER BY column;
```

This uses the Greenplum Database sorting code to produce the desired order, which is usually much faster than an index scan for disordered data. Then you drop the old table, use `ALTER TABLE ... RENAME` to rename *newtable* to the old name, and recreate the table's indexes. The big disadvantage of this approach is that it does not preserve OIDs, constraints, granted privileges, and other ancillary properties of the table — all such items must be manually recreated. Another disadvantage is that this way requires a sort temporary file about the same size as the table itself, so peak disk usage is about three times the table size instead of twice the table size.

Note: `CLUSTER` is not supported with append-optimized tables.

Examples

Cluster the table `employees` on the basis of its index `emp_ind`:

```
CLUSTER emp_ind ON emp;
```

Cluster a large table by recreating it and loading it in the correct index order:

```
CREATE TABLE newtable AS SELECT * FROM table ORDER BY column;
DROP table;
ALTER TABLE newtable RENAME TO table;
CREATE INDEX column_ix ON table (column);
VACUUM ANALYZE table;
```

Compatibility

There is no `CLUSTER` statement in the SQL standard.

See Also

CREATE TABLE AS, *CREATE INDEX*

COMMENT

Defines or change the comment of an object.

Synopsis

```
COMMENT ON
{ TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE agg_name (agg_type [, ...]) |
  CAST (sourcetype AS targettype) |
  CONSTRAINT constraint_name ON table_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  FILESPACE object_name |
  FUNCTION func_name ([[argmode] [argname] argtype [, ...]]) |
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  OPERATOR op (leftoperand_type, rightoperand_type) |
  OPERATOR CLASS object_name USING index_method |
  [PROCEDURAL] LANGUAGE object_name |
  RESOURCE QUEUE object_name |
  ROLE object_name |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  TABLESPACE object_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name }
IS 'text'
```

Description

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

Comments can be easily retrieved with the psql meta-commands `\dd`, `\d+`, and `\l+`. Other user interfaces to retrieve comments can be built atop the same built-in functions that psql uses, namely `obj_description`, `col_description`, and `shobj_description`.

Parameters

object_name
table_name.column_name
agg_name
constraint_name
func_name
op
rule_name
trigger_name

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

Note: Greenplum Database does not support triggers.

agg_type

An input data type on which the aggregate function operates. To reference a zero-argument aggregate function, write `*` in place of the list of input data types.

sourcetype

The name of the source data type of the cast.

targettype

The name of the target data type of the cast.

argmode

The mode of a function argument: either `IN`, `OUT`, or `INOUT`. If omitted, the default is `IN`. Note that `COMMENT ON FUNCTION` does not actually pay any attention to `OUT` arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the `IN` and `INOUT` arguments.

argname

The name of a function argument. Note that `COMMENT ON FUNCTION` does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any.

large_object_oid

The OID of the large object.

PROCEDURAL

This is a noise word.

text

The new comment, written as a string literal; or `NULL` to drop the comment.

Notes

There is presently no security mechanism for comments: any user connected to a database can see all the comments for objects in that database (although only superusers can change comments for objects that they do not own). For shared objects such as databases, roles, and tablespaces comments are stored globally and any user connected to any database can see all the comments for shared objects. Therefore, do not put security-critical information in comments.

Examples

Attach a comment to the table `mytable`:

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

Remove it again:

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

Compatibility

There is no `COMMENT` statement in the SQL standard.

COMMIT

Commits the current transaction.

Synopsis

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

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.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

Notes

Use *ROLLBACK* to abort a transaction.

Issuing `COMMIT` when not inside a transaction does no harm, but it will provoke a warning message.

Examples

To commit the current transaction and make all changes permanent:

```
COMMIT;
```

Compatibility

The SQL standard only specifies the two forms `COMMIT` and `COMMIT WORK`. Otherwise, this command is fully conforming.

See Also

BEGIN, *END*, *START TRANSACTION*, *ROLLBACK*

COPY

Copies data between a file and a table.

Synopsis

```
COPY table [(column [, ...])] FROM {'file' | STDIN}
     [ [WITH]
       [OIDS]
       [HEADER]
       [DELIMITER [ AS ] 'delimiter']
       [NULL [ AS ] 'null string']
       [ESCAPE [ AS ] 'escape' | 'OFF']
       [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
       [CSV [QUOTE [ AS ] 'quote']
            [FORCE NOT NULL column [, ...]]
       [FILL MISSING FIELDS]
       [[LOG ERRORS [INTO error_table] [KEEP]
       SEGMENT REJECT LIMIT count [ROWS | PERCENT] ]

COPY {table [(column [, ...])] | (query)} TO {'file' | STDOUT}
      [ [WITH]
        [OIDS]
        [HEADER]
        [DELIMITER [ AS ] 'delimiter']
        [NULL [ AS ] 'null string']
        [ESCAPE [ AS ] 'escape' | 'OFF']
        [CSV [QUOTE [ AS ] 'quote']
             [FORCE QUOTE column [, ...]] ]
      [IGNORE EXTERNAL PARTITIONS ]
```

Description

`COPY` moves data between Greenplum Database tables and standard file-system files. `COPY TO` copies the contents of a table to a file, while `COPY FROM` copies data from a file to a table (appending the data to whatever is in the table already). `COPY TO` can also copy the results of a `SELECT` query.

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

`COPY` with a file name instructs the Greenplum Database master host to directly read from or write to a file. The file must be accessible to the master host and the name must be specified from the viewpoint of the master host. When `STDIN` or `STDOUT` is specified, data is transmitted via the connection between the client and the master.

If `SEGMENT REJECT LIMIT` is used, then a `COPY FROM` operation will operate in single row error isolation mode. In this release, single row error isolation mode only applies to rows in the input file with format errors — for example, extra or missing attributes, attributes of a wrong data type, or invalid client encoding sequences. Constraint errors such as violation of a `NOT NULL`, `CHECK`, or `UNIQUE` constraint will still be handled in 'all-or-nothing' input mode. The user can specify the number of error rows acceptable (on a per-segment basis), after which the entire `COPY FROM` operation will be aborted and no rows will be loaded. Note that the count of error rows is per-segment, not per entire load operation. If the per-segment reject limit is not reached, then all rows not containing an error will be loaded. If the limit is not reached, all good rows will be loaded and any error rows discarded. If you would like to keep error rows for further examination, you can optionally declare an error table using the `LOG ERRORS INTO` clause. Any rows containing a format error would then be logged to the specified error table.

Outputs

On successful completion, a `COPY` command returns a command tag of the form, where *count* is the number of rows copied:

```
COPY count
```

If running a `COPY FROM` command in single row error isolation mode, the following notice message will be returned if any rows were not loaded due to format errors, where *count* is the number of rows rejected:

```
NOTICE: Rejected count badly formatted rows.
```

Parameters

table

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

column

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

query

A `SELECT` or `VALUES` command whose results are to be copied. Note that parentheses are required around the query.

file

The absolute path name of the input or output file.

STDIN

Specifies that input comes from the client application.

STDOUT

Specifies that output goes to the client application.

OIDS

Specifies copying the OID for each row. (An error is raised if OIDS is specified for a table that does not have OIDs, or in the case of copying a query.)

delimiter

The single ASCII character that separates columns within each row (line) of the file. The default is a tab character in text mode, a comma in `CSV` mode.

null string

The string that represents a null value. The default is `\N` (backslash-N) in text mode, and a empty value with no quotes in `CSV` mode. You might prefer an empty string even in text mode for cases where you don't want to distinguish nulls from empty strings. When using `COPY FROM`, 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 with `COPY TO`.

escape

Specifies the single character that is used for C escape sequences (such as `\n`,`\t`,`\100`, and so on) and for quoting data characters that might otherwise be taken as row or column delimiters. Make sure to choose an escape character that is not used anywhere in your actual column data. The default escape character is `\` (backslash) for text files or `"` (double quote) for CSV files, however it is possible to specify any other character to represent an escape. It is also possible to disable escaping on text-formatted files by specifying the value '`OFF'` as the escape value. This is very useful for data such as web log data that has many embedded backslashes that are not intended to be escapes.

NEWLINE

Specifies the newline used in your data files — `LF` (Line feed, 0x0A), `CR` (Carriage return, 0x0D), or `CRLF` (Carriage return plus line feed, 0x0D 0x0A). If not specified, a Greenplum

Database segment will detect the newline type by looking at the first row of data it receives and using the first newline type encountered.

CSV

Selects Comma Separated Value (CSV) mode.

HEADER

Specifies that a file contains a header line with the names of each column in the file. On output, the first line contains the column names from the table, and on input, the first line is ignored.

quote

Specifies the quotation character in CSV mode. The default is double-quote.

FORCE QUOTE

In `CSV COPY TO` mode, forces quoting to be used for all non-`NULL` values in each specified column. `NULL` output is never quoted.

FORCE NOT NULL

In `CSV COPY FROM` mode, process each specified column as though it were quoted and hence not a `NULL` value. For the default null string in `CSV` mode (nothing between two delimiters), this causes missing values to be evaluated as zero-length strings.

FILL MISSING FIELDS

In `COPY FROM` more for both `TEXT` and `CSV`, specifying `FILL MISSING FIELDS` will set missing trailing field values to `NULL` (instead of reporting an error) when a row of data has missing data fields at the end of a line or row. Blank rows, fields with a `NOT NULL` constraint, and trailing delimiters on a line will still report an error.

LOG ERRORS [INTO *error_table*] [KEEP]

This is an optional clause that can precede a `SEGMENT REJECT LIMIT` clause to log information about rows with formatting errors. The `INTO` *`error_table`* clause specifies an error table where rows with formatting errors will be logged when running in single row error isolation mode.

If the `INTO` *`error_table`* clause is not specified, the error log information is stored internally (not in an error table). Error log information that is stored internally is accessed with the Greenplum Database built-in SQL function `gp_read_error_log()`.

If the *error_table* specified already exists, it is used. If it does not exist, it is created. If *error_table* exists and does not have a random distribution (the `DISTRIBUTED RANDOMLY` clause was not specified when creating the table), an error is returned.

If the command generates the error table and no errors are produced, the default is to drop the error table after the operation completes unless `KEEP` is specified. If the table is created and the error limit is exceeded, the entire transaction is rolled back and no error data is saved. If you want the error table to persist in this case, create the error table prior to running the `COPY`.

See *Notes* for information about the error log information and built-in functions for viewing and managing error log information.

> **Note:** The optional `INTO` *`error_table`* clause is deprecated and will not be supported in a future release. Only internal error logs will be supported.

SEGMENT REJECT LIMIT count [ROWS | PERCENT]

Runs a `COPY FROM` operation in single row error isolation mode. If the input rows have format errors they will be discarded provided that the reject limit count is not reached on any Greenplum Database segment instance during the load operation. The reject limit count can be specified as number of rows (the default) or percentage of total rows (1-100). If `PERCENT` is used, each segment starts calculating the bad row percentage only after the number of rows specified by the parameter `gp_reject_percent_threshold` has been processed. The default for `gp_reject_percent_threshold` is 300 rows. Constraint errors such as violation of a `NOT NULL`, `CHECK`, or `UNIQUE` constraint will still be handled in 'all-or-nothing' input mode. If the limit is not reached, all good rows will be loaded and any error rows discarded.

Note: Greenplum Database limits the initial number of rows that can contain formatting errors if the `SEGMENT REJECT LIMIT` is not triggered first or is not specified. If the first 1000 rows are rejected, the `COPY` operation is stopped and rolled back.

The limit for the number of initial rejected rows can be changed with the Greenplum Database server configuration parameter `gp_initial_bad_row_limit`. See *Server Configuration Parameters* for information about the parameter.

IGNORE EXTERNAL PARTITIONS

When copying data from partitioned tables, data are not copied from leaf child partitions that are external tables. A message is added to the log file when data are not copied.

If this clause is not specified and Greenplum Database attempts to copy data from a leaf child partition that is an external table, an error is returned.

See the next section "Notes" for information about specifying an SQL query to copy data from leaf child partitions that are external tables.

Notes

`COPY` can only be used with tables, not with external tables or views. However, you can write `COPY (SELECT * FROM viewname) TO ...`

To copy data from a partitioned table with a leaf child partition that is an external table, use an SQL query to copy the data. For example, if the table `my_sales` contains a with a leaf child partition that is an external table, this command `COPY my_sales TO stdout` returns an error. This command sends the data to stdout:

```
COPY (SELECT * from my_sales ) TO stdout
```

The `BINARY` key word causes all data to be stored/read as binary format rather than as text. It is somewhat faster than the normal text mode, but a binary-format file is less portable across machine architectures and Greenplum Database versions. Also, you cannot run `COPY FROM` in single row error isolation mode if the data is in binary format.

You must have `SELECT` privilege on the table whose values are read by `COPY TO`, and insert privilege on the table into which values are inserted by `COPY FROM`.

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

`COPY FROM` will invoke any triggers and check constraints on the destination table. However, it will not invoke rewrite rules. Note that in this release, violations of constraints are not evaluated for single row error isolation mode.

`COPY` input and output is affected by `DateStyle`. To ensure portability to other Greenplum Database installations that might use non-default `DateStyle` settings, `DateStyle` should be set to ISO before using `COPY TO`.

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

`VACUUM` to recover the wasted space. Another option would be to use single row error isolation mode to filter out error rows while still loading good rows.

When you specify `LOG ERRORS INTO` *`error_table`*, Greenplum Database creates the table *error_table* that contains errors that occur while reading the external table. The table is defined as follows:

```
CREATE TABLE error_table_name ( cmdtime timestamptz, relname text,
    filename text, linenum int, bytenum int, errmsg text,
    rawdata text, rawbytes bytea ) DISTRIBUTED RANDOMLY;
```

You can view the information in the table with SQL commands.

For error log data that is stored internally when the `INTO` *`error_table`* is not specified:

- Use the built-in SQL function `gp_read_error_log('`*`table_name`*`')`. It requires `SELECT` privilege on *table_name*. This example displays the error log information for data loaded into table `ext_expenses` with a `COPY` command:

  ```
  SELECT * from gp_read_error_log('ext_expenses');
  ```

  The error log contains the same columns as the error table.

  The function returns `FALSE` if *table_name* does not exist.
- If error log data exists for the specified table, the new error log data is appended to existing error log data. The error log information is not replicated to mirror segments.
- Use the built-in SQL function `gp_truncate_error_log('`*`table_name`*`')` to delete the error log data for *table_name*. It requires the table owner privilege This example deletes the error log information captured when moving data into the table `ext_expenses`:

  ```
  SELECT gp_truncate_error_log('ext_expenses');
  ```

  The function returns `FALSE` if *table_name* does not exist.

  Specify the `*` wildcard character to delete error log information for existing tables in the current database. Specify the string `*.*` to delete all database error log information, including error log information that was not deleted due to previous database issues. If * is specified, database owner privilege is required. If `*.*` is specified, operating system super-user privilege is required.

When a Greenplum Database user who is not a superuser runs a `COPY` command, the command can be controlled by a resource queue. The resource queue must be configured with the `ACTIVE_STATEMENTS` parameter that specifies a maximum limit on the number of queries that can be executed by roles assigned to that queue. Greenplum Database does not apply a cost value or memory value to a `COPY` command, resource queues with only cost or memory limits do not affect the running of `COPY` commands.

A non-superuser can runs can run these types of `COPY` commands:

- `COPY FROM` command where the source is `stdin`
- `COPY TO` command where the destination is `stdout`

For information about resource queues, see "Workload Management with Resource Queues" in the *Greenplum Database Administrator Guide*.

File Formats

File formats supported by `COPY`.

Text Format

When `COPY` is used without the `BINARY` or `CSV` options, the data read or written is a text file with one line per table row. Columns in a row are separated by the *delimiter* character (tab by default). The column values themselves are strings generated by the output function, or acceptable to the input function, of each attribute's data type. The specified null string is used in place of columns that are null. `COPY FROM` will raise an error if any line of the input file contains more or fewer columns than are expected. If `OIDS` is specified, the OID is read or written as the first column, preceding the user data columns.

The data file has two reserved characters that have special meaning to `COPY`:

- The designated delimiter character (tab by default), which is used to separate fields in the data file.
- A UNIX-style line feed (`\n` or `0x0a`), which is used to designate a new row in the data file. It is strongly recommended that applications generating `COPY` data convert data line feeds to UNIX-style line feeds rather than Microsoft Windows style carriage return line feeds (`\r\n` or `0x0a 0x0d`).

If your data contains either of these characters, you must escape the character so `COPY` treats it as data and not as a field separator or new row.

By default, the escape character is a \ (backslash) for text-formatted files and a `"` (double quote) for csv-formatted files. If you want to use a different escape character, you can do so using the `ESCAPE AS` clause. Make sure to choose an escape character that is not used anywhere in your data file as an actual data value. You can also disable escaping in text-formatted files by using `ESCAPE 'OFF'`.

For example, suppose you have a table with three columns and you want to load the following three fields using COPY.

- percentage sign = %
- vertical bar = |
- backslash = \

Your designated *delimiter* character is `|` (pipe character), and your designated *escape* character is `*` (asterisk). The formatted row in your data file would look like this:

```
percentage sign = % | vertical bar = *| | backslash = \
```

Notice how the pipe character that is part of the data has been escaped using the asterisk character (*). Also notice that we do not need to escape the backslash since we are using an alternative escape character.

The following characters must be preceded by the escape character if they appear as part of a column value: the escape character itself, newline, carriage return, and the current delimiter character. You can specify a different escape character using the `ESCAPE AS` clause.

CSV Format

This format is used for importing and exporting the Comma Separated Value (CSV) file format used by many other programs, such as spreadsheets. Instead of the escaping used by Greenplum Database standard text mode, it produces and recognizes the common CSV escaping mechanism.

The values in each record are separated by the `DELIMITER` character. If the value contains the delimiter character, the `QUOTE` character, the `ESCAPE` character (which is double quote by default), the `NULL` string, a carriage return, or line feed character, then the whole value is prefixed and suffixed by the `QUOTE` character. You can also use `FORCE QUOTE` to force quotes when outputting non-`NULL` values in specific columns.

The CSV format has no standard way to distinguish a `NULL` value from an empty string. Greenplum Database `COPY` handles this by quoting. A `NULL` is output as the `NULL` string and is not quoted, while a data value matching the `NULL` string is quoted. Therefore, using the default settings, a `NULL` is written as an unquoted empty string, while an empty string is written with double quotes (""). Reading values follows similar rules. You can use `FORCE NOT NULL` to prevent `NULL` input comparisons for specific columns.

Because backslash is not a special character in the `CSV` format, `\.`, the end-of-data marker, could also appear as a data value. To avoid any misinterpretation, a `\.` data value appearing as a lone entry on a line is automatically quoted on output, and on input, if quoted, is not interpreted as the end-of-data marker. If you are loading a file created by another application that has a single unquoted column and might have a value of `\.`, you might need to quote that value in the input file.

Note: In `CSV` mode, all characters are significant. A quoted value surrounded by white space, or any characters other than `DELIMITER`, will include those characters. This can cause errors if you import data from a system that pads CSV lines with white space out to some fixed width. If such a situation arises you might need to preprocess the CSV file to remove the trailing white space, before importing the data into Greenplum Database.

Note: `CSV` mode will both recognize and produce CSV files with quoted values containing embedded carriage returns and line feeds. Thus the files are not strictly one line per table row like text-mode files.

Note: Many programs produce strange and occasionally perverse CSV files, so the file format is more a convention than a standard. Thus you might encounter some files that cannot be imported using this mechanism, and `COPY` might produce files that other programs cannot process.

Binary Format

The `BINARY` format consists of a file header, zero or more tuples containing the row data, and a file trailer. Headers and data are in network byte order.

- **File Header** — The file header consists of 15 bytes of fixed fields, followed by a variable-length header extension area. The fixed fields are:
  - **Signature** — 11-byte sequence PGCOPY\n\377\r\n\0 — note that the zero byte 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 end-of-line-translation filters, dropped zero bytes, dropped high bits, or parity changes.)
  - **Flags field** — 32-bit integer 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 in network byte order (most significant byte first), as are all the integer fields used in the file format. 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 is defined, and the rest must be zero (Bit 16: 1 if data has OIDs, 0 if not).
  - **Header extension area length** — 32-bit integer, length in bytes of remainder of header, not including self. Currently, this is 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.
- **Tuples** — Each tuple begins with a 16-bit integer 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 a 32-bit length word followed by that many bytes of field data. (The length word does not include itself, and can be zero.) As a special case, -1 indicates a NULL field value. No value bytes follow in the NULL case.

  There is no alignment padding or any other extra data between fields.

  Presently, all data values in a COPY BINARY file are assumed to be in binary format (format code one). It is anticipated that a future extension may add a header field that allows per-column format codes to be specified.

  If OIDs are included in the file, 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 length word — this will allow handling of 4-byte vs. 8-byte OIDs without too much pain, and will allow OIDs to be shown as null if that ever proves desirable.
- **File Trailer** — The file trailer consists of a 16-bit integer 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.

Examples

Copy a table to the client using the vertical bar (|) as the field delimiter:

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

Copy data from a file into the `country` table:

```
COPY country FROM '/home/usr1/sql/country_data';
```

Copy into a file just the countries whose names start with 'A':

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO
'/home/usr1/sql/a_list_countries.copy';
```

Create an error table called `err_sales` to use with single row error isolation mode:

```
CREATE TABLE err_sales ( cmdtime timestamptz, relname text,
filename text, linenum int, bytenum int, errmsg text, rawdata text, rawbytes bytea )
 DISTRIBUTED RANDOMLY;
```

Copy data from a file into the `sales` table using single row error isolation mode:

```
COPY sales FROM '/home/usr1/sql/sales_data' LOG ERRORS INTO
err_sales SEGMENT REJECT LIMIT 10 ROWS;
```

Compatibility

There is no `COPY` statement in the SQL standard.

See Also

CREATE EXTERNAL TABLE

CREATE AGGREGATE

Defines a new aggregate function.

Synopsis

```
CREATE [ORDERED] AGGREGATE name (input_data_type [ , ... ])
      ( SFUNC = sfunc,
        STYPE = state_data_type
        [, PREFUNC = prefunc]
        [, FINALFUNC = ffunc]
        [, INITCOND = initial_condition]
        [, SORTOP = sort_operator] )
```

Description

`CREATE AGGREGATE` defines a new aggregate function. Some basic and commonly-used aggregate functions such as `count`, `min`, `max`, `sum`, `avg` and so on are already provided in Greenplum Database. 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 types. Two aggregate functions in the same schema can have the same name if they operate on different input types. The name and input data types of an aggregate function must also be distinct from the name and input data types of every ordinary function in the same schema.

An aggregate function is made from one, two or three ordinary functions (all of which must be `IMMUTABLE` functions):

- A state transition function *sfunc*
- An optional preliminary segment-level calculation function *prefunc*
- An optional final calculation function *ffunc*

These functions are used as follows:

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

You can specify `PREFUNC` as method for optimizing aggregate execution. By specifying `PREFUNC`, the aggregate can be executed in parallel on segments first and then on the master. When a two-level execution is performed, `SFUNC` is executed on the segments to generate partial aggregate results, and `PREFUNC` is executed on the master to aggregate the partial results from segments. If single-level aggregation is performed, all the rows are sent to the master and `sfunc` is applied to the rows.

Single-level aggregation and two-level aggregation are equivalent execution strategies. Either type of aggregation can be implemented in a query plan. When you implement the functions `prefunc` and `sfunc`, you must ensure that the invocation of `sfunc` on the segment instances followed by `prefunc` on the master produce the same result as single-level aggregation that sends all the rows to the master and then applies only the `sfunc` to the rows.

Greenplum Database creates a temporary variable of data type *stype* to hold the current internal state of the aggregate function. At each input row, the aggregate argument values are calculated and the state transition function is invoked with the current state value and the new argument values to calculate a new internal state value. After all the rows have been processed, the final function is invoked once to calculate the aggregate return value. If there is no final function then the ending state value is returned as-is.

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

If the state transition function is declared `STRICT`, then it cannot be called with `NULL` inputs. With such a transition function, aggregate execution behaves as follows. Rows with any 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 at the first row with all non-null input values, the first argument value replaces the state value, and the transition function is invoked at subsequent rows with all non-null input values. This is useful for implementing aggregates like `max`. Note that this behavior is only available when *state_data_type* is the same as the first *input_data_type*. When these types are different, you must supply a non-null initial condition or use a nonstrict transition function.

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

If the final function is declared `STRICT`, then it will not be called when the ending state value is `NULL`; instead a `NULL` result will be returned automatically. (This is the normal behavior of `STRICT` functions.) In any case the final function has the option of returning a `NULL` value. For example, the final function for `avg` returns `NULL` when it sees there were zero input rows.

Single argument aggregate functions, such as min or max, can sometimes be optimized by looking into an index instead of scanning every input row. If this aggregate can be so optimized, indicate it by specifying a sort operator. The basic requirement is that the aggregate must yield the first element in the sort ordering induced by the operator; in other words:

```
SELECT agg(col) FROM tab;
```

must be equivalent to:

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

Further assumptions are that the aggregate function ignores `NULL` inputs, and that it delivers a `NULL` result if and only if there were no non-null inputs. Ordinarily, a data type's `<` operator is the proper sort operator for `MIN`, and `>` is the proper sort operator for `MAX`. Note that the optimization will never actually take effect unless the specified operator is the "less than" or "greater than" strategy member of a B-tree index operator class.

Ordered Aggregates

If the optional qualification `ORDERED` appears, the created aggregate function is an *ordered aggregate*. In this case, the preliminary aggregation function, `prefunc` cannot be specified.

An ordered aggregate is called with the following syntax.

```
name ( arg [ , ... ] [ORDER BY sortspec [ , ...]] )
```

If the optional `ORDER BY` is omitted, a system-defined ordering is used. The transition function `sfunc` of an ordered aggregate function is called on its input arguments in the specified order and on a single segment. There is a new column `aggordered` in the *`pg_aggregate`* table to indicate the aggregate function is defined as an ordered aggregate.

Parameters

name

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

input_data_type

An input data type on which this aggregate function operates. To create a zero-argument aggregate function, write * in place of the list of input data types. An example of such an aggregate is count(*).

sfunc

The name of the state transition function to be called for each input row. For an N-argument aggregate function, the *sfunc* must take N+1 arguments, the first being of type *state_data_type* and the rest matching the declared input data types of the aggregate. The function must return a value of type *state_data_type*. This function takes the current state value and the current input data values, and returns the next state value.

state_data_type

The data type for the aggregate state value.

prefunc

The name of a preliminary aggregation function. This is a function of two arguments, both of type *state_data_type*. It must return a value of *state_data_type*. A preliminary function takes two transition state values and returns a new transition state value representing the combined aggregation. In Greenplum Database, if the result of the aggregate function is computed in a segmented fashion, the preliminary aggregation function is invoked on the individual internal states in order to combine them into an ending internal state.

Note that this function is also called in hash aggregate mode within a segment. Therefore, if you call this aggregate function without a preliminary function, hash aggregate is never chosen. Since hash aggregate is efficient, consider defining preliminary function whenever possible.

ffunc

The name of the final function called to compute the aggregate result after all input rows have been traversed. The function must take a single argument of type `state_data_type`. The return 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 result, and the return type is *state_data_type*.

initial_condition

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

sort_operator

The associated sort operator for a MIN- or MAX-like aggregate function. This is just an operator name (possibly schema-qualified). The operator is assumed to have the same input data types as the aggregate function (which must be a single-argument aggregate function).

Notes

The ordinary functions used to define a new aggregate function must be defined first. Note that in this release of Greenplum Database, it is required that the *sfunc*, *ffunc*, and *prefunc* functions used to create the aggregate are defined as `IMMUTABLE`.

If a user-defined aggregate is used in a window expression, a `prefunc` function must be defined for the aggregate.

If the value of the Greenplum Database server configuration parameter `gp_enable_multiphase_agg` is `off`, only single-level aggregation is performed.

Any compiled code (shared library files) for custom functions must be placed in the same location on every host in your Greenplum Database array (master and all segments). This location must also be in the `LD_LIBRARY_PATH` so that the server can locate the files.

Example

The following simple example creates an aggregate function that computes the sum of two columns.

Before creating the aggregate function, create two functions that are used as the `SFUNC` and `PREFUNC` functions of the aggregate function.

This function is specified as the `SFUNC` function in the aggregate function.

```
CREATE FUNCTION mysfunc_accum(numeric, numeric, numeric)
  RETURNS numeric
   AS 'select $1 + $2 + $3'
   LANGUAGE SQL
   IMMUTABLE
   RETURNS NULL ON NULL INPUT;
```

This function is specified as the `PREFUNC` function in the aggregate function.

```
CREATE FUNCTION mypre_accum(numeric, numeric )
  RETURNS numeric
   AS 'select $1 + $2'
   LANGUAGE SQL
   IMMUTABLE
   RETURNS NULL ON NULL INPUT;
```

This `CREATE AGGREGATE` command creates the aggregate function that adds two columns.

```
CREATE AGGREGATE agg_prefunc(numeric, numeric) (
   SFUNC = mysfunc_accum,
   STYPE = numeric,
   PREFUNC = mypre_accum,
   INITCOND = 0 );
```

The following commands create a table, adds some rows, and runs the aggregate function.

```
create table t1 (a int, b int) DISTRIBUTED BY (a);
insert into t1 values
   (10, 1),
   (20, 2),
   (30, 3);
select agg_prefunc(a, b) from t1;
```

This `EXPLAIN` command shows two phase aggregation.

```
explain select agg_prefunc(a, b) from t1;

QUERY PLAN
--------------------------------------------------------------------------
Aggregate (cost=1.10..1.11 rows=1 width=32)
 -> Gather Motion 2:1 (slice1; segments: 2) (cost=1.04..1.08 rows=1
      width=32)
     -> Aggregate (cost=1.04..1.05 rows=1 width=32)
       -> Seq Scan on t1 (cost=0.00..1.03 rows=2 width=8)
 (4 rows)
```

Compatibility

`CREATE AGGREGATE` is a Greenplum Database language extension. The SQL standard does not provide for user-defined aggregate functions.

See Also

ALTER AGGREGATE, *DROP AGGREGATE*, *CREATE FUNCTION*

CREATE CAST

Defines a new cast.

Synopsis

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

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

Description

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

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

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

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

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

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

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

will be allowed if the cast from type `integer` to type `text` is marked `AS ASSIGNMENT`, otherwise not. The term *assignment cast* is typically used to describe this kind of cast.

If the cast is marked `AS IMPLICIT` then it can be invoked implicitly in any context, whether assignment or internally in an expression. The term *implicit cast* is typically used to describe this kind of cast. For example, since `||` takes `text` operands,

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

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

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

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

To be able to create a cast, you must own the source or the target data type. To create a binary-compatible cast, you must be superuser.

Parameters

sourcetype

The name of the source data type of the cast.

targettype

The name of the target data type of the cast.

funcname(argtypes)

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

Cast implementation functions may have one to three arguments. The first argument type must be identical to the cast's source type. The second argument, if present, must be type `integer`; it receives the type modifier associated with the destination type, or `-1` if there is none. The third argument, if present, must be type `boolean`; it receives `true` if the cast is an explicit cast, `false` otherwise. The SQL specification demands different behaviors for explicit and implicit casts in some cases. This argument is supplied for functions that must implement such casts. It is not recommended that you design your own data types this way.

Ordinarily a cast must have different source and target data types. However, it is allowed to declare a cast with identical source and target types if it has a cast implementation function with more than one argument. This is used to represent type-specific length coercion functions in the system catalogs. The named function is used to coerce a value of the type to the type modifier value given by its second argument. (Since the grammar presently permits only certain built-in data types to have type modifiers, this feature is of no use for user-defined target types.)

When a cast has different source and target types and a function that takes more than one argument, it represents converting from one type to another and applying a length coercion in a single step. When no such entry is available, coercion to a type that uses a type modifier involves two steps, one to convert between data types and a second to apply the modifier.

WITHOUT FUNCTION

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

AS ASSIGNMENT

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

AS IMPLICIT

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

Notes

Note that in this release of Greenplum Database, user-defined functions used in a user-defined cast must be defined as `IMMUTABLE`. Any compiled code (shared library files) for custom functions must be placed in the same location on every host in your Greenplum Database array (master and all segments). This location must also be in the `LD_LIBRARY_PATH` so that the server can locate the files.

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

It is recommended that you follow the convention of naming cast implementation functions after the target data type, as the built-in cast implementation functions are named. Many users are used to being able to cast data types using a function-style notation, that is *typename*`(x)`.

Examples

To create a cast from type `text` to type `int4` using the function `int4(text)` (This cast is already predefined in the system.):

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

Compatibility

The `CREATE CAST` command conforms to the SQL standard, except that SQL does not make provisions for binary-compatible types or extra arguments to implementation functions. `AS IMPLICIT` is a Greenplum Database extension, too.

See Also

CREATE FUNCTION, *CREATE TYPE*, *DROP CAST*

CREATE CONVERSION

Defines a new encoding conversion.

Synopsis

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

Description

`CREATE CONVERSION` defines a new conversion between character set encodings. Conversion names may be used in the convert function to specify a particular encoding conversion. Also, conversions that are marked `DEFAULT` can be used for automatic encoding conversion between client and server. For this purpose, two conversions, from encoding A to B and from encoding B to A, must be defined.

To create a conversion, you must have `EXECUTE` privilege on the function and `CREATE` privilege on the destination schema.

Parameters

DEFAULT

Indicates that this conversion is the default for this particular source to destination encoding. There should be only one default encoding in a schema for the encoding pair.

name

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

source_encoding

The source encoding name.

dest_encoding

The destination encoding name.

funcname

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

```
conv_proc(
    integer,  -- source encoding ID
    integer,  -- destination encoding ID
    cstring,  -- source string (null terminated C string)
    internal, -- destination (fill with a null terminated C string)
    integer   -- source string length
) RETURNS void;
```

Notes

Note that in this release of Greenplum Database, user-defined functions used in a user-defined conversion must be defined as `IMMUTABLE`. Any compiled code (shared library files) for custom functions must be placed in the same location on every host in your Greenplum Database array (master and all segments). This location must also be in the `LD_LIBRARY_PATH` so that the server can locate the files.

Examples

To create a conversion from encoding `UTF8` to `LATIN1` using `myfunc`:

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc;
```

Compatibility

There is no `CREATE CONVERSION` statement in the SQL standard.

See Also

ALTER CONVERSION, *CREATE FUNCTION*, *DROP CONVERSION*

CREATE DATABASE

Creates a new database.

Synopsis

```
CREATE DATABASE name [ [WITH] [OWNER [=] dbowner]
                       [TEMPLATE [=] template]
                       [ENCODING [=] encoding]
                       [TABLESPACE [=] tablespace]
                       [CONNECTION LIMIT [=] connlimit ] ]
```

Description

`CREATE DATABASE` creates a new database. To create a database, you must be a superuser or have the special `CREATEDB` privilege.

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

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 clean database containing only the standard objects predefined by Greenplum Database. This is useful if you wish to avoid copying any installation-local objects that may have been added to `template1`.

Parameters

name

The name of a database to create.

dbowner

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

template

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

encoding

Character set encoding to use in the new database. Specify a string constant (such as `'SQL_ASCII'`), an integer encoding number, or `DEFAULT` to use the default encoding. For more information, see *Character Set Support*.

tablespace

The name of the tablespace that will be associated with the new database, or `DEFAULT` to use the template database's tablespace. This tablespace will be the default tablespace used for objects created in this database.

connlimit

The maximum number of concurrent connections posible. The default of -1 means there is no limitation.

Notes

`CREATE DATABASE` cannot be executed inside a transaction block.

When you copy a database by specifying its name as the template, no other sessions can be connected to the template database while it is being copied. New connections to the template database are locked out until `CREATE DATABASE` completes.

The `CONNECTION LIMIT` is not enforced against superusers.

Examples

To create a new database:

```
CREATE DATABASE gpdb;
```

To create a database `sales` owned by user `salesapp` with a default tablespace of `salesspace`:

```
CREATE DATABASE sales OWNER salesapp TABLESPACE salesspace;
```

To create a database `music` which supports the ISO-8859-1 character set:

```
CREATE DATABASE music ENCODING 'LATIN1';
```

Compatibility

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

See Also

ALTER DATABASE, *DROP DATABASE*

CREATE DOMAIN

Defines a new domain.

Synopsis

```
CREATE DOMAIN name [AS] data_type [DEFAULT expression]
       [CONSTRAINT constraint_name
       | NOT NULL | NULL
       | CHECK (expression) [...]]
```

Description

`CREATE DOMAIN` creates a new domain. A domain is essentially a data type with optional constraints (restrictions on the allowed set of values). The user who defines a domain becomes its owner. The domain name must be unique among the data types and domains existing in its schema.

Domains are useful for abstracting common constraints on fields into a single location for maintenance. For example, several tables might contain email address columns, all requiring the same `CHECK` constraint to verify the address syntax. It is easier to define a domain rather than setting up a column constraint for each table that has an email column.

Parameters

name

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

data_type

The underlying data type of the domain. This may include array specifiers.

DEFAULT *expression*

Specifies a default value for columns of the domain data type. The value is any variable-free expression (but subqueries are not allowed). The data type of the default expression must match the data type of the domain. If no default value is specified, then the default value is the null value. The default expression will be used in any insert operation that does not specify a value for the column. If a default value is defined for a particular column, it overrides any default associated with the domain. In turn, the domain default overrides any default value associated with the underlying data type.

CONSTRAINT *constraint_name*

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

NOT NULL

Values of this domain are not allowed to be null.

NULL

Values of this domain are allowed to be null. This is the default. This clause is only intended for compatibility with nonstandard SQL databases. Its use is discouraged in new applications.

CHECK *(expression)*

`CHECK` clauses specify integrity constraints or tests which values of the domain must satisfy. Each constraint must be an expression producing a Boolean result. It should use the key word `VALUE` to refer to the value being tested. Currently, `CHECK` expressions cannot contain subqueries nor refer to variables other than `VALUE`.

Examples

Create the `us_zip_code` data type. A regular expression test is used to verify that the value looks like a valid US zip code.

```
CREATE DOMAIN us_zip_code AS TEXT CHECK
       ( VALUE ~ '^\\d{5}$' OR VALUE ~ '^\\d{5}-\\d{4}$' );
```

Compatibility

`CREATE DOMAIN` conforms to the SQL standard.

See Also

ALTER DOMAIN, *DROP DOMAIN*

CREATE EXTERNAL TABLE

Defines a new external table.

Synopsis

```
CREATE [READABLE] EXTERNAL TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
      LOCATION ('file://seghost[:port]/path/file' [, ...])
        | ('gpfdist://filehost[:port]/file_pattern[#transform]' [, ...]
        | ('gpfdists://filehost[:port]/file_pattern[#transform]'
            [, ...])
        | ('gphdfs://hdfs_host[:port]/path/file')
        | ('s3://S3_endpoint/bucket_name/[S3_prefix]
            [config=config_file]')
      FORMAT 'TEXT'
            [( [HEADER]
               [DELIMITER [AS] 'delimiter' | 'OFF']
               [NULL [AS] 'null string']
               [ESCAPE [AS] 'escape' | 'OFF']
               [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
               [FILL MISSING FIELDS] )]
           | 'CSV'
            [( [HEADER]
               [QUOTE [AS] 'quote']
               [DELIMITER [AS] 'delimiter']
               [NULL [AS] 'null string']
               [FORCE NOT NULL column [, ...]]
               [ESCAPE [AS] 'escape']
               [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
               [FILL MISSING FIELDS] )]
           | 'AVRO'
           | 'PARQUET'

           | 'CUSTOM' (Formatter=<formatter specifications>)
     [ ENCODING 'encoding' ]
     [ [LOG ERRORS [INTO error_table]] SEGMENT REJECT LIMIT count
       [ROWS | PERCENT] ]

CREATE [READABLE] EXTERNAL WEB TABLE table_name
    ( column_name data_type [, ...] | LIKE other_table )
      LOCATION ('http://webhost[:port]/path/file' [, ...])
    | EXECUTE 'command' [ON ALL
                          | MASTER
                          | number_of_segments
                          | HOST ['segment_hostname']
                          | SEGMENT segment_id ]
      FORMAT 'TEXT'
            [( [HEADER]
               [DELIMITER [AS] 'delimiter' | 'OFF']
               [NULL [AS] 'null string']
               [ESCAPE [AS] 'escape' | 'OFF']
               [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
               [FILL MISSING FIELDS] )]
           | 'CSV'
            [( [HEADER]
               [QUOTE [AS] 'quote']
               [DELIMITER [AS] 'delimiter']
               [NULL [AS] 'null string']
               [FORCE NOT NULL column [, ...]]
               [ESCAPE [AS] 'escape']
               [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
               [FILL MISSING FIELDS] )]
           | 'CUSTOM' (Formatter=<formatter specifications>)
     [ ENCODING 'encoding' ]
     [ [LOG ERRORS [INTO error_table]] SEGMENT REJECT LIMIT count
```

```
        [ROWS | PERCENT] ]

CREATE WRITABLE EXTERNAL TABLE table_name
     ( column_name data_type [, ...] | LIKE other_table )
      LOCATION('gpfdist://outputhost[:port]/filename[#transform]'
       | ('gpfdists://outputhost[:port]/file_pattern[#transform]'
           [, ...])
       | ('gphdfs://hdfs_host[:port]/path')
       FORMAT 'TEXT'
                 [( [DELIMITER [AS] 'delimiter']
                 [NULL [AS] 'null string']
                 [ESCAPE [AS] 'escape' | 'OFF'] )]
           | 'CSV'
                 [([QUOTE [AS] 'quote']
                 [DELIMITER [AS] 'delimiter']
                 [NULL [AS] 'null string']
                 [FORCE QUOTE column [, ...]] ]
                 [ESCAPE [AS] 'escape'] )]
            | 'AVRO'
            | 'PARQUET'

            | 'CUSTOM' (Formatter=<formatter specifications>)
     [ ENCODING 'write_encoding' ]
     [ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]

CREATE WRITABLE EXTERNAL WEB TABLE table_name
     ( column_name data_type [, ...] | LIKE other_table )
     EXECUTE 'command' [ON ALL]
     FORMAT 'TEXT'
                 [( [DELIMITER [AS] 'delimiter']
                 [NULL [AS] 'null string']
                 [ESCAPE [AS] 'escape' | 'OFF'] )]
           | 'CSV'
                 [([QUOTE [AS] 'quote']
                 [DELIMITER [AS] 'delimiter']
                 [NULL [AS] 'null string']
                 [FORCE QUOTE column [, ...]] ]
                 [ESCAPE [AS] 'escape'] )]
            | 'CUSTOM' (Formatter=<formatter specifications>)
     [ ENCODING 'write_encoding' ]
     [ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]
```

Description

See "Loading and Unloading Data" in the *Greenplum Database Administrator Guide* for detailed information about external tables.

`CREATE EXTERNAL TABLE` or `CREATE EXTERNAL WEB TABLE` creates a new readable external table definition in Greenplum Database. Readable external tables are typically used for fast, parallel data loading. Once an external table is defined, you can query its data directly (and in parallel) using SQL commands. For example, you can select, join, or sort external table data. You can also create views for external tables. DML operations (`UPDATE`, `INSERT`, `DELETE`, or `TRUNCATE`) are not allowed on readable external tables, and you cannot create indexes on readable external tables.

`CREATE WRITABLE EXTERNAL TABLE` or `CREATE WRITABLE EXTERNAL WEB TABLE` creates a new writable external table definition in Greenplum Database. Writable external tables are typically used for unloading data from the database into a set of files or named pipes. Writable external web tables can also be used to output data to an executable program. Writable external tables can also be used as output targets for Greenplum parallel MapReduce calculations. Once a writable external table is defined, data can be selected from database tables and inserted into the writable external table. Writable external tables only allow `INSERT` operations – `SELECT`, `UPDATE`, `DELETE` or `TRUNCATE` are not allowed.

The main difference between regular external tables and web external tables is their data sources. Regular readable external tables access static flat files, whereas web external tables access dynamic data sources – either on a web server or by executing OS commands or scripts.

The `FORMAT` clause is used to describe how the external table files are formatted. Valid file formats are delimited text (`TEXT`) and comma separated values (`CSV`) format, similar to the formatting options available with the PostgreSQL *`COPY`* command. If the data in the file does not use the default column delimiter, escape character, null string and so on, you must specify the additional formatting options so that the data in the external file is read correctly by Greenplum Database. For information about using a custom format, see "Loading and Unloading Data" in the *Greenplum Database Administrator Guide*.

For the `gphdfs` protocol, you can also specify the `AVRO` or `PARQUET` in the `FORMAT` clause to read or write Avro or Parquet format files. For information about Avro or Parquet file, see *HDFS File Format Support for the gphdfs Protocol*.

When accessing files from an Amazon Web Services (AWS) S3 bucket with the `s3` protocol, only readable external tables are supported. Before you can create an external table that reads from the S3 bucket, you must configure Greenplum Database to support the protocol. See *s3 Protocol Configuration*.

Parameters

READABLE | WRITABLE

Specifies the type of external table, readable being the default. Readable external tables are used for loading data into Greenplum Database. Writable external tables are used for unloading data.

WEB

Creates a readable or writeable web external table definition in Greenplum Database. There are two forms of readable web external tables – those that access files via the `http://` protocol or those that access data by executing OS commands. Writable web external tables output data to an executable program that can accept an input stream of data. Web external tables are not rescannable during query execution.

table_name

The name of the new external table.

column_name

The name of a column to create in the external table definition. Unlike regular tables, external tables do not have column constraints or default values, so do not specify those.

LIKE *other_table*

The `LIKE` clause specifies a table from which the new external table automatically copies all column names, data types and Greenplum distribution policy. If the original table specifies any column constraints or default column values, those will not be copied over to the new external table definition.

data_type

The data type of the column.

LOCATION *('protocol://host[:port]/path/file' [, ...])*

For readable external tables, specifies the URI of the external data source(s) to be used to populate the external table or web table. Regular readable external tables allow the `gpfdist` or `file` protocols. Web external tables allow the `http` protocol. If `port` is omitted, port `8080` is assumed for `http` and `gpfdist` protocols, and port 9000 for the `gphdfs` protocol. If using the `gpfdist` protocol, the `path` is relative to the directory from which `gpfdist` is serving files (the directory specified when you started the `gpfdist` program). Also, `gpfdist` can use wildcards (or other C-style pattern matching) to denote multiple files in a directory. For example:

```
'gpfdist://filehost:8081/*'
'gpfdist://masterhost/my_load_file'
'file://seghost1/dbfast1/external/myfile.txt'
'http://intranet.mycompany.com/finance/expenses.csv'
```

If you are using MapR clusters with the `gphdfs` protocol, you specify a specific cluster and the file:

- To specify the default cluster, the first entry in the MapR configuration file `/opt/mapr/conf/mapr-clusters.conf`, specify the location of your table with this syntax:

  ```
  LOCATION ('gphdfs:///file_path')
  ```

  The *file_path* is the path to the file.
- To specify another MapR cluster listed in the configuration file, specify the file with this syntax:

  ```
  LOCATION ('gphdfs:///mapr/cluster_name/file_path')
  ```

  The *cluster_name* is the name of the cluster specified in the configuration file and *file_path* is the path to the file.

For writable external tables, specifies the URI location of the `gpfdist` process that will collect data output from the Greenplum segments and write it to the named file. The `path` is relative to the directory from which `gpfdist` is serving files (the directory specified when you started the `gpfdist` program). If multiple `gpfdist` locations are listed, the segments sending data will be evenly divided across the available output locations. For example:

```
'gpfdist://outputhost:8081/data1.out',
'gpfdist://outputhost:8081/data2.out'
```

With two `gpfdist` locations listed as in the above example, half of the segments would send their output data to the `data1.out` file and the other half to the `data2.out` file.

If you specify `gphdfs` protocol to read or write file to a Hadoop file system (HDFS), you can read or write an Avro or Parquet format file by specifying the `FORMAT` clause with either `AVRO` or `PARQUET`.

For information about the options when specifying a location for an Avro or Parquet file, see *HDFS File Format Support for the gphdfs Protocol*.

When you create a readable external table with the `s3` protocol, only the `TEXT` and `CSV` formats are supported. Before you can create external tables with the `s3` protocol, you must configure Greenplum Database. For information about configuring Greenplum Database, see *s3 Protocol Configuration*.

For the `s3` protocol, you specify a location for files and an optional configuration file in the `LOCATION` clause. This is the syntax of the clause.

```
's3://S3_endpoint/bucket_name/[S3_prefix] [config=config_file_location]'
```

The AWS *S3_endpoint* and S3 *bucket_name* define the location of the files. If needed, the `S3_prefix` specify the group of files in the bucket. For information about the S3 endpoints see the Amazon S3 documentation *http://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region*. For information about S3 buckets and folders, see the Amazon S3 documentation *http://aws.amazon.com/documentation/s3/*.

If you specify an *S3_prefix*, the `s3` protocol selects the files that have the specified S3 file prefix. The `s3` protocol does not use the slash character (`/`) as delimiter. For example, these files have `domain` as the *S3_endpoint*, and `test1` as the *bucket_name*.

```
s3://domain/test1/abc
s3://domain/test1/abc/
s3://domain/test1/abc/xx
s3://domain/test1/abcdef
s3://domain/test1/abcdefff
```

- If the file location is `s3://domain/test1/abc`, the `s3` protocol selects all 5 files.
- If the file location is `s3://domain/test1/abc/`, the `s3` protocol selects the files `s3://domain/test1/abc/` and `s3://domain/test1/abc/xx`.
- If the file location is `s3://domain/test1/abcd` the `s3` protocol selects the files `s3://domain/test1/abcdef` and `s3://domain/test1/abcdefff`

For information about the S3 prefix, see the Amazon S3 documentation *Listing Keys Hierarchically Using a Prefix and Delimiter*.

All the files specified by the S3 file location (*S3_endpoint*/*bucket_name*/*S3_prefix*) are used as the source for the external table and must have the same format and each file must contain complete data rows. A data row cannot be split between files. The S3 file permissions must be `Open/Download` and `View` for the S3 user ID that is accessing the files.

The `config` parameter specifies the location of the required `s3` protocol configuration file that contains AWS connection credentials and communication parameters. See *s3 Protocol Configuration File*.

This is an example readable external table definition with the `s3` protocol.

```
CREATE READABLE EXTERNAL TABLE S3TBL (date text, time text, amt int)
   location('s3://s3-us-west-2.amazonaws.com/s3test.pivotal.io/dataset1/
normal/
       config=/home/gpadmin/aws_s3/s3.conf')
   FORMAT 'csv';
```

The S3 bucket is at the S3 endpoint `s3-us-west-2.amazonaws.com` and the S3 bucket name is `s3test.pivotal.io`. The S3 prefix for the files in the bucket is `/dataset1/normal/`. The configuration file is in `/home/gpadmin/s3/s3.conf` on all Greenplum Database segments.

EXECUTE *'command'* [ON ...]

Allowed for readable web external tables or writable external tables only. For readable web external tables, specifies the OS command to be executed by the segment instances. The *command* can be a single OS command or a script. The `ON` clause is used to specify which segment instances will execute the given command.

- ON ALL is the default. The command will be executed by every active (primary) segment instance on all segment hosts in the Greenplum Database system. If the command executes a script, that script must reside in the same location on all of the segment hosts and be executable by the Greenplum superuser (`gpadmin`).
- ON MASTER runs the command on the master host only.

  Note: Logging is not supported for web external tables when the `ON MASTER` clause is specified.
- ON *number* means the command will be executed by the specified number of segments. The particular segments are chosen randomly at runtime by the Greenplum Database system. If the command executes a script, that script must reside in the same location on all of the segment hosts and be executable by the Greenplum superuser (`gpadmin`).
- HOST means the command will be executed by one segment on each segment host (once per segment host), regardless of the number of active segment instances per host.
- HOST *segment_hostname* means the command will be executed by all active (primary) segment instances on the specified segment host.
- SEGMENT *segment_id* means the command will be executed only once by the specified segment. You can determine a segment instance's ID by looking at the *content* number in the system catalog table *gp_segment_configuration*. The *content* ID of the Greenplum Database master is always `-1`.

For writable external tables, the *command* specified in the `EXECUTE` clause must be prepared to have data piped into it. Since all segments that have data to send will write their output to the specified command or program, the only available option for the `ON` clause is `ON ALL`.

FORMAT 'TEXT | CSV | AVRO | PARQUET' ***(options)***

Specifies the format of the external or web table data - either plain text (`TEXT`) or comma separated values (`CSV`) format.

The `AVRO` and `PARQUET` formats are supported only with the `gphdfs` protocol.

For information about the options when specifying the `AVRO` and `PARQUET` file format, see *HDFS File Format Support for the gphdfs Protocol*.

DELIMITER

Specifies a single ASCII character that separates columns within each row (line) of data. The default is a tab character in `TEXT` mode, a comma in `CSV` mode. In `TEXT` mode for readable external tables, the delimiter can be set to `OFF` for special use cases in which unstructured data is loaded into a single-column table.

NULL

Specifies the string that represents a `NULL` value. The default is `\N` (backslash-N) in `TEXT` mode, and an empty value with no quotations in `CSV` mode. You might prefer an empty string even in `TEXT` mode for cases where you do not want to distinguish `NULL` values from empty strings. When using external and web tables, any data item that matches this string will be considered a `NULL` value.

As an example for the `text` format, this `FORMAT` clause can be used to specify that the string of two single quotes (`''`) is a `NULL` value.

```
FORMAT 'text' (delimiter ',' null '\'\'\'\'' )
```

ESCAPE

Specifies the single character that is used for C escape sequences (such as `\n`,`\t`,`\100`, and so on) and for escaping data characters that might otherwise be taken as row or column delimiters. Make sure to choose an escape character that is not used anywhere in your actual column data. The default escape character is a \ (backslash) for text-formatted files and a `"` (double quote) for csv-formatted files, however it is possible to specify another character to represent an escape. It is also possible to disable escaping in text-formatted files by specifying the value `'OFF'` as the escape value. This is very useful for data such as text-formatted web log data that has many embedded backslashes that are not intended to be escapes.

NEWLINE

Specifies the newline used in your data files – `LF` (Line feed, 0x0A), `CR` (Carriage return, 0x0D), or `CRLF` (Carriage return plus line feed, 0x0D 0x0A). If not specified, a Greenplum Database segment will detect the newline type by looking at the first row of data it receives and using the first newline type encountered.

HEADER

For readable external tables, specifies that the first line in the data file(s) is a header row (contains the names of the table columns) and should not be included as data for the table. If using multiple data source files, all files must have a header row.

QUOTE

Specifies the quotation character for `CSV` mode. The default is double-quote (`"`).

FORCE NOT NULL

In `CSV` mode, processes each specified column as though it were quoted and hence not a `NULL` value. For the default null string in `CSV` mode (nothing between two delimiters), this causes missing values to be evaluated as zero-length strings.

FORCE QUOTE

In `CSV` mode for writable external tables, forces quoting to be used for all non-`NULL` values in each specified column. `NULL` output is never quoted.

FILL MISSING FIELDS

In both `TEXT` and `CSV` mode for readable external tables, specifying `FILL MISSING FIELDS` will set missing trailing field values to `NULL` (instead of reporting an error) when a row of data has missing data fields at the end of a line or row. Blank rows, fields with a `NOT NULL` constraint, and trailing delimiters on a line will still report an error.

ENCODING *'encoding'*

Character set encoding to use for the external table. Specify a string constant (such as `'SQL_ASCII'`), an integer encoding number, or `DEFAULT` to use the default client encoding. See *Character Set Support*.

LOG ERRORS [INTO *error_table*]

This is an optional clause that can precede a `SEGMENT REJECT LIMIT` clause to log information about rows with formatting errors. The optional `INTO error_table` clause specifies an error table where rows with formatting errors will be logged when running in single row error isolation mode.

If the `INTO error_table` clause is not specified, the error log information is stored internally (not in an error table). Error log information that is stored internally is accessed with the Greenplum Database built-in SQL function `gp_read_error_log()`.

If the *error_table* specified already exists, it is used. If it does not exist, it is created. If *error_table* exists and does not have a random distribution (the `DISTRIBUTED RANDOMLY` clause was not specified when creating the table), an error is returned.

See *Notes* for information about the error log information and built-in functions for viewing and managing error log information.

The *error_table* cannot be modified with the `ALTER TABLE` command.

> **Note:** The optional `INTO error_table` clause is deprecated and will not be supported in the next major release. Only internal error logs will be supported.

SEGMENT REJECT LIMIT *count* [ROWS | PERCENT]

Runs a `COPY FROM` operation in single row error isolation mode. If the input rows have format errors they will be discarded provided that the reject limit count is not reached on any Greenplum segment instance during the load operation. The reject limit count can be specified as number of rows (the default) or percentage of total rows (1-100). If `PERCENT` is used, each segment starts calculating the bad row percentage only after the number of rows specified by the parameter `gp_reject_percent_threshold` has been processed. The default for `gp_reject_percent_threshold` is 300 rows. Constraint errors such as violation of a `NOT NULL`, `CHECK`, or `UNIQUE` constraint will still be handled in "all-or-nothing" input mode. If the limit is not reached, all good rows will be loaded and any error rows discarded.

> **Note:** When reading an external table, Greenplum Database limits the initial number of rows that can contain formatting errors if the `SEGMENT REJECT LIMIT` is not triggered first or is not specified. If the first 1000 rows are rejected, the `COPY` operation is stopped and rolled back.
>
> The limit for the number of initial rejected rows can be changed with the Greenplum Database server configuration parameter `gp_initial_bad_row_limit`. See *Server Configuration Parameters* for information about the parameter.

DISTRIBUTED BY (*column*, [...])
DISTRIBUTED RANDOMLY

Used to declare the Greenplum Database distribution policy for a writable external table. By default, writable external tables are distributed randomly. If the source table you are exporting data from has a hash distribution policy, defining the same distribution key column(s) for the writable external table will improve unload performance by eliminating the need to move rows over the interconnect. When you issue an unload command such as `INSERT INTO` *wex_table* `SELECT * FROM` *source_table*, the rows that are unloaded can be sent directly from the segments to the output location if the two tables have the same hash distribution policy.

Examples

Start the `gpfdist` file server program in the background on port `8081` serving files from directory `/var/data/staging`:

```
gpfdist -p 8081 -d /var/data/staging -l /home/gpadmin/log &
```

Create a readable external table named `ext_customer` using the `gpfdist` protocol and any text formatted files (`*.txt`) found in the `gpfdist` directory. The files are formatted with a pipe (`|`) as the column delimiter and an empty space as `NULL`. Also access the external table in single row error isolation mode:

```
CREATE EXTERNAL TABLE ext_customer
   (id int, name text, sponsor text)
   LOCATION ( 'gpfdist://filehost:8081/*.txt' )
   FORMAT 'TEXT' ( DELIMITER '|' NULL ' ')
   LOG ERRORS INTO err_customer SEGMENT REJECT LIMIT 5;
```

Create the same readable external table definition as above, but with CSV formatted files:

```
CREATE EXTERNAL TABLE ext_customer
   (id int, name text, sponsor text)
   LOCATION ( 'gpfdist://filehost:8081/*.csv' )
   FORMAT 'CSV' ( DELIMITER ',' );
```

Create a readable external table named `ext_expenses` using the `file` protocol and several CSV formatted files that have a header row:

```
CREATE EXTERNAL TABLE ext_expenses (name text, date date,
amount float4, category text, description text)
LOCATION (
'file://seghost1/dbfast/external/expenses1.csv',
'file://seghost1/dbfast/external/expenses2.csv',
'file://seghost2/dbfast/external/expenses3.csv',
'file://seghost2/dbfast/external/expenses4.csv',
'file://seghost3/dbfast/external/expenses5.csv',
'file://seghost3/dbfast/external/expenses6.csv'
)
FORMAT 'CSV' ( HEADER );
```

Create a readable web external table that executes a script once per segment host:

```
CREATE EXTERNAL WEB TABLE log_output (linenum int, message
text)  EXECUTE '/var/load_scripts/get_log_data.sh' ON HOST
 FORMAT 'TEXT' (DELIMITER '|');
```

Create a writable external table named `sales_out` that uses `gpfdist` to write output data to a file named `sales.out`. The files are formatted with a pipe (`|`) as the column delimiter and an empty space as `NULL`.

```
CREATE WRITABLE EXTERNAL TABLE sales_out (LIKE sales)
   LOCATION ('gpfdist://etl1:8081/sales.out')
   FORMAT 'TEXT' ( DELIMITER '|' NULL ' ')
   DISTRIBUTED BY (txn_id);
```

Create a writable external web table that pipes output data received by the segments to an executable script named `to_adreport_etl.sh`:

```
CREATE WRITABLE EXTERNAL WEB TABLE campaign_out
(LIKE campaign)
 EXECUTE '/var/unload_scripts/to_adreport_etl.sh'
```

```
 FORMAT 'TEXT' (DELIMITER '|');
```

Use the writable external table defined above to unload selected data:

```
INSERT INTO campaign_out SELECT * FROM campaign WHERE
customer_id=123;
```

Notes

Greenplum database can log information about rows with formatting errors in an error table or internally. When you specify `LOG ERRORS INTO error_table`, Greenplum Database creates the table *error_table* that contains errors that occur while reading the external table. The table is defined as follows:

```
CREATE TABLE error_table_name ( cmdtime timestamptz, relname text,
    filename text, linenum int, bytenum int, errmsg text,
    rawdata text, rawbytes bytea ) DISTRIBUTED RANDOMLY;
```

You can view the information in the table with SQL commands.

For error log data that is stored internally when the `INTO error_table` is not specified:

- Use the built-in SQL function `gp_read_error_log('table_name')`. It requires `SELECT` privilege on *table_name*.

  This example displays the error log information for data copied into the table `ext_expenses`:

  ```
  SELECT * from gp_read_error_log('ext_expenses');
  ```

  The error log contains the same columns as the error table.

  The function returns `FALSE` if `table_name` does not exist.
- If error log data exists for the specified table, the new error log data is appended to existing error log data. The error log information is not replicated to mirror segments.
- Use the built-in SQL function `gp_truncate_error_log('table_name')` to delete the error log data for *table_name*. It requires the table owner privilege. This example deletes the error log information captured when moving data into the table `ext_expenses`:

  ```
  SELECT gp_truncate_error_log('ext_expenses');
  ```

  The function returns `FALSE` if *table_name* does not exist.

  Specify the * wildcard character to delete error log information for existing tables in the current database. Specify the string `*.*` to delete all database error log information, including error log information that was not deleted due to previous database issues. If * is specified, database owner privilege is required. If `*.*` is specified, database super-user privilege is required.

HDFS File Format Support for the gphdfs Protocol

If you specify `gphdfs` protocol to read or write file to a Hadoop file system (HDFS), you can read or write an Avro or Parquet format file by specifying the file format with the `FORMAT` clause.

To read data from or write data to an Avro or Parquet file, you create an external table with the `CREATE EXTERNAL TABLE` command and specify the location of the Avro file in the `LOCATION` clause and `'AVRO'` in the `FORMAT` clause. This example is for a readable external table that reads from an Avro file.

```
CREATE EXTERNAL TABLE tablename (column_spec) LOCATION ( 'gphdfs://location') FORMAT
 'AVRO'
```

The *location* can be a file name or a directory containing a set of files. For the file name you can specify the wildcard character * to match any number of characters. If the location specifies multiple files when reading files, Greenplum Database uses the schema in the first file that is read as the schema for the other files.

As part of the *location* parameter you can specify options for reading or writing the file. After the file name, you can specify parameters with the HTTP query string syntax that starts with ? and uses & between field value pairs.

For this example *location* parameter, this URL sets compression parameters for an Avro format writeable external table.

```
'gphdfs://myhdfs:8081/avro/singleAvro/array2.avro?
compress=true&compression_type=block&codec=snappy' FORMAT 'AVRO'
```

See "Loading and Unloading Data" in the *Greenplum Database Administrator Guide* for information about reading and writing the Avro and Parquet format files with external tables.

Avro Files

For readable external tables, the only valid parameter is `schema`. When reading multiple Avro files, you can specify a file that contains an Avro schema. See "Avro Schema Overrides" in the Greenplum Database

For writable external tables, you can specify `schema`, `namespace`, and parameters for compression.

Table 3: Avro Format External Table location Parameters

Parameter	Value	Readable/ Writeable	Default Value
schema	*URL_to_schema_file*	Read and Write	None. For a readable external table • The specified schema overrides the schema in the Avro file. See "Avro Schema Overrides" • If not specified, Greenplum Database uses the Avro file schema. For a writeable external table • Uses the specified schema when creating the Avro file. • If not specified, Greenplum Database creates a schema according to the external table definition.
namespace	*avro_namespace*	Write only	`public.avro` If specified, a valid Avro *namespace*.
compress	`true` or `false`	Write only	`false`
compression_type	`block`	Write only	Optional. For `avro` format, `compression_type` must be `block` if `compress` is `true`.
codec	`deflate` or `snappy`	Write only	`deflate`
codec_level (`deflate` codec only)	integer between 1 and 9	Write only	`6` The level controls the trade-off between speed and compression. Valid values are 1 to 9, where 1 is the fastest and 9 is the most compressed.

This set of parameters specify `snappy` compression:

```
'compress=true&codec=snappy'
```

These two sets of parameters specify `deflate` compression and are equivalent:

```
'compress=true&codec=deflate&codec_level=1'
'compress=true&codec_level=1'
```

Parquet Files

For external tables, you can add parameters after the file specified in the *location*. This table lists the valid parameters and values.

Table 4: Parquet Format External Table location Parameters

Option	Values	Readable/ Writeable	Default Value
schema	*URL_to_schema*	Write only	None. If not specified, Greenplum Database creates a schema according to the external table definition.
pagesize	> 1024 Bytes	Write only	1 MB
rowgroupsize	> 1024 Bytes	Write only	8 MB
version	`v1`, `v2`	Write only	`v1`
codec	`UNCOMPRESSED`, `GZIP`, `LZO`, `snappy`	Write only	`UNCOMPRESSED`
dictionaryenable[1]	`true`, `false`	Write only	false
dictionarypagesize[1]	> 1024 Bytes	Write only	512 KB

Note:

1. Creates an internal dictionary. Enabling a dictionary might improve Parquet file compression if text columns contain similar or duplicate data.

s3 Protocol Configuration

The `s3` protocol is used with a URI to specify the location of files in an Amazon Simple Storage Service (Amazon S3) bucket. The protocol downloads all files specified by the `LOCATION` clause. Each Greenplum Database segment instance downloads one file at a time using several threads. The segment instances download files until all files have been downloaded.

Before you create a readable external table with the `s3` protocol, you must configure the Greenplum Database system.

- Configure the database to support the `s3` protocol.
- Create and install the `s3` protocol configuration file on all the Greenplum Database segments.

To configure a database to support the `s3` protocol

1. Create a function to access the `s3` protocol library.

   In each Greenplum database that accesses an S3 bucket with the `s3` protocol, create a function for the protocol:

   ```
   CREATE OR REPLACE FUNCTION read_from_s3() RETURNS integer AS
   ```

```
    '$libdir/gps3ext.so', 's3_import'
LANGUAGE C STABLE;
```

2. Declare the `s3` protocol and specify the function that is used to read from an S3 bucket.

```
CREATE PROTOCOL s3 (readfunc = read_from_s3);
```

Note: The protocol name `s3` must be the same as the protocol of the URL specified for the readable external table you create to access an S3 resource.

The function is called by every segment. All Greenplum Database segments must have access to the S3 bucket.

To create and install the `s3` protocol configuration file

1. Create a configuration file with the S3 configuration information.
2. Install the file in the same location for all Greenplum Database segments on all hosts.

You can specify the location of the file with the `config` parameter in the `LOCATION` clause of the `EXTERNAL TABLE` command. This is the default location:

```
gpseg_data-dir/gpseg-prefixN/s3/s3.conf
```

The *gpseg-data-dir* is the path to the Greenplum Database segment data directory, the *gpseg-prefix* is the segment prefix, and *N* is the segment ID. The segment data directory, prefix, and ID are set when you initialize a Greenplum Database system.

If you have multiple segment instances on segment hosts, you can simplify the configuration by creating a single location on each segment host. Then you specify the absolute path to the location with the `config` parameter in the `s3` protocol `LOCATION` clause.

s3 Protocol Limitations

These are `s3` protocol limitations:

- Only the S3 path-style URL is supported.

```
s3://S3_endpoint/bucketname/[S3_prefix]
```

- Only the S3 endpoint is supported. The protocol does not support virtual hosting of S3 buckets (binding a domain name to an S3 bucket).
- AWS signature version 2 and version 4 signing process are supported.

  For information about the S3 endpoints supported by each signing process, see *http://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region*.
- S3 encryption is not supported. The S3 file property **Server Side Encryption** must be `None`.

Note: To take advantage of the parallel processing performed by the Greenplum Database segment instances, the files in the S3 location should be similar in size and the number of files should allow for multiple segments to download the data from the S3 location. For example, if the Greenplum Database system consists of 16 segments and there was sufficient network bandwidth, creating 16 files in the S3 location allows each segment to download a file from the S3 location. In contrast, if the location contained only 1 or 2 files, only 1 or 2 segments download data.

s3 Protocol Configuration File

When using the `s3` protocol, the `s3` protocol configuration file is required on all Greenplum Database segments. The default location is

```
gpseg_data_dir/gpseg-prefixN/s3/s3.conf
```

The *gpseg_data_dir* is the path to the Greenplum Database segment data directory, the *gpseg-prefix* is the segment prefix, and *N* is the segment ID. The segment data directory, prefix, and ID are set when you initialize a Greenplum Database system.

If you have multiple segment instances on segment hosts, you can simplify the configuration by creating a single location on each segment host. Then you specify the absolute path to the location with the `config` parameter in the `s3` protocol `LOCATION` clause. This example specifies a location in the `gpadmin` home directory.

```
config=/home/gpadmin/s3/s3.conf
```

All segment instances on the hosts use the file `/home/gpadmin/s3/s3.conf`.

The `s3` protocol configuration file is a text file that consists of a `[default]` section and parameters. This is an example configuration file.

```
[default]
secret = "secret"
accessid = "user access id"
connections = 3
chunksize = 67108864
```

s3 Configuration File Parameters

accessid

Required. AWS S3 ID to access the S3 bucket.

secret

Required. AWS S3 passcode for the S3 ID to access the S3 bucket.

chunksize

The buffer size for each segment thread. The default is 64MB. The minimum is 2MB and the maximum is128MB.

threadnum

The maximum number of concurrent connections a segment can create when downloading data from the S3 bucket. The default is 4. The minimum is 1 and the maximum is 8.

encryption

Use connections that are secured with Secure Sockets Layer (SSL). Default value is `true`. The values `true`, `t`, `on`, `yes`, and `y` (case insensitive) are treated as `true`. Any other value is treated as `false`.

low_speed_limit

The download speed lower limit, in bytes per second. The default speed is 10240 (10K). If the download speed is slower than the limit for longer than the time specified by `low_speed_time`, the download connection is aborted and retried. After 3 retries, the `s3` protocol returns an error. A value of 0 specifies no lower limit.

low_speed_time

When the connection speed is less than `low_speed_limit`, the amount of time, in minutes, to wait before aborting a download from an S3 bucket. The default is 1 minute. A value of 0 specifies no time limit.

Note: You must ensure that there is sufficient memory on the Greenplum Database segment hosts when the `s3` protocol to accesses the files. Greenplum Database allocates `connections * chunksize` memory on each segment host when accessing S3 files.

Compatibility

`CREATE EXTERNAL TABLE` is a Greenplum Database extension. The SQL standard makes no provisions for external tables.

See Also

CREATE TABLE AS, *CREATE TABLE*, *COPY*, *SELECT INTO*, *INSERT*

CREATE FUNCTION

Defines a new function.

Synopsis

```
CREATE [OR REPLACE] FUNCTION name
    ( [ [argmode] [argname] argtype [, ...] ] )
      [ RETURNS { [ SETOF ] rettype
        | TABLE ([{ argname argtype | LIKE other table }
          [, ...]])
        } ]
    { LANGUAGE langname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER
    | AS 'definition'
    | AS 'obj_file', 'link_symbol' } ...
    [ WITH ({ DESCRIBE = describe_function
           } [, ...] ) ]
```

Description

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

The name of the new function must not match any existing function with the same argument types in the same schema. However, functions of different argument types may share a name (overloading).

To update the definition of an existing function, use `CREATE OR REPLACE FUNCTION`. It is not possible to change the name or argument types of a function this way (this would actually create a new, distinct function). Also, `CREATE OR REPLACE FUNCTION` will not let you change the return type of an existing function. To do that, you must drop and recreate the function. If you drop and then recreate a function, you will have to drop existing objects (rules, views, triggers, and so on) that refer to the old function. Use `CREATE OR REPLACE FUNCTION` to change a function definition without breaking objects that refer to the function.

For more information about creating functions, see the *User Defined Functions* section of the PostgreSQL documentation.

Limited Use of VOLATILE and STABLE Functions

To prevent data from becoming out-of-sync across the segments in Greenplum Database, any function classified as `STABLE` or `VOLATILE` cannot be executed at the segment level if it contains SQL or modifies the database in any way. For example, functions such as `random()` or `timeofday()` are not allowed to execute on distributed data in Greenplum Database because they could potentially cause inconsistent data between the segment instances.

To ensure data consistency, `VOLATILE` and `STABLE` functions can safely be used in statements that are evaluated on and execute from the master. For example, the following statements are always executed on the master (statements without a `FROM` clause):

```
SELECT setval('myseq', 201);
SELECT foo();
```

In cases where a statement has a `FROM` clause containing a distributed table and the function used in the `FROM` clause simply returns a set of rows, execution may be allowed on the segments:

```
SELECT * FROM foo();
```

One exception to this rule are functions that return a table reference (`rangeFuncs`) or functions that use the `refCursor` data type. Note that you cannot return a `refcursor` from any kind of function in Greenplum Database.

Parameters

name

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

argmode

The mode of an argument: either `IN`, `OUT`, or `INOUT`. If omitted, the default is `IN`.

argname

The name of an argument. Some languages (currently only PL/pgSQL) let you use the name in the function body. For other languages the name of an input argument is just extra documentation. But the name of an output argument is significant, since it defines the column name in the result row type. (If you omit the name for an output argument, the system will choose a default column name.)

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any. The argument types may be base, composite, or domain types, or may reference the type of a table column.

Depending on the implementation language it may also be allowed to specify pseudotypes such as `cstring`. Pseudotypes indicate that the actual argument type is either incompletely specified, or outside the set of ordinary SQL data types.

The type of a column is referenced by writing `tablename.columnname%TYPE`. Using this feature can sometimes help make a function independent of changes to the definition of a table.

rettype

The return data type (optionally schema-qualified). The return type can be a base, composite, or domain type, or may reference the type of a table column. Depending on the implementation language it may also be allowed to specify pseudotypes such as `cstring`. If the function is not supposed to return a value, specify `void` as the return type.

When there are `OUT` or `INOUT` parameters, the `RETURNS` clause may be omitted. If present, it must agree with the result type implied by the output parameters: `RECORD` if there are multiple output parameters, or the same type as the single output parameter.

The `SETOF` modifier indicates that the function will return a set of items, rather than a single item.

The type of a column is referenced by writing `tablename.columnname%TYPE`.

langname

The name of the language that the function is implemented in. May be `SQL`, `C`, `internal`, or the name of a user-defined procedural language. See *CREATE LANGUAGE* for the procedural languages supported in Greenplum Database. For backward compatibility, the name may be enclosed by single quotes.

IMMUTABLE
STABLE
VOLATILE

These attributes inform the query optimizer about the behavior of the function. At most one choice may be specified. If none of these appear, `VOLATILE` is the default assumption. Since Greenplum Database currently has limited use of `VOLATILE` functions, if a function is truly `IMMUTABLE`, you must declare it as so to be able to use it without restrictions.

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

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

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

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

`CALLED ON NULL INPUT` (the default) indicates that the function will be called normally when some of its arguments are null. It is then the function author's responsibility to check for null values if necessary and respond appropriately. `RETURNS NULL ON NULL INPUT` or `STRICT` indicates that the function always returns null whenever any of its arguments are null. If this parameter is specified, the function is not executed when there are null arguments; instead a null result is assumed automatically.

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

`SECURITY INVOKER` (the default) indicates that the function is to be executed with the privileges of the user that calls it. `SECURITY DEFINER` specifies that the function is to be executed with the privileges of the user that created it. The key word `EXTERNAL` is allowed for SQL conformance, but it is optional since, unlike in SQL, this feature applies to all functions not just external ones.

definition

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

obj_file, link_symbol

This form of the `AS` clause is used for dynamically loadable 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 name of the function in the C language source code. If the link symbol is omitted, it is assumed to be the same as the name of the SQL function being defined. It is recommended to locate shared libraries either relative to `$libdir` (which is located at `$GPHOME/lib`) or through the dynamic library path (set by the `dynamic_library_path` server configuration parameter). This simplifies version upgrades if the new installation is at a different location.

describe_function

The name of a callback function to execute when a query that calls this function is parsed. The callback function returns a tuple descriptor that indicates the result type.

Notes

Any compiled code (shared library files) for custom functions must be placed in the same location on every host in your Greenplum Database array (master and all segments). This location must also be in the `LD_LIBRARY_PATH` so that the server can locate the files. It is recommended to locate shared libraries either relative to `$libdir` (which is located at `$GPHOME/lib`) or through the dynamic library path (set by the `dynamic_library_path` server configuration parameter) on all master segment instances in the Greenplum array.

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

Greenplum Database allows function overloading. The same name can be used for several different functions so long as they have distinct argument types. However, the C names of all functions must be different, so you must give overloaded C functions different C names (for example, use the argument types as part of the C names).

Two functions are considered the same if they have the same names and input argument types, ignoring any OUT parameters. Thus for example these declarations conflict:

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, out text) ...
```

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

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

It is often helpful to use dollar quoting to write the function definition string, rather than the normal single quote syntax. Without dollar quoting, any single quotes or backslashes in the function definition must be escaped by doubling them. A dollar-quoted string constant consists of a dollar sign (`$`), an optional tag of zero or more characters, another dollar sign, an arbitrary sequence of characters that makes up the string content, a dollar sign, the same tag that began this dollar quote, and a dollar sign. Inside the dollar-quoted string, single quotes, backslashes, or any character can be used without escaping. The string content is always written literally. For example, here are two different ways to specify the string "Dianne's horse" using dollar quoting:

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

Using Functions With Queries on Distributed Data

In some cases, Greenplum Database does not support using functions in a query where the data in a table specified in the `FROM` clause is distributed over Greenplum Database segments. As an example, this SQL query contains the function `func()`:

```
SELECT func(a) FROM table1;
```

The function is not supported for use in the query if all of the following conditions are met:

- The data of table `table1` is distributed over Greenplum Database segments.
- The function `func()` reads or modifies data from distributed tables.
- The function `func()` returns more than one row or takes an argument (`a`) that comes from `table1`.

If any of the conditions are not met, the function is supported. Specifically, the function is supported if any of the following conditions apply:

- The function `func()` does not access data from distributed tables, or accesses data that is only on the Greenplum Database master.
- The table `table1` is a master only table.

- The function `func()` returns only one row and only takes input arguments that are constant values. The function is supported if it can be changed to require no input arguments.

Examples

A very simple addition function:

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

Increment an integer, making use of an argument name, in PL/pgSQL:

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS
integer AS $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;
```

Return a record containing multiple output parameters:

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;
SELECT * FROM dup(42);
```

You can do the same thing more verbosely with an explicitly named composite type:

```
CREATE TYPE dup_result AS (f1 int, f2 text);
CREATE FUNCTION dup(int) RETURNS dup_result
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;
SELECT * FROM dup(42);
```

Compatibility

`CREATE FUNCTION` is defined in SQL:1999 and later. The Greenplum Database version is similar but not fully compatible. The attributes are not portable, neither are the different available languages.

For compatibility with some other database systems, *argmode* can be written either before or after *argname*. But only the first way is standard-compliant.

See Also

ALTER FUNCTION, *DROP FUNCTION*, *LOAD*

CREATE GROUP

Defines a new database role.

Synopsis

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

where *option* can be:

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE rolename [, ...]
    | IN GROUP rolename [, ...]
    | ROLE rolename [, ...]
    | ADMIN rolename [, ...]
    | USER rolename [, ...]
    | SYSID uid
```

Description

As of Greenplum Database release 2.2, `CREATE GROUP` has been replaced by *CREATE ROLE*, although it is still accepted for backwards compatibility.

Compatibility

There is no `CREATE GROUP` statement in the SQL standard.

See Also

CREATE ROLE

CREATE INDEX

Defines a new index.

Synopsis

```
CREATE [UNIQUE] INDEX name ON table
       [USING btree|bitmap|gist]
       ( {column | (expression)} [opclass] [, ...] )
       [ WITH ( FILLFACTOR = value ) ]
       [TABLESPACE tablespace]
       [WHERE predicate]
```

Description

`CREATE INDEX` constructs an index on the specified table. Indexes are primarily used to enhance database performance (though inappropriate use can result in slower performance).

The key field(s) for the index are specified as column names, or alternatively as expressions written in parentheses. Multiple fields can be specified if the index method supports multicolumn indexes.

An index field can be an expression computed from the values of one or more columns of the table row. This feature can be used to obtain fast access to data based on some transformation of the basic data. For example, an index computed on `upper(col)` would allow the clause `WHERE upper(col) = 'JIM'` to use an index.

Greenplum Database provides the index methods B-tree, bitmap, and GiST. Users can also define their own index methods, but that is fairly complicated.

When the `WHERE` clause is present, a partial index is created. A partial index is an index that contains entries for only a portion of a table, usually a portion that is more useful for indexing than the rest of the table. For example, if you have a table that contains both billed and unbilled orders where the unbilled orders take up a small fraction of the total table and yet is most often selected, you can improve performance by creating an index on just that portion.

The expression used in the `WHERE` clause may refer only to columns of the underlying table, but it can use all columns, not just the ones being indexed. Subqueries and aggregate expressions are also forbidden in `WHERE`. The same restrictions apply to index fields that are expressions.

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

Parameters

UNIQUE

Checks for duplicate values in the table when the index is created and each time data is added. Duplicate entries will generate an error. Unique indexes only apply to B-tree indexes. In Greenplum Database, unique indexes are allowed only if the columns of the index key are the same as (or a superset of) the Greenplum distribution key. On partitioned tables, a unique index is only supported within an individual partition - not across all partitions.

name

The name of the index to be created. The index is always created in the same schema as its parent table.

table

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

btree | bitmap | gist

The name of the index method to be used. Choices are `btree`, `bitmap`, and `gist`. The default method is `btree`.

column

The name of a column of the table on which to create the index. Only the B-tree, bitmap, and GiST index methods support multicolumn indexes.

expression

An expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax. However, the parentheses may be omitted if the expression has the form of a function call.

opclass

The name of an operator class. The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on four-byte integers would use the `int4_ops` class (this operator class includes comparison functions for four-byte integers). In practice the default operator class for the column'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, a complex-number data type could be sorted by either 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.

FILLFACTOR

The fillfactor for an index is a percentage that determines how full the index method will try to pack index pages. For B-trees, leaf pages are filled to this percentage during initial index build, and also when extending the index at the right (largest key values). If pages subsequently become completely full, they will be split, leading to gradual degradation in the index's efficiency.

B-trees use a default fillfactor of 90, but any value from 10 to 100 can be selected. If the table is static then fillfactor 100 is best to minimize the index's physical size, but for heavily updated tables a smaller fillfactor is better to minimize the need for page splits. The other index methods use fillfactor in different but roughly analogous ways; the default fillfactor varies between methods.

tablespace

The tablespace in which to create the index. If not specified, the default tablespace is used.

predicate

The constraint expression for a partial index.

Notes

When an index is created on a partitioned table, the index is propagated to all the child tables created by Greenplum Database. Creating an index on a table that is created by Greenplum Database for use by a partitioned table is not supported.

`UNIQUE` indexes are allowed only if the index columns are the same as (or a superset of) the Greenplum distribution key columns.

`UNIQUE` indexes are not allowed on append-only tables.

A `UNIQUE` index can be created on a partitioned table. However, uniqueness is enforced only within a partition; uniqueness is not enforced between partitions. For example, for a partitioned table with partitions that are based on year and a subpartitions that are based on quarter, uniqueness is enforced only on each individual quarter partition. Uniqueness is not enforced between quarter partitions

Indexes are not used for `IS NULL` clauses by default. The best way to use indexes in such cases is to create a partial index using an `IS NULL` predicate.

`bitmap` indexes perform best for columns that have between 100 and 100,000 distinct values. For a column with more than 100,000 distinct values, the performance and space efficiency of a bitmap index decline. The size of a bitmap index is proportional to the number of rows in the table times the number of distinct values in the indexed column.

Columns with fewer than 100 distinct values usually do not benefit much from any type of index. For example, a gender column with only two distinct values for male and female would not be a good candidate for an index.

Prior releases of Greenplum Database also had an R-tree index method. This method has been removed because it had no significant advantages over the GiST method. If `USING rtree` is specified, `CREATE INDEX` will interpret it as `USING gist`.

For more information on the GiST index type, refer to the *PostgreSQL documentation*.

The use of hash and GIN indexes has been disabled in Greenplum Database.

Examples

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

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

To create a bitmap index on the column `gender` in the table `employee`:

```
CREATE INDEX gender_bmp_idx ON employee USING bitmap
(gender);
```

To create an index on the expression `lower(title)`, allowing efficient case-insensitive searches:

```
CREATE INDEX lower_title_idx ON films ((lower(title)));
```

To create an index with non-default fill factor:

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH
(fillfactor = 70);
```

To create an index on the column `code` in the table `films` and have the index reside in the tablespace `indexspace`:

```
CREATE INDEX code_idx ON films(code) TABLESPACE indexspace;
```

Compatibility

`CREATE INDEX` is a Greenplum Database language extension. There are no provisions for indexes in the SQL standard.

Greenplum Database does not support the concurrent creation of indexes (`CONCURRENTLY` keyword not supported).

See Also

ALTER INDEX, *DROP INDEX*, *CREATE TABLE*, *CREATE OPERATOR CLASS*

CREATE LANGUAGE

Defines a new procedural language.

Synopsis

```
CREATE [PROCEDURAL] LANGUAGE name

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

Description

`CREATE LANGUAGE` registers a new procedural language with a Greenplum database. Subsequently, functions and trigger procedures can be defined in this new language. You must be a superuser to register a new language. The PL/pgSQL language is already registered in all databases by default.

`CREATE LANGUAGE` effectively associates the language name with a call handler that is responsible for executing functions written in that language. For a function written in a procedural language (a language other than C or SQL), the database server has no built-in knowledge about how to interpret the function's source code. The task is passed to a special handler that knows the details of the language. The handler could either do all the work of parsing, syntax analysis, execution, and so on or it could serve as a bridge between Greenplum Database and an existing implementation of a programming language. The handler itself is a C language function compiled into a shared object and loaded on demand, just like any other C function. There are currently four procedural language packages included in the standard Greenplum Database distribution: PL/pgSQL, PL/Perl, PL/Python, and PL/Java. A language handler has also been added for PL/R, but the PL/R language package is not pre-installed with Greenplum Database. See the topic on *Procedural Languages* in the PostgreSQL documentation for more information on developing functions using these procedural languages.

The PL/Perl, PL/Java, and PL/R libraries require the correct versions of Perl, Java, and R to be installed, respectively.

On RHEL and SUSE platforms, download the appropriate extensions from *Pivotal Network*, then install the extensions using the Greenplum Package Manager (`gppkg`) utility to ensure that all dependencies are installed as well as the extensions. See the Greenplum Database Utility Guide for details about `gppkg`.

There are two forms of the `CREATE LANGUAGE` command. In the first form, the user specifies the name of the desired language and the Greenplum Database server uses the `pg_pltemplate` system catalog to determine the correct parameters. In the second form, the user specifies the language parameters as well as the language name. You can use the second form to create a language that is not defined in `pg_pltemplate`.

When the server finds an entry in the `pg_pltemplate` catalog for the given language name, it will use the catalog data even if the command includes language parameters. This behavior simplifies loading of old dump files, which are likely to contain out-of-date information about language support functions.

Parameters

TRUSTED

Ignored if the server has an entry for the specified language name in *pg_pltemplate*. Specifies that the call handler for the language is safe and does not offer an unprivileged user any functionality to bypass access restrictions. If this key word is omitted when registering the language, only users with the superuser privilege can use this language to create new functions.

PROCEDURAL

This is a noise word.

name

The name of the new procedural language. The language name is case insensitive. The name must be unique among the languages in the database. Built-in support is included for `plpgsql`, `plperl`, `plpython`, `plpythonu`, and `plr`. The languages `plpgsql` (PL/pgSQL) and `plpythonu` (PL/Python) are installed by default in Greenplum Database.

HANDLER *call_handler*

Ignored if the server has an entry for the specified language name in `pg_pltemplate`. The name of a previously registered function that will be called to execute the procedural language functions. The call handler for a procedural language must be written in a compiled language such as C with version 1 call convention and registered with Greenplum Database as a function taking no arguments and returning the `language_handler` type, a placeholder type that is simply used to identify the function as a call handler.

VALIDATOR *valfunction*

Ignored if the server has an entry for the specified language name in `pg_pltemplate`. *valfunction* is the name of a previously registered function that will be called when a new function in the language is created, to validate the new function. If no validator function is specified, then a new function will not be checked when it is created. The validator function must take one argument of type `oid`, which will be the OID of the to-be-created function, and will typically return `void`.

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

Notes

The PL/pgSQL and PL/Python language extensions are installed by default in Greenplum Database.

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

To create functions in a procedural language, a user must have the `USAGE` privilege for the language. By default, `USAGE` is granted to `PUBLIC` (everyone) for trusted languages. This may be revoked if desired.

Procedural languages are local to individual databases. You create and drop languages for individual databases.

The call handler function and the validator function (if any) must already exist if the server does not have an entry for the language in `pg_pltemplate`. But when there is an entry, the functions need not already exist; they will be automatically defined if not present in the database.

Any shared library that implements a language must be located in the same `LD_LIBRARY_PATH` location on all segment hosts in your Greenplum Database array.

Examples

The preferred way of creating any of the standard procedural languages:

```
CREATE LANGUAGE plpgsql;
CREATE LANGUAGE plr;
```

For a language not known in the `pg_pltemplate` catalog:

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

```
    HANDLER plsample_call_handler;
```

Compatibility

`CREATE LANGUAGE` is a Greenplum Database extension.

See Also

ALTER LANGUAGE, *CREATE FUNCTION*, *DROP LANGUAGE*

CREATE OPERATOR

Defines a new operator.

Synopsis

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

Description

`CREATE OPERATOR` defines a new operator. The user who defines an operator becomes its owner.

The operator name is a sequence of up to `NAMEDATALEN`-1 (63 by default) characters from the following list: `+ - * / < > = ~ ! @ # % ^ & | ` ?`

There are a few restrictions on your choice of name:

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

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

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 must be defined. For right unary operators, only `LEFTARG` should be defined, while for left unary operators only `RIGHTARG` should be defined.

The *funcname* procedure must have been previously defined using `CREATE FUNCTION`, must be `IMMUTABLE`, and must be defined to accept the correct number of arguments (either one or two) of the indicated types.

The other clauses specify optional operator optimization clauses. These clauses should be provided whenever appropriate to speed up queries that use the operator. But if you provide them, you must be sure that they are correct. Incorrect use of an optimization clause can result in server process crashes, subtly wrong output, or other unexpected results. You can always leave out an optimization clause if you are not sure about it.

Parameters

name

The (optionally schema-qualified) name of the operator to be defined. Two operators in the same schema can have the same name if they operate on different data types.

funcname

The function used to implement this operator (must be an `IMMUTABLE` function).

lefttype

The data type of the operator's left operand, if any. This option would be omitted for a left-unary operator.

righttype

The data type of the operator's right operand, if any. This option would be omitted for a right-unary operator.

com_op

The optional `COMMUTATOR` clause names an operator that is the commutator of the operator being defined. We say that operator A is the commutator of operator B if (x A y) equals (y B x) for all possible input values x, y. Notice that B is also the commutator of A. For example, operators `<` and `>` for a particular data type are usually each others commutators, and operator + is usually commutative with itself. But operator `-` is usually not commutative with anything. The left operand type of a commutable operator is the same as the right operand type of its commutator, and vice versa. So the name of the commutator operator is all that needs to be provided in the `COMMUTATOR` clause.

neg_op

The optional `NEGATOR` clause names an operator that is the negator of the operator being defined. We say that operator A is the negator of operator B if both return Boolean results and (x A y) equals NOT (x B y) for all possible inputs x, y. Notice that B is also the negator of A. For example, `<` and `>=` are a negator pair for most data types. An operator's negator must have the same left and/or right operand types as the operator to be defined, so only the operator name need be given in the `NEGATOR` clause.

res_proc

The optional `RESTRICT` names a restriction selectivity estimation function for the operator. Note that this is a function name, not an operator name. `RESTRICT` clauses only make sense for binary operators that return `boolean`. The idea behind a restriction selectivity estimator is to guess what fraction of the rows in a table will satisfy a `WHERE`-clause condition of the form:

```
column OP constant
```

for the current operator and a particular constant value. This assists the optimizer by giving it some idea of how many rows will be eliminated by `WHERE` clauses that have this form.

You can usually just use one of the following system standard estimator functions for many of your own operators:

`eqsel` for =

`neqsel` for <>

`scalarltsel` for < or <=

`scalargtsel` for > or >=

join_proc

The optional `JOIN` clause names a join selectivity estimation function for the operator. Note that this is a function name, not an operator name. `JOIN` clauses only make sense for binary operators that return `boolean`. The idea behind a join selectivity estimator is to guess what fraction of the rows in a pair of tables will satisfy a `WHERE`-clause condition of the form

```
table1.column1 OP table2.column2
```

for the current operator. This helps the optimizer by letting it figure out which of several possible join sequences is likely to take the least work.

You can usually just use one of the following system standard join selectivity estimator functions for many of your own operators:

`eqjoinsel` for =

`neqjoinsel` for <>

`scalarltjoinsel` for < or <=

`scalargtjoinsel` for > or >=

`areajoinsel` for 2D area-based comparisons

`positionjoinsel` for 2D position-based comparisons

`contjoinsel` for 2D containment-based comparisons

HASHES

The optional `HASHES` clause tells the system that it is permissible to use the hash join method for a join based on this operator. `HASHES` only makes sense for a binary operator that returns `boolean`. The hash join operator can only return true for pairs of left and right values that hash to the same hash code. If two values get put in different hash buckets, the join will never compare them at all, implicitly assuming that the result of the join operator must be false. So it never makes sense to specify `HASHES` for operators that do not represent equality.

To be marked `HASHES`, the join operator must appear in a hash index operator class. Attempts to use the operator in hash joins will fail at run time if no such operator class exists. The system needs the operator class to find the data-type-specific hash function for the operator's input data type. You must also supply a suitable hash function before you can create the operator class. Care should be exercised when preparing a hash function, because there are machine-dependent ways in which it might fail to do the right thing.

MERGES

The `MERGES` clause, if present, tells the system that it is permissible to use the merge-join method for a join based on this operator. `MERGES` only makes sense for a binary operator that returns `boolean`, and in practice the operator must represent equality for some data type or pair of data types.

Merge join is based on the idea of sorting the left- and right-hand tables into order and then scanning them in parallel. So, both data types must be capable of being fully ordered, and the join operator must be one that can only succeed for pairs of values that fall at the same place in the sort order. In practice this means that the join operator must behave like equality. It is possible to merge-join two distinct data types so long as they are logically compatible. For example, the smallint-versus-integer equality operator is merge-joinable. We only need sorting operators that will bring both data types into a logically compatible sequence.

Execution of a merge join requires that the system be able to identify four operators related to the merge-join equality operator: less-than comparison for the left operand data type, less-than comparison for the right operand data type, less-than comparison between the two data types, and greater-than comparison between the two data types. It is possible to specify these operators individually by name, as the `SORT1`, `SORT2`, `LTCMP`, and `GTCMP` options respectively. The system will fill in the default names if any of these are omitted when `MERGES` is specified.

left_sort_op

If this operator can support a merge join, the less-than operator that sorts the left-hand data type of this operator. `<` is the default if not specified.

right_sort_op

If this operator can support a merge join, the less-than operator that sorts the right-hand data type of this operator. `<` is the default if not specified.

less_than_op

If this operator can support a merge join, the less-than operator that compares the input data types of this operator. `<` is the default if not specified.

greater_than_op

If this operator can support a merge join, the greater-than operator that compares the input data types of this operator. `>` is the default if not specified.

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

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

Notes

Any functions used to implement the operator must be defined as `IMMUTABLE`.

Examples

Here is an example of creating an operator for adding two complex numbers, assuming we have already created the definition of type `complex`. First define the function that does the work, then define the operator:

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'filename', 'complex_add'
    LANGUAGE C IMMUTABLE STRICT;
CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
```

To use this operator in a query:

```
SELECT (a + b) AS c FROM test_complex;
```

Compatibility

`CREATE OPERATOR` is a Greenplum Database language extension. The SQL standard does not provide for user-defined operators.

See Also

CREATE FUNCTION, *CREATE TYPE*, *ALTER OPERATOR*, *DROP OPERATOR*

CREATE OPERATOR CLASS

Defines a new operator class.

Synopsis

```
CREATE OPERATOR CLASS name [DEFAULT] FOR TYPE data_type
  USING index_method AS
  {
  OPERATOR strategy_number op_name [(op_type, op_type)] [RECHECK]
  | FUNCTION support_number funcname (argument_type [, ...] )
  | STORAGE storage_type
  } [, ... ]
```

Description

`CREATE OPERATOR CLASS` creates a new operator class. An operator class defines how a particular data type can be used with an index. The operator class specifies that certain operators will fill particular roles or strategies for this data type and this index method. The operator class also specifies the support procedures to be used by the index method when the operator class is selected for an index column. All the operators and functions used by an operator class must be defined before the operator class is created. Any functions used to implement the operator class must be defined as `IMMUTABLE`.

`CREATE OPERATOR CLASS` does not presently check whether the operator class definition includes all the operators and functions required by the index method, nor whether the operators and functions form a self-consistent set. It is the user's responsibility to define a valid operator class.

You must be a superuser to create an operator class.

Parameters

name

The (optionally schema-qualified) name of the operator class to be defined. Two operator classes in the same schema can have the same name only if they are for different index methods.

DEFAULT

Makes the operator class the default operator class for its data type. At most one operator class can be the default for a specific data type and index method.

data_type

The column data type that this operator class is for.

index_method

The name of the index method this operator class is for. Choices are `btree`, `bitmap`, and `gist`.

strategy_number

The operators associated with an operator class are identified by *strategy numbers*, which serve to identify the semantics of each operator within the context of its operator class. For example, B-trees impose a strict ordering on keys, lesser to greater, and so operators like *less than* and *greater than or equal to* are interesting with respect to a B-tree. These strategies can be thought of as generalized operators. Each operator class specifies which actual operator corresponds to each strategy for a particular data type and interpretation of the index semantics. The corresponding strategy numbers for each index method are as follows:

Table 5: B-tree and Bitmap Strategies

Operation	Strategy Number
less than	1
less than or equal	2
equal	3
greater than or equal	4
greater than	5

Table 6: GiST Two-Dimensional Strategies (R-Tree)

Operation	Strategy Number
strictly left of	1
does not extend to right of	2
overlaps	3
does not extend to left of	4
strictly right of	5
same	6
contains	7
contained by	8
does not extend above	9
strictly below	10
strictly above	11
does not extend below	12

operator_name

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

op_type

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

RECHECK

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

support_number

Index methods require additional support routines in order to work. These operations are administrative routines used internally by the index methods. As with strategies, the operator class identifies which specific functions should play each of these roles for a given data type and semantic interpretation. The index method defines the set of functions it needs, and the operator class identifies the correct functions to use by assigning them to the *support function numbers* as follows:

Table 7: B-tree and Bitmap Support Functions

Function	Support Number
Compare two keys and return an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second.	1

Table 8: GiST Support Functions

Function	Support Number
consistent - determine whether key satisfies the query qualifier.	1
union - compute union of a set of keys.	2
compress - compute a compressed representation of a key or value to be indexed.	3
decompress - compute a decompressed representation of a compressed key.	4
penalty - compute penalty for inserting new key into subtree with given subtree's key.	5
picksplit - determine which entries of a page are to be moved to the new page and compute the union keys for resulting pages.	6
equal - compare two keys and return true if they are equal.	7

funcname

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

argument_types

The parameter data type(s) of the function.

storage_type

The data type actually stored in the index. Normally this is the same as the column data type, but the GiST index method allows it to be different. The `STORAGE` clause must be omitted unless the index method allows a different type to be used.

Notes

Because the index machinery does not check access permissions on functions before using them, including a function or operator in an operator class is the same as granting public execute permission on it. This is usually not an issue for the sorts of functions that are useful in an operator class.

The operators should not be defined by SQL functions. A SQL function is likely to be inlined into the calling query, which will prevent the optimizer from recognizing that the query matches an index.

Any functions used to implement the operator class must be defined as `IMMUTABLE`.

Examples

The following example command defines a GiST index operator class for the data type `_int4` (array of int4):

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

Compatibility

`CREATE OPERATOR CLASS` is a Greenplum Database extension. There is no `CREATE OPERATOR CLASS` statement in the SQL standard.

See Also

ALTER OPERATOR CLASS, *DROP OPERATOR CLASS*, *CREATE FUNCTION*

CREATE RESOURCE QUEUE

Defines a new resource queue.

Synopsis

```
CREATE RESOURCE QUEUE name WITH (queue_attribute=value [, ... ])
```

where *queue_attribute* is:

```
    ACTIVE_STATEMENTS=integer
        [ MAX_COST=float [COST_OVERCOMMIT={TRUE|FALSE}] ]
        [ MIN_COST=float ]
        [ PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX} ]
        [ MEMORY_LIMIT='memory_units' ]

 | MAX_COST=float [ COST_OVERCOMMIT={TRUE|FALSE} ]
        [ ACTIVE_STATEMENTS=integer ]
        [ MIN_COST=float ]
        [ PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX} ]
        [ MEMORY_LIMIT='memory_units' ]
```

Description

Creates a new resource queue for Greenplum Database workload management. A resource queue must have either an `ACTIVE_STATEMENTS` or a `MAX_COST` value (or it can have both). Only a superuser can create a resource queue.

Resource queues with an `ACTIVE_STATEMENTS` threshold set a maximum limit on the number of queries that can be executed by roles assigned to that queue. It controls the number of active queries that are allowed to run at the same time. The value for `ACTIVE_STATEMENTS` should be an integer greater than 0.

Resource queues with a `MAX_COST` threshold set a maximum limit on the total cost of queries that can be executed by roles assigned to that queue. Cost is measured in the *estimated total cost* for the query as determined by the Greenplum Database query planner (as shown in the `EXPLAIN` output for a query). Therefore, an administrator must be familiar with the queries typically executed on the system in order to set an appropriate cost threshold for a queue. Cost is measured in units of disk page fetches; 1.0 equals one sequential disk page read. The value for `MAX_COST` is specified as a floating point number (for example 100.0) or can also be specified as an exponent (for example 1e+2). If a resource queue is limited based on a cost threshold, then the administrator can allow `COST_OVERCOMMIT=TRUE` (the default). This means that a query that exceeds the allowed cost threshold will be allowed to run but only when the system is idle. If `COST_OVERCOMMIT=FALSE` is specified, queries that exceed the cost limit will always be rejected and never allowed to run. Specifying a value for `MIN_COST` allows the administrator to define a cost for small queries that will be exempt from resource queueing.

If a value is not defined for `ACTIVE_STATEMENTS` or `MAX_COST`, it is set to `-1` by default (meaning no limit). After defining a resource queue, you must assign roles to the queue using the *ALTER ROLE* or *CREATE ROLE* command.

You can optionally assign a `PRIORITY` to a resource queue to control the relative share of available CPU resources used by queries associated with the queue in relation to other resource queues. If a value is not defined for `PRIORITY`, queries associated with the queue have a default priority of `MEDIUM`.

Resource queues with an optional `MEMORY_LIMIT` threshold set a maximum limit on the amount of memory that all queries submitted through a resource queue can consume on a segment host. This determines the total amount of memory that all worker processes of a query can consume on a segment host during query execution. Greenplum recommends that `MEMORY_LIMIT` be used in conjunction with `ACTIVE_STATEMENTS` rather than with `MAX_COST`. The default amount of memory allotted per query on statement-based queues is: `MEMORY_LIMIT / ACTIVE_STATEMENTS`. The default amount of memory allotted per query on cost-based queues is: `MEMORY_LIMIT * (query_cost / MAX_COST)`.

The default memory allotment can be overridden on a per-query basis using the `statement_mem` server configuration parameter, provided that `MEMORY_LIMIT` or `max_statement_mem` is not exceeded. For example, to allocate more memory to a particular query:

```
=> SET statement_mem='2GB';
=> SELECT * FROM my_big_table WHERE column='value' ORDER BY id;
=> RESET statement_mem;
```

The `MEMORY_LIMIT` value for all of your resource queues should not exceed the amount of physical memory of a segment host. If workloads are staggered over multiple queues, memory allocations can be oversubscribed. However, queries can be cancelled during execution if the segment host memory limit specified in `gp_vmem_protect_limit` is exceeded.

For information about `statement_mem`, `max_statement`, and `gp_vmem_protect_limit`, see *Server Configuration Parameters*.

Parameters

name

The name of the resource queue.

ACTIVE_STATEMENTS *integer*

Resource queues with an `ACTIVE_STATEMENTS` threshold limit the number of queries that can be executed by roles assigned to that queue. It controls the number of active queries that are allowed to run at the same time. The value for `ACTIVE_STATEMENTS` should be an integer greater than 0.

MEMORY_LIMIT *'memory_units'*

Sets the total memory quota for all statements submitted from users in this resource queue. Memory units can be specified in kB, MB or GB. The minimum memory quota for a resource queue is 10MB. There is no maximum, however the upper boundary at query execution time is limited by the physical memory of a segment host. The default is no limit (`-1`).

MAX_COST *float*

Resource queues with a `MAX_COST` threshold set a maximum limit on the total cost of queries that can be executed by roles assigned to that queue. Cost is measured in the *estimated total cost* for the query as determined by the Greenplum Database query optimizer (as shown in the `EXPLAIN` output for a query). Therefore, an administrator must be familiar with the queries typically executed on the system in order to set an appropriate cost threshold for a queue. Cost is measured in units of disk page fetches; 1.0 equals one sequential disk page read. The value for `MAX_COST` is specified as a floating point number (for example 100.0) or can also be specified as an exponent (for example 1e+2).

COST_OVERCOMMIT *boolean*

If a resource queue is limited based on `MAX_COST`, then the administrator can allow `COST_OVERCOMMIT` (the default). This means that a query that exceeds the allowed cost threshold will be allowed to run but only when the system is idle. If `COST_OVERCOMMIT=FALSE` is specified, queries that exceed the cost limit will always be rejected and never allowed to run.

MIN_COST *float*

The minimum query cost limit of what is considered a small query. Queries with a cost under this limit will not be queued and run immediately. Cost is measured in the *estimated total cost* for the query as determined by the Greenplum Database query planner (as shown in the `EXPLAIN` output for a query). Therefore, an administrator must be familiar with the queries typically executed on the system in order to set an appropriate cost for what is considered a small query. Cost is measured in units of disk page fetches; 1.0 equals one sequential disk page read. The value for `MIN_COST` is specified as a floating point number (for example 100.0) or can also be specified as an exponent (for example 1e+2).

PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX}

Sets the priority of queries associated with a resource queue. Queries or statements in queues with higher priority levels will receive a larger share of available CPU resources in case of contention. Queries in low-priority queues may be delayed while higher priority queries are executed. If no priority is specified, queries associated with the queue have a priority of `MEDIUM`.

Notes

Use the `gp_toolkit.gp_resqueue_status` system view to see the limit settings and current status of a resource queue:

```
SELECT * from gp_toolkit.gp_resqueue_status WHERE
rsqname='queue_name';
```

There is also another system view named `pg_stat_resqueues` which shows statistical metrics for a resource queue over time. To use this view, however, you must enable the `stats_queue_level` server configuration parameter. See "Managing Workload and Resources" in the *Greenplum Database Administrator Guide* for more information about using resource queues.

`CREATE RESOURCE QUEUE` cannot be run within a transaction.

Examples

Create a resource queue with an active query limit of 20:

```
CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=20);
```

Create a resource queue with an active query limit of 20 and a total memory limit of 2000MB (each query will be allocated 100MB of segment host memory at execution time):

```
CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=20,
MEMORY_LIMIT='2000MB');
```

Create a resource queue with a query cost limit of 3000.0:

```
CREATE RESOURCE QUEUE myqueue WITH (MAX_COST=3000.0);
```

Create a resource queue with a query cost limit of 3^{10} (or 30000000000.0) and do not allow overcommit. Allow small queries with a cost under 500 to run immediately:

```
CREATE RESOURCE QUEUE myqueue WITH (MAX_COST=3e+10,
COST_OVERCOMMIT=FALSE, MIN_COST=500.0);
```

Create a resource queue with both an active query limit and a query cost limit:

```
CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=30,
MAX_COST=5000.00);
```

Create a resource queue with an active query limit of 5 and a maximum priority setting:

```
CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=5,
PRIORITY=MAX);
```

Compatibility

`CREATE RESOURCE QUEUE` is a Greenplum Database extension. There is no provision for resource queues or workload management in the SQL standard.

See Also

ALTER ROLE, *CREATE ROLE*, *ALTER RESOURCE QUEUE*, *DROP RESOURCE QUEUE*

CREATE ROLE

Defines a new database role (user or group).

Synopsis

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

where *option* can be:

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEEXTTABLE | NOCREATEEXTTABLE
      [ ( attribute='value'[, ...] ) ]
           where attributes and value are:
           type='readable'|'writable'
           protocol='gpfdist'|'http'
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE rolename [, ...]
    | ROLE rolename [, ...]
    | ADMIN rolename [, ...]
    | RESOURCE QUEUE queue_name
    | [ DENY deny_point ]
    | [ DENY BETWEEN deny_point AND deny_point]
```

Description

`CREATE ROLE` adds a new role to a Greenplum Database system. A role is an entity that can own database objects and have database privileges. A role can be considered a user, a group, or both depending on how it is used. You must have `CREATEROLE` privilege or be a database superuser to use this command.

Note that roles are defined at the system-level and are valid for all databases in your Greenplum Database system.

Parameters

name

The name of the new role.

SUPERUSER
NOSUPERUSER

If `SUPERUSER` is specified, the role being defined will be a superuser, who can override all access restrictions within the database. Superuser status is dangerous and should be used only when really needed. You must yourself be a superuser to create a new superuser. `NOSUPERUSER` is the default.

CREATEDB
NOCREATEDB

If `CREATEDB` is specified, the role being defined will be allowed to create new databases. `NOCREATEDB` (the default) will deny a role the ability to create databases.

CREATEROLE
NOCREATEROLE

If `CREATEDB` is specified, the role being defined will be allowed to create new roles, alter other roles, and drop other roles. `NOCREATEROLE` (the default) will deny a role the ability to create roles or modify roles other than their own.

CREATEEXTTABLE
NOCREATEEXTTABLE

If `CREATEEXTTABLE` is specified, the role being defined is allowed to create external tables. The default `type` is `readable` and the default `protocol` is `gpfdist` if not specified. `NOCREATEEXTTABLE` (the default) denies the role the ability to create external tables. Note that external tables that use the `file` or `execute` protocols can only be created by superusers.

INHERIT
NOINHERIT

If specified, `INHERIT` (the default) allows the role to use whatever database privileges have been granted to all roles it is directly or indirectly a member of. With `NOINHERIT`, membership in another role only grants the ability to `SET ROLE` to that other role.

LOGIN
NOLOGIN

If specified, `LOGIN` allows a role to log in to a database. A role having the `LOGIN` attribute can be thought of as a user. Roles with `NOLOGIN` (the default) are useful for managing database privileges, and can be thought of as groups.

CONNECTION LIMIT *connlimit*

The number maximum of concurrent connections this role can make. The default of -1 means there is no limitation.

PASSWORD *password*

Sets the user password for roles with the `LOGIN` attribute. If you do not plan to use password authentication you can omit this option. If no password is specified, the password will be set to null and password authentication will always fail for that user. A null password can optionally be written explicitly as `PASSWORD NULL`.

ENCRYPTED
UNENCRYPTED

These key words control whether the password is stored encrypted in the system catalogs. (If neither is specified, the default behavior is determined by the configuration parameter *password_encryption*.) If the presented password string is already in MD5-encrypted format, then it is stored encrypted as-is, regardless of whether `ENCRYPTED` or `UNENCRYPTED` is specified (since the system cannot decrypt the specified encrypted password string). This allows reloading of encrypted passwords during dump/restore.

Note that older clients may lack support for the MD5 authentication mechanism that is needed to work with passwords that are stored encrypted.

VALID UNTIL *'timestamp'*

The VALID UNTIL clause sets a date and time after which the role's password is no longer valid. If this clause is omitted the password will never expire.

IN ROLE *rolename*

Adds the new role as a member of the named roles. Note that there is no option to add the new role as an administrator; use a separate `GRANT` command to do that.

ROLE *rolename*

Adds the named roles as members of this role, making this new role a group.

ADMIN *rolename*

The `ADMIN` clause is like `ROLE`, but the named roles are added to the new role `WITH ADMIN OPTION`, giving them the right to grant membership in this role to others.

RESOURCE QUEUE *queue_name*

The name of the resource queue to which the new user-level role is to be assigned. Only roles with `LOGIN` privilege can be assigned to a resource queue. The special keyword `NONE` means that the role is assigned to the default resource queue. A role can only belong to one resource queue.

DENY *deny_point*
DENY BETWEEN *deny_point* AND *deny_point*

The `DENY` and `DENY BETWEEN` keywords set time-based constraints that are enforced at login. `DENY` sets a day or a day and time to deny access. `DENY BETWEEN` sets an interval during which access is denied. Both use the parameter *deny_point* that has the following format:

```
DAY day [ TIME 'time' ]
```

The two parts of the `deny_point` parameter use the following formats:

For `day`:

```
{'Sunday' | 'Monday' | 'Tuesday' |'Wednesday' | 'Thursday' | 'Friday' |
'Saturday' | 0-6 }
```

For `time`:

```
{ 00-23 : 00-59 | 01-12 : 00-59 { AM | PM }}
```

The `DENY BETWEEN` clause uses two *deny_point* parameters:

```
DENY BETWEEN deny_point AND deny_point
```

For more information and examples about time-based constraints, see "Managing Roles and Privileges" in the *Greenplum Database Administrator Guide*.

Notes

The preferred way to add and remove role members (manage groups) is to use *`GRANT`* and *`REVOKE`*.

The `VALID UNTIL` clause defines an expiration time for a password only, not for the role. The expiration time is not enforced when logging in using a non-password-based authentication method.

The `INHERIT` attribute governs inheritance of grantable privileges (access privileges for database objects and role memberships). It does not apply to the special role attributes set by `CREATE ROLE` and `ALTER ROLE`. For example, being a member of a role with `CREATEDB` privilege does not immediately grant the ability to create databases, even if `INHERIT` is set. These privileges/attributes are never inherited: `SUPERUSER`, `CREATEDB`, `CREATEROLE`, `CREATEEXTTABLE`, `LOGIN`, and `RESOURCE QUEUE`. The attributes must be set on each user-level role.

The `INHERIT` attribute is the default for reasons of backwards compatibility. In prior releases of Greenplum Database, users always had access to all privileges of groups they were members of. However, `NOINHERIT` provides a closer match to the semantics specified in the SQL standard.

Be careful with the `CREATEROLE` privilege. There is no concept of inheritance for the privileges of a `CREATEROLE`-role. That means that even if a role does not have a certain privilege but is allowed to create other roles, it can easily create another role with different privileges than its own (except for creating roles with superuser privileges). For example, if a role has the `CREATEROLE` privilege but not the `CREATEDB` privilege, it can create a new role with the `CREATEDB` privilege. Therefore, regard roles that have the `CREATEROLE` privilege as almost-superuser-roles.

The `CONNECTION LIMIT` option is never enforced for superusers.

Caution must be exercised when specifying an unencrypted password with this command. The password will be transmitted to the server in clear-text, and it might also be logged in the client's command history or the server log. The client program `createuser`, however, transmits the password encrypted. Also, psql contains a command `\password` that can be used to safely change the password later.

Examples

Create a role that can log in, but don't give it a password:

```
CREATE ROLE jonathan LOGIN;
```

Create a role that belongs to a resource queue:

```
CREATE ROLE jonathan LOGIN RESOURCE QUEUE poweruser;
```

Create a role with a password that is valid until the end of 2009 (`CREATE USER` is the same as `CREATE ROLE` except that it implies `LOGIN`):

```
CREATE USER joelle WITH PASSWORD 'jw8s0F4' VALID UNTIL '2010-01-01';
```

Create a role that can create databases and manage other roles:

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

Create a role that does not allow login access on Sundays:

```
CREATE ROLE user3 DENY DAY 'Sunday';
```

Compatibility

The SQL standard defines the concepts of users and roles, but it regards them as distinct concepts and leaves all commands defining users to be specified by the database implementation. In Greenplum Database users and roles are unified into a single type of object. Roles therefore have many more optional attributes than they do in the standard.

`CREATE ROLE` is in the SQL standard, but the standard only requires the syntax:

```
CREATE ROLE name [WITH ADMIN rolename]
```

Allowing multiple initial administrators, and all the other options of `CREATE ROLE`, are Greenplum Database extensions.

The behavior specified by the SQL standard is most closely approximated by giving users the `NOINHERIT` attribute, while roles are given the `INHERIT` attribute.

See Also

SET ROLE, *ALTER ROLE*, *DROP ROLE*, *GRANT*, *REVOKE*, *CREATE RESOURCE QUEUE*

CREATE RULE

Defines a new rewrite rule.

Synopsis

```
CREATE [OR REPLACE] RULE name AS ON event
  TO table [WHERE condition]
  DO [ALSO | INSTEAD] { NOTHING | command | (command; command
  ...) }
```

Description

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

The Greenplum Database rule system allows one to define an alternate action to be performed on insertions, updates, or deletions in database tables. A rule causes additional or alternate commands to be executed when a given command on a given table is executed. Rules can be used on views as well. It is important to realize that a rule is really a command transformation mechanism, or command macro. The transformation happens before the execution of the commands starts. It does not operate independently for each physical row as does a trigger.

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

You can create the illusion of an updatable view by defining `ON INSERT`, `ON UPDATE`, and `ON DELETE` rules to replace update actions on the view with appropriate updates on other tables.

There is a catch if you try to use conditional rules for view updates: there must be an unconditional `INSTEAD` rule for each action you wish to allow on the view. If the rule is conditional, or is not `INSTEAD`, then the system will still reject attempts to perform the update action, because it thinks it might end up trying to perform the action on the dummy table of the view in some cases. If you want to handle all the useful cases in conditional rules, add an unconditional `DO INSTEAD NOTHING` rule to ensure that the system understands it will never be called on to update the dummy table. Then make the conditional rules non-`INSTEAD`; in the cases where they are applied, they add to the default `INSTEAD NOTHING` action. (This method does not currently work to support `RETURNING` queries, however.)

Parameters

name

The name of a rule to create. This must be distinct from the name of any other rule for the same table. Multiple rules on the same table and same event type are applied in alphabetical name order.

event

The event is one of `SELECT`, `INSERT`, `UPDATE`, or `DELETE`.

table

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

condition

Any SQL conditional expression (returning boolean). The condition expression may not refer to any tables except `NEW` and `OLD`, and may not contain aggregate functions. `NEW` and `OLD` refer to values in the referenced table. `NEW` is valid in `ON INSERT` and `ON UPDATE` rules to refer to the new row being inserted or updated. `OLD` is valid in `ON UPDATE` and `ON DELETE` rules to refer to the existing row being updated or deleted.

INSTEAD

`INSTEAD` indicates that the commands should be executed instead of the original command.

ALSO

`ALSO` indicates that the commands should be executed in addition to the original command. If neither `ALSO` nor `INSTEAD` is specified, `ALSO` is the default.

command

The command or commands that make up the rule action. Valid commands are `SELECT`, `INSERT`, `UPDATE`, or `DELETE`. The special table names `NEW` and `OLD` may be used to refer to values in the referenced table. `NEW` is valid in `ON INSERT` and `ONUPDATE` rules to refer to the new row being inserted or updated. `OLD` is valid in `ON UPDATE` and `ON DELETE` rules to refer to the existing row being updated or deleted.

Notes

You must be the owner of a table to create or change rules for it.

It is very important to take care to avoid circular rules. Recursive rules are not validated at rule create time, but will report an error at execution time.

Examples

Create a rule that inserts rows into the child table `b2001` when a user tries to insert into the partitioned parent table `rank`:

```
CREATE RULE b2001 AS ON INSERT TO rank WHERE gender='M' and
year='2001' DO INSTEAD INSERT INTO b2001 VALUES (NEW.id,
NEW.rank, NEW.year, NEW.gender, NEW.count);
```

Compatibility

`CREATE RULE` is a Greenplum Database language extension, as is the entire query rewrite system.

See Also

DROP RULE, *CREATE TABLE*, *CREATE VIEW*

CREATE SCHEMA

Defines a new schema.

Synopsis

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

CREATE SCHEMA AUTHORIZATION rolename [schema_element [ ... ]]
```

Description

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

A schema is essentially a namespace: it contains named objects (tables, data types, functions, and operators) whose names may duplicate those of other objects existing in other schemas. Named objects are accessed either by qualifying their names with the schema name as a prefix, or by setting a search path that includes the desired schema(s). A `CREATE` command specifying an unqualified object name creates the object in the current schema (the one at the front of the search path, which can be determined with the function `current_schema`).

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

Parameters

schema_name

The name of a schema to be created. If this is omitted, the user name is used as the schema name. The name cannot begin with `pg_`, as such names are reserved for system catalog schemas.

rolename

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

schema_element

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

Note: Greenplum Database does not support triggers.

Notes

To create a schema, the invoking user must have the `CREATE` privilege for the current database or be a superuser.

Examples

Create a schema:

```
CREATE SCHEMA myschema;
```

Create a schema for role `joe` (the schema will also be named `joe`):

```
CREATE SCHEMA AUTHORIZATION joe;
```

Compatibility

The SQL standard allows a `DEFAULT CHARACTER SET` clause in `CREATE SCHEMA`, as well as more subcommand types than are presently accepted by Greenplum Database.

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

According to the SQL standard, the owner of a schema always owns all objects within it. Greenplum Database allows schemas to contain objects owned by users other than the schema owner. This can happen only if the schema owner grants the `CREATE` privilege on the schema to someone else.

See Also

ALTER SCHEMA, *DROP SCHEMA*

CREATE SEQUENCE

Defines a new sequence generator.

Synopsis

```
CREATE [TEMPORARY | TEMP] SEQUENCE name
       [INCREMENT [BY] value]
       [MINVALUE minvalue | NO MINVALUE]
       [MAXVALUE maxvalue | NO MAXVALUE]
       [START [ WITH ] start]
       [CACHE cache]
       [[NO] CYCLE]
       [OWNED BY { table.column | NONE }]
```

Description

`CREATE SEQUENCE` creates a new sequence number generator. This involves creating and initializing a new special single-row table. The generator will be owned by the user issuing the command.

If a schema name is given, then the sequence is created in the specified schema. Otherwise it is created in the current schema. Temporary sequences exist in a special schema, so a schema name may not be given when creating a temporary sequence. The sequence name must be distinct from the name of any other sequence, table, index, or view in the same schema.

After a sequence is created, you use the `nextval` function to operate on the sequence. For example, to insert a row into a table that gets the next value of a sequence:

```
INSERT INTO distributors VALUES (nextval('myserial'), 'acme');
```

You can also use the function `setval` to operate on a sequence, but only for queries that do not operate on distributed data. For example, the following query is allowed because it resets the sequence counter value for the sequence generator process on the master:

```
SELECT setval('myserial', 201);
```

But the following query will be rejected in Greenplum Database because it operates on distributed data:

```
INSERT INTO product VALUES (setval('myserial', 201), 'gizmo');
```

In a regular (non-distributed) database, functions that operate on the sequence go to the local sequence table to get values as they are needed. In Greenplum Database, however, keep in mind that each segment is its own distinct database process. Therefore the segments need a single point of truth to go for sequence values so that all segments get incremented correctly and the sequence moves forward in the right order. A sequence server process runs on the master and is the point-of-truth for a sequence in a Greenplum distributed database. Segments get sequence values at runtime from the master.

Because of this distributed sequence design, there are some limitations on the functions that operate on a sequence in Greenplum Database:

- `lastval` and `currval` functions are not supported.
- `setval` can only be used to set the value of the sequence generator on the master, it cannot be used in subqueries to update records on distributed table data.
- `nextval` sometimes grabs a block of values from the master for a segment to use, depending on the query. So values may sometimes be skipped in the sequence if all of the block turns out not to be needed at the segment level. Note that a regular PostgreSQL database does this too, so this is not something unique to Greenplum Database.

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

```
SELECT * FROM sequence_name;
```

to examine the parameters and current state of a sequence. In particular, the *last_value* field of the sequence shows the last value allocated by any session.

Parameters

TEMPORARY | TEMP

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

name

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

increment

Specifies which value is added to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is 1.

minvalue
NO MINVALUE

Determines the minimum value a sequence can generate. If this clause is not supplied or `NO MINVALUE` is specified, then defaults will be used. The defaults are 1 and -263-1 for ascending and descending sequences, respectively.

maxvalue
NO MAXVALUE

Determines the maximum value for the sequence. If this clause is not supplied or `NO MAXVALUE` is specified, then default values will be used. The defaults are 263-1 and -1 for ascending and descending sequences, respectively.

start

Allows the sequence to begin anywhere. The default starting value is *minvalue* for ascending sequences and *maxvalue* for descending ones.

cache

Specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum (and default) value is 1 (no cache).

CYCLE
NO CYCLE

Allows the sequence to wrap around when the *maxvalue* (for ascending) or *minvalue* (for descending) has been reached. If the limit is reached, the next number generated will be the *minvalue* (for ascending) or *maxvalue* (for descending). If `NO CYCLE` is specified, any calls to `nextval` after the sequence has reached its maximum value will return an error. If not specified, `NO CYCLE` is the default.

OWNED BY *table.column*
OWNED BY NONE

Causes the sequence to be associated with a specific table column, such that if that column (or its whole table) is dropped, the sequence will be automatically dropped as well. The specified table must have the same owner and be in the same schema as the sequence. `OWNED BY NONE`, the default, specifies that there is no such association.

Notes

Sequences are based on bigint arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807).

Although multiple sessions are guaranteed to allocate distinct sequence values, the values may be generated out of sequence when all the sessions are considered. For example, session A might reserve values 1..10 and return `nextval=1`, then session B might reserve values 11..20 and return `nextval=11` before session A has generated nextval=2. Thus, 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 session, whether or not it has yet been returned by `nextval`.

Examples

Create a sequence named *myseq*:

```
CREATE SEQUENCE myseq START 101;
```

Insert a row into a table that gets the next value:

```
INSERT INTO distributors VALUES (nextval('myseq'), 'acme');
```

Reset the sequence counter value on the master:

```
SELECT setval('myseq', 201);
```

Illegal use of `setval` in Greenplum Database (setting sequence values on distributed data):

```
INSERT INTO product VALUES (setval('myseq', 201), 'gizmo');
```

Compatibility

`CREATE SEQUENCE` conforms to the SQL standard, with the following exceptions:

- The `AS` *`data_type`* expression specified in the SQL standard is not supported.
- Obtaining the next value is done using the `nextval()` function instead of the `NEXT VALUE FOR` expression specified in the SQL standard.
- The `OWNED BY` clause is a Greenplum Database extension.

See Also

ALTER SEQUENCE, *DROP SEQUENCE*

CREATE TABLE

Defines a new table.

Note: Referential integrity syntax (foreign key constraints) is accepted but not enforced.

Synopsis

```
CREATE [[GLOBAL | LOCAL] {TEMPORARY | TEMP}] TABLE table_name (
[ { column_name data_type [ DEFAULT default_expr ]
   [column_constraint [ ... ]
[ ENCODING ( storage_directive [,...] ) ]
]
   | table_constraint
   | LIKE other_table [{INCLUDING | EXCLUDING}
                      {DEFAULTS | CONSTRAINTS}] ...}
   [, ... ] ]
   )
   [ INHERITS ( parent_table [, ... ] ) ]
   [ WITH ( storage_parameter=value [, ... ] )
   [ ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP} ]
   [ TABLESPACE tablespace ]
   [ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]
   [ PARTITION BY partition_type (column)
       [ SUBPARTITION BY partition_type (column) ]
          [ SUBPARTITION TEMPLATE ( template_spec ) ]
       [...]
    ( partition_spec )
        | [ SUBPARTITION BY partition_type (column) ]
          [...]
    ( partition_spec
      [ ( subpartition_spec
           [(...)]
         ) ]
    )
```

where *column_constraint* is:

```
   [CONSTRAINT constraint_name]
   NOT NULL | NULL
   | UNIQUE [USING INDEX TABLESPACE tablespace]
            [WITH ( FILLFACTOR = value )]
   | PRIMARY KEY [USING INDEX TABLESPACE tablespace]
                 [WITH ( FILLFACTOR = value )]
   | CHECK ( expression )
   | REFERENCES table_name [ ( column_name [, ... ] ) ]
            [ key_match_type ]
            [ key_action ]
```

where *storage_directive* for a column is:

```
   COMPRESSTYPE={ZLIB | QUICKLZ | RLE_TYPE | NONE}
    [COMPRESSLEVEL={0-9} ]
    [BLOCKSIZE={8192-2097152} ]
```

where *storage_parameter* for the table is:

```
   APPENDONLY={TRUE|FALSE}
   BLOCKSIZE={8192-2097152}
   ORIENTATION={COLUMN|ROW}
   CHECKSUM={TRUE|FALSE}
   COMPRESSTYPE={ZLIB|QUICKLZ|RLE_TYPE|NONE}
   COMPRESSLEVEL={0-9}
   FILLFACTOR={10-100}
```

```
    OIDS[=TRUE|FALSE]
```

and *table_constraint* is:

```
    [CONSTRAINT constraint_name]
    UNIQUE ( column_name [, ... ] )
           [USING INDEX TABLESPACE tablespace]
           [WITH ( FILLFACTOR=value )]
    | PRIMARY KEY ( column_name [, ... ] )
                  [USING INDEX TABLESPACE tablespace]
                  [WITH ( FILLFACTOR=value )]
    | CHECK ( expression )
    | FOREIGN KEY ( column_name [, ... ] )
             REFERENCES table_name [ ( column_name [, ... ] ) ]
             [ key_match_type ]
             [ key_action ]
             [ key_checking_mode ]
```

where *key_match_type* is:

```
     MATCH FULL
   | SIMPLE
```

where *key_action* is:

```
     ON DELETE
   | ON UPDATE
   | NO ACTION
   | RESTRICT
   | CASCADE
   | SET NULL
   | SET DEFAULT
```

where *key_checking_mode* is:

```
     DEFERRABLE
   | NOT DEFERRABLE
   | INITIALLY DEFERRED
   | INITIALLY IMMEDIATE
```

where *partition_type* is:

```
     LIST
   | RANGE
```

where *partition_specification* is:

```
partition_element [, ...]
```

and *partition_element* is:

```
    DEFAULT PARTITION name
   | [PARTITION name] VALUES (list_value [,...] )
   | [PARTITION name]
     START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
     [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
     [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
   | [PARTITION name]
     END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
     [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
[ TABLESPACE tablespace ]
```

where *subpartition_spec* or *template_spec* is:

```
subpartition_element [, ...]
```

and *subpartition_element* is:

```
   DEFAULT SUBPARTITION name
  | [SUBPARTITION name] VALUES (list_value [,...] )
  | [SUBPARTITION name]
     START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
     [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
     [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
  | [SUBPARTITION name]
     END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
     [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
[ TABLESPACE tablespace ]
```

where *storage_parameter* for a partition is:

```
   APPENDONLY={TRUE|FALSE}
   BLOCKSIZE={8192-2097152}
   ORIENTATION={COLUMN|ROW}
   CHECKSUM={TRUE|FALSE}
   COMPRESSTYPE={ZLIB|QUICKLZ|RLE_TYPE|NONE}
   COMPRESSLEVEL={1-9}
   FILLFACTOR={10-100}
   OIDS[=TRUE|FALSE]
```

Description

`CREATE TABLE` creates an initially empty table in the current database. The user who issues the command owns the table.

If you specify a schema name, Greenplum creates the table in the specified schema. Otherwise Greenplum creates the table in the current schema. Temporary tables exist in a special schema, so you cannot specify a schema name when creating a temporary table. Table names must be distinct from the name of any other table, external table, sequence, index, or view in the same schema.

The optional constraint clauses specify conditions that new or updated rows must satisfy for an insert or update operation to succeed. A constraint is an SQL object that helps define the set of valid values in the table in various ways. Constraints apply to tables, not to partitions. You cannot add a constraint to a partition or subpartition.

Referential integrity constraints (foreign keys) are accepted but not enforced. The information is kept in the system catalogs but is otherwise ignored.

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

When creating a table, there is an additional clause to declare the Greenplum Database distribution policy. If a `DISTRIBUTED BY` or `DISTRIBUTED RANDOMLY` clause is not supplied, then Greenplum assigns a hash distribution policy to the table using either the `PRIMARY KEY` (if the table has one) or the first column of the table as the distribution key. Columns of geometric or user-defined data types are not eligible as Greenplum distribution key columns. If a table does not have a column of an eligible data type, the rows are distributed based on a round-robin or random distribution. To ensure an even distribution of data in your Greenplum Database system, you want to choose a distribution key that is unique for each record, or if that is not possible, then choose `DISTRIBUTED RANDOMLY`.

The `PARTITION BY` clause allows you to divide the table into multiple sub-tables (or parts) that, taken together, make up the parent table and share its schema. Though the sub-tables exist as independent tables, the Greenplum Database restricts their use in important ways. Internally, partitioning is implemented as a special form of inheritance. Each child table partition is created with a distinct `CHECK` constraint which limits the data the table can contain, based on some defining criteria. The `CHECK` constraints are also used by the query optimizer to determine which table partitions to scan in order to satisfy a given query predicate. These partition constraints are managed automatically by the Greenplum Database.

Parameters

GLOBAL | LOCAL

These keywords are present for SQL standard compatibility, but have no effect in Greenplum Database.

TEMPORARY | TEMP

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

table_name

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

column_name

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

data_type

The data type of the column. This may include array specifiers.

For table columns that contain textual data, Pivotal recommends specifying the data type `VARCHAR` or `TEXT`. Specifying the data type `CHAR` is not recommended. In Greenplum Database, the data types `VARCHAR` or `TEXT` handles padding added to the data (space characters added after the last non-space character) as significant characters, the data type `CHAR` does not. See *Notes*.

DEFAULT *default_expr*

The `DEFAULT` clause assigns a default data value for the column whose column definition it appears within. The value is any variable-free expression (subqueries and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column. The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a column, then the default is null.

ENCODING (*storage_directive* [, ...])

For a column, the optional `ENCODING` clause specifies the type of compression and block size for the column data. See *storage_options* for `COMPRESSTYPE`, `COMPRESSLEVEL`, and `BLOCKSIZE` values.

The clause is valid only for append-optimized, column-oriented tables.

Column compression settings are inherited from the table level to the partition level to the subpartition level. The lowest-level settings have priority.

INHERITS

The optional `INHERITS` clause specifies a list of tables from which the new table automatically inherits all columns. Use of `INHERITS` creates a persistent relationship between the new child table and its parent table(s). Schema modifications to the parent(s) normally propagate to children as well, and by default the data of the child table is included in scans of the parent(s).

In Greenplum Database, the `INHERITS` clause is not used when creating partitioned tables. Although the concept of inheritance is used in partition hierarchies, the inheritance structure of a partitioned table is created using the *`PARTITION BY`* clause.

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

LIKE *other_table* [{INCLUDING | EXCLUDING} {DEFAULTS | CONSTRAINTS}]

The `LIKE` clause specifies a table from which the new table automatically copies all column names, data types, not-null constraints, and distribution policy. Storage properties like append-optimized or partition structure are not copied. Unlike `INHERITS`, the new table and original table are completely decoupled after creation is complete.

Default expressions for the copied column definitions will only be copied if `INCLUDING DEFAULTS` is specified. The default behavior is to exclude default expressions, resulting in the copied columns in the new table having null defaults.

Not-null constraints are always copied to the new table. `CHECK` constraints will only be copied if `INCLUDING CONSTRAINTS` is specified; other types of constraints will *never* be copied. Also, no distinction is made between column constraints and table constraints — when constraints are requested, all check constraints are copied.

Note also that unlike `INHERITS`, copied columns and constraints are not merged with similarly named columns and constraints. If the same name is specified explicitly or in another `LIKE` clause an error is signalled.

CONSTRAINT *constraint_name*

An optional name for a column or table constraint. If the constraint is violated, the constraint name is present in error messages, so constraint names like *column must be positive* can be used to communicate helpful constraint information to client applications. (Double-quotes are needed to specify constraint names that contain spaces.) If a constraint name is not specified, the system generates a name.

> **Note:** The specified *constraint_name* is used for the constraint, but a system-generated unique name is used for the index name. In some prior releases, the provided name was used for both the constraint name and the index name.

NULL | NOT NULL

Specifies if the column is or is not allowed to contain null values. `NULL` is the default.

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

The `UNIQUE` constraint specifies that a group of one or more columns of a table may contain only unique values. The behavior of the unique table constraint is the same as that for column constraints, with the additional capability to span multiple columns. For the purpose of a unique constraint, null values are not considered equal. The column(s) that are unique must contain all the columns of the Greenplum distribution key. In addition, the `<key>` must contain all the columns in the partition key if the table is partitioned. Note that a `<key>` constraint in a partitioned table is not the same as a simple `UNIQUE INDEX`.

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

The primary key constraint specifies that a column or columns of a table may contain only unique (non-duplicate), non-null values. Technically, `PRIMARY KEY` is merely a combination of `UNIQUE` and `NOT NULL`, but identifying a set of columns as primary key also provides metadata about the design of the schema, as a primary key implies that other tables may rely on this set of columns as a unique identifier for rows. For a table to have a primary key, it must be hash distributed (not randomly distributed), and the primary key The column(s) that are unique must contain all the columns of the Greenplum distribution key. In addition, the `<key>` must contain all the columns in the partition key if the table is partitioned. Note that a `<key>` constraint in a partitioned table is not the same as a simple `UNIQUE INDEX`.

CHECK (*expression*)

The `CHECK` clause specifies an expression producing a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to `TRUE` or `UNKNOWN` succeed. Should any row of an insert or update operation produce a `FALSE` result an error exception is raised and the insert or update does not alter the database. A check constraint specified as a column constraint should reference that column's value only, while an expression appearing in a table constraint may reference multiple columns. `CHECK` expressions cannot contain subqueries nor refer to variables other than columns of the current row.

REFERENCES *table_name* [(*column_name* [, ...])]
[*key_match_type*] [*key_action*]
FOREIGN KEY (*column_name* [, ...])
REFERENCES *table_name* [(*column_name* [, ...])
[*key_match_type*] [*key_action* [*key_checking_mode*]

The `REFERENCES` and `FOREIGN KEY` clauses specify referential integrity constraints (foreign key constraints). Greenplum accepts referential integrity constraints as specified in PostgreSQL syntax but does not enforce them. See the PostgreSQL documentation for information about referential integrity constraints.

WITH (*storage_option*=*value*)

The `WITH` clause can be used to set storage options for the table or its indexes. Note that you can also set storage parameters on a particular partition or subpartition by declaring the `WITH` clause in the partition specification. The lowest-level settings have priority.

The defaults for some of the table storage options can be specified with the server configuration parameter `gp_default_storage_options`. For information about setting default storage options, see *Notes*.

The following storage options are available:

APPENDONLY — Set to `TRUE` to create the table as an append-optimized table. If `FALSE` or not declared, the table will be created as a regular heap-storage table.

BLOCKSIZE — Set to the size, in bytes for each block in a table. The `BLOCKSIZE` must be between 8192 and 2097152 bytes, and be a multiple of 8192. The default is 32768.

ORIENTATION — Set to `column` for column-oriented storage, or `row` (the default) for row-oriented storage. This option is only valid if `APPENDONLY=TRUE`. Heap-storage tables can only be row-oriented.

CHECKSUM — This option is valid only for append-optimized tables (`APPENDONLY=TRUE`). The value `TRUE` is the default and enables CRC checksum validation for append-optimized tables. The checksum is calculated during block creation and is stored on disk. Checksum validation is performed during block reads. If the checksum calculated during the read does not match the stored checksum, the transaction is aborted. If you set the value to `FALSE` to disable checksum validation, checking the table data for on-disk corruption will not be performed.

COMPRESSTYPE — Set to `ZLIB` (the default), `RLE-TYPE`, or `QUICKLZ` to specify the type of compression used. The value `NONE`disables compression. QuickLZ uses less CPU power and compresses data faster at a lower compression ratio than zlib. Conversely, zlib provides more compact compression ratios at lower speeds. This option is only valid if `APPENDONLY=TRUE`.

The value `RLE_TYPE` is supported only if `ORIENTATION =column` is specified, Greenplum Database uses the run-length encoding (RLE) compression algorithm. RLE compresses data better than the zlib or QuickLZ compression algorithm when the same data value occurs in many consecutive rows.

For columns of type `BIGINT`, `INTEGER`, `DATE`, `TIME`, or `TIMESTAMP`, delta compression is also applied if the `COMPRESSTYPE` option is set to `RLE-TYPE` compression. The delta compression algorithm is based on the delta between column values in consecutive rows and is designed to improve compression when data is loaded in sorted order or the compression is applied to column data that is in sorted order.

For information about using table compression, see "Choosing the Table Storage Model" in the *Greenplum Database Administrator Guide*.

COMPRESSLEVEL — For zlib compression of append-optimized tables, set to an integer value between 1 (fastest compression) to 9 (highest compression ratio). QuickLZ compression level can only be set to 1. If not declared, the default is 1. For `RLE_TYPE`, the compression level can be set an integer value between 1 (fastest compression) to 4 (highest compression ratio).

This option is valid only if `APPENDONLY=TRUE`.

FILLFACTOR — See *`CREATE INDEX`* for more information about this index storage parameter.

OIDS — Set to `OIDS=FALSE` (the default) so that rows do not have object identifiers assigned to them. Greenplum strongly recommends that you do not enable OIDS when creating a table. On large tables, such as those in a typical Greenplum Database system, using OIDs for table rows can cause wrap-around of the 32-bit OID counter. Once the counter wraps around, OIDs can no longer be assumed to be unique, which not only makes them useless to user applications, but can also cause problems in the Greenplum Database system catalog tables. In addition, excluding OIDs from a table reduces the space required to store the table on disk by 4 bytes per row, slightly improving performance. OIDS are not allowed on partitioned tables or append-optimized column-oriented tables.

ON COMMIT

The behavior of temporary tables at the end of a transaction block can be controlled using `ON COMMIT`. The three options are:

PRESERVE ROWS - No special action is taken at the ends of transactions for temporary tables. This is the default behavior.

DELETE ROWS - All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic `TRUNCATE` is done at each commit.

DROP - The temporary table will be dropped at the end of the current transaction block.

TABLESPACE *tablespace*

The name of the tablespace in which the new table is to be created. If not specified, the database's default tablespace is used.

USING INDEX TABLESPACE *tablespace*

This clause allows selection of the tablespace in which the index associated with a `UNIQUE` or `PRIMARY KEY` constraint will be created. If not specified, the database's default tablespace is used.

DISTRIBUTED BY (*column*, [...])
DISTRIBUTED RANDOMLY

Used to declare the Greenplum Database distribution policy for the table. `DISTIBUTED BY` uses hash distribution with one or more columns declared as the distribution key. For the most even data distribution, the distribution key should be the primary key of the table or a unique column (or set of columns). If that is not possible, then you may choose `DISTRIBUTED RANDOMLY`, which will send the data round-robin to the segment instances.

The Greenplum Database server configuration parameter `gp_create_table_random_default_distribution` controls the default table distribution policy if the `DISTRIBUTED BY` clause is not specified when you create a table. Greenplum Database follows these rules to create a table if a distribution policy is not specified.

If the value of the parameter is `off` (the default), Greenplum Database chooses the table distribution key based on the command. If the `LIKE` or `INHERITS` clause is specified in table creation command, the created table uses the same distribution key as the source or parent table.

If the value of the parameter is set to `on`, Greenplum Database follows these rules:

- If `PRIMARY KEY` or `UNIQUE` columns are not specified, the distribution of the table is random (`DISTRIBUTED RANDOMLY`). Table distribution is random even if the table creation command contains the `LIKE` or `INHERITS` clause.
- If `PRIMARY KEY` or `UNIQUE` columns are specified, a `DISTRIBUTED BY` clause must also be specified. If a `DISTRIBUTED BY` clause is not specified as part of the table creation command, the command fails.

For information about the parameter, see "Server Configuration Parameters."

PARTITION BY

Declares one or more columns by which to partition the table.

When creating a partitioned table, Greenplum Database creates the root partitioned table (the root partition) with the specified table name. Greenplum Database also creates a hierarchy of tables, child tables, that are the subpartitions based on the partitioning options that you specify. The Greenplum Database *pg_partition** system views contain information about the subpartition tables.

For each partition level (each hierarchy level of tables), a partitioned table can have a maximum of 32,767 partitions.

> **Note:** Greenplum Database stores partitioned table data in the leaf child tables, the lowest-level tables in the hierarchy of child tables for use by the partitioned table.

partition_type

Declares partition type: `LIST` (list of values) or `RANGE` (a numeric or date range).

partition_specification

Declares the individual partitions to create. Each partition can be defined individually or, for range partitions, you can use the `EVERY` clause (with a `START` and optional `END` clause) to define an increment pattern to use to create the individual partitions.

`DEFAULT PARTITION` ***`name`*** — Declares a default partition. When data does not match to an existing partition, it is inserted into the default partition. Partition designs that do not have a default partition will reject incoming rows that do not match to an existing partition.

`PARTITION` ***`name`*** — Declares a name to use for the partition. Partitions are created using the following naming convention: *`parentname_level#_prt_givenname`*.

`VALUES` — For list partitions, defines the value(s) that the partition will contain.

`START` — For range partitions, defines the starting range value for the partition. By default, start values are `INCLUSIVE`. For example, if you declared a start date of

'`2008-01-01`', then the partition would contain all dates greater than or equal to '`2008-01-01`'. Typically the data type of the `START` expression is the same type as the partition key column. If that is not the case, then you must explicitly cast to the intended data type.

`END` — For range partitions, defines the ending range value for the partition. By default, end values are `EXCLUSIVE`. For example, if you declared an end date of '`2008-02-01`', then the partition would contain all dates less than but not equal to '`2008-02-01`'. Typically the data type of the `END` expression is the same type as the partition key column. If that is not the case, then you must explicitly cast to the intended data type.

`EVERY` — For range partitions, defines how to increment the values from `START` to `END` to create individual partitions. Typically the data type of the `EVERY` expression is the same type as the partition key column. If that is not the case, then you must explicitly cast to the intended data type.

`WITH` — Sets the table storage options for a partition. For example, you may want older partitions to be append-optimized tables and newer partitions to be regular heap tables.

`TABLESPACE` — The name of the tablespace in which the partition is to be created.

SUBPARTITION BY

Declares one or more columns by which to subpartition the first-level partitions of the table. The format of the subpartition specification is similar to that of a partition specification described above.

SUBPARTITION TEMPLATE

Instead of declaring each subpartition definition individually for each partition, you can optionally declare a subpartition template to be used to create the subpartitions (lower level child tables). This subpartition specification would then apply to all parent partitions.

Notes

- In Greenplum Database (a Postgres-based system) the data types `VARCHAR` or `TEXT` handles padding added to the textual data (space characters added after the last non-space character) as significant characters, the data type `CHAR` does not.

  In Greenplum Database, values of type `CHAR(n)` are padded with trailing spaces to the specified width *n*. The values are stored and displayed with the spaces. However, the padding spaces are treated as semantically insignificant. When the values are distributed, the trailing spaces are disregarded. The trailing spaces are also treated as semantically insignificant when comparing two values of data type `CHAR`, and the trailing spaces are removed when converting a character value to one of the other string types.
- Using OIDs in new applications is not recommended: where possible, using a `SERIAL` or other sequence generator as the table's primary key is preferred. However, if your application does make use of OIDs to identify specific rows of a table, it is recommended to create a unique constraint on the OID column of that table, to ensure that OIDs in the table will indeed uniquely identify rows even after counter wrap-around. Avoid assuming that OIDs are unique across tables; if you need a database-wide unique identifier, use the combination of table OID and row OID for the purpose.
- Greenplum Database has some special conditions for primary key and unique constraints with regards to columns that are the *distribution key* in a Greenplum table. For a unique constraint to be enforced in Greenplum Database, the table must be hash-distributed (not `DISTRIBUTED RANDOMLY`), and the constraint columns must be the same as (or a superset of) the table's distribution key columns. Also, the distribution key must be a left-subset of the constraint columns with the columns in the correct order. For example, if the primary key is (a,b,c), the distribution key can be only one of the following: (a), (a,b), or (a,b,c).

  A primary key constraint is simply a combination of a unique constraint and a not-null constraint.

Greenplum Database automatically creates an index for each unique constraint or primary key constraint to enforce uniqueness. Thus, it is not necessary to create an index explicitly for primary key columns.

Foreign key constraints are not supported in Greenplum Database.

For inherited tables, unique constraints, primary key constraints, indexes and table privileges are *not* inherited in the current implementation.

- For append-optimized tables, `UPDATE` and `DELETE` are not allowed in a serializable transaction and will cause the transaction to abort. `CLUSTER`, `DECLARE...FORUPDATE`, and triggers are not supported with append-optimized tables.
- To insert data into a partitioned table, you specify the root partitioned table, the table created with the `CREATE TABLE` command. You also can specify a leaf child table of the partitioned table in an `INSERT` command. An error is returned if the data is not valid for the specified leaf child table. Specifying a child table that is not a leaf child table in the `INSERT` command is not supported. Execution of other DML commands such as `UPDATE` and `DELETE` on any child table of a partitioned table is not supported. These commands must be executed on the root partitioned table, the table created with the `CREATE TABLE` command.
- The default values for these table storage options can be specified with the server configuration parameter `gp_default_storage_option`.
  - `APPENDONLY`
  - `BLOCKSIZE`
  - `CHECKSUM`
  - `COMPRESSTYPE`
  - `COMPRESSLEVEL`
  - `ORIENTATION`

  The defaults can be set for a database, schema, and user. For information about setting storage options, see the server configuration parameter `gp_default_storage_options`.

  Important: The current Greenplum Database legacy optimizer allows list partitions with multi-column (composite) partition keys. The Pivotal Query Optimizer does not support composite keys, so Pivotal does not recommend using composite partition keys.

Examples

Create a table named `rank` in the schema named `baby` and distribute the data using the columns `rank`, `gender`, and `year`:

```
CREATE TABLE baby.rank (id int, rank int, year smallint,
gender char(1), count int ) DISTRIBUTED BY (rank, gender,
year);
```

Create table films and table distributors (the primary key will be used as the Greenplum distribution key by default):

```
CREATE TABLE films (
code        char(5) CONSTRAINT firstkey PRIMARY KEY,
title       varchar(40) NOT NULL,
did         integer NOT NULL,
date_prod   date,
kind        varchar(10),
len         interval hour to minute
);

CREATE TABLE distributors (
did    integer PRIMARY KEY DEFAULT nextval('serial'),
name   varchar(40) NOT NULL CHECK (name <> '')
```

```
);
```

Create a gzip-compressed, append-optimized table:

```
CREATE TABLE sales (txn_id int, qty int, date date)
WITH (appendonly=true, compresslevel=5)
DISTRIBUTED BY (txn_id);
```

Create a three level partitioned table using subpartition templates and default partitions at each level:

```
CREATE TABLE sales (id int, year int, month int, day int,
region text)
DISTRIBUTED BY (id)
PARTITION BY RANGE (year)

  SUBPARTITION BY RANGE (month)
    SUBPARTITION TEMPLATE (
       START (1) END (13) EVERY (1),
       DEFAULT SUBPARTITION other_months )

  SUBPARTITION BY LIST (region)
    SUBPARTITION TEMPLATE (
       SUBPARTITION usa VALUES ('usa'),
       SUBPARTITION europe VALUES ('europe'),
       SUBPARTITION asia VALUES ('asia'),
       DEFAULT SUBPARTITION other_regions)

( START (2002) END (2010) EVERY (1),
  DEFAULT PARTITION outlying_years);
```

Compatibility

`CREATE TABLE` command conforms to the SQL standard, with the following exceptions:

- **Temporary Tables** — In the SQL standard, temporary tables are defined just once and automatically exist (starting with empty contents) in every session that needs them. Greenplum Database instead requires each session to issue its own `CREATE TEMPORARY TABLE` command for each temporary table to be used. This allows different sessions to use the same temporary table name for different purposes, whereas the standard's approach constrains all instances of a given temporary table name to have the same table structure.

  The standard's distinction between global and local temporary tables is not in Greenplum Database. Greenplum Database will accept the `GLOBAL` and `LOCAL` keywords in a temporary table declaration, but they have no effect.

  If the `ON COMMIT` clause is omitted, the SQL standard specifies that the default behavior as `ON COMMIT DELETE ROWS`. However, the default behavior in Greenplum Database is `ON COMMIT PRESERVE ROWS`. The `ON COMMIT DROP` option does not exist in the SQL standard.
- **Column Check Constraints** — The SQL standard says that `CHECK` column constraints may only refer to the column they apply to; only `CHECK` table constraints may refer to multiple columns. Greenplum Database does not enforce this restriction; it treats column and table check constraints alike.
- **NULL Constraint** — The `NULL` constraint is a Greenplum Database extension to the SQL standard that is included for compatibility with some other database systems (and for symmetry with the `NOT NULL` constraint). Since it is the default for any column, its presence is not required.
- **Inheritance** — Multiple inheritance via the `INHERITS` clause is a Greenplum Database language extension. SQL:1999 and later define single inheritance using a different syntax and different semantics. SQL:1999-style inheritance is not yet supported by Greenplum Database.
- **Partitioning** — Table partitioning via the `PARTITION BY` clause is a Greenplum Database language extension.
- **Zero-column tables** — Greenplum Database allows a table of no columns to be created (for example, `CREATE TABLE foo();`). This is an extension from the SQL standard, which does not allow zero-column

tables. Zero-column tables are not in themselves very useful, but disallowing them creates odd special cases for `ALTER TABLE DROP COLUMN`, so Greenplum decided to ignore this spec restriction.

- **WITH clause** — The `WITH` clause is a Greenplum Database extension; neither storage parameters nor OIDs are in the standard.
- **Tablespaces** — The Greenplum Database concept of tablespaces is not part of the SQL standard. The clauses `TABLESPACE` and `USING INDEX TABLESPACE` are extensions.
- **Data Distribution** — The Greenplum Database concept of a parallel or distributed database is not part of the SQL standard. The `DISTRIBUTED` clauses are extensions.

See Also

ALTER TABLE, *DROP TABLE*, *CREATE EXTERNAL TABLE*, *CREATE TABLE AS*

CREATE TABLE AS

Defines a new table from the results of a query.

Synopsis

```
CREATE [ [GLOBAL | LOCAL] {TEMPORARY | TEMP} ] TABLE table_name
   [(column_name [, ...] )]
   [ WITH ( storage_parameter=value [, ... ] ) ]
   [ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP}]
   [TABLESPACE tablespace]
   AS query
   [DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY]
```

where *storage_parameter* is:

```
   APPENDONLY={TRUE|FALSE}
   BLOCKSIZE={8192-2097152}
   ORIENTATION={COLUMN|ROW}
   COMPRESSTYPE={ZLIB|QUICKLZ}
   COMPRESSLEVEL={1-9 | 1}
   FILLFACTOR={10-100}
   OIDS[=TRUE|FALSE]
```

Description

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

`CREATE TABLE AS` creates a new table and evaluates the query just once to fill the new table initially. The new table will not track subsequent changes to the source tables of the query.

Parameters

GLOBAL | LOCAL

These keywords are present for SQL standard compatibility, but have no effect in Greenplum Database.

TEMPORARY | TEMP

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

table_name

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

column_name

The name of a column in the new table. If column names are not provided, they are taken from the output column names of the query. If the table is created from an `EXECUTE` command, a column name list cannot be specified.

WITH (*storage_parameter=value*)

The `WITH` clause can be used to set storage options for the table or its indexes. Note that you can also set different storage parameters on a particular partition or subpartition by declaring the `WITH` clause in the partition specification. The following storage options are available:

APPENDONLY — Set to `TRUE` to create the table as an append-optimized table. If `FALSE` or not declared, the table will be created as a regular heap-storage table.

BLOCKSIZE — Set to the size, in bytes for each block in a table. The `BLOCKSIZE` must be between 8192 and 2097152 bytes, and be a multiple of 8192. The default is 32768.

ORIENTATION — Set to `column` for column-oriented storage, or `row` (the default) for row-oriented storage. This option is only valid if `APPENDONLY=TRUE`. Heap-storage tables can only be row-oriented.

COMPRESSTYPE — Set to `ZLIB` (the default) or `QUICKLZ` to specify the type of compression used. QuickLZ uses less CPU power and compresses data faster at a lower compression ratio than zlib. Conversely, zlib provides more compact compression ratios at lower speeds. This option is only valid if `APPENDONLY=TRUE`.

COMPRESSLEVEL — For zlib compression of append-optimized tables, set to a value between 1 (fastest compression) to 9 (highest compression ratio). QuickLZ compression level can only be set to 1. If not declared, the default is 1. This option is only valid if `APPENDONLY=TRUE`.

FILLFACTOR — See *`CREATE INDEX`* for more information about this index storage parameter.

OIDS — Set to `OIDS=FALSE` (the default) so that rows do not have object identifiers assigned to them. Greenplum strongly recommends that you do not enable OIDS when creating a table. On large tables, such as those in a typical Greenplum Database system, using OIDs for table rows can cause wrap-around of the 32-bit OID counter. Once the counter wraps around, OIDs can no longer be assumed to be unique, which not only makes them useless to user applications, but can also cause problems in the Greenplum Database system catalog tables. In addition, excluding OIDs from a table reduces the space required to store the table on disk by 4 bytes per row, slightly improving performance. OIDS are not allowed on column-oriented tables.

ON COMMIT

The behavior of temporary tables at the end of a transaction block can be controlled using `ON COMMIT`. The three options are:

PRESERVE ROWS — No special action is taken at the ends of transactions for temporary tables. This is the default behavior.

DELETE ROWS — All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic `TRUNCATE` is done at each commit.

DROP — The temporary table will be dropped at the end of the current transaction block.

TABLESPACE *tablespace*

The tablespace is the name of the tablespace in which the new table is to be created. If not specified, the database's default tablespace is used.

AS *query*

A *`SELECT`* or *`VALUES`* command, or an *`EXECUTE`* command that runs a prepared `SELECT` or `VALUES` query.

DISTRIBUTED BY (*column*, [...])
DISTRIBUTED RANDOMLY

Used to declare the Greenplum Database distribution policy for the table. `DISTIBUTED BY` uses hash distribution with one or more columns declared as the distribution key. For the most even data distribution, the distribution key should be the primary key of the table or a unique column (or set of columns). If that is not possible, then you may choose DISTRIBUTED RANDOMLY, which will send the data round-robin to the segment instances.

The Greenplum Database server configuration parameter gp_create_table_random_default_distribution controls the default table distribution policy if the DISTRIBUTED BY clause is not specified when you create a table. Greenplum Database follows these rules to create a table if a distribution policy is not specified.

- If the legacy query optimizer creates the table, and the value of the parameter is off, the table distribution policy is determined based on the command.
- If the legacy query optimizer creates the table, and the value of the parameter is on, the table distribution policy is random.
- If the Pivotal Query Optimizer creates the table, the table distribution policy is random. The parameter value has no affect.

For information about the parameter, see "Server Configuration Parameters." For information about the legacy query optimizer and the Pivotal Query Optimizer, see "Querying Data" in the *Greenplum Database Administrator Guide*.

Notes

This command is functionally similar to *SELECT INTO*, but it is preferred since it is less likely to be confused with other uses of the SELECT INTO syntax. Furthermore, CREATE TABLE AS offers a superset of the functionality offered by SELECT INTO.

CREATE TABLE AS can be used for fast data loading from external table data sources. See *CREATE EXTERNAL TABLE*.

Examples

Create a new table films_recent consisting of only recent entries from the table films:

```
CREATE TABLE films_recent AS SELECT * FROM films WHERE
date_prod >= '2007-01-01';
```

Create a new temporary table films_recent, consisting of only recent entries from the table films, using a prepared statement. The new table has OIDs and will be dropped at commit:

```
PREPARE recentfilms(date) AS SELECT * FROM films WHERE
date_prod > $1;
CREATE TEMP TABLE films_recent WITH (OIDS) ON COMMIT DROP AS
EXECUTE recentfilms('2007-01-01');
```

Compatibility

CREATE TABLE AS conforms to the SQL standard, with the following exceptions:

- The standard requires parentheses around the subquery clause; in Greenplum Database, these parentheses are optional.
- The standard defines a WITH [NO] DATA clause; this is not currently implemented by Greenplum Database. The behavior provided by Greenplum Database is equivalent to the standard's WITH DATA case. WITH NO DATA can be simulated by appending LIMIT 0 to the query.
- Greenplum Database handles temporary tables differently from the standard; see CREATE TABLE for details.
- The WITH clause is a Greenplum Database extension; neither storage parameters nor OIDs are in the standard.
- The Greenplum Database concept of tablespaces is not part of the standard. The TABLESPACE clause is an extension.

See Also

CREATE EXTERNAL TABLE, *CREATE EXTERNAL TABLE*, *EXECUTE*, *SELECT*, *SELECT INTO*, *VALUES*

CREATE TABLESPACE

Defines a new tablespace.

Synopsis

```
CREATE TABLESPACE tablespace_name [OWNER username]
       FILESPACE filespace_name
```

Description

`CREATE TABLESPACE` registers a new tablespace for your Greenplum Database system. The tablespace name must be distinct from the name of any existing tablespace in the system.

A tablespace allows superusers to define an alternative location on the file system where the data files containing database objects (such as tables and indexes) may reside.

A user with appropriate privileges can pass a tablespace name to *CREATE DATABASE*, *CREATE TABLE*, or *CREATE INDEX* to have the data files for these objects stored within the specified tablespace.

In Greenplum Database, there must be a file system location defined for the master, each primary segment, and each mirror segment in order for the tablespace to have a location to store its objects across an entire Greenplum system. This collection of file system locations is defined in a filespace object. A filespace must be defined before you can create a tablespace. See `gpfilespace` in the *Greenplum Database Utility Guide* for more information.

Parameters

tablespacename

The name of a tablespace to be created. The name cannot begin with `pg_` or `gp_`, as such names are reserved for system tablespaces.

OWNER *username*

The name of the user who will own the tablespace. If omitted, defaults to the user executing the command. Only superusers may create tablespaces, but they can assign ownership of tablespaces to non-superusers.

FILESPACE

The name of a Greenplum Database filespace that was defined using the `gpfilespace` management utility.

Notes

You must first create a filespace to be used by the tablespace. See `gpfilespace` in the *Greenplum Database Utility Guide* for more information.

Tablespaces are only supported on systems that support symbolic links.

`CREATE TABLESPACE` cannot be executed inside a transaction block.

Examples

Create a new tablespace by specifying the corresponding filespace to use:

```
CREATE TABLESPACE mytblspace FILESPACE myfilespace;
```

Compatibility

CREATE TABLESPACE is a Greenplum Database extension.

See Also

CREATE DATABASE, *CREATE TABLE*, *CREATE INDEX*, *DROP TABLESPACE*, *ALTER TABLESPACE*, gpfilespace in the *Greenplum Database Utility Guide*

CREATE TYPE

Defines a new data type.

Synopsis

```
CREATE TYPE name AS ( attribute_name data_type [, ... ] )

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

CREATE TYPE name
```

Description

`CREATE TYPE` registers a new data type for use in the current database. The user who defines a type becomes its owner.

If a schema name is given then the type is created in the specified schema. Otherwise it is created in the current schema. The type name must be distinct from the name of any existing type or domain in the same schema. The type name must also be distinct from the name of any existing table in the same schema.

Composite Types

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

Base Types

The second form of `CREATE TYPE` creates a new base type (scalar type). The parameters may appear in any order, not only that shown in the syntax, and most are optional. You must register two or more functions (using `CREATE FUNCTION`) before defining the type. The support functions *input_function* and *output_function* are required, while the functions *receive_function*, *send_function* and *analyze_function* are optional. Generally these functions have to be coded in C or another low-level language. In Greenplum Database, any function used to implement a data type must be defined as `IMMUTABLE`.

The *input_function* converts the type's external textual representation to the internal representation used by the operators and functions defined for the type. *output_function* performs the reverse transformation. The input function may be declared as taking one argument of type `cstring`, or as taking three arguments of types `cstring`, `oid`, `integer`. The first argument is the input text as a C string, the second argument is the type's own OID (except for array types, which instead receive their element type's OID), and the third is the `typmod` of the destination column, if known (`-1` will be passed if not). The input function must return a value of the data type itself. Usually, an input function should be declared `STRICT`; if it is not, it will be called with a `NULL` first parameter when reading a `NULL` input value. The function must still return `NULL` in this case, unless it raises an error. (This case is mainly meant to support domain input functions, which may need to reject `NULL` inputs.) The output function must be declared as taking one argument of the new data type. The output function must return type `cstring`. Output functions are not invoked for `NULL` values.

The optional *receive_function* converts the type's external binary representation to the internal representation. If this function is not supplied, the type cannot participate in binary input. The binary representation should be chosen to be cheap to convert to internal form, while being reasonably portable. (For example, the standard integer data types use network byte order as the external binary representation, while the internal representation is in the machine's native byte order.) The receive function should perform adequate checking to ensure that the value is valid. The receive function may be declared as taking one argument of type `internal`, or as taking three arguments of types `internal`, `oid`, `integer`. The first argument is a pointer to a `StringInfo` buffer holding the received byte string; the optional arguments are the same as for the text input function. The receive function must return a value of the data type itself. Usually, a receive function should be declared `STRICT`; if it is not, it will be called with a `NULL` first parameter when reading a NULL input value. The function must still return `NULL` in this case, unless it raises an error. (This case is mainly meant to support domain receive functions, which may need to reject `NULL` inputs.) Similarly, the optional *send_function* converts from the internal representation to the external binary representation. If this function is not supplied, the type cannot participate in binary output. The send function must be declared as taking one argument of the new data type. The send function must return type `bytea`. Send functions are not invoked for `NULL` values.

You should at this point be wondering how the input and output functions can be declared to have results or arguments of the new type, when they have to be created before the new type can be created. The answer is that the type should first be defined as a shell type, which is a placeholder type that has no properties except a name and an owner. This is done by issuing the command `CREATE TYPE` *name*, with no additional parameters. Then the I/O functions can be defined referencing the shell type. Finally, `CREATE TYPE` with a full definition replaces the shell entry with a complete, valid type definition, after which the new type can be used normally.

While the details of the new type's internal representation are only known to the I/O functions and other functions you create to work with the type, there are several properties of the internal representation that must be declared to Greenplum Database. Foremost of these is *internallength*. Base data types can be fixed-length, in which case *internallength* is a positive integer, or variable length, indicated by setting *internallength* to `VARIABLE`. (Internally, this is represented by setting `typlen` to `-1`.) The internal representation of all variable-length types must start with a 4-byte integer giving the total length of this value of the type.

The optional flag `PASSEDBYVALUE` indicates that values of this data type are passed by value, rather than by reference. You may not pass by value types whose internal representation is larger than the size of the `Datum` type (4 bytes on most machines, 8 bytes on a few).

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

The *storage* parameter allows selection of storage strategies for variable-length data types. (Only `plain` is allowed for fixed-length types.) `plain` specifies that data of the type will always be stored in-line and not compressed. `extended` specifies that 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 strategy 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.)

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

To indicate that a type is an array, specify the type of the array elements using the `ELEMENT` key word. For example, to define an array of 4-byte integers (int4), specify `ELEMENT = int4`. More details about array types appear below.

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

Array Types

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

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

Parameters

name

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

attribute_name

The name of an attribute (column) for the composite type.

data_type

The name of an existing data type to become a column of the composite type.

input_function

The name of a function that converts data from the type's external textual form to its internal form.

output_function

The name of a function that converts data from the type's internal form to its external textual form.

receive_function

The name of a function that converts data from the type's external binary form to its internal form.

send_function

The name of a function that converts data from the type's internal form to its external binary form.

internallength

A numeric constant that specifies the length in bytes of the new type's internal representation. The default assumption is that it is variable-length.

alignment

The storage alignment requirement of the data type. Must be one of `char`, `int2`, `int4`, or `double`. The default is `int4`.

storage

The storage strategy for the data type. Must be one of `plain`, `external`, `extended`, or `main`. The default is `plain`.

default

The default value for the data type. If this is omitted, the default is null.

element

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

delimiter

The delimiter character to be used between values in arrays made of this type.

Notes

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

Because there are no restrictions on use of a data type once it's been created, creating a base type is tantamount to granting public execute permission on the functions mentioned in the type definition. (The creator of the type is therefore required to own these functions.) This is usually not an issue for the sorts of functions that are useful in a type definition. But you might want to think twice before designing a type in a way that would require 'secret' information to be used while converting it to or from external form.

Before Greenplum Database version 2.4, the syntax `CREATE TYPE` *name* did not exist. The way to create a new base type was to create its input function first. In this approach, Greenplum Database will first see the name of the new data type as the return type of the input function. The shell type is implicitly created in this situation, and then it can be referenced in the definitions of the remaining I/O functions. This approach still works, but is deprecated and may be disallowed in some future release. Also, to avoid accidentally cluttering the catalogs with shell types as a result of simple typos in function definitions, a shell type will only be made this way when the input function is written in C.

Examples

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

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

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

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS
... ;

CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS
... ;

CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

If the internal structure of `box` were an array of four `float4` elements, we might instead use:

```
CREATE TYPE box (
```

```
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

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

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

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

CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

Compatibility

This `CREATE TYPE` command is a Greenplum Database extension. There is a `CREATE TYPE` statement in the SQL standard that is rather different in detail.

See Also

CREATE FUNCTION, *ALTER TYPE*, *DROP TYPE*, *CREATE DOMAIN*

CREATE USER

Defines a new database role with the `LOGIN` privilege by default.

Synopsis

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

where *option* can be:

```
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE rolename [, ...]
    | IN GROUP rolename [, ...]
    | ROLE rolename [, ...]
    | ADMIN rolename [, ...]
    | USER rolename [, ...]
    | SYSID uid    | RESOURCE QUEUE queue_name
```

Description

As of Greenplum Database release 2.2, `CREATE USER` has been replaced by *`CREATE ROLE`*, although it is still accepted for backwards compatibility.

The only difference between `CREATE ROLE` and `CREATE USER` is that `LOGIN` is assumed by default with `CREATE USER`, whereas `NOLOGIN` is assumed by default with `CREATE ROLE`.

Compatibility

There is no `CREATE USER` statement in the SQL standard.

See Also

`CREATE ROLE`

CREATE VIEW

Defines a new view.

Synopsis

```
CREATE [OR REPLACE] [TEMP | TEMPORARY] VIEW name
       [ ( column_name [, ...] ) ]
       AS query
```

Description

`CREATE VIEW` defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

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

If a schema name is given then the view is created in the specified schema. Otherwise it is created in the current schema. Temporary views exist in a special schema, so a schema name may not be given when creating a temporary view. The name of the view must be distinct from the name of any other view, table, sequence, or index in the same schema.

Parameters

TEMPORARY | TEMP

If specified, the view is created as a temporary view. Temporary views are automatically dropped at the end of the current session. Existing permanent relations with the same name are not visible to the current session while the temporary view exists, unless they are referenced with schema-qualified names. If any of the tables referenced by the view are temporary, the view is created as a temporary view (whether `TEMPORARY` is specified or not).

name

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

column_name

An optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.

query

A *`SELECT`* or *`VALUES`* command which will provide the columns and rows of the view.

Notes

Views in Greenplum Database are read only. The system will not allow an insert, update, or delete on a view. You can get the effect of an updatable view by creating rewrite rules on the view into appropriate actions on other tables. For more information see `CREATE RULE`.

Be careful that the names and data types of the view's columns will be assigned the way you want. For example:

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

is bad form in two ways: the column name defaults to `?column?`, and the column data type defaults to `unknown`. If you want a string literal in a view's result, use something like:

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

Access to tables referenced in the view is determined by permissions of the view owner not the current user (even if the current user is a superuser). This can be confusing in the case of superusers, since superusers typically have access to all objects. In the case of a view, even superusers must be explicitly granted access to tables referenced in the view if they are not the owner of the view.

However, functions called in the view are treated the same as if they had been called directly from the query using the view. Therefore the user of a view must have permissions to call any functions used by the view.

If you create a view with an `ORDER BY` clause, the `ORDER BY` clause is ignored when you do a `SELECT` from the view.

Examples

Create a view consisting of all comedy films:

```
CREATE VIEW comedies AS SELECT * FROM films WHERE kind =
'comedy';
```

Create a view that gets the top ten ranked baby names:

```
CREATE VIEW topten AS SELECT name, rank, gender, year FROM
names, rank WHERE rank < '11' AND names.id=rank.id;
```

Compatibility

The SQL standard specifies some additional capabilities for the `CREATE VIEW` statement that are not in Greenplum Database. The optional clauses for the full SQL command in the standard are:

- **CHECK OPTION** — This option has to do with updatable views. All `INSERT` and `UPDATE` commands on the view will be checked to ensure data satisfy the view-defining condition (that is, the new data would be visible through the view). If they do not, the update will be rejected.
- **LOCAL** — Check for integrity on this view.
- **CASCADED** — Check for integrity on this view and on any dependent view. `CASCADED` is assumed if neither `CASCADED` nor `LOCAL` is specified.

`CREATE OR REPLACE VIEW` is a Greenplum Database language extension. So is the concept of a temporary view.

See Also

SELECT, *DROP VIEW*

DEALLOCATE

Deallocates a prepared statement.

Synopsis

```
DEALLOCATE [PREPARE] name
```

Description

`DEALLOCATE` is used to deallocate a previously prepared SQL statement. If you do not explicitly deallocate a prepared statement, it is deallocated when the session ends.

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

Parameters

PREPARE

Optional key word which is ignored.

name

The name of the prepared statement to deallocate.

Examples

Deallocated the previously prepared statement named `insert_names`:

```
DEALLOCATE insert_names;
```

Compatibility

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

See Also

EXECUTE, *PREPARE*

DECLARE

Defines a cursor.

Synopsis

```
DECLARE name [BINARY] [INSENSITIVE] [NO SCROLL] CURSOR
     [{WITH | WITHOUT} HOLD]
     FOR query [FOR READ ONLY]
```

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, the same as a `SELECT` would produce. Since data is stored natively in binary format, the system must do a conversion to produce the text format. Once the information comes back in text form, the client application may need to convert it to a binary format to manipulate it. In addition, data in the text format is often larger in size than in the binary format. Binary cursors return the data in a binary representation that may be more easily manipulated. Nevertheless, if you intend to display the data as text anyway, retrieving it in text form will save you some effort on the client side.

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 field containing the internal representation of the value (in big-endian byte order).

Binary cursors should be used carefully. Many applications, including psql, are not prepared to handle binary cursors and expect data to come back in the text format.

> **Note:**
>
> When the client application uses the 'extended query' protocol to issue a `FETCH` command, the Bind protocol message specifies whether data is to be retrieved in text or binary format. This choice overrides the way that the cursor is defined. The concept of a binary cursor as such is thus obsolete when using extended query protocol — any cursor can be treated as either text or binary.

A cursor can be specified in the `WHERE CURRENT OF` clause of the *UPDATE* or *DELETE* command to update or delete table data.

Parameters

name

The name of the cursor to be created.

BINARY

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

INSENSITIVE

Indicates that data retrieved from the cursor should be unaffected by updates to the tables underlying the cursor while the cursor exists. In Greenplum Database, all cursors are insensitive. This key word currently has no effect and is present for compatibility with the SQL standard.

NO SCROLL

A cursor cannot be used to retrieve rows in a nonsequential fashion. This is the default behavior in Greenplum Database, since scrollable cursors (`SCROLL`) are not supported.

WITH HOLD
WITHOUT HOLD

`WITH HOLD` specifies that the cursor may continue to be used after the transaction that created it successfully commits. `WITHOUT HOLD` specifies that the cursor cannot be used outside of the transaction that created it. `WITHOUT HOLD` is the default.

`WITH HOLD` cannot not be specified when the `query` includes a `FOR UPDATE` or `FOR SHARE` clause.

query

A *`SELECT`* or *`VALUES`* command which will provide the rows to be returned by the cursor.

If the cursor is used in the `WHERE CURRENT OF` clause of the *`UPDATE`* or *`DELETE`* command, the `SELECT` command must satisfy the following conditions:

- Cannot reference a view or external table.
- References only one table.

  The table must be updatable. For example, the following are not updatable: table functions, set-returning functions, append-only tables, columnar tables.
- Cannot contain any of the following:
  - A grouping clause
  - A set operation such as `UNION ALL` or `UNION DISTINCT`
  - A sorting clause
  - A windowing clause
  - A join or a self-join

  Specifying the `FOR UPDATE` clause in the `SELECT` command prevents other sessions from changing the rows between the time they are fetched and the time they are updated. Without the `FOR UPDATE` clause, a subsequent use of the `UPDATE` or `DELETE` command with the `WHERE CURRENT OF` clause has no effect if the row was changed since the cursor was created.

  > **Note:** Specifying the `FOR UPDATE` clause in the `SELECT` command locks the entire table, not just the selected rows.

FOR READ ONLY

`FOR READ ONLY` indicates that the cursor is used in a read-only mode.

Notes

Unless `WITH HOLD` is specified, the cursor created by this command can only be used within the current transaction. Thus, `DECLARE` without `WITH HOLD` is useless outside a transaction block: the cursor would survive only to the completion of the statement. Therefore Greenplum Database reports an error if this command is used outside a transaction block. Use `BEGIN`, `COMMIT` and `ROLLBACK` to define a transaction block.

If `WITH HOLD` is specified and the transaction that created the cursor successfully commits, the cursor can continue to be accessed by subsequent transactions in the same session. (But if the creating transaction is aborted, the cursor is removed.) A cursor created with `WITH HOLD` is closed when an explicit `CLOSE` command is issued on it, or the session ends. In the current implementation, the rows represented by a held cursor are copied into a temporary file or memory area so that they remain available for subsequent transactions.

If you create a cursor with the `DECLARE` command in a transaction, you cannot use the `SET` command in the transaction until you close the cursor with the `CLOSE` command.

Scrollable cursors are not currently supported in Greenplum Database. You can only use `FETCH` to move the cursor position forward, not backwards.

`DECLARE...FORUPDATE` is not supported with append-optimized tables.

You can see all available cursors by querying the `pg_cursors` system view.

Examples

Declare a cursor:

```
DECLARE mycursor CURSOR FOR SELECT * FROM mytable;
```

Compatibility

SQL standard allows cursors only in embedded SQL and in modules. Greenplum Database permits cursors to be used interactively.

Greenplum Database does not implement an `OPEN` statement for cursors. A cursor is considered to be open when it is declared.

The SQL standard allows cursors to move both forward and backward. All Greenplum Database cursors are forward moving only (not scrollable).

Binary cursors are a Greenplum Database extension.

See Also

CLOSE, *DELETE*, *FETCH*, *MOVE*, *SELECT*, *UPDATE*

DELETE

Deletes rows from a table.

Synopsis

```
DELETE FROM [ONLY] table [[AS] alias]
      [USING usinglist]
      [WHERE condition | WHERE CURRENT OF cursor_name ]
```

Description

`DELETE` deletes rows that satisfy the `WHERE` clause from the specified table. If the `WHERE` clause is absent, the effect is to delete all rows in the table. The result is a valid, but empty table.

By default, `DELETE` will delete rows in the specified table and all its child tables. If you wish to delete only from the specific table mentioned, you must use the `ONLY` clause.

There are two ways to delete rows in a table using information contained in other tables in the database: using sub-selects, or specifying additional tables in the `USING` clause. Which technique is more appropriate depends on the specific circumstances.

If the `WHERE CURRENT OF` clause is specified, the row that is deleted is the one most recently fetched from the specified cursor.

You must have the `DELETE` privilege on the table to delete from it.

Outputs

On successful completion, a `DELETE` command returns a command tag of the form

```
DELETE count
```

The count is the number of rows deleted. If count is 0, no rows matched the condition (this is not considered an error).

Parameters

ONLY

If specified, delete rows from the named table only. When not specified, any tables inheriting from the named table are also processed.

table

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

alias

A substitute name for the target table. When an alias is provided, it completely hides the actual name of the table. For example, given `DELETE FROM foo AS f`, the remainder of the `DELETE` statement must refer to this table as `f` not `foo`.

usinglist

A list of table expressions, allowing columns from other tables to appear in the `WHERE` condition. This is similar to the list of tables that can be specified in the `FROM` Clause of a *`SELECT`* statement; for example, an alias for the table name can be specified. Do not repeat the target table in the `usinglist`, unless you wish to set up a self-join.

condition

An expression returning a value of type `boolean`, which determines the rows that are to be deleted.

cursor_name

The name of the cursor to use in a `WHERE CURRENT OF` condition. The row to be deleted is the one most recently fetched from this cursor. The cursor must be a simple (non-join, non-aggregate) query on the `DELETE` target table. See *DECLARE* for more information about creating cursors.

`WHERE CURRENT OF` cannot be specified together with a Boolean condition.

See *DECLARE* for more information about creating cursors.

Notes

Greenplum Database lets you reference columns of other tables in the `WHERE` condition by specifying the other tables in the `USING` clause. For example, to the name `Hannah` from the `rank` table, one might do:

```
DELETE FROM rank USING names WHERE names.id = rank.id AND
name = 'Hannah';
```

What is essentially happening here is a join between `rank` and `names`, with all successfully joined rows being marked for deletion. This syntax is not standard. However, this join style is usually easier to write and faster to execute than a more standard sub-select style, such as:

```
DELETE FROM rank WHERE id IN (SELECT id FROM names WHERE name
= 'Hannah');
```

When using `DELETE` to remove all the rows of a table (for example: `DELETE * FROM table;`), Greenplum Database adds an implicit `TRUNCATE` command (when user permissions allow). The added `TRUNCATE` command frees the disk space occupied by the deleted rows without requiring a `VACUUM` of the table. This improves scan performance of subsequent queries, and benefits ELT workloads that frequently insert and delete from temporary tables.

Execution of `UPDATE` and `DELETE` commands directly on a specific partition (child table) of a partitioned table is not supported. Instead, these commands must be executed on the root partitioned table, the table created with the `CREATE TABLE` command.

Examples

Delete all films but musicals:

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

Clear the table films:

```
DELETE FROM films;
```

Delete using a join:

```
DELETE FROM rank USING names WHERE names.id = rank.id AND
name = 'Hannah';
```

Compatibility

This command conforms to the SQL standard, except that the `USING` clause is a Greenplum Database extension.

See Also

DECLARE, *TRUNCATE*

DROP AGGREGATE

Removes an aggregate function.

Synopsis

```
DROP AGGREGATE [IF EXISTS] name ( type [, ...] ) [CASCADE | RESTRICT]
```

Description

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

Parameters

IF EXISTS

Do not throw an error if the aggregate does not exist. A notice is issued in this case.

name

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

type

An input data type on which the aggregate function operates. To reference a zero-argument aggregate function, write `*` in place of the list of input data types.

CASCADE

Automatically drop objects that depend on the aggregate function.

RESTRICT

Refuse to drop the aggregate function if any objects depend on it. This is the default.

Examples

To remove the aggregate function `myavg` for type `integer`:

```
DROP AGGREGATE myavg(integer);
```

Compatibility

There is no `DROP AGGREGATE` statement in the SQL standard.

See Also

ALTER AGGREGATE, *CREATE AGGREGATE*

DROP CAST

Removes a cast.

Synopsis

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

Description

`DROP CAST` will delete a previously defined cast. To be able to drop a cast, you must own the source or the target data type. These are the same privileges that are required to create a cast.

Parameters

IF EXISTS

Do not throw an error if the cast does not exist. A notice is issued in this case.

sourcetype

The name of the source data type of the cast.

targettype

The name of the target data type of the cast.

CASCADE
RESTRICT

These keywords have no effect since there are no dependencies on casts.

Examples

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

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

Compatibility

There `DROP CAST` command conforms to the SQL standard.

See Also

CREATE CAST

DROP CONVERSION

Removes a conversion.

Synopsis

```
DROP CONVERSION [IF EXISTS] name [CASCADE | RESTRICT]
```

Description

`DROP CONVERSION` removes a previously defined conversion. To be able to drop a conversion, you must own the conversion.

Parameters

IF EXISTS

Do not throw an error if the conversion does not exist. A notice is issued in this case.

name

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

CASCADE
RESTRICT

These keywords have no effect since there are no dependencies on conversions.

Examples

Drop the conversion named `myname`:

```
DROP CONVERSION myname;
```

Compatibility

There is no `DROP CONVERSION` statement in the SQL standard.

See Also

ALTER CONVERSION, *CREATE CONVERSION*

DROP DATABASE

Removes a database.

Synopsis

```
DROP DATABASE [IF EXISTS] name
```

Description

`DROP DATABASE` drops a database. It removes the catalog entries for the database and deletes the directory containing the data. It can only be executed by the database owner. Also, it cannot be executed while you or anyone else are connected to the target database. (Connect to `template1` or any other database to issue this command.)

> **Warning:** `DROP DATABASE` cannot be undone. Use it with care!

Parameters

IF EXISTS

Do not throw an error if the database does not exist. A notice is issued in this case.

name

The name of the database to remove.

Notes

`DROP DATABASE` cannot be executed inside a transaction block.

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

Examples

Drop the database named `testdb`:

```
DROP DATABASE testdb;
```

Compatibility

There is no `DROP DATABASE` statement in the SQL standard.

See Also

ALTER DATABASE, *CREATE DATABASE*

DROP DOMAIN

Removes a domain.

Synopsis

```
DROP DOMAIN [IF EXISTS] name [, ...]  [CASCADE | RESTRICT]
```

Description

`DROP DOMAIN` removes a previously defined domain. You must be the owner of a domain to drop it.

Parameters

IF EXISTS

Do not throw an error if the domain does not exist. A notice is issued in this case.

name

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

CASCADE

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

RESTRICT

Refuse to drop the domain if any objects depend on it. This is the default.

Examples

Drop the domain named `zipcode`:

```
DROP DOMAIN zipcode;
```

Compatibility

This command conforms to the SQL standard, except for the `IF EXISTS` option, which is a Greenplum Database extension.

See Also

ALTER DOMAIN, *CREATE DOMAIN*

DROP EXTERNAL TABLE

Removes an external table definition.

Synopsis

```
DROP EXTERNAL [WEB] TABLE [IF EXISTS] name [CASCADE | RESTRICT]
```

Description

`DROP EXTERNAL TABLE` drops an existing external table definition from the database system. The external data sources or files are not deleted. To execute this command you must be the owner of the external table.

Parameters

WEB

Optional keyword for dropping external web tables.

IF EXISTS

Do not throw an error if the external table does not exist. A notice is issued in this case.

name

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

CASCADE

Automatically drop objects that depend on the external table (such as views).

RESTRICT

Refuse to drop the external table if any objects depend on it. This is the default.

Examples

Remove the external table named `staging` if it exists:

```
DROP EXTERNAL TABLE IF EXISTS staging;
```

Compatibility

There is no `DROP EXTERNAL TABLE` statement in the SQL standard.

See Also

CREATE EXTERNAL TABLE

DROP FILESPACE

Removes a filespace.

Synopsis

```
DROP FILESPACE [IF EXISTS] filespacename
```

Description

`DROP FILESPACE` removes a filespace definition and its system-generated data directories from the system.

A filespace can only be dropped by its owner or a superuser. The filespace must be empty of all tablespace objects before it can be dropped. It is possible that tablespaces in other databases may still be using a filespace even if no tablespaces in the current database are using the filespace.

Parameters

IF EXISTS

Do not throw an error if the filespace does not exist. A notice is issued in this case.

tablespacename

The name of the filespace to remove.

Examples

Remove the tablespace `myfs`:

```
DROP FILESPACE myfs;
```

Compatibility

There is no `DROP FILESPACE` statement in the SQL standard or in PostgreSQL.

See Also

ALTER FILESPACE, *DROP TABLESPACE*, `gpfilespace` in the *Greenplum Database Utility Guide*

DROP FUNCTION

Removes a function.

Synopsis

```
DROP FUNCTION [IF EXISTS] name ( [ [argmode] [argname] argtype
    [, ...] ] ) [CASCADE | RESTRICT]
```

Description

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

Parameters

IF EXISTS

Do not throw an error if the function does not exist. A notice is issued in this case.

name

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

argmode

The mode of an argument: either `IN`, `OUT`, or `INOUT`. If omitted, the default is IN. Note that `DROP FUNCTION` does not actually pay any attention to `OUT` arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the `IN` and `INOUT` arguments.

argname

The name of an argument. Note that `DROP FUNCTION` does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any.

CASCADE

Automatically drop objects that depend on the function such as operators.

RESTRICT

Refuse to drop the function if any objects depend on it. This is the default.

Examples

Drop the square root function:

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

Compatibility

A `DROP FUNCTION` statement is defined in the SQL standard, but it is not compatible with this command.

See Also

CREATE FUNCTION, *ALTER FUNCTION*

DROP GROUP

Removes a database role.

Synopsis

```
DROP GROUP [IF EXISTS] name [, ...]
```

Description

`DROP GROUP` is an obsolete command, though still accepted for backwards compatibility. Groups (and users) have been superseded by the more general concept of roles. See *DROP ROLE* for more information.

Parameters

IF EXISTS

Do not throw an error if the role does not exist. A notice is issued in this case.

name

The name of an existing role.

Compatibility

There is no `DROP GROUP` statement in the SQL standard.

See Also

DROP ROLE

DROP INDEX

Removes an index.

Synopsis

```
DROP INDEX [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

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

Parameters

IF EXISTS

Do not throw an error if the index does not exist. A notice is issued in this case.

name

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

CASCADE

Automatically drop objects that depend on the index.

RESTRICT

Refuse to drop the index if any objects depend on it. This is the default.

Examples

Remove the index `title_idx`:

```
DROP INDEX title_idx;
```

Compatibility

`DROP INDEX` is a Greenplum Database language extension. There are no provisions for indexes in the SQL standard.

See Also

ALTER INDEX, *CREATE INDEX*, *REINDEX*

DROP LANGUAGE

Removes a procedural language.

Synopsis

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

Description

`DROP LANGUAGE` will remove the definition of the previously registered procedural language. You must be a superuser to drop a language.

Parameters

PROCEDURAL

Optional keyword - has no effect.

IF EXISTS

Do not throw an error if the language does not exist. A notice is issued in this case.

name

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

CASCADE

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

RESTRICT

Refuse to drop the language if any objects depend on it. This is the default.

Examples

Remove the procedural language `plsample`:

```
DROP LANGUAGE plsample;
```

Compatibility

There is no `DROP LANGUAGE` statement in the SQL standard.

See Also

ALTER LANGUAGE, *CREATE LANGUAGE*

DROP OPERATOR

Removes an operator.

Synopsis

```
DROP OPERATOR [IF EXISTS] name ( {lefttype | NONE} ,
    {righttype | NONE} ) [CASCADE | RESTRICT]
```

Description

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

Parameters

IF EXISTS

Do not throw an error if the operator does not exist. A notice is issued in this case.

name

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

lefttype

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

righttype

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

CASCADE

Automatically drop objects that depend on the operator.

RESTRICT

Refuse to drop the operator if any objects depend on it. This is the default.

Examples

Remove the power operator `a^b` for type `integer`:

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

Remove the left unary bitwise complement operator `~b` for type `bit`:

```
DROP OPERATOR ~ (none, bit);
```

Remove the right unary factorial operator `x!` for type `bigint`:

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

Compatibility

There is no `DROP OPERATOR` statement in the SQL standard.

See Also

ALTER OPERATOR, *CREATE OPERATOR*

DROP OPERATOR CLASS

Removes an operator class.

Synopsis

```
DROP OPERATOR CLASS [IF EXISTS] name USING index_method [CASCADE | RESTRICT]
```

Description

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

Parameters

IF EXISTS

Do not throw an error if the operator class does not exist. A notice is issued in this case.

name

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

index_method

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

CASCADE

Automatically drop objects that depend on the operator class.

RESTRICT

Refuse to drop the operator class if any objects depend on it. This is the default.

Examples

Remove the B-tree operator class `widget_ops`:

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

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

Compatibility

There is no `DROP OPERATOR CLASS` statement in the SQL standard.

See Also

ALTER OPERATOR CLASS, *CREATE OPERATOR CLASS*

DROP OWNED

Removes database objects owned by a database role.

Synopsis

```
DROP OWNED BY name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP OWNED` drops all the objects in the current database that are owned by one of the specified roles. Any privileges granted to the given roles on objects in the current database will also be revoked.

Parameters

name

The name of a role whose objects will be dropped, and whose privileges will be revoked.

CASCADE

Automatically drop objects that depend on the affected objects.

RESTRICT

Refuse to drop the objects owned by a role if any other database objects depend on one of the affected objects. This is the default.

Notes

`DROP OWNED` is often used to prepare for the removal of one or more roles. Because `DROP OWNED` only affects the objects in the current database, it is usually necessary to execute this command in each database that contains objects owned by a role that is to be removed.

Using the `CASCADE` option may make the command recurse to objects owned by other users.

The `REASSIGN OWNED` command is an alternative that reassigns the ownership of all the database objects owned by one or more roles.

Examples

Remove any database objects owned by the role named `sally`:

```
DROP OWNED BY sally;
```

Compatibility

The `DROP OWNED` statement is a Greenplum Database extension.

See Also

REASSIGN OWNED, *DROP ROLE*

DROP RESOURCE QUEUE

Removes a resource queue.

Synopsis

```
DROP RESOURCE QUEUE queue_name
```

Description

This command removes a workload management resource queue from Greenplum Database. To drop a resource queue, the queue cannot have any roles assigned to it, nor can it have any statements waiting in the queue. Only a superuser can drop a resource queue.

Parameters

queue_name

The name of a resource queue to remove.

Notes

Use *ALTER ROLE* to remove a user from a resource queue.

To see all the currently active queries for all resource queues, perform the following query of the `pg_locks` table joined with the `pg_roles` and `pg_resqueue` tables:

```
SELECT rolname, rsqname, locktype, objid, transaction, pid,
mode, granted FROM pg_roles, pg_resqueue, pg_locks WHERE
pg_roles.rolresqueue=pg_locks.objid AND
pg_locks.objid=pg_resqueue.oid;
```

To see the roles assigned to a resource queue, perform the following query of the `pg_roles` and `pg_resqueue` system catalog tables:

```
SELECT rolname, rsqname FROM pg_roles, pg_resqueue WHERE
pg_roles.rolresqueue=pg_resqueue.oid;
```

Examples

Remove a role from a resource queue (and move the role to the default resource queue, `pg_default`):

```
ALTER ROLE bob RESOURCE QUEUE NONE;
```

Remove the resource queue named `adhoc`:

```
DROP RESOURCE QUEUE adhoc;
```

Compatibility

The `DROP RESOURCE QUEUE` statement is a Greenplum Database extension.

See Also

ALTER RESOURCE QUEUE, *CREATE RESOURCE QUEUE*, *ALTER ROLE*

DROP ROLE

Removes a database role.

Synopsis

```
DROP ROLE [IF EXISTS] name [, ...]
```

Description

`DROP ROLE` removes the specified role(s). To drop a superuser role, you must be a superuser yourself. To drop non-superuser roles, you must have `CREATEROLE` privilege.

A role cannot be removed if it is still referenced in any database; an error will be raised if so. Before dropping the role, you must drop all the objects it owns (or reassign their ownership) and revoke any privileges the role has been granted. The `REASSIGN OWNED` and `DROP OWNED` commands can be useful for this purpose.

However, it is not necessary to remove role memberships involving the role; `DROP ROLE` automatically revokes any memberships of the target role in other roles, and of other roles in the target role. The other roles are not dropped nor otherwise affected.

Parameters

IF EXISTS

Do not throw an error if the role does not exist. A notice is issued in this case.

name

The name of the role to remove.

Examples

Remove the roles named `sally` and `bob`:

```
DROP ROLE sally, bob;
```

Compatibility

The SQL standard defines `DROP ROLE`, but it allows only one role to be dropped at a time, and it specifies different privilege requirements than Greenplum Database uses.

See Also

REASSIGN OWNED, *DROP OWNED*, *CREATE ROLE*, *ALTER ROLE*, *SET ROLE*

DROP RULE

Removes a rewrite rule.

Synopsis

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

Description

`DROP RULE` drops a rewrite rule from a table or view.

Parameters

IF EXISTS

Do not throw an error if the rule does not exist. A notice is issued in this case.

name

The name of the rule to remove.

relation

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

CASCADE

Automatically drop objects that depend on the rule.

RESTRICT

Refuse to drop the rule if any objects depend on it. This is the default.

Examples

Remove the rewrite rule `sales_2006` on the table `sales`:

```
DROP RULE sales_2006 ON sales;
```

Compatibility

There is no `DROP RULE` statement in the SQL standard.

See Also

CREATE RULE

DROP SCHEMA

Removes a schema.

Synopsis

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

Description

`DROP SCHEMA` removes schemas from the database. A schema can only be dropped by its owner or a superuser. Note that the owner can drop the schema (and thereby all contained objects) even if he does not own some of the objects within the schema.

Parameters

IF EXISTS

Do not throw an error if the schema does not exist. A notice is issued in this case.

name

The name of the schema to remove.

CASCADE

Automatically drops any objects contained in the schema (tables, functions, etc.).

RESTRICT

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

Examples

Remove the schema `mystuff` from the database, along with everything it contains:

```
DROP SCHEMA mystuff CASCADE;
```

Compatibility

`DROP SCHEMA` is fully conforming with the SQL standard, except that the standard only allows one schema to be dropped per command. Also, the `IF EXISTS` option is a Greenplum Database extension.

See Also

CREATE SCHEMA, *ALTER SCHEMA*

DROP SEQUENCE

Removes a sequence.

Synopsis

```
DROP SEQUENCE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP SEQUENCE` removes a sequence generator table. You must own the sequence to drop it (or be a superuser).

Parameters

IF EXISTS

Do not throw an error if the sequence does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the sequence to remove.

CASCADE

Automatically drop objects that depend on the sequence.

RESTRICT

Refuse to drop the sequence if any objects depend on it. This is the default.

Examples

Remove the sequence `myserial`:

```
DROP SEQUENCE myserial;
```

Compatibility

`DROP SEQUENCE` is fully conforming with the SQL standard, except that the standard only allows one sequence to be dropped per command. Also, the `IF EXISTS` option is a Greenplum Database extension.

See Also

ALTER SEQUENCE, *CREATE SEQUENCE*

DROP TABLE

Removes a table.

Synopsis

```
DROP TABLE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP TABLE` removes tables from the database. Only its owner may drop a table. To empty a table of rows without removing the table definition, use `DELETE` or `TRUNCATE`.

`DROP TABLE` always removes any indexes, rules, triggers, and constraints that exist for the target table. However, to drop a table that is referenced by a view, `CASCADE` must be specified. `CASCADE` will remove a dependent view entirely.

Parameters

IF EXISTS

Do not throw an error if the table does not exist. A notice is issued in this case.

name

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

CASCADE

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

RESTRICT

Refuse to drop the table if any objects depend on it. This is the default.

Examples

Remove the table `mytable`:

```
DROP TABLE mytable;
```

Compatibility

`DROP TABLE` is fully conforming with the SQL standard, except that the standard only allows one table to be dropped per command. Also, the `IF EXISTS` option is a Greenplum Database extension.

See Also

CREATE TABLE, *ALTER TABLE*, *TRUNCATE*

DROP TABLESPACE

Removes a tablespace.

Synopsis

```
DROP TABLESPACE [IF EXISTS] tablespacename
```

Description

`DROP TABLESPACE` removes a tablespace from the system.

A tablespace can only be dropped by its owner or a superuser. The tablespace must be empty of all database objects before it can be dropped. It is possible that objects in other databases may still reside in the tablespace even if no objects in the current database are using the tablespace.

Parameters

IF EXISTS

Do not throw an error if the tablespace does not exist. A notice is issued in this case.

tablespacename

The name of the tablespace to remove.

Examples

Remove the tablespace `mystuff`:

```
DROP TABLESPACE mystuff;
```

Compatibility

`DROP TABLESPACE` is a Greenplum Database extension.

See Also

CREATE TABLESPACE, *ALTER TABLESPACE*

DROP TYPE

Removes a data type.

Synopsis

```
DROP TYPE [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP TYPE` will remove a user-defined data type. Only the owner of a type can remove it.

Parameters

IF EXISTS

Do not throw an error if the type does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the data type to remove.

CASCADE

Automatically drop objects that depend on the type (such as table columns, functions, operators).

RESTRICT

Refuse to drop the type if any objects depend on it. This is the default.

Examples

Remove the data type `box`;

```
DROP TYPE box;
```

Compatibility

This command is similar to the corresponding command in the SQL standard, apart from the `IF EXISTS` option, which is a Greenplum Database extension. But note that the `CREATE TYPE` command and the data type extension mechanisms in Greenplum Database differ from the SQL standard.

See Also

ALTER TYPE, *CREATE TYPE*

DROP USER

Removes a database role.

Synopsis

```
DROP USER [IF EXISTS] name [, ...]
```

Description

`DROP USER` is an obsolete command, though still accepted for backwards compatibility. Groups (and users) have been superseded by the more general concept of roles. See *`DROP ROLE`* for more information.

Parameters

IF EXISTS

Do not throw an error if the role does not exist. A notice is issued in this case.

name

The name of an existing role.

Compatibility

There is no `DROP USER` statement in the SQL standard. The SQL standard leaves the definition of users to the implementation.

See Also

`DROP ROLE`

DROP VIEW

Removes a view.

Synopsis

```
DROP VIEW [IF EXISTS] name [, ...] [CASCADE | RESTRICT]
```

Description

`DROP VIEW` will remove an existing view. Only the owner of a view can remove it.

Parameters

IF EXISTS

Do not throw an error if the view does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the view to remove.

CASCADE

Automatically drop objects that depend on the view (such as other views).

RESTRICT

Refuse to drop the view if any objects depend on it. This is the default.

Examples

Remove the view `topten`;

```
DROP VIEW topten;
```

Compatibility

`DROP VIEW` is fully conforming with the SQL standard, except that the standard only allows one view to be dropped per command. Also, the `IF EXISTS` option is a Greenplum Database extension.

See Also

CREATE VIEW

END

Commits the current transaction.

Synopsis

```
END [WORK | TRANSACTION]
```

Description

`END` commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs. This command is a Greenplum Database extension that is equivalent to *`COMMIT`*.

Parameters

WORK
TRANSACTION

Optional keywords. They have no effect.

Examples

Commit the current transaction:

```
END;
```

Compatibility

`END` is a Greenplum Database extension that provides functionality equivalent to *`COMMIT`*, which is specified in the SQL standard.

See Also

`BEGIN`, *`ROLLBACK`*, *`COMMIT`*

EXECUTE

Executes a prepared SQL statement.

Synopsis

```
EXECUTE name [ (parameter [, ...] ) ]
```

Description

`EXECUTE` is used to execute a previously prepared statement. Since prepared statements only exist for the duration of a session, the prepared statement must have been created by a `PREPARE` statement executed earlier in the current session.

If the `PREPARE` statement that created the statement specified some parameters, a compatible set of parameters must be passed to the `EXECUTE` statement, or else an error is raised. Note that (unlike functions) prepared statements are not overloaded based on the type or number of their parameters; the name of a prepared statement must be unique within a database session.

For more information on the creation and usage of prepared statements, see `PREPARE`.

Parameters

name

The name of the prepared statement to execute.

parameter

The actual value of a parameter to the prepared statement. This must be an expression yielding a value that is compatible with the data type of this parameter, as was determined when the prepared statement was created.

Examples

Create a prepared statement for an `INSERT` statement, and then execute it:

```
PREPARE fooplan (int, text, bool, numeric) AS INSERT INTO
foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

Compatibility

The SQL standard includes an `EXECUTE` statement, but it is only for use in embedded SQL. This version of the `EXECUTE` statement also uses a somewhat different syntax.

See Also

DEALLOCATE, *PREPARE*

EXPLAIN

Shows the query plan of a statement.

Synopsis

```
EXPLAIN [ANALYZE] [VERBOSE] statement
```

Description

`EXPLAIN` displays the query plan that the Greenplum planner generates for the supplied statement. Query plans are a tree plan of nodes. Each node in the plan represents a single operation, such as table scan, join, aggregation or a sort.

Plans should be read from the bottom up as each node feeds rows into the node directly above it. The bottom nodes of a plan are usually table scan operations (sequential, index or bitmap index scans). If the query requires joins, aggregations, or sorts (or other operations on the raw rows) then there will be additional nodes above the scan nodes to perform these operations. The topmost plan nodes are usually the Greenplum Database motion nodes (redistribute, explicit redistribute, broadcast, or gather motions). These are the operations responsible for moving rows between the segment instances during query processing.

The output of `EXPLAIN` has one line for each node in the plan tree, showing the basic node type plus the following cost estimates that the planner made for the execution of that plan node:

- **cost** — measured in units of disk page fetches; that is, 1.0 equals one sequential disk page read. The first estimate is the start-up cost (cost of getting to the first row) and the second is the total cost (cost of getting all rows). Note that the total cost assumes that all rows will be retrieved, which may not always be the case (if using `LIMIT` for example).
- **rows** — the total number of rows output by this plan node. This is usually less than the actual number of rows processed or scanned by the plan node, reflecting the estimated selectivity of any `WHERE` clause conditions. Ideally the top-level nodes estimate will approximate the number of rows actually returned, updated, or deleted by the query.
- **width** — total bytes of all the rows output by this plan node.

It is important to note that the cost of an upper-level node includes the cost of all its child nodes. The topmost node of the plan has the estimated total execution cost for the plan. This is this number that the planner seeks to minimize. It is also important to realize that the cost only reflects things that the query optimizer cares about. In particular, the cost does not consider the time spent transmitting result rows to the client.

`EXPLAIN ANALYZE` causes the statement to be actually executed, not only planned. The `EXPLAIN ANALYZE` plan shows the actual results along with the planner's estimates. This is useful for seeing whether the planner's estimates are close to reality. In addition to the information shown in the `EXPLAIN` plan, `EXPLAIN ANALYZE` will show the following additional information:

- The total elapsed time (in milliseconds) that it took to run the query.
- The number of *workers* (segments) involved in a plan node operation. Only segments that return rows are counted.
- The maximum number of rows returned by the segment that produced the most rows for an operation. If multiple segments produce an equal number of rows, the one with the longest *time to end* is the one chosen.
- The segment id number of the segment that produced the most rows for an operation.

- For relevant operations, the *work_mem* used by the operation. If `work_mem` was not sufficient to perform the operation in memory, the plan will show how much data was spilled to disk and how many passes over the data were required for the lowest performing segment. For example:

```
Work_mem used: 64K bytes avg, 64K bytes max (seg0).
Work_mem wanted: 90K bytes avg, 90K bytes max (seg0) to abate workfile
I/O affecting 2 workers.
[seg0] pass 0: 488 groups made from 488 rows; 263 rows written to
workfile
[seg0] pass 1: 263 groups made from 263 rows
```

- The time (in milliseconds) it took to retrieve the first row from the segment that produced the most rows, and the total time taken to retrieve all rows from that segment. The *<time> to first row* may be omitted if it is the same as the *<time> to end*.

  Important: Keep in mind that the statement is actually executed when `EXPLAIN ANALYZE` is used. Although `EXPLAIN ANALYZE` will discard any output that a `SELECT` would return, other side effects of the statement will happen as usual. If you wish to use `EXPLAIN ANALYZE` on a DML statement without letting the command affect your data, use this approach:

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

Parameters

name

The name of the prepared statement to execute.

parameter

The actual value of a parameter to the prepared statement. This must be an expression yielding a value that is compatible with the data type of this parameter, as was determined when the prepared statement was created.

Notes

In order to allow the query optimizer to make reasonably informed decisions when optimizing queries, the `ANALYZE` statement should be run to record statistics about the distribution of data within the table. If you have not done this (or if the statistical distribution of the data in the table has changed significantly since the last time `ANALYZE` was run), the estimated costs are unlikely to conform to the real properties of the query, and consequently an inferior query plan may be chosen.

For more information about query profiling, see "Query Profiling" in the *Greenplum Database Administrator Guide*.

Examples

To illustrate how to read an `EXPLAIN` query plan, consider the following example for a very simple query:

```
EXPLAIN SELECT * FROM names WHERE name = 'Joelle';
                     QUERY PLAN
------------------------------------------------------------
Gather Motion 2:1 (slice1) (cost=0.00..20.88 rows=1 width=13)

   -> Seq Scan on 'names' (cost=0.00..20.88 rows=1 width=13)
         Filter: name::text ~~ 'Joelle'::text
```

If we read the plan from the bottom up, the query optimizer starts by doing a sequential scan of the `names` table. Notice that the `WHERE` clause is being applied as a *filter* condition. This means that the scan operation checks the condition for each row it scans, and outputs only the ones that pass the condition.

The results of the scan operation are passed up to a *gather motion* operation. In Greenplum Database, a gather motion is when segments send rows up to the master. In this case we have 2 segment instances sending to 1 master instance (2:1). This operation is working on `slice1` of the parallel query execution plan. In Greenplum Database a query plan is divided into *slices* so that portions of the query plan can be worked on in parallel by the segments.

The estimated startup cost for this plan is `00.00` (no cost) and a total cost of `20.88` disk page fetches. The planner is estimating that this query will return one row.

Compatibility

There is no `EXPLAIN` statement defined in the SQL standard.

See Also

ANALYZE

FETCH

Retrieves rows from a query using a cursor.

Synopsis

```
FETCH [ forward_direction { FROM | IN } ] cursorname
```

where *forward_direction* can be empty or one of:

```
    NEXT
    FIRST
    LAST
    ABSOLUTE count
    RELATIVE count
    count
    ALL
    FORWARD
    FORWARD count
    FORWARD ALL
```

Description

`FETCH` retrieves rows using a previously-created cursor.

A cursor has an associated position, which is used by `FETCH`. The cursor position can be before the first row of the query result, on any particular row of the result, or after the last row of the result. When created, a cursor is positioned before the first row. After fetching some rows, the cursor is positioned on the row most recently retrieved. If `FETCH` runs off the end of the available rows then the cursor is left positioned after the last row. `FETCH ALL` will always leave the cursor positioned after the last row.

The forms `NEXT`, `FIRST`, `LAST`, `ABSOLUTE`, `RELATIVE` fetch a single row after moving the cursor appropriately. If there is no such row, an empty result is returned, and the cursor is left positioned before the first row or after the last row as appropriate.

The forms using `FORWARD` retrieve the indicated number of rows moving in the forward direction, leaving the cursor positioned on the last-returned row (or after all rows, if the count exceeds the number of rows available). Note that it is not possible to move a cursor position backwards in Greenplum Database, since scrollable cursors are not supported. You can only move a cursor forward in position using `FETCH`.

`RELATIVE 0` and `FORWARD 0` request fetching the current row without moving the cursor, that is, re-fetching the most recently fetched row. This will succeed unless the cursor is positioned before the first row or after the last row, in which case no row is returned.

Outputs

On successful completion, a `FETCH` command returns a command tag of the form

```
FETCH count
```

The count is the number of rows fetched (possibly zero). Note that in `psql`, the command tag will not actually be displayed, since `psql` displays the fetched rows instead.

Parameters

forward_direction

Defines the fetch direction and number of rows to fetch. Only forward fetches are allowed in Greenplum Database. It can be one of the following:

NEXT

Fetch the next row. This is the default if direction is omitted.

FIRST

Fetch the first row of the query (same as `ABSOLUTE 1`). Only allowed if it is the first `FETCH` operation using this cursor.

LAST

Fetch the last row of the query (same as `ABSOLUTE -1`).

ABSOLUTE *count*

Fetch the specified row of the query. Position after last row if count is out of range. Only allowed if the row specified by *count* moves the cursor position forward.

RELATIVE *count*

Fetch the specified row of the query *count* rows ahead of the current cursor position. `RELATIVE 0` re-fetches the current row, if any. Only allowed if *count* moves the cursor position forward.

count

Fetch the next *count* number of rows (same as `FORWARD count`).

ALL

Fetch all remaining rows (same as `FORWARD ALL`).

FORWARD

Fetch the next row (same as `NEXT`).

FORWARD *count*

Fetch the next *count* number of rows. `FORWARD 0` re-fetches the current row.

FORWARD ALL

Fetch all remaining rows.

cursorname

The name of an open cursor.

Notes

Greenplum Database does not support scrollable cursors, so you can only use `FETCH` to move the cursor position forward.

`ABSOLUTE` fetches are not any faster than navigating to the desired row with a relative move: the underlying implementation must traverse all the intermediate rows anyway.

Updating data via a cursor is currently not supported by Greenplum Database.

`DECLARE` is used to define a cursor. Use `MOVE` to change cursor position without retrieving data.

Examples

-- Start the transaction:

```
BEGIN;
```

-- Set up a cursor:

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

-- Fetch the first 5 rows in the cursor `mycursor`:

```
FETCH FORWARD 5 FROM mycursor;
 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
```

-- Close the cursor and end the transaction:

```
CLOSE mycursor;
COMMIT;
```

Compatibility

SQL standard allows cursors only in embedded SQL and in modules. Greenplum Database permits cursors to be used interactively.

The variant of `FETCH` described here returns the data as if it were a `SELECT` result rather than placing it in host variables. Other than this point, `FETCH` is fully upward-compatible with the SQL standard.

The `FETCH` forms involving `FORWARD`, as well as the forms `FETCH` count and `FETCHALL`, in which `FORWARD` is implicit, are Greenplum Database extensions. `BACKWARD` is not supported.

The SQL standard allows only `FROM` preceding the cursor name; the option to use `IN` is an extension.

See Also

DECLARE, *CLOSE*, *MOVE*

GRANT

Defines access privileges.

Synopsis

```
GRANT { {SELECT | INSERT | UPDATE | DELETE | REFERENCES |
TRIGGER | TRUNCATE } [,...] | ALL [PRIVILEGES] }
    ON [TABLE] tablename [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { {USAGE | SELECT | UPDATE} [,...] | ALL [PRIVILEGES] }
    ON SEQUENCE sequencename [, ...]
    TO { rolename | PUBLIC } [, ...] [WITH GRANT OPTION]

GRANT { {CREATE | CONNECT | TEMPORARY | TEMP} [,...] | ALL
[PRIVILEGES] }
    ON DATABASE dbname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { EXECUTE | ALL [PRIVILEGES] }
    ON FUNCTION funcname ( [ [argmode] [argname] argtype [, ...]
] ) [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { USAGE | ALL [PRIVILEGES] }
    ON LANGUAGE langname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { {CREATE | USAGE} [,...] | ALL [PRIVILEGES] }
    ON SCHEMA schemaname [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT { CREATE | ALL [PRIVILEGES] }
    ON TABLESPACE tablespacename [, ...]
    TO {rolename | PUBLIC} [, ...] [WITH GRANT OPTION]

GRANT parent_role [, ...]
    TO member_role [, ...] [WITH ADMIN OPTION]

GRANT { SELECT | INSERT | ALL [PRIVILEGES] }
    ON PROTOCOL protocolname
    TO username
```

Description

The `GRANT` command has two basic variants: one that grants privileges on a database object (table, view, sequence, database, function, procedural language, schema, or tablespace), and one that grants membership in a role.

GRANT on Database Objects

This variant of the `GRANT` command gives specific privileges on a database object to one or more roles. These privileges are added to those already granted, if any.

The key word `PUBLIC` indicates that the privileges are to be granted to all roles, including those that may be created later. `PUBLIC` may be thought of as an implicitly defined group-level role that always includes all roles. Any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to `PUBLIC`.

If `WITH GRANT OPTION` is specified, the recipient of the privilege may in turn grant it to others. Without a grant option, the recipient cannot do that. Grant options cannot be granted to `PUBLIC`.

There is no need to grant privileges to the owner of an object (usually the role that created it), as the owner has all privileges by default. The right to drop an object, or to alter its definition in any way is not described by a grantable privilege; it is inherent in the owner, and cannot be granted or revoked. The owner implicitly has all grant options for the object, too.

Depending on the type of object, the initial default privileges may include granting some privileges to `PUBLIC`. The default is no public access for tables, schemas, and tablespaces; `CONNECT` privilege and `TEMP` table creation privilege for databases; `EXECUTE` privilege for functions; and `USAGE` privilege for languages. The object owner may of course revoke these privileges.

GRANT on Roles

This variant of the `GRANT` command grants membership in a role to one or more other roles. Membership in a role is significant because it conveys the privileges granted to a role to each of its members.

If `WITH ADMIN OPTION` is specified, the member may in turn grant membership in the role to others, and revoke membership in the role as well. Database superusers can grant or revoke membership in any role to anyone. Roles having `CREATEROLE` privilege can grant or revoke membership in any role that is not a superuser.

Unlike the case with privileges, membership in a role cannot be granted to `PUBLIC`.

GRANT on Protocols

After creating a custom protocol, specify `CREATE TRUSTED PROTOCOL` to be able to allowing any user besides the owner to access it. If the protocol is not trusted, you cannot give any other user permission to use it to read or write data. After a `TRUSTED` protocol is created, you can specify which other users can access it with the `GRANT` command.

- To allow a user to create a readable external table with a trusted protocol

```
GRANT SELECT ON PROTOCOL protocolname TO username
```

- To allow a user to create a writable external table with a trusted protocol

```
GRANT INSERT ON PROTOCOL protocolname TO username
```

- To allow a user to create both readable and writable external table with a trusted protocol

```
GRANT ALL ON PROTOCOL protocolname TO username
```

Parameters

SELECT

Allows `SELECT` from any column of the specified table, view, or sequence. Also allows the use of `COPY TO`. For sequences, this privilege also allows the use of the `currval` function.

INSERT

Allows `INSERT` of a new row into the specified table. Also allows `COPY FROM`.

UPDATE

Allows `UPDATE` of any column of the specified table. `SELECT ... FOR UPDATE` and `SELECT ... FOR SHARE` also require this privilege (as well as the `SELECT` privilege). For sequences, this privilege allows the use of the `nextval` and `setval` functions.

DELETE

Allows `DELETE` of a row from the specified table.

REFERENCES

This keyword is accepted, although foreign key constraints are currently not supported in Greenplum Database. To create a foreign key constraint, it is necessary to have this privilege on both the referencing and referenced tables.

TRIGGER

Allows the creation of a trigger on the specified table.

Note: Greenplum Database does not support triggers.

TRUNCATE

Allows `TRUNCATE` of all rows from the specified table.

CREATE

For databases, allows new schemas to be created within the database.

For schemas, allows new objects to be created within the schema. To rename an existing object, you must own the object and have this privilege for the containing schema.

For tablespaces, allows tables and indexes to be created within the tablespace, and allows databases to be created that have the tablespace as their default tablespace. (Note that revoking this privilege will not alter the placement of existing objects.)

CONNECT

Allows the user to connect to the specified database. This privilege is checked at connection startup (in addition to checking any restrictions imposed by `pg_hba.conf`).

TEMPORARY
TEMP

Allows temporary tables to be created while using the database.

EXECUTE

Allows the use of the specified function and the use of any operators that are implemented on top of the function. This is the only type of privilege that is applicable to functions. (This syntax works for aggregate functions, as well.)

USAGE

For procedural languages, allows the use of the specified language for the creation of functions in that language. This is the only type of privilege that is applicable to procedural languages.

For schemas, allows access to objects contained in the specified schema (assuming that the objects' own privilege requirements are also met). Essentially this allows the grantee to look up objects within the schema.

For sequences, this privilege allows the use of the `currval` and `nextval` functions.

ALL PRIVILEGES

Grant all of the available privileges at once. The `PRIVILEGES` key word is optional in Greenplum Database, though it is required by strict SQL.

PUBLIC

A special group-level role that denotes that the privileges are to be granted to all roles, including those that may be created later.

WITH GRANT OPTION

The recipient of the privilege may in turn grant it to others.

WITH ADMIN OPTION

The member of a role may in turn grant membership in the role to others.

Notes

Database superusers can access all objects regardless of object privilege settings. One exception to this rule is view objects. Access to tables referenced in the view is determined by permissions of the view owner not the current user (even if the current user is a superuser).

If a superuser chooses to issue a `GRANT` or `REVOKE` command, the command is performed as though it were issued by the owner of the affected object. In particular, privileges granted via such a command will appear to have been granted by the object owner. For role membership, the membership appears to have been granted by the containing role itself.

`GRANT` and `REVOKE` can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges `WITH GRANT OPTION` on the object. In this case the privileges will be recorded as having been granted by the role that actually owns the object or holds the privileges `WITH GRANT OPTION`.

Granting permission on a table does not automatically extend permissions to any sequences used by the table, including sequences tied to `SERIAL` columns. Permissions on a sequence must be set separately.

Greenplum Database does not support granting or revoking privileges for individual columns of a table. One possible workaround is to create a view having just the desired columns and then grant privileges to that view.

Use psql's `\z` meta-command to obtain information about existing privileges for an object.

Examples

Grant insert privilege to all roles on table `mytable`:

```
GRANT INSERT ON mytable TO PUBLIC;
```

Grant all available privileges to role `sally` on the view `topten`. Note that while the above will indeed grant all privileges if executed by a superuser or the owner of `topten`, when executed by someone else it will only grant those permissions for which the granting role has grant options.

```
GRANT ALL PRIVILEGES ON topten TO sally;
```

Grant membership in role `admins` to user `joe`:

```
GRANT admins TO joe;
```

Compatibility

The `PRIVILEGES` key word in is required in the SQL standard, but optional in Greenplum Database. The SQL standard does not support setting the privileges on more than one object per command.

Greenplum Database allows an object owner to revoke his own ordinary privileges: for example, a table owner can make the table read-only to himself by revoking his own `INSERT`, `UPDATE`, `DELETE`, and `TRUNCATE` privileges. This is not possible according to the SQL standard. Greenplum Database treats the owner's privileges as having been granted by the owner to himself; therefore he can revoke them too. In the SQL standard, the owner's privileges are granted by an assumed *system* entity.

The SQL standard allows setting privileges for individual columns within a table.

The SQL standard provides for a `USAGE` privilege on other kinds of objects: character sets, collations, translations, domains.

Privileges on databases, tablespaces, schemas, and languages are Greenplum Database extensions.

See Also

REVOKE

INSERT

Creates new rows in a table.

Synopsis

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

Description

`INSERT` inserts new rows into a table. One can insert one or more rows specified by value expressions, or zero or more rows resulting from a query.

The target column names may be listed in any order. If no list of column names is given at all, the default is the columns of the table in their declared order. The values supplied by the `VALUES` clause or query are associated with the explicit or implicit column list left-to-right.

Each column not present in the explicit or implicit column list will be filled with a default value, either its declared default value or null if there is no default.

If the expression for any column is not of the correct data type, automatic type conversion will be attempted.

You must have `INSERT` privilege on a table in order to insert into it.

Outputs

On successful completion, an `INSERT` command returns a command tag of the form:

```
INSERT oid count
```

The *count* is the number of rows inserted. If count is exactly one, and the target table has OIDs, then *oid* is the OID assigned to the inserted row. Otherwise *oid* is zero.

Parameters

table

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

column

The name of a column in table. The column name can be qualified with a subfield name or array subscript, if needed. (Inserting into only some fields of a composite column leaves the other fields null.)

DEFAULT VALUES

All columns will be filled with their default values.

expression

An expression or value to assign to the corresponding column.

DEFAULT

The corresponding column will be filled with its default value.

query

A query (`SELECT` statement) that supplies the rows to be inserted. Refer to the `SELECT` statement for a description of the syntax.

Notes

To insert data into a partitioned table, you specify the root partitioned table, the table created with the `CREATE TABLE` command. You also can specify a leaf child table of the partitioned table in an `INSERT` command. An error is returned if the data is not valid for the specified leaf child table. Specifying a child table that is not a leaf child table in the `INSERT` command is not supported. Execution of other DML commands such as `UPDATE` and `DELETE` on any child table of a partitioned table is not supported. These commands must be executed on the root partitioned table, the table created with the `CREATE TABLE` command.

For append-optimized tables, Greenplum Database supports a maximum of 127 concurrent `INSERT` transactions into a single append-optimized table.

Examples

Insert a single row into table `films`:

```
INSERT INTO films VALUES ('UA502', 'Bananas', 105,
'1971-07-13', 'Comedy', '82 minutes');
```

In this example, the `length` column is omitted and therefore it will have the default value:

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

This example uses the `DEFAULT` clause for the `date_prod` column rather than specifying a value:

```
INSERT INTO films VALUES ('UA502', 'Bananas', 105, DEFAULT,
'Comedy', '82 minutes');
```

To insert a row consisting entirely of default values:

```
INSERT INTO films DEFAULT VALUES;
```

To insert multiple rows using the multirow `VALUES` syntax:

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

This example inserts some rows into table `films` from a table `tmp_films` with the same column layout as `films`:

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod <
'2004-05-07';
```

Compatibility

`INSERT` conforms to the SQL standard. The case in which a column name list is omitted, but not all the columns are filled from the `VALUES` clause or query, is disallowed by the standard.

Possible limitations of the *query* clause are documented under `SELECT`.

See Also

`COPY`, *`SELECT`*, *`CREATE EXTERNAL TABLE`*

LOAD

Loads or reloads a shared library file.

Synopsis

```
LOAD 'filename'
```

Description

This command loads a shared library file into the Greenplum Database server address space. If the file had been loaded previously, it is first unloaded. This command is primarily useful to unload and reload a shared library file that has been changed since the server first loaded it. To make use of the shared library, function(s) in it need to be declared using the `CREATE FUNCTION` command.

The file name is specified in the same way as for shared library names in `CREATE FUNCTION`; in particular, one may rely on a search path and automatic addition of the system's standard shared library file name extension.

Note that in Greenplum Database the shared library file (`.so` file) must reside in the same path location on every host in the Greenplum Database array (masters, segments, and mirrors).

Only database superusers can load shared library files.

Parameters

filename

The path and file name of a shared library file. This file must exist in the same location on all hosts in your Greenplum Database array.

Examples

Load a shared library file:

```
LOAD '/usr/local/greenplum-db/lib/myfuncs.so';
```

Compatibility

`LOAD` is a Greenplum Database extension.

See Also

CREATE FUNCTION

LOCK

Locks a table.

Synopsis

```
LOCK [TABLE] name [, ...] [IN lockmode MODE] [NOWAIT]
```

where *lockmode* is one of:

```
    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
  | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

Description

`LOCK TABLE` obtains a table-level lock, waiting if necessary for any conflicting locks to be released. If `NOWAIT` is specified, `LOCK TABLE` does not wait to acquire the desired lock: if it cannot be acquired immediately, the command is aborted and an error is emitted. Once obtained, the lock is held for the remainder of the current transaction. There is no `UNLOCK TABLE` command; locks are always released at transaction end.

When acquiring locks automatically for commands that reference tables, Greenplum Database always uses the least restrictive lock mode possible. `LOCK TABLE` provides for cases when you might need more restrictive locking. For example, suppose an application runs a transaction at the *Read Committed* isolation level and needs to ensure that data in a table remains stable for the duration of the transaction. To achieve this you could obtain `SHARE` lock mode over the table before querying. This will prevent concurrent data changes and ensure subsequent reads of the table see a stable view of committed data, because `SHARE` lock mode conflicts with the `ROW EXCLUSIVE` lock acquired by writers, and your `LOCK TABLE name IN SHARE MODE` statement will wait until any concurrent holders of `ROW EXCLUSIVE` mode locks commit or roll back. Thus, once you obtain the lock, there are no uncommitted writes outstanding; furthermore none can begin until you release the lock.

To achieve a similar effect when running a transaction at the *Serializable* isolation level, you have to execute the `LOCK TABLE` statement before executing any `SELECT` or data modification statement. A serializable transaction's view of data will be frozen when its first `SELECT` or data modification statement begins. A `LOCK TABLE` later in the transaction will still prevent concurrent writes — but it won't ensure that what the transaction reads corresponds to the latest committed values.

If a transaction of this sort is going to change the data in the table, then it should use `SHARE ROW EXCLUSIVE` lock mode instead of `SHARE` mode. This ensures that only one transaction of this type runs at a time. Without this, a deadlock is possible: two transactions might both acquire `SHARE` mode, and then be unable to also acquire `ROW EXCLUSIVE` mode to actually perform their updates. Note that a transaction's own locks never conflict, so a transaction can acquire `ROW EXCLUSIVE` mode when it holds `SHARE` mode — but not if anyone else holds `SHARE` mode. To avoid deadlocks, make sure all transactions acquire locks on the same objects in the same order, and if multiple lock modes are involved for a single object, then transactions should always acquire the most restrictive mode first.

Parameters

name

The name (optionally schema-qualified) of an existing table to lock.

If multiple tables are given, tables are locked one-by-one in the order specified in the `LOCK TABLE` command.

lockmode

The lock mode specifies which locks this lock conflicts with. If no lock mode is specified, then ACCESS EXCLUSIVE, the most restrictive mode, is used. Lock modes are as follows:

- ACCESS SHARE — Conflicts with the ACCESS EXCLUSIVE lock mode only. The commands SELECT and ANALYZE automatically acquire a lock of this mode on referenced tables. In general, any query that only reads a table and does not modify it will acquire this lock mode.
- ROW SHARE — Conflicts with the EXCLUSIVE and ACCESS EXCLUSIVE lock modes. The SELECT FOR UPDATE and SELECT FOR SHARE commands automatically acquire a lock of this mode on the target table(s) (in addition to ACCESS SHARE locks on any other tables that are referenced but not selected FOR UPDATE/FOR SHARE).
- ROW EXCLUSIVE — Conflicts with the SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. The commands INSERT and COPY automatically acquire this lock mode on the target table (in addition to ACCESS SHARE locks on any other referenced tables).
- SHARE UPDATE EXCLUSIVE — Conflicts with the SHARE UPDATEEXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode protects a table against concurrent schema changes and VACUUM runs. Acquired automatically by VACUUM (without FULL).
- SHARE — Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode protects a table against concurrent data changes. Acquired automatically by CREATE INDEX.
- SHARE ROW EXCLUSIVE — Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This lock mode is not automatically acquired by any Greenplum Database command.
- EXCLUSIVE — Conflicts with the ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode allows only concurrent ACCESS SHARE locks, i.e., only reads from the table can proceed in parallel with a transaction holding this lock mode. This lock mode is automatically acquired for UPDATE and DELETE in Greenplum Database (which is more restrictive locking than in regular PostgreSQL).
- ACCESS EXCLUSIVE — Conflicts with locks of all modes (ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHAREROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE). This mode guarantees that the holder is the only transaction accessing the table in any way. Acquired automatically by the ALTER TABLE, DROP TABLE, REINDEX, CLUSTER, and VACUUM FULL commands. This is also the default lock mode for LOCK TABLE statements that do not specify a mode explicitly.

NOWAIT

Specifies that LOCK TABLE should not wait for any conflicting locks to be released: if the specified lock(s) cannot be acquired immediately without waiting, the transaction is aborted.

Notes

LOCK TABLE ... IN ACCESS SHARE MODE requires SELECT privileges on the target table. All other forms of LOCK require UPDATE and/or DELETE privileges.

LOCK TABLE is useful only inside a transaction block (BEGIN/COMMIT pair), since the lock is dropped as soon as the transaction ends. A LOCK TABLE command appearing outside any transaction block forms a self-contained transaction, so the lock will be dropped as soon as it is obtained.

LOCK TABLE only deals with table-level locks, and so the mode names involving ROW are all misnomers. These mode names should generally be read as indicating the intention of the user to acquire row-level locks within the locked table. Also, ROW EXCLUSIVE mode is a sharable table lock. Keep in mind that all the lock modes have identical semantics so far as LOCK TABLE is concerned, differing only in the rules about which modes conflict with which. For information on how to acquire an actual row-level lock, see the `FOR UPDATE/FOR SHARE` clause in the *SELECT* reference documentation.

Examples

Obtain a `SHARE` lock on the `films` table when going to perform inserts into the `films_user_comments` 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 table when performing 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

There is no `LOCK TABLE` in the SQL standard, which instead uses `SET TRANSACTION` to specify concurrency levels on transactions. Greenplum Database supports that too.

Except for `ACCESS SHARE`, `ACCESS EXCLUSIVE`, and `SHARE UPDATE EXCLUSIVE` lock modes, the Greenplum Database lock modes and the `LOCK TABLE` syntax are compatible with those present in Oracle.

See Also

BEGIN, *SET TRANSACTION*, *SELECT*

MOVE

Positions a cursor.

Synopsis

```
MOVE [ forward_direction {FROM | IN} ] cursorname
```

where *forward_direction* can be empty or one of:

```
    NEXT
    FIRST
    LAST
    ABSOLUTE count
    RELATIVE count
    count
    ALL
    FORWARD
    FORWARD count
    FORWARD ALL
```

Description

`MOVE` repositions a cursor without retrieving any data. `MOVE` works exactly like the *`FETCH`* command, except it only positions the cursor and does not return rows.

Note that it is not possible to move a cursor position backwards in Greenplum Database, since scrollable cursors are not supported. You can only move a cursor forward in position using `MOVE`.

Outputs

On successful completion, a `MOVE` command returns a command tag of the form

```
MOVE count
```

The count is the number of rows that a `FETCH` command with the same parameters would have returned (possibly zero).

Parameters

forward_direction

See *`FETCH`* for more information.

cursorname

The name of an open cursor.

Examples

-- Start the transaction:

```
BEGIN;
```

-- Set up a cursor:

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

-- Move forward 5 rows in the cursor `mycursor`:

```
MOVE FORWARD 5 IN mycursor;
```

```
MOVE 5
```

--Fetch the next row after that (row 6):

```
FETCH 1 FROM mycursor;
 code  | title  | did | date_prod  |  kind  |  len
-------+--------+-----+------------+--------+-------
 P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)
```

-- Close the cursor and end the transaction:

```
CLOSE mycursor;
COMMIT;
```

Compatibility

There is no `MOVE` statement in the SQL standard.

See Also

DECLARE, *FETCH*, *CLOSE*

PREPARE

Prepare a statement for execution.

Synopsis

```
PREPARE name [ (datatype [, ...] ) ] AS statement
```

Description

`PREPARE` creates a prepared statement, possibly with unbound parameters. A prepared statement is a server-side object that can be used to optimize performance. A prepared statement may be subsequently executed with a binding for its parameters. Greenplum Database may choose to replan the query for different executions of the same prepared statement.

Prepared statements can take parameters: values that are substituted into the statement when it is executed. When creating the prepared statement, refer to parameters by position, using `$1`, `$2`, etc. A corresponding list of parameter data types can optionally be specified. When a parameter's data type is not specified or is declared as unknown, the type is inferred from the context in which the parameter is used (if possible). When executing the statement, specify the actual values for these parameters in the `EXECUTE` statement.

Prepared statements only last for the duration of the current database session. When the session ends, the prepared statement is forgotten, so it must be recreated before being used again. This also means that a single prepared statement cannot be used by multiple simultaneous database clients; however, each client can create their own prepared statement to use. The prepared statement can be manually cleaned up using the *`DEALLOCATE`* command.

Prepared statements have the largest performance advantage when a single session is being used to execute a large number of similar statements. The performance difference will be particularly significant if the statements are complex to plan or rewrite, for example, if the query involves a join of many tables or requires the application of several rules. If the statement is relatively simple to plan and rewrite but relatively expensive to execute, the performance advantage of prepared statements will be less noticeable.

Parameters

name

An arbitrary name given to this particular prepared statement. It must be unique within a single session and is subsequently used to execute or deallocate a previously prepared statement.

datatype

The data type of a parameter to the prepared statement. If the data type of a particular parameter is unspecified or is specified as unknown, it will be inferred from the context in which the parameter is used. To refer to the parameters in the prepared statement itself, use `$1`, `$2`, etc.

statement

Any `SELECT`, `INSERT`, `UPDATE`, `DELETE`, or `VALUES` statement.

Notes

In some situations, the query plan produced for a prepared statement will be inferior to the query plan that would have been chosen if the statement had been submitted and executed normally. This is because when the statement is planned and the planner attempts to determine the optimal query plan, the actual values of any parameters specified in the statement are unavailable. Greenplum Database collects statistics on the distribution of data in the table, and can use constant values in a statement to make guesses about the likely result of executing the statement. Since this data is unavailable when planning prepared statements with parameters, the chosen plan may be suboptimal. To examine the query plan Greenplum Database has chosen for a prepared statement, use `EXPLAIN`.

For more information on query planning and the statistics collected by Greenplum Database for that purpose, see the `ANALYZE` documentation.

You can see all available prepared statements of a session by querying the `pg_prepared_statements` system view.

Examples

Create a prepared statement for an `INSERT` statement, and then execute it:

```
PREPARE fooplan (int, text, bool, numeric) AS INSERT INTO
foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

Create a prepared statement for a `SELECT` statement, and then execute it. Note that the data type of the second parameter is not specified, so it is inferred from the context in which `$2` is used:

```
PREPARE usrrptplan (int) AS SELECT * FROM users u, logs l
WHERE u.usrid=$1 AND u.usrid=l.usrid AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

Compatibility

The SQL standard includes a `PREPARE` statement, but it is only for use in embedded SQL. This version of the `PREPARE` statement also uses a somewhat different syntax.

See Also

EXECUTE, *DEALLOCATE*

REASSIGN OWNED

Changes the ownership of database objects owned by a database role.

Synopsis

```
REASSIGN OWNED BY old_role [, ...] TO new_role
```

Description

`REASSIGN OWNED` reassigns all the objects in the current database that are owned by *old_row* to *new_role*. Note that it does not change the ownership of the database itself.

Parameters

old_role

The name of a role. The ownership of all the objects in the current database owned by this role will be reassigned to *new_role*.

new_role

The name of the role that will be made the new owner of the affected objects.

Notes

`REASSIGN OWNED` is often used to prepare for the removal of one or more roles. Because `REASSIGN OWNED` only affects the objects in the current database, it is usually necessary to execute this command in each database that contains objects owned by a role that is to be removed.

The `DROP OWNED` command is an alternative that drops all the database objects owned by one or more roles.

The `REASSIGN OWNED` command does not affect the privileges granted to the old roles in objects that are not owned by them. Use `DROP OWNED` to revoke those privileges.

Examples

Reassign any database objects owned by the role named `sally` and `bob` to `admin`;

```
REASSIGN OWNED BY sally, bob TO admin;
```

Compatibility

The `REASSIGN OWNED` statement is a Greenplum Database extension.

See Also

DROP OWNED, *DROP ROLE*

REINDEX

Rebuilds indexes.

Synopsis

```
REINDEX {INDEX | TABLE | DATABASE | SYSTEM} name
```

Description

`REINDEX` rebuilds an index using the data stored in the index's table, replacing the old copy of the index. There are several scenarios in which to use `REINDEX`:

- An index has become bloated, that it is contains many empty or nearly-empty pages. This can occur with B-tree indexes in Greenplum Database under certain uncommon access patterns. `REINDEX` provides a way to reduce the space consumption of the index by writing a new version of the index without the dead pages.
- You have altered the fillfactor storage parameter for an index, and wish to ensure that the change has taken full effect.

Parameters

INDEX

Recreate the specified index.

TABLE

Recreate all indexes of the specified table. If the table has a secondary TOAST table, that is reindexed as well.

DATABASE

Recreate all indexes within the current database. Indexes on shared system catalogs are skipped. This form of `REINDEX` cannot be executed inside a transaction block.

SYSTEM

Recreate all indexes on system catalogs within the current database. Indexes on user tables are not processed. Also, indexes on shared (global) system catalogs are skipped. This form of `REINDEX` cannot be executed inside a transaction block.

name

The name of the specific index, table, or database to be reindexed. Index and table names may be schema-qualified. Presently, `REINDEX DATABASE` and `REINDEX SYSTEM` can only reindex the current database, so their parameter must match the current database's name.

Notes

`REINDEX` is similar to a drop and recreate of the index in that the index contents are rebuilt from scratch. However, the locking considerations are rather different. `REINDEX` locks out writes but not reads of the index's parent table. It also takes an exclusive lock on the specific index being processed, which will block reads that attempt to use that index. In contrast, `DROP INDEX` momentarily takes exclusive lock on the parent table, blocking both writes and reads. The subsequent `CREATE INDEX` locks out writes but not reads; since the index is not there, no read will attempt to use it, meaning that there will be no blocking but reads may be forced into expensive sequential scans. Another important point is that the drop/create approach invalidates any cached query plans that use the index, while `REINDEX` does not.

Reindexing a single index or table requires being the owner of that index or table. Reindexing a database requires being the owner of the database (note that the owner can therefore rebuild indexes of tables owned by other users). Of course, superusers can always reindex anything.

If you suspect that shared global system catalog indexes are corrupted, they can only be reindexed in Greenplum utility mode. The typical symptom of a corrupt shared index is "index is not a btree" errors, or else the server crashes immediately at startup due to reliance on the corrupted indexes. Contact Greenplum Customer Support for assistance in this situation.

Examples

Rebuild a single index:

```
REINDEX INDEX my_index;
```

Rebuild all the indexes on the table `my_table`:

```
REINDEX TABLE my_table;
```

Compatibility

There is no `REINDEX` command in the SQL standard.

See Also

CREATE INDEX, *DROP INDEX*, *VACUUM*

RELEASE SAVEPOINT

Destroys a previously defined savepoint.

Synopsis

```
RELEASE [SAVEPOINT] savepoint_name
```

Description

`RELEASE SAVEPOINT` destroys a savepoint previously defined in the current transaction.

Destroying a savepoint makes it unavailable as a rollback point, but it has no other user visible behavior. It does not undo the effects of commands executed after the savepoint was established. (To do that, see *ROLLBACK TO SAVEPOINT*.) Destroying a savepoint when it is no longer needed may allow the system to reclaim some resources earlier than transaction end.

`RELEASE SAVEPOINT` also destroys all savepoints that were established *after* the named savepoint was established.

Parameters

savepoint_name

The name of the savepoint to destroy.

Examples

To establish and later destroy a savepoint:

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

The above transaction will insert both 3 and 4.

Compatibility

This command conforms to the SQL standard. The standard specifies that the key word `SAVEPOINT` is mandatory, but Greenplum Database allows it to be omitted.

See Also

BEGIN, *SAVEPOINT*, *ROLLBACK TO SAVEPOINT*, *COMMIT*

RESET

Restores the value of a system configuration parameter to the default value.

Synopsis

```
RESET configuration_parameter

RESET ALL
```

Description

`RESET` restores system configuration parameters to their default values. `RESET` is an alternative spelling for `SET configuration_parameter TO DEFAULT`.

The default value is defined as the value that the parameter would have had, had no `SET` ever been issued for it in the current session. The actual source of this value might be a compiled-in default, the master `postgresql.conf` configuration file, command-line options, or per-database or per-user default settings. See *Server Configuration Parameters* for more information.

Parameters

configuration_parameter

The name of a system configuration parameter. See *Server Configuration Parameters* for details.

ALL

Resets all settable configuration parameters to their default values.

Examples

Set the `statement_mem` configuration parameter to its default value:

```
RESET statement_mem;
```

Compatibility

`RESET` is a Greenplum Database extension.

See Also

SET

REVOKE

Removes access privileges.

Synopsis

```
REVOKE [GRANT OPTION FOR] { {SELECT | INSERT | UPDATE | DELETE
       | REFERENCES | TRIGGER | TRUNCATE } [,...] | ALL [PRIVILEGES] }
       ON [TABLE] tablename [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { {USAGE | SELECT | UPDATE} [,...]
       | ALL [PRIVILEGES] }
       ON SEQUENCE sequencename [, ...]
       FROM { rolename | PUBLIC } [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { {CREATE | CONNECT
       | TEMPORARY | TEMP} [,...] | ALL [PRIVILEGES] }
       ON DATABASE dbname [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] {EXECUTE | ALL [PRIVILEGES]}
       ON FUNCTION funcname ( [[argmode] [argname] argtype
                              [, ...]] ) [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] {USAGE | ALL [PRIVILEGES]}
       ON LANGUAGE langname [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [ CASCADE | RESTRICT ]

REVOKE [GRANT OPTION FOR] { {CREATE | USAGE} [,...]
       | ALL [PRIVILEGES] }
       ON SCHEMA schemaname [, ...]
       FROM {rolename | PUBLIC} [, ...]
       [CASCADE | RESTRICT]

REVOKE [GRANT OPTION FOR] { CREATE | ALL [PRIVILEGES] }
       ON TABLESPACE tablespacename [, ...]
       FROM { rolename | PUBLIC } [, ...]
       [CASCADE | RESTRICT]

REVOKE [ADMIN OPTION FOR] parent_role [, ...]
       FROM member_role [, ...]
       [CASCADE | RESTRICT]
```

Description

`REVOKE` command revokes previously granted privileges from one or more roles. The key word `PUBLIC` refers to the implicitly defined group of all roles.

See the description of the *GRANT* command for the meaning of the privilege types.

Note that any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to `PUBLIC`. Thus, for example, revoking `SELECT` privilege from `PUBLIC` does not necessarily mean that all roles have lost `SELECT` privilege on the object: those who have it granted directly or via another role will still have it.

If `GRANT OPTION FOR` is specified, only the grant option for the privilege is revoked, not the privilege itself. Otherwise, both the privilege and the grant option are revoked.

If a role holds a privilege with grant option and has granted it to other roles then the privileges held by those other roles are called dependent privileges. If the privilege or the grant option held by the first role is being revoked and dependent privileges exist, those dependent privileges are also revoked if `CASCADE` is specified, else the revoke action will fail. This recursive revocation only affects privileges that were granted through a chain of roles that is traceable to the role that is the subject of this `REVOKE` command. Thus, the affected roles may effectively keep the privilege if it was also granted through other roles.

When revoking membership in a role, `GRANT OPTION` is instead called `ADMIN OPTION`, but the behavior is similar.

Parameters

See *GRANT*.

Examples

Revoke insert privilege for the public on table `films`:

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

Revoke all privileges from role `sally` on view `topten`. Note that this actually means revoke all privileges that the current role granted (if not a superuser).

```
REVOKE ALL PRIVILEGES ON topten FROM sally;
```

Revoke membership in role `admins` from user `joe`:

```
REVOKE admins FROM joe;
```

Compatibility

The compatibility notes of the *GRANT* command also apply to `REVOKE`.

Either `RESTRICT` or `CASCADE` is required according to the standard, but Greenplum Database assumes `RESTRICT` by default.

See Also

GRANT

ROLLBACK

Aborts the current transaction.

Synopsis

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

Description

`ROLLBACK` rolls back the current transaction and causes all the updates made by the transaction to be discarded.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

Notes

Use `COMMIT` to successfully end the current transaction.

Issuing `ROLLBACK` when not inside a transaction does no harm, but it will provoke a warning message.

Examples

To discard all changes made in the current transaction:

```
ROLLBACK;
```

Compatibility

The SQL standard only specifies the two forms `ROLLBACK` and `ROLLBACK WORK`. Otherwise, this command is fully conforming.

See Also

BEGIN, *COMMIT*, *SAVEPOINT*, *ROLLBACK TO SAVEPOINT*

ROLLBACK TO SAVEPOINT

Rolls back the current transaction to a savepoint.

Synopsis

```
ROLLBACK [WORK | TRANSACTION] TO [SAVEPOINT] savepoint_name
```

Description

This command will roll back all commands that were executed after the savepoint was established. The savepoint remains valid and can be rolled back to again later, if needed.

`ROLLBACK TO SAVEPOINT` implicitly destroys all savepoints that were established after the named savepoint.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

savepoint_name

The name of a savepoint to roll back to.

Notes

Use `RELEASE SAVEPOINT` to destroy a savepoint without discarding the effects of commands executed after it was established.

Specifying a savepoint name that has not been established is an error.

Cursors have somewhat non-transactional behavior with respect to savepoints. Any cursor that is opened inside a savepoint will be closed when the savepoint is rolled back. If a previously opened cursor is affected by a `FETCH` command inside a savepoint that is later rolled back, the cursor position remains at the position that `FETCH` left it pointing to (that is, `FETCH` is not rolled back). Closing a cursor is not undone by rolling back, either. A cursor whose execution causes a transaction to abort is put in a can't-execute state, so while the transaction can be restored using `ROLLBACK TO SAVEPOINT`, the cursor can no longer be used.

Examples

To undo the effects of the commands executed after `my_savepoint` was established:

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

Cursor positions are not affected by a savepoint rollback:

```
BEGIN;
DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;
SAVEPOINT foo;
FETCH 1 FROM foo;
column
----------
        1
ROLLBACK TO SAVEPOINT foo;
FETCH 1 FROM foo;
column
----------
```

```
        2
COMMIT;
```

Compatibility

The SQL standard specifies that the key word `SAVEPOINT` is mandatory, but Greenplum Database (and Oracle) allow it to be omitted. SQL allows only `WORK`, not `TRANSACTION`, as a noise word after `ROLLBACK`. Also, SQL has an optional clause `AND [NO] CHAIN` which is not currently supported by Greenplum Database. Otherwise, this command conforms to the SQL standard.

See Also

BEGIN, *COMMIT*, *SAVEPOINT*, *RELEASE SAVEPOINT*, *ROLLBACK*

SAVEPOINT

Defines a new savepoint within the current transaction.

Synopsis

```
SAVEPOINT savepoint_name
```

Description

`SAVEPOINT` establishes a new savepoint within the current transaction.

A savepoint is a special mark inside a transaction that allows all commands that are executed after it was established to be rolled back, restoring the transaction state to what it was at the time of the savepoint.

Parameters

savepoint_name

The name of the new savepoint.

Notes

Use *ROLLBACK TO SAVEPOINT* to rollback to a savepoint. Use *RELEASE SAVEPOINT* to destroy a savepoint, keeping the effects of commands executed after it was established.

Savepoints can only be established when inside a transaction block. There can be multiple savepoints defined within a transaction.

Examples

To establish a savepoint and later undo the effects of all commands executed after it was established:

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

The above transaction will insert the values 1 and 3, but not 2.

To establish and later destroy a savepoint:

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

The above transaction will insert both 3 and 4.

Compatibility

SQL requires a savepoint to be destroyed automatically when another savepoint with the same name is established. In Greenplum Database, the old savepoint is kept, though only the more recent one will be used when rolling back or releasing. (Releasing the newer savepoint will cause the older one to again become accessible to *ROLLBACK TO SAVEPOINT* and *RELEASE SAVEPOINT*.) Otherwise, SAVEPOINT is fully SQL conforming.

See Also

BEGIN, *COMMIT*, *ROLLBACK*, *RELEASE SAVEPOINT*, *ROLLBACK TO SAVEPOINT*

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 grouping_element [, ...]]
  [HAVING condition [, ...]]
  [WINDOW window_name AS (window_specification)]
  [{UNION | INTERSECT | EXCEPT} [ALL] select]
  [ORDER BY expression [ASC | DESC | USING operator] [, ...]]
  [LIMIT {count | ALL}]
  [OFFSET start]
  [FOR {UPDATE | SHARE} [OF table_name [, ...]] [NOWAIT] [...]]
```

where *grouping_element* can be one of:

```
  ()
  expression
  ROLLUP (expression [,...])
  CUBE (expression [,...])
  GROUPING SETS ((grouping_element [, ...]))
```

where *window_specification* can be:

```
  [window_name]
  [PARTITION BY expression [, ...]]
  [ORDER BY expression [ASC | DESC | USING operator] [, ...]
     [{RANGE | ROWS}
          { UNBOUNDED PRECEDING
          | expression PRECEDING
          | CURRENT ROW
          | BETWEEN window_frame_bound AND window_frame_bound }]]
                    where window_frame_bound can be one of:
                        UNBOUNDED PRECEDING
                        expression PRECEDING
                        CURRENT ROW
                        expression FOLLOWING
                        UNBOUNDED FOLLOWING
```

where *from_item* can be one of:

```
[ONLY] table_name [[AS] alias [( column_alias [, ...] )]]
(select) [AS] alias [( column_alias [, ...] )]
function_name ( [argument [, ...]] ) [AS] alias
             [( column_alias [, ...]
                | column_definition [, ...] )]
function_name ( [argument [, ...]] ) AS
              ( column_definition [, ...] )
from_item [NATURAL] join_type from_item
          [ON join_condition | USING ( join_column [, ...] )]
```

Description

`SELECT` retrieves rows from zero or more tables. The general processing of `SELECT` is as follows:

1. All elements in the `FROM` list are computed. (Each element in the `FROM` list is a real or virtual table.) If more than one element is specified in the `FROM` list, they are cross-joined together.

2. If the WHERE clause is specified, all rows that do not satisfy the condition are eliminated from the output.
3. If the GROUP BY clause is specified, the output is divided into groups of rows that match on one or more of the defined grouping elements. If the HAVING clause is present, it eliminates groups that do not satisfy the given condition.
4. If a window expression is specified (and optional WINDOW clause), the output is organized according to the positional (row) or value-based (range) window frame.
5. DISTINCT eliminates duplicate rows from the result. DISTINCT ON eliminates rows that match on all the specified expressions. ALL (the default) will return all candidate rows, including duplicates.
6. The actual output rows are computed using the SELECT output expressions for each selected row.
7. Using the operators UNION, INTERSECT, and EXCEPT, the output of more than one SELECT statement can be combined to form a single result set. The UNION operator returns all rows that are in one or both of the result sets. The INTERSECT operator returns all rows that are strictly in both result sets. The EXCEPT operator returns the rows that are in the first result set but not in the second. In all three cases, duplicate rows are eliminated unless ALL is specified.
8. If the ORDER BY clause is specified, the returned rows are sorted in the specified order. If ORDER BY is not given, the rows are returned in whatever order the system finds fastest to produce.
9. If the LIMIT or OFFSET clause is specified, the SELECT statement only returns a subset of the result rows.
10. If FOR UPDATE or FOR SHARE is specified, the SELECT statement locks the entire table against concurrent updates.

You must have SELECT privilege on a table to read its values. The use of FOR UPDATE or FOR SHARE requires UPDATE privilege as well.

Parameters

The SELECT List

The SELECT list (between the key words SELECT and FROM) specifies expressions that form the output rows of the SELECT statement. The expressions can (and usually do) refer to columns computed in the FROM clause.

Using the clause [AS] *output_name*, another name can be specified for an output column. 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 not in the WHERE or HAVING clauses; there you must write out the expression instead. The AS keyword is optional in most cases (such as when declaring an alias for column names, constants, function calls, and simple unary operator expressions). In cases where the declared alias is a reserved SQL keyword, the *output_name* must be enclosed in double quotes to avoid ambiguity.

An *expression* in the SELECT list can be a constant value, a column reference, an operator invocation, a function call, an aggregate expression, a window expression, a scalar subquery, and so on. A number of constructs can be classified as an expression but do not follow any general syntax rules. These generally have the semantics of a function or operator. For information about SQL value expressions and function calls, see "Querying Data" in the *Greenplum Database Administrator Guide*.

Instead of an expression, * can be written in the output list as a shorthand for all the columns of the selected rows. Also, you can write *table_name*.* as a shorthand for the columns coming from just that table.

The FROM Clause

The FROM clause specifies one or more source tables for the SELECT. If multiple sources are specified, the result is the Cartesian product (cross join) of all the sources. But usually qualification conditions are added to restrict the returned rows to a small subset of the Cartesian product. The FROM clause can contain the following elements:

table_name

The name (optionally schema-qualified) of an existing table or view. If ONLY is specified, only that table is scanned. If ONLY is not specified, the table and all its descendant tables (if any) are scanned.

alias

A substitute name for the `FROM` item containing the alias. An alias is used for brevity or to eliminate ambiguity for self-joins (where the same table is scanned multiple times). When an alias is provided, it completely hides the actual name of the table or function; for example given `FROM foo AS f`, the remainder of the `SELECT` must refer to this `FROM` item as `f` not `foo`. If an alias is written, a column alias list can also be written to provide substitute names for one or more columns of the table.

select

A sub-`SELECT` can appear in the `FROM` clause. This acts as though its output were created as a temporary table for the duration of this single `SELECT` command. Note that the sub-`SELECT` must be surrounded by parentheses, and an alias must be provided for it. A `VALUES` command can also be used here. See "Non-standard Clauses" in the *Compatibility* section for limitations of using correlated sub-selects in Greenplum Database.

function_name

Function calls can appear in the `FROM` clause. (This is especially useful for functions that return result sets, but any function can be used.) This acts as though its output were created as a temporary table for the duration of this single `SELECT` command. An alias may also be used. If an alias is written, a column alias list can also be written to provide substitute names for one or more attributes of the function's composite return type. If the function has been defined as returning the record data type, then an alias or the key word `AS` must be present, followed by a column definition list in the form `( column_name data_type [, ... ] )`. The column definition list must match the actual number and types of columns returned by the function.

join_type

One of:

- **[INNER] JOIN**
- **LEFT [OUTER] JOIN**
- **RIGHT [OUTER] JOIN**
- **FULL [OUTER] JOIN**
- **CROSS JOIN**

For the `INNER` and `OUTER` join types, a join condition must be specified, namely exactly one of `NATURAL`, `ON` *`join_condition`*, or `USING (` *`join_column`* `[, ...])`. See below for the meaning. For `CROSS JOIN`, none of these clauses may appear.

A `JOIN` clause combines two `FROM` items. Use parentheses if necessary to determine the order of nesting. In the absence of parentheses, `JOIN`s nest left-to-right. In any case `JOIN` binds more tightly than the commas separating `FROM` items.

`CROSS JOIN` and `INNER JOIN` produce a simple Cartesian product, the same result as you get from listing the two items at the top level of `FROM`, but restricted by the join condition (if any). `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 could not 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 join condition), plus one copy of each row in the left-hand table for which there was no right-hand row that passed the join condition. This left-hand row is extended to the full width of the joined table by inserting null values for the right-hand columns. Note that only the `JOIN` clause's own condition is considered while deciding which rows have matches. Outer 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).

ON *join_condition*

join_condition is an expression resulting in a value of type `boolean` (similar to a `WHERE` clause) that specifies which rows in a join are considered to match.

USING (*join_column* [, ...])

A clause of the form `USING ( a, b, ... )` is shorthand for `ON 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

`NATURAL` is shorthand for a `USING` list that mentions all columns in the two tables that have the same names.

The WHERE Clause

The optional `WHERE` clause has the general form:

```
WHERE condition
```

where *condition* is any expression that evaluates to a result of type `boolean`. Any row that does not satisfy this condition will be eliminated from the output. A row satisfies the condition if it returns true when the actual row values are substituted for any variable references.

The GROUP BY Clause

The optional `GROUP BY` clause has the general form:

```
GROUP BY grouping_element [, ...]
```

where *grouping_element* can be one of:

```
()
expression
ROLLUP (expression [,...])
CUBE (expression [,...])
GROUPING SETS ((grouping_element [, ...]))
```

`GROUP BY` will condense into a single row all selected rows that share the same values for the grouped expressions. *expression* can be an input column name, or the name or ordinal number of an output column (`SELECT` list item), or 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.

Aggregate functions, if any are used, 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` list expressions to refer to ungrouped columns except within aggregate functions, since there would be more than one possible value to return for an ungrouped column.

Greenplum Database has the following additional OLAP grouping extensions (often referred to as *supergroups*):

ROLLUP

A `ROLLUP` grouping is an extension to the `GROUP BY` clause that creates aggregate subtotals that roll up from the most detailed level to a grand total, following a list of grouping columns (or expressions). `ROLLUP` takes an ordered list of grouping columns, calculates the standard aggregate values specified in the `GROUP BY` clause, then creates progressively higher-level subtotals, moving from right to left through the list. Finally, it creates a grand total. A `ROLLUP` grouping can be thought of as a series of grouping sets. For example:

```
GROUP BY ROLLUP (a,b,c)
```

is equivalent to:

```
GROUP BY GROUPING SETS( (a,b,c), (a,b), (a), () )
```

Notice that the *n* elements of a `ROLLUP` translate to *n*+1 grouping sets. Also, the order in which the grouping expressions are specified is significant in a `ROLLUP`.

CUBE

A `CUBE` grouping is an extension to the `GROUP BY` clause that creates subtotals for all of the possible combinations of the given list of grouping columns (or expressions). In terms of multidimensional analysis, `CUBE` generates all the subtotals that could be calculated for a data cube with the specified dimensions. For example:

```
GROUP BY CUBE (a,b,c)
```

is equivalent to:

```
GROUP BY GROUPING SETS( (a,b,c), (a,b), (a,c), (b,c), (a),
(b), (c), () )
```

Notice that *n* elements of a `CUBE` translate to 2n grouping sets. Consider using `CUBE` in any situation requiring cross-tabular reports. `CUBE` is typically most suitable in queries that use columns from multiple dimensions rather than columns representing different levels of a single dimension. For instance, a commonly requested cross-tabulation might need subtotals for all the combinations of month, state, and product.

GROUPING SETS

You can selectively specify the set of groups that you want to create using a `GROUPING SETS` expression within a `GROUP BY` clause. This allows precise specification across multiple dimensions without computing a whole `ROLLUP` or `CUBE`. For example:

```
GROUP BY GROUPING SETS( (a,c), (a,b) )
```

If using the grouping extension clauses `ROLLUP`, `CUBE`, or `GROUPING SETS`, two challenges arise. First, how do you determine which result rows are subtotals, and then the exact level of aggregation for a given subtotal. Or, how do you differentiate between result rows that contain both stored `NULL` values and "NULL" values created by the `ROLLUP` or `CUBE`. Secondly, when duplicate grouping sets are specified in the `GROUP BY` clause, how do you determine which result rows are duplicates? There are two additional grouping functions you can use in the `SELECT` list to help with this:

- **grouping(column [, ...])** — The `grouping` function can be applied to one or more grouping attributes to distinguish super-aggregated rows from regular grouped rows. This can be helpful in distinguishing a "NULL" representing the set of all values in a super-aggregated row from a `NULL` value in a regular row. Each argument in this function produces a bit — either `1` or `0`, where `1` means the result row is super-aggregated, and `0` means the result row is from a regular grouping. The `grouping` function returns an integer by treating these bits as a binary number and then converting it to a base-10 integer.
- **group_id()** — For grouping extension queries that contain duplicate grouping sets, the `group_id` function is used to identify duplicate rows in the output. All *unique* grouping set output rows will have a group_id value of 0. For each duplicate grouping set detected, the `group_id` function assigns a group_id number greater than 0. All output rows in a particular duplicate grouping set are identified by the same group_id number.

The WINDOW Clause

The `WINDOW` clause is used to define a window that can be used in the `OVER()` expression of a window function such as `rank` or `avg`. For example:

```
SELECT vendor, rank() OVER (mywindow) FROM sale
GROUP BY vendor
WINDOW mywindow AS (ORDER BY sum(prc*qty));
```

A `WINDOW` clause has this general form:

```
WINDOW window_name AS (window_specification)
```

where *window_specification* can be:

```
[window_name]
[PARTITION BY expression [, ...]]
[ORDER BY expression [ASC | DESC | USING operator] [, ...]
    [{RANGE | ROWS}
      { UNBOUNDED PRECEDING
      | expression PRECEDING
      | CURRENT ROW
      | BETWEEN window_frame_bound AND window_frame_bound }]]
             where window_frame_bound can be one of:
               UNBOUNDED PRECEDING
               expression PRECEDING
               CURRENT ROW
               expression FOLLOWING
               UNBOUNDED FOLLOWING
```

window_name

Gives a name to the window specification.

PARTITION BY

The `PARTITION BY` clause organizes the result set into logical groups based on the unique values of the specified expression. When used with window functions, the functions are applied to each partition independently. For example, if you follow `PARTITION BY` with a column name, the result set is partitioned by the distinct values of that column. If omitted, the entire result set is considered one partition.

ORDER BY

The `ORDER BY` clause defines how to sort the rows in each partition of the result set. If omitted, rows are returned in whatever order is most efficient and may vary. **Note:** Columns of data types that lack a coherent ordering, such as `time`, are not good candidates for use in the `ORDER BY` clause of a window specification. Time, with or without time zone, lacks a coherent ordering because addition and subtraction do not have the expected effects. For example, the following is not generally true: `x::time < x::time + '2 hour'::interval`

ROWS | RANGE

Use either a `ROWS` or `RANGE` clause to express the bounds of the window. The window bound can be one, many, or all rows of a partition. You can express the bound of the window either in terms of a range of data values offset from the value in the current row (`RANGE`), or in terms of the number of rows offset from the current row (`ROWS`). When using the `RANGE` clause, you must also use an `ORDER BY` clause. This is because the calculation performed to produce the window requires that the values be sorted. Additionally, the `ORDER BY` clause cannot contain more than one expression, and the expression must result in either a date or a numeric value. When using the `ROWS` or `RANGE` clauses, if you specify only a starting row, the current row is used as the last row in the window.

PRECEDING — The `PRECEDING` clause defines the first row of the window using the current row as a reference point. The starting row is expressed in terms of the number of rows preceding the current row. For example, in the case of `ROWS` framing, `5 PRECEDING` sets the window to start with the fifth row preceding the current row. In the case of `RANGE` framing, it sets the window to start with the first row whose ordering column value precedes that of the current row by 5 in the given order. If the specified order is ascending by date, this will be the first row within 5 days before the current row. `UNBOUNDED PRECEDING` sets the first row in the window to be the first row in the partition.

BETWEEN — The `BETWEEN` clause defines the first and last row of the window, using the current row as a reference point. First and last rows are expressed in terms of the number of rows preceding and following the current row, respectively. For example, `BETWEEN 3 PRECEDING AND 5 FOLLOWING` sets the window to start with the third row preceding the current row, and end with the fifth row following the current row. Use `BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` to set the first and last rows in the window to be the first and last row in the partition, respectively. This is equivalent to the default behavior if no `ROW` or `RANGE` clause is specified.

FOLLOWING — The `FOLLOWING` clause defines the last row of the window using the current row as a reference point. The last row is expressed in terms of the number of rows following the current row. For example, in the case of `ROWS` framing, `5 FOLLOWING` sets the window to end with the fifth row following the current row. In the case of `RANGE` framing, it sets the window to end with the last row whose ordering column value follows that of the current row by 5 in the given order. If the specified order is ascending by date, this will be the last row within 5 days after the current row. Use `UNBOUNDED FOLLOWING` to set the last row in the window to be the last row in the partition.

If you do not specify a `ROW` or a `RANGE` clause, the window bound starts with the first row in the partition (`UNBOUNDED PRECEDING`) and ends with the current row (`CURRENT ROW`) if `ORDER BY` is used. If an `ORDER BY` is not specified, the window starts with the first row in the partition (`UNBOUNDED PRECEDING`) and ends with last row in the partition (`UNBOUNDED FOLLOWING`).

The HAVING Clause

The optional `HAVING` clause has the general form:

```
HAVING condition
```

where *condition* is the same as specified for the `WHERE` clause. `HAVING` eliminates group rows that do not satisfy the condition. `HAVING` is different from `WHERE`: `WHERE` filters individual rows before the application of `GROUP BY`, while `HAVING` filters group rows created by `GROUP BY`. Each column referenced in *condition* must unambiguously reference a grouping column, unless the reference appears within an aggregate function.

The presence of `HAVING` turns a query into a grouped query even if there is no `GROUP BY` clause. This is the same as what happens when the query contains aggregate functions but no `GROUP BY` clause. All the selected rows are considered to form a single group, and the `SELECT` list and `HAVING` clause can only reference table columns from within aggregate functions. Such a query will emit a single row if the `HAVING` condition is true, zero rows if it is not true.

The UNION Clause

The `UNION` clause has this general form:

```
select_statement UNION [ALL] select_statement
```

where *select_statement* is any `SELECT` statement without an `ORDER BY`, `LIMIT`, `FOR UPDATE`, or `FOR SHARE` clause. (`ORDER BY` and `LIMIT` can be attached to a subquery 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 set union of the rows returned by the involved `SELECT` statements. A row is in the set union of two result sets if it appears in at least one of the result sets. The two `SELECT` statements that represent the direct operands of the `UNION` must produce the same number of columns, and corresponding columns must be of compatible data types.

The result of `UNION` does not contain any duplicate rows unless the `ALL` option is specified. `ALL` prevents elimination of duplicates. (Therefore, `UNION ALL` is usually significantly quicker than `UNION`; use `ALL` when you can.)

Multiple `UNION` operators in the same `SELECT` statement are evaluated left to right, unless otherwise indicated by parentheses.

Currently, `FOR UPDATE` and `FOR SHARE` may not be specified either for a `UNION` result or for any input of a `UNION`.

The INTERSECT Clause

The `INTERSECT` clause has this general form:

```
select_statement INTERSECT [ALL] select_statement
```

where *select_statement* is any SELECT statement without an `ORDER BY`, `LIMIT`, `FOR UPDATE`, or `FOR SHARE` clause.

The `INTERSECT` operator computes the set intersection of the rows returned by the involved `SELECT` statements. A row is in the intersection of two result sets if it appears in both result sets.

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 the left table and *n* duplicates in the right table will appear min(*m*, *n*) times in the result set.

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)`.

Currently, `FOR UPDATE` and `FOR SHARE` may not be specified either for an `INTERSECT` result or for any input of an `INTERSECT`.

The EXCEPT Clause

The `EXCEPT` clause has this general form:

```
select_statement EXCEPT [ALL] select_statement
```

where *select_statement* is any `SELECT` statement without an `ORDER BY`, `LIMIT`, `FOR UPDATE`, or `FOR SHARE` clause.

The `EXCEPT` operator computes the set of rows that are in the result of the left `SELECT` statement but not in the result of the right one.

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 the left table and *n* duplicates in the right table will appear max(*m*-*n*,0) times in the result set.

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`.

Currently, `FOR UPDATE` and `FOR SHARE` may not be specified either for an `EXCEPT` result or for any input of an `EXCEPT`.

The ORDER BY Clause

The optional `ORDER BY` clause has this general form:

```
ORDER BY expression [ASC | DESC | USING operator] [, ...]
```

where *expression* can be the name or ordinal number of an output column (`SELECT` list item), or it can be an arbitrary expression formed from input-column values.

The `ORDER BY` clause causes the result rows to be sorted according to the specified expressions. If two rows are equal according to the left-most expression, they are compared according to the next expression and so on. If they are equal according to all specified expressions, they are returned in an implementation-dependent order.

The ordinal number refers to the ordinal (left-to-right) position of the result column. This feature makes it possible to define an ordering on the basis of a column that does not have a unique name. This is never absolutely necessary because it is always possible to assign a name to a result column using the `AS` clause.

It is also possible to use arbitrary expressions in the `ORDER BY` clause, including columns that do not appear in the `SELECT` result list. Thus the following statement is valid:

```
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` clause may only specify an output column name or number, not an expression.

If an `ORDER BY` expression 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 made to be compatible with the SQL standard.

Optionally one may add the key word `ASC` (ascending) or `DESC` (descending) after any expression in the `ORDER BY` clause. If not specified, `ASC` is assumed by default. Alternatively, a specific ordering operator name may be specified in the `USING` clause. `ASC` is usually equivalent to `USING <` and `DESC` is usually equivalent to `USING >`. (But the creator of a user-defined data type can define exactly what the default sort ordering is, and it might correspond to operators with other names.)

The null value sorts higher than any other value. In other words, with ascending sort order, null values sort at the end, and with descending sort order, null values sort at the beginning.

Character-string data is sorted according to the locale-specific collation order that was established when the Greenplum Database system was initialized.

The DISTINCT Clause

If `DISTINCT` is specified, all duplicate rows are removed from the result set (one row is kept from each group of duplicates). `ALL` specifies the opposite: all rows are kept. `ALL` is the default.

`DISTINCT ON (` *`expression`* `[, ...] )` keeps only the first row of each set of rows where the given expressions evaluate to equal. The `DISTINCT ON` expressions are interpreted using the same rules as for `ORDER BY`. 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
weather_reports 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 would have gotten a report from an unpredictable time for each location.

The `DISTINCT ON` expression(s) must match the left-most `ORDER BY` expression(s). The `ORDER BY` clause will normally contain additional expression(s) that determine the desired precedence of rows within each `DISTINCT ON` group.

When Greenplum Database processes queries that contain the `DISTINCT` clause, the queries are transformed into `GROUP BY` queries. In many cases, the transformation provides significant performance gains. However, when the number of distinct values is close to the total number of rows, the transformation might result in the generation of a multi-level grouping plan. In this case, there is an expected performance degradation because of the overhead introduced by the lower aggregation level.

The LIMIT Clause

The LIMIT clause consists of two independent sub-clauses:

```
LIMIT {count | ALL}
OFFSET start
```

where *count* specifies the maximum number of rows to return, while *start* specifies the number of rows to skip before starting to return rows. When both are specified, start rows are skipped before starting to count the count rows to be returned.

When using LIMIT, it is a good idea to use an ORDER BY clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query's rows — you may be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? You don't know what ordering unless you specify ORDER BY.

The query optimizer takes LIMIT into account when generating a query plan, so you are very likely to get different plans (yielding different row orders) depending on what you use for LIMIT and OFFSET. Thus, using different LIMIT/OFFSET values to select different subsets of a query result will give inconsistent results unless you enforce a predictable result ordering with ORDER BY. This is not a defect; 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.

The FOR UPDATE/FOR SHARE Clause

The FOR UPDATE clause has this form:

```
FOR UPDATE [OF table_name [, ...]] [NOWAIT]
```

The closely related FOR SHARE clause has this form:

```
FOR SHARE [OF table_name [, ...]] [NOWAIT]
```

FOR UPDATE causes the tables accessed by the SELECT statement to be locked as though for update. This prevents the table from being modified or deleted by other transactions until the current transaction ends. That is, other transactions that attempt UPDATE, DELETE, or SELECT FOR UPDATE of this table will be blocked until the current transaction ends. Also, if an UPDATE, DELETE, or SELECT FOR UPDATE from another transaction has already locked a selected table, SELECT FOR UPDATE will wait for the other transaction to complete, and will then lock and return the updated table.

To prevent the operation from waiting for other transactions to commit, use the NOWAIT option. SELECT FOR UPDATE NOWAIT reports an error, rather than waiting, if a selected row cannot be locked immediately. Note that NOWAIT applies only to the row-level lock(s) — the required ROW SHARE table-level lock is still taken in the ordinary way. You can use the NOWAIT option of LOCK if you need to acquire the table-level lock without waiting (see *LOCK*).

FOR SHARE behaves similarly, except that it acquires a shared rather than exclusive lock on the table. A shared lock blocks other transactions from performing UPDATE, DELETE, or SELECT FOR UPDATE on the table, but it does not prevent them from performing SELECT FOR SHARE.

If specific tables are named in FOR UPDATE or FOR SHARE, then only those tables are locked; any other tables used in the SELECT are simply read as usual. A FOR UPDATE or FOR SHARE clause without a table list affects all tables used in the command. If FOR UPDATE or FOR SHARE is applied to a view or subquery, it affects all tables used in the view or subquery.

Multiple FOR UPDATE and FOR SHARE clauses can be written if it is necessary to specify different locking behavior for different tables. If the same table is mentioned (or implicitly affected) by both FOR UPDATE and FOR SHARE clauses, then it is processed as FOR UPDATE. Similarly, a table is processed as NOWAIT if that is specified in any of the clauses affecting it.

Examples

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
```

To sum the column length of all films and group the results by kind:

```
SELECT kind, sum(length) AS total FROM films GROUP BY kind;
```

To sum the column length of all films, group the results by kind and show those group totals that are less than 5 hours:

```
SELECT kind, sum(length) AS total FROM films GROUP BY kind
HAVING sum(length) < interval '5 hours';
```

Calculate the subtotals and grand totals of all sales for movie kind and distributor.

```
SELECT kind, distributor, sum(prc*qty) FROM sales
GROUP BY ROLLUP(kind, distributor)
ORDER BY 1,2,3;
```

Calculate the rank of movie distributors based on total sales:

```
SELECT distributor, sum(prc*qty),
       rank() OVER (ORDER BY sum(prc*qty) DESC)
FROM sale
GROUP BY distributor ORDER BY 2 DESC;
```

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;
```

The next example shows how to obtain the union of the tables distributors and actors, restricting the results to those that begin with the letter W in each table. Only distinct rows are wanted, so the key word ALL is omitted:

```
SELECT distributors.name FROM distributors WHERE
distributors.name LIKE 'W%' UNION SELECT actors.name FROM
actors WHERE actors.name LIKE 'W%';
```

This example shows how to use a function in the FROM clause, both with and without a column definition list:

```
CREATE FUNCTION distributors(int) RETURNS SETOF distributors
AS $$ SELECT * FROM distributors WHERE did = $1; $$ LANGUAGE
SQL;
SELECT * FROM distributors(111);

CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS
$$ SELECT * FROM distributors WHERE did = $1; $$ LANGUAGE
SQL;
SELECT * FROM distributors_2(111) AS (dist_id int, dist_name
text);
```

Compatibility

The SELECT statement is compatible with the SQL standard, but there are some extensions and some missing features.

Omitted FROM Clauses

Greenplum Database allows one to omit the `FROM` clause. It has a straightforward use to compute the results of simple expressions. For example:

```
SELECT 2+2;
```

Some other SQL databases cannot do this except by introducing a dummy one-row table from which to do the `SELECT`.

Note that if a `FROM` clause is not specified, the query cannot reference any database tables. For compatibility with applications that rely on this behavior the *add_missing_from* configuration variable can be enabled.

The AS Key Word

In the SQL standard, the optional key word `AS` is just noise and can be omitted without affecting the meaning. The Greenplum Database parser requires this key word when renaming output columns because the type extensibility features lead to parsing ambiguities without it. `AS` is optional in `FROM` items, however.

Namespace Available to GROUP BY and ORDER BY

In the SQL-92 standard, an `ORDER BY` clause may only use result column names or numbers, while a `GROUP BY` clause may only use expressions based on input column names. Greenplum Database extends each of these clauses to allow the other choice as well (but it uses the standard's interpretation if there is ambiguity). Greenplum Database 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.

SQL:1999 and later use a slightly different definition which is not entirely upward compatible with SQL-92. In most cases, however, Greenplum Database will interpret an `ORDER BY` or `GROUP BY` expression the same way SQL:1999 does.

Nonstandard Clauses

The clauses `DISTINCT ON`, `LIMIT`, and `OFFSET` are not defined in the SQL standard.

Limited Use of STABLE and VOLATILE Functions

To prevent data from becoming out-of-sync across the segments in Greenplum Database, any function classified as `STABLE` or `VOLATILE` cannot be executed at the segment database level if it contains SQL or modifies the database in any way. See *CREATE FUNCTION* for more information.

See Also

EXPLAIN

SELECT INTO

Defines a new table from the results of a query.

Synopsis

```
SELECT [ALL | DISTINCT [ON ( expression [, ...] )]]
    * | expression [AS output_name] [, ...]
    INTO [TEMPORARY | TEMP] [TABLE] new_table
    [FROM from_item [, ...]]
    [WHERE condition]
    [GROUP BY expression [, ...]]
    [HAVING condition [, ...]]
    [{UNION | INTERSECT | EXCEPT} [ALL] select]
    [ORDER BY expression [ASC | DESC | USING operator] [, ...]]
    [LIMIT {count | ALL}]
    [OFFSET start]
    [FOR {UPDATE | SHARE} [OF table_name [, ...]] [NOWAIT]
    [...]]
```

Description

`SELECT INTO` creates a new table and fills it with data computed by a query. The data is not returned to the client, as it is with a normal `SELECT`. The new table's columns have the names and data types associated with the output columns of the `SELECT`.

Parameters

The majority of parameters for `SELECT INTO` are the same as *`SELECT`*.

TEMPORARY
TEMP

If specified, the table is created as a temporary table.

new_table

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

Examples

Create a new table `films_recent` consisting of only recent entries from the table `films`:

```
SELECT * INTO films_recent FROM films WHERE date_prod >=
'2006-01-01';
```

Compatibility

The SQL standard uses `SELECT INTO` to represent selecting values into scalar variables of a host program, rather than creating a new table. The Greenplum Database usage of `SELECT INTO` to represent table creation is historical. It is best to use *`CREATE TABLE AS`* for this purpose in new applications.

See Also

`SELECT`, *`CREATE TABLE AS`*

SET

Changes the value of a Greenplum Database configuration parameter.

Synopsis

```
SET [SESSION | LOCAL] configuration_parameter {TO | =} value |
    'value' | DEFAULT}

SET [SESSION | LOCAL] TIME ZONE {timezone | LOCAL | DEFAULT}
```

Description

The `SET` command changes server configuration parameters. Any configuration parameter classified as a *session* parameter can be changed on-the-fly with `SET`. `SET` affects only the value used by the current session.

If `SET` or `SET SESSION` is issued within a transaction that is later aborted, the effects of the `SET` command disappear when the transaction is rolled back. Once the surrounding transaction is committed, the effects will persist until the end of the session, unless overridden by another `SET`.

The effects of `SET LOCAL` last only till the end of the current transaction, whether committed or not. A special case is `SET` followed by `SET LOCAL` within a single transaction: the `SET LOCAL` value will be seen until the end of the transaction, but afterwards (if the transaction is committed) the `SET` value will take effect.

If you create a cursor with the `DECLARE` command in a transaction, you cannot use the `SET` command in the transaction until you close the cursor with the `CLOSE` command.

See *Server Configuration Parameters* for information about server parameters.

Parameters

SESSION

Specifies that the command takes effect for the current session. This is the default.

LOCAL

Specifies that the command takes effect for only the current transaction. After `COMMIT` or `ROLLBACK`, the session-level setting takes effect again. Note that `SET LOCAL` will appear to have no effect if it is executed outside of a transaction.

configuration_parameter

The name of a Greenplum Database configuration parameter. Only parameters classified as *session* can be changed with `SET`. See *Server Configuration Parameters* for details.

value

New value of parameter. Values can be specified as string constants, identifiers, numbers, or comma-separated lists of these. `DEFAULT` can be used to specify resetting the parameter to its default value. If specifying memory sizing or time units, enclose the value in single quotes.

TIME ZONE

`SET TIME ZONE` value is an alias for `SET timezone TO value`. The syntax `SET TIME ZONE` allows special syntax for the time zone specification. Here are examples of valid values:

`'PST8PDT'`

`'Europe/Rome'`

`-7` (time zone 7 hours west from UTC)

`INTERVAL '-08:00' HOUR TO MINUTE` (time zone 8 hours west from UTC).

LOCAL
DEFAULT

Set the time zone to your local time zone (the one that the server's operating system defaults to). See the *Time zone section of the PostgreSQL documentation* for more information about time zones in Greenplum Database.

Examples

Set the schema search path:

```
SET search_path TO my_schema, public;
```

Increase the segment host memory per query to 200 MB:

```
SET statement_mem TO '200MB';
```

Set the style of date to traditional POSTGRES with "day before month" input convention:

```
SET datestyle TO postgres, dmy;
```

Set the time zone for San Mateo, California (Pacific Time):

```
SET TIME ZONE 'PST8PDT';
```

Set the time zone for Italy:

```
SET TIME ZONE 'Europe/Rome';
```

Compatibility

`SET TIME ZONE` extends syntax defined in the SQL standard. The standard allows only numeric time zone offsets while Greenplum Database allows more flexible time-zone specifications. All other `SET` features are Greenplum Database extensions.

See Also

RESET, *SHOW*

SET ROLE

Sets the current role identifier of the current session.

Synopsis

```
SET [SESSION | LOCAL] ROLE rolename

SET [SESSION | LOCAL] ROLE NONE

RESET ROLE
```

Description

This command sets the current role identifier of the current SQL-session context to be *rolename*. The role name may be written as either an identifier or a string literal. After `SET ROLE`, permissions checking for SQL commands is carried out as though the named role were the one that had logged in originally.

The specified *rolename* must be a role that the current session user is a member of. If the session user is a superuser, any role can be selected.

The `NONE` and `RESET` forms reset the current role identifier to be the current session role identifier. These forms may be executed by any user.

Parameters

SESSION

Specifies that the command takes effect for the current session. This is the default.

LOCAL

Specifies that the command takes effect for only the current transaction. After `COMMIT` or `ROLLBACK`, the session-level setting takes effect again. Note that `SET LOCAL` will appear to have no effect if it is executed outside of a transaction.

rolename

The name of a role to use for permissions checking in this session.

NONE
RESET

Reset the current role identifier to be the current session role identifier (that of the role used to log in).

Notes

Using this command, it is possible to either add privileges or restrict privileges. If the session user role has the `INHERITS` attribute, then it automatically has all the privileges of every role that it could `SET ROLE` to; in this case `SET ROLE` effectively drops all the privileges assigned directly to the session user and to the other roles it is a member of, leaving only the privileges available to the named role. On the other hand, if the session user role has the `NOINHERITS` attribute, `SET ROLE` drops the privileges assigned directly to the session user and instead acquires the privileges available to the named role.

In particular, when a superuser chooses to `SET ROLE` to a non-superuser role, she loses her superuser privileges.

`SET ROLE` has effects comparable to `SET SESSION AUTHORIZATION`, but the privilege checks involved are quite different. Also, `SET SESSION AUTHORIZATION` determines which roles are allowable for later `SET ROLE` commands, whereas changing roles with `SET ROLE` does not change the set of roles allowed to a later `SET ROLE`.

Examples

```
SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
--------------+--------------
 peter        | peter

SET ROLE 'paul';

SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
--------------+--------------
 peter        | paul
```

Compatibility

Greenplum Database allows identifier syntax (*rolename*), while the SQL standard requires the role name to be written as a string literal. SQL does not allow this command during a transaction; Greenplum Database does not make this restriction. The `SESSION` and `LOCAL` modifiers are a Greenplum Database extension, as is the `RESET` syntax.

See Also

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION

Sets the session role identifier and the current role identifier of the current session.

Synopsis

```
SET [SESSION | LOCAL] SESSION AUTHORIZATION rolename

SET [SESSION | LOCAL] SESSION AUTHORIZATION DEFAULT

RESET SESSION AUTHORIZATION
```

Description

This command sets the session role identifier and the current role identifier of the current SQL-session context to be *rolename*. The role name may be written as either an identifier or a string literal. Using this command, it is possible, for example, to temporarily become an unprivileged user and later switch back to being a superuser.

The session role identifier is initially set to be the (possibly authenticated) role name provided by the client. The current role identifier is normally equal to the session user identifier, but may change temporarily in the context of `setuid` functions and similar mechanisms; it can also be changed by *`SET ROLE`*. The current user identifier is relevant for permission checking.

The session user identifier may be changed only if the initial session user (the authenticated user) had the superuser privilege. Otherwise, the command is accepted only if it specifies the authenticated user name.

The `DEFAULT` and `RESET` forms reset the session and current user identifiers to be the originally authenticated user name. These forms may be executed by any user.

Parameters

SESSION

Specifies that the command takes effect for the current session. This is the default.

LOCAL

Specifies that the command takes effect for only the current transaction. After `COMMIT` or `ROLLBACK`, the session-level setting takes effect again. Note that `SET LOCAL` will appear to have no effect if it is executed outside of a transaction.

rolename

The name of the role to assume.

NONE
RESET

Reset the session and current role identifiers to be that of the role used to log in.

Examples

```
SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
--------------+--------------
```

```
 paul          | paul
```

Compatibility

The SQL standard allows some other expressions to appear in place of the literal *rolename*, but these options are not important in practice. Greenplum Database allows identifier syntax (*rolename*), which SQL does not. SQL does not allow this command during a transaction; Greenplum Database does not make this restriction. The `SESSION` and `LOCAL` modifiers are a Greenplum Database extension, as is the `RESET` syntax.

See Also

SET ROLE

SET TRANSACTION

Sets the characteristics of the current transaction.

Synopsis

```
SET TRANSACTION [transaction_mode] [READ ONLY | READ WRITE]

SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode
     [READ ONLY | READ WRITE]
```

where *transaction_mode* is one of:

```
   ISOLATION LEVEL {SERIALIZABLE | READ COMMITTED | READ UNCOMMITTED}
```

Description

The `SET TRANSACTION` command sets the characteristics of the current transaction. It has no effect on any subsequent transactions.

The available transaction characteristics are the transaction isolation level and the transaction access mode (read/write or read-only).

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** — All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction.

The SQL standard defines two additional levels, `READ UNCOMMITTED` and `REPEATABLE READ`. In Greenplum Database `READ UNCOMMITTED` is treated as `READ COMMITTED`. `REPEATABLE READ` is not supported; use `SERIALIZABLE` if `REPEATABLE READ` behavior is required.

The transaction isolation level cannot be changed after the first query or data-modification statement (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH`, or `COPY`) of a transaction has been executed.

The transaction access mode determines whether the transaction is read/write or read-only. Read/write is the default. When a transaction is read-only, the following SQL commands are disallowed: `INSERT`, `UPDATE`, `DELETE`, and `COPY FROM` if the table they would write to is not a temporary table; all `CREATE`, `ALTER`, and `DROP` commands; `GRANT`, `REVOKE`, `TRUNCATE`; and `EXPLAIN ANALYZE` and `EXECUTE` if the command they would execute is among those listed. This is a high-level notion of read-only that does not prevent all writes to disk.

Parameters

SESSION CHARACTERISTICS

Sets the default transaction characteristics for subsequent transactions of a session.

SERIALIZABLE
READ COMMITTED
READ UNCOMMITTED

The SQL standard defines four transaction isolation levels: `READ COMMITTED`, `READ UNCOMMITTED`, `SERIALIZABLE`, and `REPEATABLE READ`. The default behavior is that a statement can only see rows committed before it began (`READ COMMITTED`). In Greenplum Database `READ UNCOMMITTED` is treated the same as `READ COMMITTED`. `REPEATABLE READ` is not supported; use `SERIALIZABLE` instead. `SERIALIZABLE` is the strictest transaction isolation. This level emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. Applications using this level must be prepared to retry transactions due to serialization failures.

READ WRITE
READ ONLY

Determines whether the transaction is read/write or read-only. Read/write is the default. When a transaction is read-only, the following SQL commands are disallowed: `INSERT`, `UPDATE`, `DELETE`, and `COPY FROM` if the table they would write to is not a temporary table; all `CREATE`, `ALTER`, and `DROP` commands; `GRANT`, `REVOKE`, `TRUNCATE`; and `EXPLAIN ANALYZE` and `EXECUTE` if the command they would execute is among those listed.

Notes

If `SET TRANSACTION` is executed without a prior `START TRANSACTION` or `BEGIN`, it will appear to have no effect.

It is possible to dispense with `SET TRANSACTION` by instead specifying the desired transaction_modes in `BEGIN` or `START TRANSACTION`.

The session default transaction modes can also be set by setting the configuration parameters *default_transaction_isolation* and *default_transaction_read_only*.

Examples

Set the transaction isolation level for the current transaction:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Compatibility

Both commands are defined in the SQL standard. `SERIALIZABLE` is the default transaction isolation level in the standard. In Greenplum Database the default is `READ COMMITTED`. Because of lack of predicate locking, the `SERIALIZABLE` level is not truly serializable. Essentially, a predicate-locking system prevents phantom reads by restricting what is written, whereas a multi-version concurrency control model (MVCC) as used in Greenplum Database prevents them by restricting what is read.

In the SQL standard, there is one other transaction characteristic that can be set with these commands: the size of the diagnostics area. This concept is specific to embedded SQL, and therefore is not implemented in the Greenplum Database server.

The SQL standard requires commas between successive *transaction_modes*, but for historical reasons Greenplum Database allows the commas to be omitted.

See Also

BEGIN, *LOCK*

SHOW

Shows the value of a system configuration parameter.

Synopsis

```
SHOW configuration_parameter

SHOW ALL
```

Description

`SHOW` displays the current settings of Greenplum Database system configuration parameters. You can set these parameters with the `SET` statement, or by editing the `postgresql.conf` configuration file of the Greenplum Database master. Note that some parameters viewable by `SHOW` are read-only — their values can be viewed but not set. See the Greenplum Database Reference Guide for details.

Parameters

configuration_parameter

The name of a system configuration parameter.

ALL

Shows the current value of all configuration parameters.

Examples

Show the current setting of the parameter `search_path`:

```
SHOW search_path;
```

Show the current setting of all parameters:

```
SHOW ALL;
```

Compatibility

`SHOW` is a Greenplum Database extension.

See Also

SET, *RESET*

START TRANSACTION

Starts a transaction block.

Synopsis

```
START TRANSACTION [SERIALIZABLE | READ COMMITTED | READ UNCOMMITTED]
                  [READ WRITE | READ ONLY]
```

Description

`START TRANSACTION` begins a new transaction block. If the isolation level or read/write mode is specified, the new transaction has those characteristics, as if *`SET TRANSACTION`* was executed. This is the same as the `BEGIN` command.

Parameters

SERIALIZABLE
READ COMMITTED
READ UNCOMMITTED

The SQL standard defines four transaction isolation levels: `READ COMMITTED`, `READ UNCOMMITTED`, `SERIALIZABLE`, and `REPEATABLE READ`. The default behavior is that a statement can only see rows committed before it began (`READ COMMITTED`). In Greenplum Database `READ UNCOMMITTED` is treated the same as `READ COMMITTED`. `REPEATABLE READ` is not supported; use `SERIALIZABLE` if this behavior is required. `SERIALIZABLE`, wherein all statements of the current transaction can only see rows committed before the first statement was executed in the transaction, is the strictest transaction isolation. This level emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. Applications using this level must be prepared to retry transactions due to serialization failures.

READ WRITE
READ ONLY

Determines whether the transaction is read/write or read-only. Read/write is the default. When a transaction is read-only, the following SQL commands are disallowed: `INSERT`, `UPDATE`, `DELETE`, and `COPY FROM` if the table they would write to is not a temporary table; all `CREATE`, `ALTER`, and `DROP` commands; `GRANT`, `REVOKE`, `TRUNCATE`; and `EXPLAIN ANALYZE` and `EXECUTE` if the command they would execute is among those listed.

Examples

To begin a transaction block:

```
START TRANSACTION;
```

Compatibility

In the standard, it is not necessary to issue `START TRANSACTION` to start a transaction block: any SQL command implicitly begins a block. Greenplum Database behavior can be seen as implicitly issuing a `COMMIT` after each command that does not follow `START TRANSACTION` (or `BEGIN`), and it is therefore often called 'autocommit'. Other relational database systems may offer an autocommit feature as a convenience.

The SQL standard requires commas between successive *transaction_modes*, but for historical reasons Greenplum Database allows the commas to be omitted.

See also the compatibility section of *`SET TRANSACTION`*.

See Also

BEGIN, *SET TRANSACTION*

TRUNCATE

Empties a table of all rows.

Synopsis

```
TRUNCATE [TABLE] name [, ...] [CASCADE | RESTRICT]
```

Description

`TRUNCATE` quickly removes all rows from a table or set of tables. It has the same effect as an unqualified *`DELETE`* on each table, but since it does not actually scan the tables it is faster. This is most useful on large tables.

You must have the `TRUNCATE` privilege on the table to truncate table rows.

Parameters

name

The name (optionally schema-qualified) of a table to be truncated.

CASCADE

Since this key word applies to foreign key references (which are not supported in Greenplum Database) it has no effect.

RESTRICT

Since this key word applies to foreign key references (which are not supported in Greenplum Database) it has no effect.

Notes

`TRUNCATE` will not run any user-defined `ON DELETE` triggers that might exist for the tables.

`TRUNCATE` will not truncate any tables that inherit from the named table. Only the named table is truncated, not its child tables.

`TRUNCATE` will not truncate any sub-tables of a partitioned table. If you specify a sub-table of a partitioned table, `TRUNCATE` will not remove rows from the sub-table and its child tables.

Examples

Empty the table `films`:

```
TRUNCATE films;
```

Compatibility

There is no `TRUNCATE` command in the SQL standard.

See Also

`DELETE`, *`DROP TABLE`*

UPDATE

Updates rows of a table.

Synopsis

```
UPDATE [ONLY] table [[AS] alias]
   SET {column = {expression | DEFAULT} |
   (column [, ...]) = ({expression | DEFAULT} [, ...])} [, ...]
   [FROM fromlist]
   [WHERE condition | WHERE CURRENT OF cursor_name ]
```

Description

`UPDATE` changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need be mentioned in the `SET` clause; columns not explicitly modified retain their previous values.

By default, `UPDATE` will update rows in the specified table and all its subtables. If you wish to only update the specific table mentioned, you must use the `ONLY` clause.

There are two ways to modify a table using information contained in other tables in the database: using sub-selects, or specifying additional tables in the `FROM` clause. Which technique is more appropriate depends on the specific circumstances.

If the `WHERE CURRENT OF` clause is specified, the row that is updated is the one most recently fetched from the specified cursor.

You must have the `UPDATE` privilege on the table to update it, as well as the `SELECT` privilege to any table whose values are read in the expressions or condition.

Outputs

On successful completion, an `UPDATE` command returns a command tag of the form:

```
UPDATE count
```

where *count* is the number of rows updated. If *count* is 0, no rows matched the condition (this is not considered an error).

Parameters

ONLY

If specified, update rows from the named table only. When not specified, any tables inheriting from the named table are also processed.

table

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

alias

A substitute name for the target table. When an alias is provided, it completely hides the actual name of the table. For example, given `UPDATE foo AS f`, the remainder of the `UPDATE` statement must refer to this table as `f` not `foo`.

column

The name of a column in table. The column name can be qualified with a subfield name or array subscript, if needed. Do not include the table's name in the specification of a target column.

expression

An expression to assign to the column. The expression may use the old values of this and other columns in the table.

DEFAULT

Set the column to its default value (which will be NULL if no specific default expression has been assigned to it).

fromlist

A list of table expressions, allowing columns from other tables to appear in the `WHERE` condition and the update expressions. This is similar to the list of tables that can be specified in the `FROM` clause of a `SELECT` statement. Note that the target table must not appear in the *fromlist*, unless you intend a self-join (in which case it must appear with an *alias* in the *fromlist*).

condition

An expression that returns a value of type boolean. Only rows for which this expression returns true will be updated.

cursor_name

The name of the cursor to use in a `WHERE CURRENT OF` condition. The row to be updated is the one most recently fetched from the cursor. The cursor must be a simple (non-join, non-aggregate) query on the `UPDATE` command target table. See *`DECLARE`* for more information about creating cursors.

`WHERE CURRENT OF` cannot be specified together with a Boolean condition.

output_expression

An expression to be computed and returned by the `UPDATE` command after each row is updated. The expression may use any column names of the table or table(s) listed in `FROM`. Write `*` to return all columns.

output_name

A name to use for a returned column.

Notes

`SET` is not allowed on the Greenplum distribution key columns of a table.

When a `FROM` clause is present, what essentially happens is that the target table is joined to the tables mentioned in the from list, and each output row of the join represents an update operation for the target table. When using `FROM` you should ensure that the join produces at most one output row for each row to be modified. In other words, a target row should not join to more than one row from the other table(s). If it does, then only one of the join rows will be used to update the target row, but which one will be used is not readily predictable.

Because of this indeterminacy, referencing other tables only within sub-selects is safer, though often harder to read and slower than using a join.

Execution of `UPDATE` and `DELETE` commands directly on a specific partition (child table) of a partitioned table is not supported. Instead, these commands must be executed on the root partitioned table, the table created with the `CREATE TABLE` command.

Examples

Change the word `Drama` to `Dramatic` in the column `kind` of the table `films`:

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

Adjust temperature entries and reset precipitation to its default value in one row of the table `weather`:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi =
```

```
temp_lo+15, prcp = DEFAULT
WHERE city = 'San Francisco' AND date = '2006-07-03';
```

Use the alternative column-list syntax to do the same update:

```
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1,
temp_lo+15, DEFAULT)
WHERE city = 'San Francisco' AND date = '2006-07-03';
```

Increment the sales count of the salesperson who manages the account for Acme Corporation, using the `FROM` clause syntax (assuming both tables being joined are distributed in Greenplum Database on the `id` column):

```
UPDATE employees SET sales_count = sales_count + 1 FROM
accounts
WHERE accounts.name = 'Acme Corporation'
AND employees.id = accounts.id;
```

Perform the same operation, using a sub-select in the `WHERE` clause:

```
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
  (SELECT id FROM accounts WHERE name = 'Acme Corporation');
```

Attempt to insert a new stock item along with the quantity of stock. If the item already exists, instead update the stock count of the existing item. To do this without failing the entire transaction, use savepoints.

```
BEGIN;
-- other operations
SAVEPOINT sp1;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');
-- Assume the above fails because of a unique key violation,
-- so now we issue these commands:
ROLLBACK TO sp1;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau
Lafite 2003';
-- continue with other operations, and eventually
COMMIT;
```

Compatibility

This command conforms to the SQL standard, except that the `FROM` clause is a Greenplum Database extension.

According to the standard, the column-list syntax should allow a list of columns to be assigned from a single row-valued expression, such as a sub-select:

```
UPDATE accounts SET (contact_last_name, contact_first_name) =
     (SELECT last_name, first_name FROM salesmen
      WHERE salesmen.id = accounts.sales_id);
```

This is not currently implemented — the source must be a list of independent expressions.

Some other database systems offer a `FROM` option in which the target table is supposed to be listed again within `FROM`. That is not how Greenplum Database interprets `FROM`. Be careful when porting applications that use this extension.

See Also

DECLARE, *DELETE*, *SELECT*, *INSERT*

VACUUM

Garbage-collects and optionally analyzes a database.

Synopsis

```
VACUUM [FULL] [FREEZE] [VERBOSE] [table]

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

Description

`VACUUM` reclaims storage occupied by deleted tuples. In normal Greenplum Database operation, tuples that are deleted or obsoleted by an update are not physically removed from their table; they remain present on disk until a `VACUUM` is done. Therefore it is necessary to do `VACUUM` periodically, especially on frequently-updated tables.

With no parameter, `VACUUM` processes every table in the current database. With a parameter, `VACUUM` processes only that table.

`VACUUM ANALYZE` performs a `VACUUM` and then an `ANALYZE` for each selected table. This is a handy combination form for routine maintenance scripts. See *`ANALYZE`* for more details about its processing.

`VACUUM` (without `FULL`) marks deleted and obsoleted data in tables and indexes for future reuse and reclaims space for re-use only if the space is at the end of the table and an exclusive table lock can be easily obtained. Unused space at the start or middle of a table remains as is. With heap tables, this form of the command can operate in parallel with normal reading and writing of the table, as an exclusive lock is not obtained.

With append-optimized tables, `VACUUM` compacts a table by first vacuuming the indexes, then compacting each segment file in turn, and finally vacuuming auxiliary relations and updating statistics. On each segment, visible rows are copied from the current segment file to a new segment file, and then the current segment file is scheduled to be dropped and the new segment file is made available. Plain `VACUUM` of an append-optimized table allows scans, inserts, deletes, and updates of the table while a segment file is compacted.

`VACUUM FULL` does more extensive processing, including moving of tuples across blocks to try to compact the table to the minimum number of disk blocks. This form is much slower and requires an exclusive lock on each table while it is being processed.

With append-optimized tables, `VACUUM FULL` acquires an exclusive lock and disallows inserts, updates and deletes, but `SELECT` statements can be executed during most of the compaction process. When all visible rows have been copied to the new segment file, the table is locked briefly while the new segment is made available and the old segment is set to be dropped.

Important: For information on the use of `VACUUM`, `VACUUM FULL`, and `VACUUM ANALYZE`, see *Notes*

Outputs

When `VERBOSE` is specified, `VACUUM` emits progress messages to indicate which table is currently being processed. Various statistics about the tables are printed as well.

Parameters

FULL

Selects a full vacuum, which may reclaim more space, but takes much longer and exclusively locks the table.

FREEZE

Specifying `FREEZE` is equivalent to performing `VACUUM` with the `vacuum_freeze_min_age` server configuration parameter set to zero. See *Server Configuration Parameters* for information about `vacuum_freeze_min_age`.

VERBOSE

Prints a detailed vacuum activity report for each table.

ANALYZE

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

table

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

column

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

Notes

`VACUUM` cannot be executed inside a transaction block.

Pivotal recommends that active production databases be vacuumed frequently (at least nightly), in order to remove expired rows. After adding or deleting a large number of rows, running the `VACUUM ANALYZE` command for the affected table might be useful. This updates the system catalogs with the results of all recent changes, and allows the Greenplum Database query optimizer to make better choices in planning queries.

Important: Regular PostgreSQL has a separate optional server process called the *autovacuum daemon*, whose purpose is to automate the execution of `VACUUM` and `ANALYZE` commands. This feature is currently disabled in Greenplum Database.

`VACUUM` causes a substantial increase in I/O traffic, which can cause poor performance for other active sessions. Therefore, it is advisable to vacuum the database at low usage times.

For heap tables, expired rows are held in what is called the *free space map*. The free space map must be sized large enough to cover the dead rows of all heap tables in your database. If not sized large enough, space occupied by dead rows that overflow the free space map cannot be reclaimed by a regular `VACUUM` command.

`VACUUM` commands skip external tables.

`VACUUM FULL` reclaims all expired row space, however it requires an exclusive lock on each table being processed, is a very expensive operation, and might take a long time to complete on large, distributed Greenplum Database tables. Pivotal recommends performing `VACUUM FULL` operations during database maintenance periods.

As an alternative to `VACUUM FULL`, you can re-create the table with a `CREATE TABLE AS` statement and drop the old table.

Size the free space map appropriately. You configure the free space map using the following server configuration parameters:

- `max_fsm_pages`
- `max_fsm_relations`

For append-optimized tables, `VACUUM` requires enough available disk space to accommodate the new segment file during the `VACUUM` process. If the ratio of hidden rows to total rows in a segment file is less than a threshold value (10, by default), the segment file is not compacted. The threshold value can be configured with the `gp_appendonly_compaction_threshold` server configuration parameter. `VACUUM FULL` ignores the threshold and rewrites the segment file regardless of the ratio. `VACUUM` can be disabled for append-optimized tables using the `gp_appendonly_compaction` server configuration parameter. See *Server Configuration Parameters* for information about the server configuration parameters.

If a concurrent serializable transaction is detected when an append-optimized table is being vacuumed, the current and subsequent segment files are not compacted. If a segment file has been compacted but a concurrent serializable transaction is detected in the transaction that drops the original segment file, the drop is skipped. This could leave one or two segment files in an "awaiting drop" state after the vacuum has completed.

For more information about concurrency control in Greenplum Database, see "Routine System Maintenance Tasks" in *Greenplum Database Administrator Guide*.

Examples

Vacuum all tables in the current database:

```
VACUUM;
```

Vacuum a specific table only:

```
VACUUM mytable;
```

Vacuum all tables in the current database and collect statistics for the query optimizer:

```
VACUUM ANALYZE;
```

Compatibility

There is no `VACUUM` statement in the SQL standard.

See Also

ANALYZE

VALUES

Computes a set of rows.

Synopsis

```
VALUES ( expression [, ...] ) [, ...]
   [ORDER BY sort_expression [ASC | DESC | USING operator] [, ...]]
   [LIMIT {count | ALL}] [OFFSET start]
```

Description

`VALUES` computes a row value or set of row values specified by value expressions. It is most commonly used to generate a "constant table" within a larger command, but it can be used on its own.

When more than one row is specified, all the rows must have the same number of elements. The data types of the resulting table's columns are determined by combining the explicit or inferred types of the expressions appearing in that column, using the same rules as for `UNION`.

Within larger commands, `VALUES` is syntactically allowed anywhere that `SELECT` is. Because it is treated like a `SELECT` by the grammar, it is possible to use the `ORDERBY`, `LIMIT`, and `OFFSET` clauses with a `VALUES` command.

Parameters

expression

A constant or expression to compute and insert at the indicated place in the resulting table (set of rows). In a `VALUES` list appearing at the top level of an `INSERT`, an expression can be replaced by `DEFAULT` to indicate that the destination column's default value should be inserted. `DEFAULT` cannot be used when `VALUES` appears in other contexts.

sort_expression

An expression or integer constant indicating how to sort the result rows. This expression may refer to the columns of the `VALUES` result as `column1`, `column2`, etc. For more details, see "The ORDER BY Clause" in the parameters for *`SELECT`*.

operator

A sorting operator. For more details, see "The ORDER BY Clause" in the parameters for *`SELECT`*.

LIMIT count
OFFSET start

The maximum number of rows to return. For more details, see "The LIMIT Clause" in the parameters for *`SELECT`*.

Notes

`VALUES` lists with very large numbers of rows should be avoided, as you may encounter out-of-memory failures or poor performance. `VALUES` appearing within `INSERT` is a special case (because the desired column types are known from the `INSERT`'s target table, and need not be inferred by scanning the `VALUES` list), so it can handle larger lists than are practical in other contexts.

Examples

A bare `VALUES` command:

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

This will return a table of two columns and three rows. It is effectively equivalent to:

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

More usually, `VALUES` is used within a larger SQL command. The most common use is in `INSERT`:

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

In the context of `INSERT`, entries of a `VALUES` list can be `DEFAULT` to indicate that the column default should be used here instead of specifying a value:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82
minutes'),
    ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama', DEFAULT);
```

`VALUES` can also be used where a sub-`SELECT` might be written, for example in a `FROM` clause:

```
SELECT f.* FROM films f, (VALUES('MGM', 'Horror'), ('UA',
'Sci-Fi')) AS t (studio, kind) WHERE f.studio = t.studio AND
f.kind = t.kind;
UPDATE employees SET salary = salary * v.increase FROM
(VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (depno,
target, increase) WHERE employees.depno = v.depno AND
employees.sales >= v.target;
```

Note that an `AS` clause is required when `VALUES` is used in a `FROM` clause, just as is true for `SELECT`. It is not required that the `AS` clause specify names for all the columns, but it is good practice to do so. The default column names for `VALUES` are `column1`, `column2`, etc. in Greenplum Database, but these names might be different in other database systems.

When `VALUES` is used in `INSERT`, the values are all automatically coerced to the data type of the corresponding destination column. When it is used in other contexts, it may be necessary to specify the correct data type. If the entries are all quoted literal constants, coercing the first is sufficient to determine the assumed type for all:

```
SELECT * FROM machines WHERE ip_address IN
(VALUES('192.168.0.1'::inet), ('192.168.0.10'),
('192.168.1.43'));
```

Note: For simple `IN` tests, it is better to rely on the list-of-scalars form of `IN` than to write a `VALUES` query as shown above. The list of scalars method requires less writing and is often more efficient.

Compatibility

`VALUES` conforms to the SQL standard, except that `LIMIT` and `OFFSET` are Greenplum Database extensions.

See Also

INSERT, *SELECT*

Chapter 3

SQL 2008 Optional Feature Compliance

The following table lists the features described in the 2008 SQL standard. Features that are supported in Greenplum Database are marked as YES in the 'Supported' column, features that are not implemented are marked as NO.

For information about Greenplum features and SQL compliance, see the *Greenplum Database Administrator Guide*.

Table 9: SQL 2008 Optional Feature Compliance Details

ID	Feature	Supported	Comments
B011	Embedded Ada	NO	
B012	Embedded C	NO	Due to issues with PostgreSQL `ecpg`
B013	Embedded COBOL	NO	
B014	Embedded Fortran	NO	
B015	Embedded MUMPS	NO	
B016	Embedded Pascal	NO	
B017	Embedded PL/I	NO	
B021	Direct SQL	YES	
B031	Basic dynamic SQL	NO	
B032	Extended dynamic SQL	NO	
B033	Untyped SQL-invoked function arguments	NO	
B034	Dynamic specification of cursor attributes	NO	
B035	Non-extended descriptor names	NO	
B041	Extensions to embedded SQL exception declarations	NO	
B051	Enhanced execution rights	NO	
B111	Module language Ada	NO	
B112	Module language C	NO	
B113	Module language COBOL	NO	

ID	Feature	Supported	Comments
B114	Module language Fortran	NO	
B115	Module language MUMPS	NO	
B116	Module language Pascal	NO	
B117	Module language PL/I	NO	
B121	Routine language Ada	NO	
B122	Routine language C	NO	
B123	Routine language COBOL	NO	
B124	Routine language Fortran	NO	
B125	Routine language MUMPS	NO	
B126	Routine language Pascal	NO	
B127	Routine language PL/I	NO	
B128	Routine language SQL	NO	
E011	Numeric data types	YES	
E011-01	`INTEGER` and `SMALLINT` data types	YES	
E011-02	`DOUBLE PRECISION` and `FLOAT` data types	YES	
E011-03	`DECIMAL` and `NUMERIC` data types	YES	
E011-04	Arithmetic operators	YES	
E011-05	Numeric comparison	YES	
E011-06	Implicit casting among the numeric data types	YES	
E021	Character data types	YES	
E021-01	`CHARACTER` data type	YES	
E021-02	`CHARACTER VARYING` data type	YES	
E021-03	Character literals	YES	
E021-04	`CHARACTER_LENGTH` function	YES	Trims trailing spaces from `CHARACTER` values before counting
E021-05	`OCTET_LENGTH` function	YES	
E021-06	`SUBSTRING` function	YES	
E021-07	Character concatenation	YES	

ID	Feature	Supported	Comments
E021-08	UPPER and LOWER functions	YES	
E021-09	TRIM function	YES	
E021-10	Implicit casting among the character string types	YES	
E021-11	POSITION function	YES	
E021-12	Character comparison	YES	
E031	Identifiers	YES	
E031-01	Delimited identifiers	YES	
E031-02	Lower case identifiers	YES	
E031-03	Trailing underscore	YES	
E051	Basic query specification	YES	
E051-01	SELECT DISTINCT	YES	
E051-02	GROUP BY clause	YES	
E051-03	GROUP BY can contain columns not in SELECT list	YES	
E051-04	SELECT list items can be renamed	YES	
E051-05	HAVING clause	YES	
E051-06	Qualified * in SELECT list	YES	
E051-07	Correlation names in the FROM clause	YES	
E051-08	Rename columns in the FROM clause	YES	
E061	Basic predicates and search conditions	YES	
E061-01	Comparison predicate	YES	
E061-02	BETWEEN predicate	YES	
E061-03	IN predicate with list of values	YES	
E061-04	LIKE predicate	YES	
E061-05	LIKE predicate ESCAPE clause	YES	
E061-06	NULL predicate	YES	
E061-07	Quantified comparison predicate	YES	

ID	Feature	Supported	Comments
E061-08	`EXISTS` predicate	YES	Not all uses work in Greenplum
E061-09	Subqueries in comparison predicate	YES	
E061-11	Subqueries in IN predicate	YES	
E061-12	Subqueries in quantified comparison predicate	YES	
E061-13	Correlated subqueries	YES	
E061-14	Search condition	YES	
E071	Basic query expressions	YES	
E071-01	`UNION DISTINCT` table operator	YES	
E071-02	`UNION ALL` table operator	YES	
E071-03	`EXCEPT DISTINCT` table operator	YES	
E071-05	Columns combined via table operators need not have exactly the same data type	YES	
E071-06	Table operators in subqueries	YES	
E081	Basic Privileges	NO	Partial sub-feature support
E081-01	`SELECT` privilege	YES	
E081-02	`DELETE` privilege	YES	
E081-03	`INSERT` privilege at the table level	YES	
E081-04	`UPDATE` privilege at the table level	YES	
E081-05	`UPDATE` privilege at the column level	NO	
E081-06	`REFERENCES` privilege at the table level	NO	
E081-07	`REFERENCES` privilege at the column level	NO	
E081-08	`WITH GRANT OPTION`	YES	
E081-09	`USAGE` privilege	YES	
E081-10	`EXECUTE` privilege	YES	

ID	Feature	Supported	Comments
E091	Set Functions	YES	
E091-01	AVG	YES	
E091-02	COUNT	YES	
E091-03	MAX	YES	
E091-04	MIN	YES	
E091-05	SUM	YES	
E091-06	ALL quantifier	YES	
E091-07	DISTINCT quantifier	YES	
E101	Basic data manipulation	YES	
E101-01	INSERT statement	YES	
E101-03	Searched UPDATE statement	YES	
E101-04	Searched DELETE statement	YES	
E111	Single row SELECT statement	YES	
E121	Basic cursor support	YES	
E121-01	DECLARE CURSOR	YES	
E121-02	ORDER BY columns need not be in select list	YES	
E121-03	Value expressions in ORDER BY clause	YES	
E121-04	OPEN statement	YES	
E121-06	Positioned UPDATE statement	NO	
E121-07	Positioned DELETE statement	NO	
E121-08	CLOSE statement	YES	
E121-10	FETCH statement implicit NEXT	YES	
E121-17	WITH HOLD cursors	YES	
E131	Null value support	YES	
E141	Basic integrity constraints	YES	
E141-01	NOT NULL constraints	YES	
E141-02	UNIQUE constraints of NOT NULL columns	YES	Must be the same as or a superset of the Greenplum distribution key

ID	Feature	Supported	Comments
E141-03	PRIMARY KEY constraints	YES	Must be the same as or a superset of the Greenplum distribution key
E141-04	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action	NO	
E141-06	CHECK constraints	YES	
E141-07	Column defaults	YES	
E141-08	NOT NULL inferred on PRIMARY KEY	YES	
E141-10	Names in a foreign key can be specified in any order	YES	Foreign keys can be declared but are not enforced in Greenplum
E151	Transaction support	YES	
E151-01	COMMIT statement	YES	
E151-02	ROLLBACK statement	YES	
E152	Basic SET TRANSACTION statement	YES	
E152-01	ISOLATION LEVEL SERIALIZABLE clause	YES	
E152-02	READ ONLY and READ WRITE clauses	YES	
E153	Updatable queries with subqueries	NO	
E161	SQL comments using leading double minus	YES	
E171	SQLSTATE support	YES	
E182	Module language	NO	
F021	Basic information schema	YES	
F021-01	COLUMNS view	YES	
F021-02	TABLES view	YES	
F021-03	VIEWS view	YES	
F021-04	TABLE_CONSTRAINTS view	YES	
F021-05	REFERENTIAL_CONSTRAINTS view	YES	

ID	Feature	Supported	Comments
F021-06	CHECK_CONSTRAINTS view	YES	
F031	Basic schema manipulation	YES	
F031-01	CREATE TABLE statement to create persistent base tables	YES	
F031-02	CREATE VIEW statement	YES	
F031-03	GRANT statement	YES	
F031-04	ALTER TABLE statement: ADD COLUMN clause	YES	
F031-13	DROP TABLE statement: RESTRICT clause	YES	
F031-16	DROP VIEW statement: RESTRICT clause	YES	
F031-19	REVOKE statement: RESTRICT clause	YES	
F032	CASCADE drop behavior	YES	
F033	ALTER TABLE statement: DROP COLUMN clause	YES	
F034	Extended REVOKE statement	YES	
F034-01	REVOKE statement performed by other than the owner of a schema object	YES	
F034-02	REVOKE statement: GRANT OPTION FOR clause	YES	
F034-03	REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION	YES	
F041	Basic joined table	YES	
F041-01	Inner join (but not necessarily the INNER keyword)	YES	
F041-02	INNER keyword	YES	
F041-03	LEFT OUTER JOIN	YES	
F041-04	RIGHT OUTER JOIN	YES	
F041-05	Outer joins can be nested	YES	

ID	Feature	Supported	Comments
F041-07	The inner table in a left or right outer join can also be used in an inner join	YES	
F041-08	All comparison operators are supported (rather than just =)	YES	
F051	Basic date and time	YES	
F051-01	`DATE` data type (including support of `DATE` literal)	YES	
F051-02	`TIME` data type (including support of `TIME` literal) with fractional seconds precision of at least 0	YES	
F051-03	`TIMESTAMP` data type (including support of `TIMESTAMP` literal) with fractional seconds precision of at least 0 and 6	YES	
F051-04	Comparison predicate on `DATE`, `TIME`, and `TIMESTAMP` data types	YES	
F051-05	Explicit `CAST` between datetime types and character string types	YES	
F051-06	`CURRENT_DATE`	YES	
F051-07	`LOCALTIME`	YES	
F051-08	`LOCALTIMESTAMP`	YES	
F052	Intervals and datetime arithmetic	YES	
F053	`OVERLAPS` predicate	YES	
F081	`UNION` and `EXCEPT` in views	YES	
F111	Isolation levels other than SERIALIZABLE	YES	
F111-01	`READ UNCOMMITTED` isolation level	NO	Can be declared but is treated as a synonym for `READ COMMITTED`
F111-02	`READ COMMITTED` isolation level	YES	
F111-03	`REPEATABLE READ` isolation level	NO	Use `SERIALIZABLE`

ID	Feature	Supported	Comments
F121	Basic diagnostics management	NO	
F122	Enhanced diagnostics management	NO	
F123	All diagnostics	NO	
F131-	Grouped operations	YES	
F131-01	`WHERE`, `GROUP BY`, and `HAVING` clauses supported in queries with grouped views	YES	
F131-02	Multiple tables supported in queries with grouped views	YES	
F131-03	Set functions supported in queries with grouped views	YES	
F131-04	Subqueries with `GROUP BY` and `HAVING` clauses and grouped views	YES	
F131-05	Single row `SELECT` with `GROUP BY` and `HAVING` clauses and grouped views	YES	
F171	Multiple schemas per user	YES	
F181	Multiple module support	NO	
F191	Referential delete actions	NO	
F200	`TRUNCATE TABLE` statement	YES	
F201	`CAST` function	YES	
F202	`TRUNCATE TABLE`: identity column restart option	NO	
F221	Explicit defaults	YES	
F222	`INSERT` statement: `DEFAULT VALUES` clause	YES	
F231	Privilege tables	YES	
F231-01	`TABLE_PRIVILEGES` view	YES	
F231-02	`COLUMN_PRIVILEGES` view	YES	
F231-03	`USAGE_PRIVILEGES` view	YES	

ID	Feature	Supported	Comments
F251	Domain support		
F261	CASE expression	YES	
F261-01	Simple `CASE`	YES	
F261-02	Searched `CASE`	YES	
F261-03	`NULLIF`	YES	
F261-04	`COALESCE`	YES	
F262	Extended `CASE` expression	NO	
F263	Comma-separated predicates in simple `CASE` expression	NO	
F271	Compound character literals	YES	
F281	`LIKE` enhancements	YES	
F291	`UNIQUE` predicate	NO	
F301	`CORRESPONDING` in query expressions	NO	
F302	INTERSECT table operator	YES	
F302-01	`INTERSECT DISTINCT` table operator	YES	
F302-02	`INTERSECT ALL` table operator	YES	
F304	`EXCEPT ALL` table operator		
F311	Schema definition statement	YES	Partial sub-feature support
F311-01	`CREATE SCHEMA`	YES	
F311-02	`CREATE TABLE` for persistent base tables	YES	
F311-03	`CREATE VIEW`	YES	
F311-04	`CREATE VIEW: WITH CHECK OPTION`	NO	
F311-05	`GRANT` statement	YES	
F312	`MERGE` statement	NO	
F313	Enhanced `MERGE` statement	NO	
F321	User authorization	YES	
F341	Usage Tables	NO	

ID	Feature	Supported	Comments
F361	Subprogram support	YES	
F381	Extended schema manipulation	YES	
F381-01	`ALTER TABLE` statement: `ALTER COLUMN` clause		Some limitations on altering distribution key columns
F381-02	`ALTER TABLE` statement: `ADD CONSTRAINT` clause		
F381-03	`ALTER TABLE` statement: `DROP CONSTRAINT` clause		
F382	Alter column data type	YES	Some limitations on altering distribution key columns
F391	Long identifiers	YES	
F392	Unicode escapes in identifiers	NO	
F393	Unicode escapes in literals	NO	
F394	Optional normal form specification	NO	
F401	Extended joined table	YES	
F401-01	`NATURAL JOIN`	YES	
F401-02	`FULL OUTER JOIN`	YES	
F401-04	`CROSS JOIN`	YES	
F402	Named column joins for LOBs, arrays, and multisets	NO	
F403	Partitioned joined tables	NO	
F411	Time zone specification	YES	Differences regarding literal interpretation
F421	National character	YES	
F431	Read-only scrollable cursors	YES	Forward scrolling only
01	`FETCH` with explicit `NEXT`	YES	
02	`FETCH FIRST`	NO	
03	`FETCH LAST`	YES	
04	`FETCH PRIOR`	NO	
05	`FETCH ABSOLUTE`	NO	
06	`FETCH RELATIVE`	NO	

ID	Feature	Supported	Comments
F441	Extended set function support	YES	
F442	Mixed column references in set functions	YES	
F451	Character set definition	NO	
F461	Named character sets	NO	
F471	Scalar subquery values	YES	
F481	Expanded `NULL` predicate	YES	
F491	Constraint management	YES	
F501	Features and conformance views	YES	
F501-01	`SQL_FEATURES` view	YES	
F501-02	`SQL_SIZING` view	YES	
F501-03	`SQL_LANGUAGES` view	YES	
F502	Enhanced documentation tables	YES	
F502-01	`SQL_SIZING_PROFILES` view	YES	
F502-02	`SQL_IMPLEMENTATION_INFO` view	YES	
F502-03	`SQL_PACKAGES` view	YES	
F521	Assertions	NO	
F531	Temporary tables	YES	Non-standard form
F555	Enhanced seconds precision	YES	
F561	Full value expressions	YES	
F571	Truth value tests	YES	
F591	Derived tables	YES	
F611	Indicator data types	YES	
F641	Row and table constructors	NO	
F651	Catalog name qualifiers	YES	
F661	Simple tables	NO	
F671	Subqueries in `CHECK`	NO	Intentionally ommitted
F672	Retrospective check constraints	YES	

ID	Feature	Supported	Comments
F690	Collation support	NO	
F692	Enhanced collation support	NO	
F693	SQL-session and client module collations	NO	
F695	Translation support	NO	
F696	Additional translation documentation	NO	
F701	Referential update actions	NO	
F711	`ALTER` domain	YES	
F721	Deferrable constraints	NO	
F731	`INSERT` column privileges	NO	
F741	Referential `MATCH` types	NO	No partial match
F751	View `CHECK` enhancements	NO	
F761	Session management	YES	
F762	`CURRENT_CATALOG`	NO	
F763	`CURRENT_SCHEMA`	NO	
F771	Connection management	YES	
F781	Self-referencing operations	YES	
F791	Insensitive cursors	YES	
F801	Full set function	YES	
F812	Basic flagging	NO	
F813	Extended flagging	NO	
F831	Full cursor update	NO	
F841	`LIKE_REGEX` predicate	NO	Non-standard syntax for regex
F842	`OCCURENCES_REGEX` function	NO	
F843	`POSITION_REGEX` function	NO	
F844	`SUBSTRING_REGEX` function	NO	
F845	`TRANSLATE_REGEX` function	NO	

ID	Feature	Supported	Comments
F846	Octet support in regular expression operators	NO	
F847	Nonconstant regular expressions	NO	
F850	Top-level `ORDER BY` clause in *query expression*	YES	
F851	Top-level `ORDER BY` clause in subqueries	NO	
F852	Top-level `ORDER BY` clause in views	NO	
F855	Nested `ORDER BY` clause in *query expression*	NO	
F856	Nested `FETCH FIRST` clause in *query expression*	NO	
F857	Top-level `FETCH FIRST` clause in *query expression*	NO	
F858	`FETCH FIRST` clause in subqueries	NO	
F859	Top-level `FETCH FIRST` clause in views	NO	
F860	`FETCH FIRST ROW`*count* in `FETCH FIRST` clause	NO	
F861	Top-level `RESULT OFFSET` clause in *query expression*	NO	
F862	`RESULT OFFSET` clause in subqueries	NO	
F863	Nested `RESULT OFFSET` clause in *query expression*	NO	
F864	Top-level `RESULT OFFSET` clause in views	NO	
F865	`OFFSET ROW`*count* in `RESULT OFFSET` clause	NO	
S011	Distinct data types	NO	
S023	Basic structured types	NO	
S024	Enhanced structured types	NO	
S025	Final structured types	NO	

ID	Feature	Supported	Comments
S026	Self-referencing structured types	NO	
S027	Create method by specific method name	NO	
S028	Permutable UDT options list	NO	
S041	Basic reference types	NO	
S043	Enhanced reference types	NO	
S051	Create table of type	NO	
S071	SQL paths in function and type name resolution	YES	
S091	Basic array support	NO	Greenplum has arrays, but is not fully standards compliant
S091-01	Arrays of built-in data types	NO	Partially compliant
S091-02	Arrays of distinct types	NO	
S091-03	Array expressions	NO	
S092	Arrays of user-defined types	NO	
S094	Arrays of reference types	NO	
S095	Array constructors by query	NO	
S096	Optional array bounds	NO	
S097	Array element assignment	NO	

ID	Feature	Supported	Comments
S098	`ARRAY_AGG`	Partially	Supported: Using array_agg without a window specification; for example SELECT array_agg(x) FROM ... SELECT array_agg (x order by y) FROM ... Not supported: Using array_agg as an aggregate derived window function; for example SELECT array_agg(x) over (ORDER BY y) FROM ... SELECT array_agg(x order by y) over (PARTITION BY z) FROM ... SELECT array_agg(x order by y) over (ORDER BY z) FROM ...
S111	`ONLY` in query expressions	YES	
S151	Type predicate	NO	
S161	Subtype treatment	NO	
S162	Subtype treatment for references	NO	
S201	SQL-invoked routines on arrays	NO	Functions can be passed Greenplum array types
S202	SQL-invoked routines on multisets	NO	
S211	User-defined cast functions	YES	
S231	Structured type locators	NO	
S232	Array locators	NO	
S233	Multiset locators	NO	
S241	Transform functions	NO	
S242	Alter transform statement	NO	
S251	User-defined orderings	NO	

ID	Feature	Supported	Comments
S261	Specific type method	NO	
S271	Basic multiset support	NO	
S272	Multisets of user-defined types	NO	
S274	Multisets of reference types	NO	
S275	Advanced multiset support	NO	
S281	Nested collection types	NO	
S291	Unique constraint on entire row	NO	
S301	Enhanced `UNNEST`	NO	
S401	Distinct types based on array types	NO	
S402	Distinct types based on distinct types	NO	
S403	`MAX_CARDINALITY`	NO	
S404	`TRIM_ARRAY`	NO	
T011	Timestamp in Information Schema	NO	
T021	`BINARY` and `VARBINARY` data types	NO	
T022	Advanced support for `BINARY` and `VARBINARY` data types	NO	
T023	Compound binary literal	NO	
T024	Spaces in binary literals	NO	
T031	`BOOLEAN` data type	YES	
T041	Basic `LOB` data type support	NO	
T042	Extended `LOB` data type support	NO	
T043	Multiplier T	NO	
T044	Multiplier P	NO	
T051	Row types	NO	
T052	`MAX` and `MIN` for row types	NO	
T053	Explicit aliases for all-fields reference	NO	

ID	Feature	Supported	Comments
T061	UCS support	NO	
T071	`BIGINT` data type	YES	
T101	Enhanced nullability determiniation	NO	
T111	Updatable joins, unions, and columns	NO	
T121	`WITH` (excluding `RECURSIVE`) in query expression	NO	
T122	`WITH` (excluding `RECURSIVE`) in subquery	NO	
T131	Recursive query	NO	
T132	Recursive query in subquery	NO	
T141	`SIMILAR` predicate	YES	
T151	`DISTINCT` predicate	YES	
T152	`DISTINCT` predicate with negation	NO	
T171	`LIKE` clause in table definition	YES	
T172	`AS` subquery clause in table definition	YES	
T173	Extended `LIKE` clause in table definition	YES	
T174	Identity columns	NO	
T175	Generated columns	NO	
T176	Sequence generator support	NO	
T177	Sequence generator support: simple restart option	NO	
T178	Identity columns: simple restart option	NO	
T191	Referential action `RESTRICT`	NO	
T201	Comparable data types for referential constraints	NO	
T211	Basic trigger capability	NO	

ID	Feature	Supported	Comments
T211-01	Triggers activated on `UPDATE`, `INSERT`, or `DELETE` of one base table	NO	
T211-02	`BEFORE` triggers	NO	
T211-03	`AFTER` triggers	NO	
T211-04	`FOR EACH ROW` triggers	NO	
T211-05	Ability to specify a search condition that must be true before the trigger is invoked	NO	
T211-06	Support for run-time rules for the interaction of triggers and constraints	NO	
T211-07	`TRIGGER` privilege	YES	
T211-08	Multiple triggers for the same event are executed in the order in which they were created in the catalog	NO	Intentionally omitted
T212	Enhanced trigger capability	NO	
T213	`INSTEAD OF` triggers	NO	
T231	Sensitive cursors	YES	
T241	`START TRANSACTION` statement	YES	
T251	`SET TRANSACTION` statement: `LOCAL` option	NO	
T261	Chained transactions	NO	
T271	Savepoints	YES	
T272	Enhanced savepoint management	NO	
T281	`SELECT` privilege with column granularity	NO	
T285	Enhanced derived column names	NO	
T301	Functional dependencies	NO	
T312	OVERLAY function	YES	
T321	Basic SQL-invoked routines	NO	Partial support

ID	Feature	Supported	Comments
T321-01	User-defined functions with no overloading	YES	
T321-02	User-defined stored procedures with no overloading	NO	
T321-03	Function invocation	YES	
T321-04	CALL statement	NO	
T321-05	RETURN statement	NO	
T321-06	ROUTINES view	YES	
T321-07	PARAMETERS view	YES	
T322	Overloading of SQL-invoked functions and procedures	YES	
T323	Explicit security for external routines	YES	
T324	Explicit security for SQL routines	NO	
T325	Qualified SQL parameter references	NO	
T326	Table functions	NO	
T331	Basic roles	NO	
T332	Extended roles	NO	
T351	Bracketed SQL comments (`/*...*/` comments)	YES	
T431	Extended grouping capabilities	NO	
T432	Nested and concatenated `GROUPING SETS`	NO	
T433	Multiargument `GROUPING` function	NO	
T434	`GROUP BY DISTINCT`	NO	
T441	`ABS` and `MOD` functions	YES	
T461	Symmetric `BETWEEN` predicate	YES	
T471	Result sets return value	NO	
T491	`LATERAL` derived table	NO	
T501	Enhanced `EXISTS` predicate	NO	

ID	Feature	Supported	Comments
T511	Transaction counts	NO	
T541	Updatable table references	NO	
T561	Holdable locators	NO	
T571	Array-returning external SQL-invoked functions	NO	
T572	Multiset-returning external SQL-invoked functions	NO	
T581	Regular expression substring function	YES	
T591	`UNIQUE` constraints of possibly null columns	YES	
T601	Local cursor references	NO	
T611	Elementary OLAP operations	YES	
T612	Advanced OLAP operations	NO	Partially supported
T613	Sampling	NO	
T614	`NTILE` function	YES	
T615	`LEAD` and `LAG` functions	YES	
T616	Null treatment option for `LEAD` and `LAG` functions	NO	
T617	`FIRST_VALUE` and `LAST_VALUE` function	YES	
T618	`NTH_VALUE`	NO	Function exists in Greenplum but not all options are supported
T621	Enhanced numeric functions	YES	
T631	N predicate with one list element	NO	
T641	Multiple column assignment	NO	Some syntax variants supported
T651	SQL-schema statements in SQL routines	NO	
T652	SQL-dynamic statements in SQL routines	NO	
T653	SQL-schema statements in external routines	NO	

ID	Feature	Supported	Comments
T654	SQL-dynamic statements in external routines	NO	
T655	Cyclically dependent routines	NO	
M001	Datalinks	NO	
M002	Datalinks via SQL/CLI	NO	
M003	Datalinks via Embedded SQL	NO	
M004	Foreign data support	NO	
M005	Foreign schema support	NO	
M006	GetSQLString routine	NO	
M007	TransmitRequest	NO	
M009	GetOpts and GetStatistics routines	NO	
M010	Foreign data wrapper support	NO	
M011	Datalinks via Ada	NO	
M012	Datalinks via C	NO	
M013	Datalinks via COBOL	NO	
M014	Datalinks via Fortran	NO	
M015	Datalinks via M	NO	
M016	Datalinks via Pascal	NO	
M017	Datalinks via PL/I	NO	
M018	Foreign data wrapper interface routines in Ada	NO	
M019	Foreign data wrapper interface routines in C	NO	
M020	Foreign data wrapper interface routines in COBOL	NO	
M021	Foreign data wrapper interface routines in Fortran	NO	
M022	Foreign data wrapper interface routines in MUMPS	NO	
M023	Foreign data wrapper interface routines in Pascal	NO	

ID	Feature	Supported	Comments
M024	Foreign data wrapper interface routines in PL/I	NO	
M030	SQL-server foreign data support	NO	
M031	Foreign data wrapper general routines	NO	
X010	XML type	YES	
X011	Arrays of XML type	YES	
X012	Multisets of XML type	NO	
X013	Distinct types of XML type	NO	
X014	Attributes of XML type	NO	
X015	Fields of XML type	NO	
X016	Persistent XML values	YES	
X020	XMLConcat	NO	xmlconcat2() supported
X025	XMLCast	NO	
X030	XMLDocument	NO	
X031	XMLElement	NO	
X032	XMLForest	NO	
X034	XMLAgg	YES	
X035	XMLAgg: ORDER BY option	YES	
X036	XMLComment	YES	
X037	XMLPI	NO	
X038	XMLText	NO	
X040	Basic table mapping	NO	
X041	Basic table mapping: nulls absent	NO	
X042	Basic table mapping: null as nil	NO	
X043	Basic table mapping: table as forest	NO	
X044	Basic table mapping: table as element	NO	
X045	Basic table mapping: with target namespace	NO	
X046	Basic table mapping: data mapping	NO	

ID	Feature	Supported	Comments
X047	Basic table mapping: metadata mapping	NO	
X048	Basic table mapping: base64 encoding of binary strings	NO	
X049	Basic table mapping: hex encoding of binary strings	NO	
X051	Advanced table mapping: nulls absent	NO	
X052	Advanced table mapping: null as nil	NO	
X053	Advanced table mapping: table as forest	NO	
X054	Advanced table mapping: table as element	NO	
X055	Advanced table mapping: target namespace	NO	
X056	Advanced table mapping: data mapping	NO	
X057	Advanced table mapping: metadata mapping	NO	
X058	Advanced table mapping: base64 encoding of binary strings	NO	
X059	Advanced table mapping: hex encoding of binary strings	NO	
X060	XMLParse: Character string input and CONTENT option	NO	xml() supported
X061	XMLParse: Character string input and DOCUMENT option	NO	xml() supported
X065	XMLParse: BLOB input and CONTENT option	NO	
X066	XMLParse: BLOB input and DOCUMENT option	NO	
X068	XMLSerialize: BOM	NO	
X069	XMLSerialize: INDENT	NO	

ID	Feature	Supported	Comments
X070	XMLSerialize: Character string serialization and CONTENT option	NO	text(xml) supported
X071	XMLSerialize: Character string serialization and DOCUMENT option	NO	text(xml) supported
X072	XMLSerialize: Character string serialization	NO	text(xml) supported
X073	XMLSerialize: BLOB serialization and CONTENT option	NO	
X074	XMLSerialize: BLOB serialization and DOCUMENT option	NO	
X075	XMLSerialize: BLOB serialization	NO	
X076	XMLSerialize: VERSION	NO	
X077	XMLSerialize: explicit ENCODING option	NO	
X078	XMLSerialize: explicit XML declaration	NO	
X080	Namespaces in XML publishing	NO	
X081	Query-level XML namespace declarations	NO	
X082	XML namespace declarations in DML	NO	
X083	XML namespace declarations in DDL	NO	
X084	XML namespace declarations in compound statements	NO	
X085	Predefined namespace prefixes	NO	
X086	XML namespace declarations in XMLTable	NO	
X090	XML document predicate	NO	xml_is_well_formed_ document() supported
X091	XML content predicate	NO	xml_is_well_formed_ content() supported
X096	XMLExists	NO	xmlexists() supported

ID	Feature	Supported	Comments
X100	Host language support for XML: CONTENT option	NO	
X101	Host language support for XML: DOCUMENT option	NO	
X110	Host language support for XML: VARCHAR mapping	NO	
X111	Host language support for XML: CLOB mapping	NO	
X112	Host language support for XML: BLOB mapping	NO	
X113	Host language support for XML: STRIP WHITESPACE option	NO	
X114	Host language support for XML: PRESERVE WHITESPACE option	NO	
X120	XML parameters in SQL routines	YES	
X121	XML parameters in external routines	YES	
X131	Query-level XMLBINARY clause	NO	
X132	XMLBINARY clause in DML	NO	
X133	XMLBINARY clause in DDL	NO	
X134	XMLBINARY clause in compound statements	NO	
X135	XMLBINARY clause in subqueries	NO	
X141	IS VALID predicate: data-driven case	NO	
X142	IS VALID predicate: ACCORDING TO clause	NO	
X143	IS VALID predicate: ELEMENT clause	NO	
X144	IS VALID predicate: schema location	NO	

ID	Feature	Supported	Comments
X145	IS VALID predicate outside check constraints	NO	
X151	IS VALID predicate with DOCUMENT option	NO	
X152	IS VALID predicate with CONTENT option	NO	
X153	IS VALID predicate with SEQUENCE option	NO	
X155	IS VALID predicate: NAMESPACE without ELEMENT clause	NO	
X157	IS VALID predicate: NO NAMESPACE with ELEMENT clause	NO	
X160	Basic Information Schema for registered XML Schemas	NO	
X161	Advanced Information Schema for registered XML Schemas	NO	
X170	XML null handling options	NO	
X171	NIL ON NO CONTENT option	NO	
X181	XML(DOCUMENT (UNTYPED)) type	NO	
X182	XML(DOCUMENT (ANY)) type	NO	
X190	XML(SEQUENCE) type	NO	
X191	XML(DOCUMENT (XMLSCHEMA)) type	NO	
X192	XML(CONTENT (XMLSCHEMA)) type	NO	
X200	XMLQuery	NO	
X201	XMLQuery: RETURNING CONTENT	NO	
X202	XMLQuery: RETURNING SEQUENCE	NO	
X203	XMLQuery: passing a context item	NO	

ID	Feature	Supported	Comments
X204	XMLQuery: initializing an XQuery variable	NO	
X205	XMLQuery: EMPTY ON EMPTY option	NO	
X206	XMLQuery: NULL ON EMPTY option	NO	
X211	XML 1.1 support	NO	
X221	XML passing mechanism BY VALUE	NO	
X222	XML passing mechanism BY REF	NO	
X231	XML(CONTENT (UNTYPED)) type	NO	
X232	XML(CONTENT (ANY)) type	NO	
X241	RETURNING CONTENT in XML publishing	NO	
X242	RETURNING SEQUENCE in XML publishing	NO	
X251	Persistent XML values of XML(DOCUMENT (UNTYPED)) type	NO	
X252	Persistent XML values of XML(DOCUMENT (ANY)) type	NO	
X253	Persistent XML values of XML(CONTENT (UNTYPED)) type	NO	
X254	Persistent XML values of XML(CONTENT (ANY)) type	NO	
X255	Persistent XML values of XML(SEQUENCE) type	NO	
X256	Persistent XML values of XML(DOCUMENT (XMLSCHEMA)) type	NO	
X257	Persistent XML values of XML(CONTENT (XMLSCHEMA) type	NO	
X260	XML type: ELEMENT clause	NO	

ID	Feature	Supported	Comments
X261	XML type: NAMESPACE without ELEMENT clause	NO	
X263	XML type: NO NAMESPACE with ELEMENT clause	NO	
X264	XML type: schema location	NO	
X271	XMLValidate: data-driven case	NO	
X272	XMLValidate: ACCORDING TO clause	NO	
X273	XMLValidate: ELEMENT clause	NO	
X274	XMLValidate: schema location	NO	
X281	XMLValidate: with DOCUMENT option	NO	
X282	XMLValidate with CONTENT option	NO	
X283	XMLValidate with SEQUENCE option	NO	
X284	XMLValidate NAMESPACE without ELEMENT clause	NO	
X286	XMLValidate: NO NAMESPACE with ELEMENT clause	NO	
X300	XMLTable	NO	
X301	XMLTable: derived column list option	NO	
X302	XMLTable: ordinality column option	NO	
X303	XMLTable: column default option	NO	
X304	XMLTable: passing a context item	NO	
X305	XMLTable: initializing an XQuery variable	NO	
X400	Name and identifier mapping	NO	

Chapter 4

Greenplum Environment Variables

This reference lists and describes the environment variables to set for Greenplum Database. Set these in your user's startup shell profile (such as `~/.bashrc` or `~/.bash_profile`), or in `/etc/profile` if you want to set them for all users.

- *Required Environment Variables*
- *Optional Environment Variables*

Required Environment Variables

Note: `GPHOME`, `PATH` and `LD_LIBRARY_PATH` can be set by sourcing the `greenplum_path.sh` file from your Greenplum Database installation directory.

GPHOME

This is the installed location of your Greenplum Database software. For example:

```
GPHOME=/usr/local/greenplum-db-4.3.x.x
export GPHOME
```

PATH

Your `PATH` environment variable should point to the location of the Greenplum Database `bin` directory. For example:

```
PATH=$GPHOME/bin:$PATH
export PATH
```

LD_LIBRARY_PATH

The `LD_LIBRARY_PATH` environment variable should point to the location of the Greenplum Database/ PostgreSQL library files. For example:

```
LD_LIBRARY_PATH=$GPHOME/lib
export LD_LIBRARY_PATH
```

MASTER_DATA_DIRECTORY

This should point to the directory created by the gpinitsystem utility in the master data directory location. For example:

```
MASTER_DATA_DIRECTORY=/data/master/gpseg-1
export MASTER_DATA_DIRECTORY
```

Optional Environment Variables

The following are standard PostgreSQL environment variables, which are also recognized in Greenplum Database. You may want to add the connection-related environment variables to your profile for convenience, so you do not have to type so many options on the command line for client connections. Note that these environment variables should be set on the Greenplum Database master host only.

PGAPPNAME

The name of the application that is usually set by an application when it connects to the server. This name is displayed in the activity view and in log entries. The `PGAPPNAME` environmental variable behaves the same as the `application_name` connection parameter. The default value for `application_name` is *psql*. The name cannot be longer than 63 characters.

PGDATABASE

The name of the default database to use when connecting.

PGHOST

The Greenplum Database master host name.

PGHOSTADDR

The numeric IP address of the master host. This can be set instead of or in addition to `PGHOST` to avoid DNS lookup overhead.

PGPASSWORD

The password used if the server demands password authentication. Use of this environment variable is not recommended for security reasons (some operating systems allow non-root users to see process environment variables via `ps`). Instead consider using the `~/.pgpass` file.

PGPASSFILE

The name of the password file to use for lookups. If not set, it defaults to `~/.pgpass`. See the topic about *The Password File* in the PostgreSQL documentation for more information.

PGOPTIONS

Sets additional configuration parameters for the Greenplum Database master server.

PGPORT

The port number of the Greenplum Database server on the master host. The default port is 5432.

PGUSER

The Greenplum Database user name used to connect.

PGDATESTYLE

Sets the default style of date/time representation for a session. (Equivalent to `SET datestyle TO...`)

PGTZ

Sets the default time zone for a session. (Equivalent to `SET timezone TO...`)

PGCLIENTENCODING

Sets the default client character set encoding for a session. (Equivalent to `SET client_encoding TO...`)

Chapter 5

System Catalog Reference

This reference describes the Greenplum Database system catalog tables and views. System tables prefixed with `gp_` relate to the parallel features of Greenplum Database. Tables prefixed with `pg_` are either standard PostgreSQL system catalog tables supported in Greenplum Database, or are related to features Greenplum that provides to enhance PostgreSQL for data warehousing workloads. Note that the global system catalog for Greenplum Database resides on the master instance.

- *System Tables*
- *System Views*
- *System Catalogs Definitions*

System Tables

- gp_configuration (Deprecated. See *gp_segment_configuration*.)
- *gp_configuration_history*
- *gp_db_interfaces*
- *gp_distribution_policy*
- *gp_fastsequence*
- *gp_fault_strategy*
- *gp_global_sequence*
- *gp_id*
- *gp_interfaces*
- gp_master_mirroring (Deprecated. See *pg_stat_replication*.)
- *gp_persistent_database_node*
- *gp_persistent_filespace_node*
- *gp_persistent_relation_node*
- *gp_persistent_tablespace_node*
- *gp_relation_node*
- *gp_san_configuration*
- *gp_segment_configuration*
- *gp_version_at_initdb*
- *gpexpand.status*
- *gpexpand.status_detail*
- *pg_aggregate*
- *pg_am*
- *pg_amop*
- *pg_amproc*
- *pg_appendonly*
- pg_appendonly_alter_column (not supported in 4.3)
- *pg_attrdef*
- *pg_attribute*
- *pg_auth_members*
- *pg_authid*
- pg_autovacuum (not supported in 4.2.5 and later)
- *pg_cast*
- *pg_class*
- *pg_constraint*
- *pg_conversion*
- *pg_database*
- *pg_depend*
- *pg_description*
- *pg_exttable*
- *pg_filespace*
- *pg_filespace_entry*
- pg_foreign_data_wrapper (not supported in 4.3)
- pg_foreign_server (not supported in 4.3)

- pg_foreign_table (not supported in 4.3)
- *pg_index*
- *pg_inherits*
- *pg_language*
- *pg_largeobject*
- *pg_listener*
- *pg_namespace*
- *pg_opclass*
- *pg_operator*
- *pg_partition*
- *pg_partition_rule*
- *pg_pltemplate*
- *pg_proc*
- *pg_resourcetype*
- *pg_resqueue*
- *pg_resqueuecapability*
- *pg_rewrite*
- *pg_shdepend*
- *pg_shdescription*
- *pg_stat_last_operation*
- *pg_stat_last_shoperation*
- *pg_statistic*
- *pg_tablespace*
- *pg_trigger*
- *pg_type*
- pg_user_mapping (not supported in 4.3)
- *pg_window*

System Views

Greenplum Database provides the following system views not available in PostgreSQL.

- *gp_distributed_log*
- *gp_distributed_xacts*
- *gp_pgdatabase*
- *gp_resqueue_status*
- *gp_transaction_log*
- *gpexpand.expansion_progress*
- pg_max_external_files (shows number of external table files allowed per segment host when using the file protocol)
- *pg_partition_columns*
- *pg_partition_templates*
- *pg_partitions*
- *pg_stat_activity*
- *pg_stat_replication*
- *pg_resqueue_attributes*
- pg_resqueue_status (Deprecated. Use gp_toolkit.gp_resqueue_status.)
- *pg_stat_resqueues*
- pg_user_mappings (not supported)
- session_level_memory_consumption (See "Viewing Session Memory Usage Information" in the *Greenplum Database Administrator Guide*.)

For more information about the standard system views supported in PostgreSQL and Greenplum Database, see the following sections of the PostgreSQL documentation:

- *System Views*
- *Statistics Collector Views*
- *The Information Schema*

System Catalogs Definitions

System catalog table and view definitions in alphabetical order.

gp_configuration_history

The `gp_configuration_history` table contains information about system changes related to fault detection and recovery operations. The `fts_probe` process logs data to this table, as do certain related management utilities such as `gpcheck`, `gprecoverseg`, and `gpinitsystem`. For example, when you add a new segment and mirror segment to the system, records for these events are logged to `gp_configuration_history`.

The event descriptions stored in this table may be helpful for troubleshooting serious system issues in collaboration with Greenplum support technicians.

This table is populated only on the master. This table is defined in the `pg_global` tablespace, meaning it is globally shared across all databases in the system.

Table 10: pg_catalog.gp_configuration_history

column	type	references	description
`time`	timestamp with time zone		Timestamp for the event recorded.
`dbid`	smallint	gp_segment _ configuration.dbid	System-assigned ID. The unique identifier of a segment (or master) instance.
`desc`	text		Text description of the event.

For information about `gpcheck`, `gprecoverseg`, and `gpinitsystem`, see the Greenplum Database Utility Guide.

gp_db_interfaces

The `gp_db_interfaces` table contains information about the relationship of segments to network interfaces. This information, joined with data from *gp_interfaces*, is used by the system to optimize the usage of available network interfaces for various purposes, including fault detection.

Table 11: pg_catalog.gp_db_interfaces

column	type	references	description
`dbid`	smallint	gp_segment_ configuration.dbid	System-assigned ID. The unique identifier of a segment (or master) instance.
`interfaceid`	smallint	gp_interfaces.interfaceid	System-assigned ID for a network interface.
`priority`	smallint		Priority of the network interface for this segment.

gp_distributed_log

The `gp_distributed_log` view contains status information about distributed transactions and their associated local transactions. A distributed transaction is a transaction that involves modifying data on the segment instances. Greenplum's distributed transaction manager ensures that the segments stay in synch. This view allows you to see the status of distributed transactions.

Table 12: pg_catalog.gp_distributed_log

column	type	references	description
`segment_id`	smallint	gp_segment_ configuration.content	The content id if the segment. The master is always -1 (no content).
`dbid`	small_int	gp_segment_ configuration.dbid	The unique id of the segment instance.
`distributed_xid`	xid		The global transaction id.
`distributed_id`	text		A system assigned ID for a distributed transaction.
`status`	text		The status of the distributed transaction (Committed or Aborted).
`local_transaction`	xid		The local transaction ID.

gp_distributed_xacts

The `gp_distributed_xacts` view contains information about Greenplum Database distributed transactions. A distributed transaction is a transaction that involves modifying data on the segment instances. Greenplum's distributed transaction manager ensures that the segments stay in synch. This view allows you to see the currently active sessions and their associated distributed transactions.

Table 13: pg_catalog.gp_distributed_xacts

column	type	references	description
`distributed_xid`	xid		The transaction ID used by the distributed transaction across the Greenplum Database array.
`distributed_id`	text		The distributed transaction identifier. It has 2 parts — a unique timestamp and the distributed transaction number.
`state`	text		The current state of this session with regards to distributed transactions.

column	type	references	description
gp_session_id	int		The ID number of the Greenplum Database session associated with this transaction.
xmin_distributed _ snapshot	xid		The minimum distributed transaction number found among all open transactions when this transaction was started. It is used for MVCC distributed snapshot purposes.

gp_distribution_policy

The gp_distribution_policy table contains information about Greenplum Database tables and their policy for distributing table data across the segments. This table is populated only on the master. This table is not globally shared, meaning each database has its own copy of this table.

Table 14: pg_catalog.gp_distribution_policy

column	type	references	description
localoid	oid	pg_class.oid	The table object identifier (OID).
attrnums	smallint[]	pg_attribute.attnum	The column number(s) of the distribution column(s).

gpexpand.expansion_progress

The gpexpand.expansion_progress view contains information about the status of a system expansion operation. The view provides calculations of the estimated rate of table redistribution and estimated time to completion.

Status for specific tables involved in the expansion is stored in *gpexpand.status_detail*.

Table 15: gpexpand.expansion_progress

column	type	references	description
`name`	text		Name for the data field provided. Includes: Bytes Left Bytes Done Estimated Expansion Rate Estimated Time to Completion Tables Expanded Tables Left
`value`	text		The value for the progress data. For example: `Estimated Expansion Rate - 9.75667095996092 MB/s`

gpexpand.status

The `gpexpand.status` table contains information about the status of a system expansion operation. Status for specific tables involved in the expansion is stored in *gpexpand.status_detail*.

In a normal expansion operation it is not necessary to modify the data stored in this table.

Table 16: gpexpand.status

column	type	references	description

`status`	text		Tracks the status of an expansion operation. Valid values are:
			`SETUP`
			`SETUP DONE`
			`EXPANSION STARTED`
			`EXPANSION STOPPED`
			`COMPLETED`

`updated`	timestamp with time zone		Timestamp of the last change in status.

gpexpand.status_detail

The `gpexpand.status_detail` table contains information about the status of tables involved in a system expansion operation. You can query this table to determine the status of tables being expanded, or to view the start and end time for completed tables.

This table also stores related information about the table such as the oid, disk size, and normal distribution policy and key. Overall status information for the expansion is stored in *gpexpand.status*.

In a normal expansion operation it is not necessary to modify the data stored in this table. .

Table 17: gpexpand.status_detail

column	type	references	description
`dbname`	text		Name of the database to which the table belongs.
`fq_name`	text		Fully qualified name of the table.
`schema_oid`	oid		OID for the schema of the database to which the table belongs.
`table_oid`	oid		OID of the table.
`distribution_policy`	smallint()		Array of column IDs for the distribution key of the table.
`distribution_policy_names`	text		Column names for the hash distribution key.
`distribution_policy_coloids`	text		Column IDs for the distribution keys of the table.
`storage_options`	text		Not enabled in this release. Do not update this field.
`rank`	int		Rank determines the order in which tables are expanded. The expansion utility will sort on rank and expand the lowest-ranking tables first.
`status`	text		Status of expansion for this table. Valid values are: NOT STARTED IN PROGRESS FINISHED NO LONGER EXISTS
`last updated`	timestamp with time zone		Timestamp of the last change in status for this table.

column	type	references	description
`expansion started`	timestamp with time zone		Timestamp for the start of the expansion of this table. This field is only populated after a table is successfully expanded.
`expansion finished`	timestamp with time zone		Timestamp for the completion of expansion of this table.
`source bytes`			The size of disk space associated with the source table. Due to table bloat in heap tables and differing numbers of segments after expansion, it is not expected that the final number of bytes will equal the source number. This information is tracked to help provide progress measurement to aid in duration estimation for the end-to-end expansion operation.

gp_fastsequence

The `gp_fastsequence` table contains information about append-optimized and column-oriented tables. The `last_sequence` value indicates maximum row number currently used by the table.

Table 18: pg_catalog.gp_fastsequence

column	type	references	description
`objid`	oid	pg_class.oid	Object id of the `pg_aoseg.pg_aocsseg_*` table used to track append-optimized file segments.
`objmod`	bigint		Object modifier.
`last_sequence`	bigint		The last sequence number used by the object.

gp_fault_strategy

The `gp_fault_strategy` table specifies the fault action.

Table 19: pg_catalog.gp_fault_strategy

column	type	references	description
fault_strategy	char		The mirror failover action to take when a segment failure occurs: n = nothing f = file-based failover s = SAN-based failover

gp_global_sequence

The gp_global_sequence table contains the log sequence number position in the transaction log, which is used by the file replication process to determine the file blocks to replicate from a primary to a mirror segment.

Table 20: pg_catalog.gp_global_sequence

column	type	references	description
sequence_num	bigint		Log sequence number position in the transaction log

gp_id

The gp_id system catalog table identifies the Greenplum Database system name and number of segments for the system. It also has local values for the particular database instance (segment or master) on which the table resides. This table is defined in the pg_global tablespace, meaning it is globally shared across all databases in the system.

Table 21: pg_catalog.gp_id

column	type	references	description
gpname	name		The name of this Greenplum Database system.
numsegments	integer		The number of segments in the Greenplum Database system.
dbid	integer		The unique identifier of this segment (or master) instance.

column	type	references	description
content	integer		The ID for the portion of data on this segment instance. A primary and its mirror will have the same content ID. For a segment the value is from 0-*N*, where *N* is the number of segments in Greenplum Database. For the master, the value is -1.

gp_interfaces

The gp_interfaces table contains information about network interfaces on segment hosts. This information, joined with data from *gp_db_interfaces*, is used by the system to optimize the usage of available network interfaces for various purposes, including fault detection.

Table 22: gp_interfaces

column	type	references	description
interfaceid	smallint		System-assigned ID. The unique identifier of a network interface.
address	name		Hostname address for the segment host containing the network interface. Can be a numeric IP address or a hostname.
status	smallint		Status for the network interface. A value of 0 indicates that the interface is unavailable.

gp_persistent_database_node

The gp_persistent_database_node table keeps track of the status of file system objects in relation to the transaction status of database objects. This information is used to make sure the state of the system catalogs and the file system files persisted to disk are synchronized. This information is used by the primary to mirror file replication process.

Table 23: pg_catalog.gp_persistent_database_node

column	type	references	description
tablespace_oid	oid	pg_tablespace.oid	Table space object id.
database_oid	oid	pg_database.oid	Database object id.

column	type	references	description
persistent_state	smallint		0 - free 1 - create pending 2 - created 3 - drop pending 4 - aborting create 5 - "Just in Time" create pending 6 - bulk load create pending
mirror_existence _ state	smallint		0 - none 1 - not mirrored 2 - mirror create pending 3 - mirrorcreated 4 - mirror down before create 5 - mirror down during create 6 - mirror drop pending 7 - only mirror drop remains
parent_xid	integer		Global transaction id.
persistent_serial_ num	bigint		Log sequence number position in the transaction log for a file block.
previous_free_tid	tid		Used by Greenplum Database to internally manage persistent representations of file system objects.

gp_persistent_filespace_node

The gp_persistent_filespace_node table keeps track of the status of file system objects in relation to the transaction status of filespace objects. This information is used to make sure the state of the system catalogs and the file system files persisted to disk are synchronized. This information is used by the primary to mirror file replication process.

Table 24: pg_catalog.gp_persistent_filespace_node

column	type	references	description
filespace_oid	oid	pg_filespace.oid	object id of the filespace
db_id_1	smallint		primary segment id

column	type	references	description
location_1	text		primary filesystem location
db_id_2	smallint		mirror segment id
location_2	text		mirror filesystem location
persistent_state	smallint		0 - free 1 - create pending 2 - created 3 - drop pending 4 - aborting create 5 - "Just in Time" create pending 6 - bulk load create pending
mirror_existence _ state	smallint		0 - none 1 - not mirrored 2 - mirror create pending 3 - mirrorcreated 4 - mirror down before create 5 - mirror down during create 6 - mirror drop pending 7 - only mirror drop remains
parent_xid	integer		Global transaction id.
persistent_serial_ num	bigint		Log sequence number position in the transaction log for a file block.
previous_free_tid	tid		Used by Greenplum Database to internally manage persistent representations of file system objects.

gp_persistent_relation_node

The gp_persistent_relation_node table table keeps track of the status of file system objects in relation to the transaction status of relation objects (tables, view, indexes, and so on). This information is used to make sure the state of the system catalogs and the file system files persisted to disk are synchronized. This information is used by the primary to mirror file replication process.

Table 25: pg_catalog.gp_persistent_relation_node

column	type	references	description
`tablespace_oid`	oid	pg_tablespace.oid	Tablespace object id
`database_oid`	oid	pg_database.oid	Database object id
`relfilenode_oid`	oid	pg_class.relfilenode	The object id of the relation file node.
`segment_file_num`	integer		For append-optimized tables, the append-optimized segment file number.
`relation_storage_manager`	smallint		Whether the relation is heap storage or append-optimized storage.
`persistent_state`	smallint		0 - free 1 - create pending 2 - created 3 - drop pending 4 - aborting create 5 - "Just in Time" create pending 6 - bulk load create pending
`mirror_existence_state`	smallint		0 - none 1 - not mirrored 2 - mirror create pending 3 - mirrorcreated 4 - mirror down before create 5 - mirror down during create 6 - mirror drop pending 7 - only mirror drop remains
`parent_xid`	integer		Global transaction id.
`persistent_serial_num`	bigint		Log sequence number position in the transaction log for a file block.

column	type	references	description
previous_free_tid	tid		Used by Greenplum Database to internally manage persistent representations of file system objects.

gp_persistent_tablespace_node

The gp_persistent_tablespace_node table keeps track of the status of file system objects in relation to the transaction status of tablespace objects. This information is used to make sure the state of the system catalogs and the file system files persisted to disk are synchronized. This information is used by the primary to mirror file replication process

Table 26: pg_catalog.gp_persistent_tablespace_node

column	type	references	description
filespace_oid	oid	pg_filespace.oid	Filespace object id
tablespace_oid	oid	pg_tablespace.oid	Tablespace object id
persistent_state	smallint		0 - free 1 - create pending 2 - created 3 - drop pending 4 - aborting create 5 - "Just in Time" create pending 6 - bulk load create pending
mirror_existence_state	smallint		0 - none 1 - not mirrored 2 - mirror create pending 3 - mirrorcreated 4 - mirror down before create 5 - mirror down during create 6 - mirror drop pending 7 - only mirror drop remains
parent_xid	integer		Global transaction id.
persistent_serial_num	bigint		Log sequence number position in the transaction log for a file block.

column	type	references	description
`previous_free_tid`	tid		Used by Greenplum Database to internally manage persistent representations of file system objects.

gp_pgdatabase

The `gp_pgdatabase` view shows status information about the Greenplum segment instances and whether they are acting as the mirror or the primary. This view is used internally by the Greenplum fault detection and recovery utilities to determine failed segments.

Table 27: pg_catalog.gp_pgdatabase

column	type	references	description
`dbid`	smallint	gp_segment_ configuration.dbid	System-assigned ID. The unique identifier of a segment (or master) instance.
`isprimary`	boolean	gp_segment_ configuration.role	Whether or not this instance is active. Is it currently acting as the primary segment (as opposed to the mirror).
`content`	smallint	gp_segment_ configuration.content	The ID for the portion of data on an instance. A primary segment instance and its mirror will have the same content ID. For a segment the value is from 0-*N*, where *N* is the number of segments in Greenplum Database. For the master, the value is -1.
`definedprimary`	boolean	gp_segment_ configuration.preferred_ role	Whether or not this instance was defined as the primary (as opposed to the mirror) at the time the system was initialized.

gp_relation_node

The `gp_relation_node` table contains information about the file system objects for a relation (table, view, index, and so on).

Table 28: pg_catalog.gp_relation_node

column	type	references	description
relfilenode_oid	oid	pg_class.relfilenode	The object id of the relation file node.
segment_file_num	integer		For append-optimized tables, the append-optimized segment file number.
persistent_tid	tid		Used by Greenplum Database to internally manage persistent representations of file system objects.
persistent_serial_num	bigint		Log sequence number position in the transaction log for a file block.

gp_resqueue_status

The `gp_toolkit.gp_resqueue_status` view allows administrators to see status and activity for a workload management resource queue. It shows how many queries are waiting to run and how many queries are currently active in the system from a particular resource queue.

Table 29: gp_toolkit.gp_resqueue_status

column	type	references	description
queueid	oid	gp_toolkit.gp_resqueue_queueid	The ID of the resource queue.
rsqname	name	gp_toolkit.gp_resqueue_rsqname	The name of the resource queue.
rsqcountlimit	real	gp_toolkit.gp_resqueue_rsqcountlimit	The active query threshold of the resource queue. A value of -1 means no limit.
rsqcountvalue	real	gp_toolkit.gp_resqueue_rsqcountvalue	The number of active query slots currently being used in the resource queue.
rsqcostlimit	real	gp_toolkit.gp_resqueue_rsqcostlimit	The query cost threshold of the resource queue. A value of -1 means no limit.
rsqcostvalue	real	gp_toolkit.gp_resqueue_rsqcostvalue	The total cost of all statements currently in the resource queue.
rsqmemorylimit	real	gp_toolkit.gp_resqueue_rsqmemorylimit	The memory limit for the resource queue.

column	type	references	description
rsqmemoryvalue	real	gp_toolkit.gp_resqueue_rsqmemoryvalue	The total memory used by all statements currently in the resource queue.
rsqwaiters	integer	gp_toolkit.gp_resqueue_rsqwaiter	The number of statements currently waiting in the resource queue.
rsqholders	integer	gp_toolkit.gp_resqueue_rsqholders	The number of statements currently running on the system from this resource queue.

gp_san_configuration

The gp_san_configuration table contains mount-point information for SAN failover.

Table 30: pg_catalog.gp_san_configuration

column	type	references	description
mountid	smallint		A value that identifies the mountpoint for the primary and mirror hosts. This is the primary key which is referred to by the value that appears in the san_mounts structure in gp_segment_configuration.
active_host	char		The current active host. p indidcates primary, and m indicates mirror.
san_type	char		The type of shared storage in use. n indidcates NFS, and e indicates EMC SAN.
primary_host	text		The name of the primary host system
primary_mountpoint	text		The mount point for the primary host.

column	type	references	description
primary_device	text		A string specifying the device to mount on the primary mountpoint. For NFS, this string is similar to: nfs-server:/exported/fs. For EMC this is a larger string that includes the WWN for the storage processor, the storage-processor IP, and the storage-group name. The primary_device field is identical to the mirror_device field.
mirror_host	text		The name or the mirror/backup host system.
mirror_mountpoint	text		The mount point for the mirror/backup host.
mirror_device	text		A string specifying the device to mount on the mirror mountpoint. For NFS, this string is similar to: nfs-server:/exported/fs. For EMC this is a larger string that includes the WWN for the storage processor, the storage-processor IP, and the storage-group name. The mirror_device field is identical to the primary_device field.

gp_segment_configuration

The gp_segment_configuration table contains information about mirroring and segment configuration.

Table 31: pg_catalog.gp_segment_configuration

column	type	references	description
dbid	smallint		The unique identifier of a segment (or master) instance.

column	type	references	description
`content`	smallint		The content identifier for a segment instance. A primary segment instance and its corresponding mirror will always have the same content identifier. For a segment the value is from 0-*N*, where *N* is the number of primary segments in the system. For the master, the value is always -1.
`role`	char		The role that a segment is currently running as. Values are `p` (primary) or `m` (mirror).
`preferred_role`	char		The role that a segment was originally assigned at initialization time. Values are `p` (primary) or `m` (mirror).
`mode`	char		The synchronization status of a segment with its mirror copy. Values are `s` (synchronized), `c` (change logging), or `r` (resyncing).
`status`	char		The fault status of a segment. Values are `u` (up) or `d` (down).
`port`	integer		The TCP port the database server listener process is using.
`hostname`	text		The hostname of a segment host.
`address`	text		The hostname used to access a particular segment on a segment host. This value may be the same as `hostname` in systems upgraded from 3.x or on systems that do not have per-interface hostnames configured.

column	type	references	description
`replication_port`	integer		The TCP port the file block replication process is using to keep primary and mirror segments synchronized.
`san_mounts`	int2vector	*gp_san_ configuration*. oid	An array of references to the *gp_san_ configuration* table. Only used on systems that were initialized using sharred storage.

gp_transaction_log

The `gp_transaction_log` view contains status information about transactions local to a particular segment. This view allows you to see the status of local transactions.

Table 32: pg_catalog.gp_transaction_log

column	type	references	description
`segment_id`	smallint	gp_segment_ configuration.content	The content id if the segment. The master is always -1 (no content).
`dbid`	smallint	gp_segment_ configuration.dbid	The unique id of the segment instance.
`transaction`	xid		The local transaction ID.
`status`	text		The status of the local transaction (Committed or Aborted).

gp_version_at_initdb

The `gp_version_at_initdb` table is populated on the master and each segment in the Greenplum Database system. It identifies the version of Greenplum Database used when the system was first initialized. This table is defined in the `pg_global` tablespace, meaning it is globally shared across all databases in the system.

Table 33: pg_catalog.gp_version

column	type	references	description
`schemaversion`	integer		Schema version number.
`productversion`	text		Product version number.

pg_aggregate

The `pg_aggregate` table stores information about aggregate functions. An aggregate function is a function that operates on a set of values (typically one column from each row that matches a query condition) and returns a single value computed from all these values. Typical aggregate functions are `sum`, `count`, and max. Each entry in pg_aggregate is an extension of an entry in pg_proc. The pg_proc entry carries the aggregate's name, input and output data types, and other information that is similar to ordinary functions.

Table 34: pg_catalog.pg_aggregate

column	type	references	description
aggfnoid	regproc	pg_proc.oid	Aggregate function OID
aggtransfn	regproc	pg_proc.oid	Transition function OID
aggprelimfn	regproc		Preliminary function OID (zero if none)
aggfinalfn	regproc	pg_proc.oid	Final function OID (zero if none)
agginitval	text		The initial value of the transition state. This is a text field containing the initial value in its external string representation. If this field is NULL, the transition state value starts out NULL
agginvtransfn	regproc	pg_proc.oid	The OID in pg_proc of the inverse function of *aggtransfn*
agginvprelimfn	regproc	pg_proc.oid	The OID in pg_proc of the inverse function of aggprelimfn
aggordered	Boolean		If true, the aggregate is defined as ORDERED.
aggsortop	oid	pg_operator.oid	Associated sort operator OID (zero if none)
aggtranstype	oid	pg_type.oid	Data type of the aggregate function's internal transition (state) data

pg_am

The pg_am table stores information about index access methods. There is one row for each index access method supported by the system.

Table 35: pg_catalog.pg_am

column	type	references	description
amname	name		Name of the access method
amstrategies	int2		Number of operator strategies for this access method

column	type	references	description
`amsupport`	int2		Number of support routines for this access method
`amorderstrategy`	int2		Zero if the index offers no sort order, otherwise the strategy number of the strategy operator that describes the sort order
`amcanunique`	boolean		Does the access method support unique indexes?
`amcanmulticol`	boolean		Does the access method support multicolumn indexes?
`amoptionalkey`	boolean		Does the access method support a scan without any constraint for the first index column?
`amindexnulls`	boolean		Does the access method support null index entries?
`amstorage`	boolean		Can index storage data type differ from column data type?
`amclusterable`	boolean		Can an index of this type be clustered on?
`aminsert`	regproc	pg_proc.oid	"Insert this tuple" function
`ambeginscan`	regproc	pg_proc.oid	"Start new scan" function
`amgettuple`	regproc	pg_proc.oid	"Next valid tuple" function
`amgetmulti`	regproc	pg_proc.oid	"Fetch multiple tuples" function
`amrescan`	regproc	pg_proc.oid	"Restart this scan" function
`amendscan`	regproc	pg_proc.oid	"End this scan" function
`ammarkpos`	regproc	pg_proc.oid	"Mark current scan position" function
`amrestrpos`	regproc	pg_proc.oid	"Restore marked scan position" function
`ambuild`	regproc	pg_proc.oid	"Build new index" function
`ambulkdelete`	regproc	pg_proc.oid	Bulk-delete function

column	type	references	description
amvacuumcleanup	regproc	pg_proc.oid	Post-VACUUM cleanup function
amcostestimate	regproc	pg_proc.oid	Function to estimate cost of an index scan
amoptions	regproc	pg_proc.oid	Function to parse and validate reloptions for an index

pg_amop

The pg_amop table stores information about operators associated with index access method operator classes. There is one row for each operator that is a member of an operator class.

Table 36: pg_catalog.pg_amop

column	type	references	description
amopclaid	oid	pg_opclass.oid	The index operator class this entry is for
amopsubtype	oid	pg_type.oid	Subtype to distinguish multiple entries for one strategy; zero for default
amopstrategy	int2		Operator strategy number
amopreqcheck	boolean		Index hit must be rechecked
amopopr	oid	pg_operator.oid	OID of the operator

pg_amproc

The pg_amproc table stores information about support procedures associated with index access method operator classes. There is one row for each support procedure belonging to an operator class.

Table 37: pg_catalog.pg_amproc

column	type	references	description
amopclaid	oid	pg_opclass.oid	The index operator class this entry is for
amprocsubtype	oid	pg_type.oid	Subtype, if cross-type routine, else zero
amprocnum	int2		Support procedure number
amproc	regproc	pg_proc.oid	OID of the procedure

pg_appendonly

The pg_appendonly table contains information about the storage options and other characteristics of append-optimized tables.

Table 38: pg_catalog.pg_appendonly

column	type	references	description
relid	oid		The table object identifier (OID) of the compressed table.
blocksize	integer		Block size used for compression of append-optimized tables. Valid values are 8K - 2M. Default is 32K.
safefswritesize	integer		Minimum size for safe write operations to append-optimized tables in a non-mature file system. Commonly set to a multiple of the extent size of the file system; for example, Linux ext3 is 4096 bytes, so a value of 32768 is commonly used.
compresslevel	smallint		The compression level, with compression ratio increasing from 1 to 9. When quicklz is specified for compresstype, valid values are 1 or 3. With zlib specified, valid values are 1-9.
majorversion	smallint		The major version number of the pg_ appendonly table.
minorversion	smallint		The minor version number of the pg_ appendonly table.
checksum	boolean		A checksum value that is stored to compare the state of a block of data at compression time and at scan time to ensure data integrity.
compresstype	text		Type of compression used to compress append-optimized tables. Valid values are zlib (gzip compression) and quicklz.

column	type	references	description
`columnstore`	boolean		`1` for column-oriented storage, `0` for row-oriented storage.
`segrelid`	oid		Table on-disk segment file id.
`segidxid`	oid		Index on-disk segment file id.
`blkdirrelid`	oid		Block used for on-disk column-oriented table file.
`blkdiridxid`	oid		Block used for on-disk column-oriented index file.
`visimaprelid`	oid		Visibility map for the table.
`visimapidxid`	oid		B-tree index on the visibility map.

pg_attrdef

The `pg_attrdef` table stores column default values. The main information about columns is stored in *pg_attribute*. Only columns that explicitly specify a default value (when the table is created or the column is added) will have an entry here.

Table 39: pg_catalog.pg_attrdef

column	type	references	description
`adrelid`	oid	pg_class.oid	The table this column belongs to
`adnum`	int2	pg_attribute.attnum	The number of the column
`adbin`	text		The internal representation of the column default value
`adsrc`	text		A human-readable representation of the default value. This field is historical, and is best not used.

pg_attribute

The `pg_attribute` table stores information about table columns. There will be exactly one `pg_attribute` row for every column in every table in the database. (There will also be attribute entries for indexes, and all objects that have `pg_class` entries.) The term attribute is equivalent to column.

Table 40: pg_catalog.pg_attribute

column	type	references	description
`attrelid`	oid	pg_class.oid	The table this column belongs to
`attname`	name		The column name
`atttypid`	oid	pg_type.oid	The data type of this column
`attstattarget`	int4		Controls the level of detail of statistics accumulated for this column by `ANALYZE`. A zero value indicates that no statistics should be collected. A negative value says to use the system default statistics target. The exact meaning of positive values is data type-dependent. For scalar data types, it is both the target number of "most common values" to collect, and the target number of histogram bins to create.
`attlen`	int2		A copy of pg_type.typlen of this column's type.
`attnum`	int2		The number of the column. Ordinary columns are numbered from 1 up. System columns, such as oid, have (arbitrary) negative numbers.
`attndims`	int4		Number of dimensions, if the column is an array type; otherwise `0`. (Presently, the number of dimensions of an array is not enforced, so any nonzero value effectively means it is an array)
`attcacheoff`	int4		Always `-1` in storage, but when loaded into a row descriptor in memory this may be updated to cache the offset of the attribute within the row

column	type	references	description
`atttypmod`	int4		Records type-specific data supplied at table creation time (for example, the maximum length of a varchar column). It is passed to type-specific input functions and length coercion functions. The value will generally be `-1` for types that do not need it.
`attbyval`	boolean		A copy of pg_type. typbyval of this column's type
`attstorage`	char		Normally a copy of pg_type.typstorage of this column's type. For TOAST-able data types, this can be altered after column creation to control storage policy.
`attalign`	char		A copy of pg_type. typalign of this column's type
`attnotnull`	boolean		This represents a not-null constraint. It is possible to change this column to enable or disable the constraint.
`atthasdef`	boolean		This column has a default value, in which case there will be a corresponding entry in the pg_attrdef catalog that actually defines the value
`attisdropped`	boolean		This column has been dropped and is no longer valid. A dropped column is still physically present in the table, but is ignored by the parser and so cannot be accessed via SQL
`attislocal`	boolean		This column is defined locally in the relation. Note that a column may be locally defined and inherited simultaneously

column	type	references	description
attinhcount	int4		The number of direct ancestors this column has. A column with a nonzero number of ancestors cannot be dropped nor renamed

pg_attribute_encoding

The pg_attribute_encoding system catalog table contains column storage information.

Table 41: pg_catalog.pg_attribute_encoding

column	type	modifers	storage	description
attrelid	oid	not null	plain	Foreign key to pg_attribute.attrelid
attnum	smallint	not null	plain	Foreign key to pg_attribute.attnum
attoptions	text [ ]		extended	The options

pg_auth_members

The pg_auth_members system catalog table shows the membership relations between roles. Any non-circular set of relationships is allowed. Because roles are system-wide, pg_auth_members is shared across all databases of a Greenplum Database system.

Table 42: pg_catalog.pg_auth_members

column	type	references	description
roleid	oid	pg_authid.oid	ID of the parent-level (group) role
member	oid	pg_authid.oid	ID of a member role
grantor	oid	pg_authid.oid	ID of the role that granted this membership
admin_option	boolean		True if role member may grant membership to others

pg_authid

The pg_authid table contains information about database authorization identifiers (roles). A role subsumes the concepts of users and groups. A user is a role with the rolcanlogin flag set. Any role (with or without rolcanlogin) may have other roles as members. See *pg_auth_members*.

Since this catalog contains passwords, it must not be publicly readable. *pg_roles* is a publicly readable view on pg_authid that blanks out the password field.

Because user identities are system-wide, pg_authid is shared across all databases in a Greenplum Database system: there is only one copy of pg_authid per system, not one per database.

Table 43: pg_catalog.pg_authid

column	type	references	description
`rolname`	name		Role name
`rolsuper`	boolean		Role has superuser privileges
`rolinherit`	boolean		Role automatically inherits privileges of roles it is a member of
`rolcreaterole`	boolean		Role may create more roles
`rolcreatedb`	boolean		Role may create databases
`rolcatupdate`	boolean		Role may update system catalogs directly. (Even a superuser may not do this unless this column is true)
`rolcanlogin`	boolean		Role may log in. That is, this role can be given as the initial session authorization identifier
`rolconnlimit`	int4		For roles that can log in, this sets maximum number of concurrent connections this role can make. `-1` means no limit
`rolpassword`	text		Password (possibly encrypted); NULL if none
`rolvaliduntil`	timestamptz		Password expiry time (only used for password authentication); NULL if no expiration
`rolconfig`	text[]		Session defaults for server configuration parameters
`relresqueue`	oid		Object ID of the associated resource queue ID in *pg_resqueue*
`rolcreaterextgpfd`	boolean		Privilege to create read external tables with the `gpfdist` or `gpfdists` protocol
`rolcreaterexhttp`	boolean		Privilege to create read external tables with the `http` protocol

column	type	references	description
`rolcreatewextgpfd`	boolean		Privilege to create write external tables with the `gpfdist` or `gpfdists` protocol
`rolcreaterexthdfs`	boolean		Privilege to create read external tables with the `gphdfs` protocol
`rolcreatewexthdfs`	boolean		Privilege to create write external tables with the `gphdfs` protocol

pg_cast

The `pg_cast` table stores data type conversion paths, both built-in paths and those defined with `CREATE CAST`. The cast functions listed in `pg_cast` must always take the cast source type as their first argument type, and return the cast destination type as their result type. A cast function can have up to three arguments. The second argument, if present, must be type `integer`; it receives the type modifier associated with the destination type, or `-1` if there is none. The third argument, if present, must be type `boolean`; it receives `true` if the cast is an explicit cast, `false` otherwise.

It is legitimate to create a `pg_cast` entry in which the source and target types are the same, if the associated function takes more than one argument. Such entries represent 'length coercion functions' that coerce values of the type to be legal for a particular type modifier value. Note however that at present there is no support for associating non-default type modifiers with user-created data types, and so this facility is only of use for the small number of built-in types that have type modifier syntax built into the grammar.

When a `pg_cast` entry has different source and target types and a function that takes more than one argument, it represents converting from one type to another and applying a length coercion in a single step. When no such entry is available, coercion to a type that uses a type modifier involves two steps, one to convert between data types and a second to apply the modifier.

Table 44: pg_catalog.pg_cast

column	type	references	description
`castsource`	oid	pg_type.oid	OID of the source data type.
`casttarget`	oid	pg_type.oid	OID of the target data type.
`castfunc`	oid	pg_proc.oid	The OID of the function to use to perform this cast. Zero is stored if the data types are binary compatible (that is, no run-time operation is needed to perform the cast).

column	type	references	description
`castcontext`	char		Indicates what contexts the cast may be invoked in. `e` means only as an explicit cast (using `CAST` or `::` syntax). `a` means implicitly in assignment to a target column, as well as explicitly. `i` means implicitly in expressions, as well as the other cases.

pg_class

The system catalog table `pg_class` catalogs tables and most everything else that has columns or is otherwise similar to a table (also known as *relations*). This includes indexes (see also *pg_index*), sequences, views, composite types, and TOAST tables. Not all columns are meaningful for all relation types.

Table 45: pg_catalog.pg_class

column	type	references	description
`relname`	name		Name of the table, index, view, etc.
`relnamespace`	oid	pg_namespace.oid	The OID of the namespace (schema) that contains this relation
`reltype`	oid	pg_type.oid	The OID of the data type that corresponds to this table's row type, if any (zero for indexes, which have no pg_type entry)
`relowner`	oid	pg_authid.oid	Owner of the relation
`relam`	oid	pg_am.oid	If this is an index, the access method used (B-tree, Bitmap, hash, etc.)
`relfilenode`	oid		Name of the on-disk file of this relation; `0` if none.
`reltablespace`	oid	pg_tablespace.oid	The tablespace in which this relation is stored. If zero, the database's default tablespace is implied. (Not meaningful if the relation has no on-disk file.)

column	type	references	description
relpages	int4		Size of the on-disk representation of this table in pages (of 32K each). This is only an estimate used by the planner. It is updated by VACUUM, ANALYZE, and a few DDL commands.
reltuples	float4		Number of rows in the table. This is only an estimate used by the planner. It is updated by VACUUM, ANALYZE, and a few DDL commands.
reltoastrelid	oid	pg_class.oid	OID of the TOAST table associated with this table, 0 if none. The TOAST table stores large attributes "out of line" in a secondary table.
reltoastidxid	oid	pg_class.oid	For a TOAST table, the OID of its index. 0 if not a TOAST table.
relaosegidxid	oid		Deprecated in Greenplum Database 3. 4.
relaosegrelid	oid		Deprecated in Greenplum Database 3. 4.
relhasindex	boolean		True if this is a table and it has (or recently had) any indexes. This is set by CREATE INDEX, but not cleared immediately by DROP INDEX. VACUUM will clear if it finds the table has no indexes.
relisshared	boolean		True if this table is shared across all databases in the system. Only certain system catalog tables are shared.

column	type	references	description
`relkind`	char		The type of object `r` = heap or append-optimized table, `i` = index, `S` = sequence, `v` = view, `c` = composite type, `t` = TOAST value, `o` = internal append-optimized segment files and EOFs, `c` = composite type, `u` = uncataloged temporary heap table
`relstorage`	char		The storage mode of a table `a`= append-optimized, `h` = heap, `v` = virtual, `x`= external table.
`relnatts`	int2		Number of user columns in the relation (system columns not counted). There must be this many corresponding entries in pg_attribute.
`relchecks`	int2		Number of check constraints on the table.
`reltriggers`	int2		Number of triggers on the table.
`relukeys`	int2		Unused
`relfkeys`	int2		Unused
`relrefs`	int2		Unused
`relhasoids`	boolean		True if an OID is generated for each row of the relation.
`relhaspkey`	boolean		True if the table has (or once had) a primary key.
`relhasrules`	boolean		True if table has rules.
`relhassubclass`	boolean		True if table has (or once had) any inheritance children.

column	type	references	description
relfrozenxid	xid		All transaction IDs before this one have been replaced with a permanent (frozen) transaction ID in this table. This is used to track whether the table needs to be vacuumed in order to prevent transaction ID wraparound or to allow pg_clog to be shrunk. Zero (InvalidTransactionId) if the relation is not a table.
relacl	aclitem[]		Access privileges assigned by GRANT and REVOKE.
reloptions	text[]		Access-method-specific options, as "keyword= value" strings.

pg_compression

The pg_compression system catalog table describes the compression methods available..

Table 46: pg_catalog.pg_compression

column	type	modifers	storage	description
compname	name	not null	plain	Name of the compression
compconstructor	regproc	not null	plain	Name of compression constructor
compdestructor	regproc	not null	plain	Name of compression destructor
compcompressor	regproc	not null	plain	Name of the compressor
compdecompressor	regproc	not null	plain	Name of the decompressor
compvalidator	regproc	not null	plain	Name of the compression validator
compowner	oid	not null	plain	oid from pg_authid

pg_constraint

The `pg_constraint` system catalog table stores check, primary key, unique, and foreign key constraints on tables. Column constraints are not treated specially. Every column constraint is equivalent to some table constraint. Not-null constraints are represented in the *pg_attribute* catalog table. Check constraints on domains are stored here, too.

Table 47: pg_catalog.pg_constraint

column	type	references	description
`conname`	name		Constraint name (not necessarily unique!)
`connamespace`	oid	pg_namespace.oid	The OID of the namespace (schema) that contains this constraint.
`contype`	char		`c` = check constraint, `f` = foreign key constraint, `p` = primary key constraint, `u` = unique constraint.
`condeferrable`	boolean		Is the constraint deferrable?
`condeferred`	boolean		Is the constraint deferred by default?
`conrelid`	oid	pg_class.oid	The table this constraint is on; 0 if not a table constraint.
`contypid`	oid	pg_type.oid	The domain this constraint is on; 0 if not a domain constraint.
`confrelid`	oid	pg_class.oid	If a foreign key, the referenced table; else 0.
`confupdtype`	char		Foreign key update action code.
`confdeltype`	char		Foreign key deletion action code.
`confmatchtype`	char		Foreign key match type.
`conkey`	int2[]	pg_attribute.attnum	If a table constraint, list of columns which the constraint constrains.
`confkey`	int2[]	pg_attribute.attnum	If a foreign key, list of the referenced columns.
`conbin`	text		If a check constraint, an internal representation of the expression.

column	type	references	description
`consrc`	text		If a check constraint, a human-readable representation of the expression. This is not updated when referenced objects change; for example, it won't track renaming of columns. Rather than relying on this field, it is best to use `pg_get_constraintdef()` to extract the definition of a check constraint.

pg_conversion

The `pg_conversion` system catalog table describes the available encoding conversion procedures as defined by `CREATE CONVERSION`.

Table 48: pg_catalog.pg_conversion

column	type	references	description
`conname`	name		Conversion name (unique within a namespace).
`connamespace`	oid	pg_namespace.oid	The OID of the namespace (schema) that contains this conversion.
`conowner`	oid	pg_authid.oid	Owner of the conversion.
`conforencoding`	int4		Source encoding ID.
`contoencoding`	int4		Destination encoding ID.
`conproc`	regproc	pg_proc.oid	Conversion procedure.
`condefault`	boolean		True if this is the default conversion.

pg_database

The `pg_database` system catalog table stores information about the available databases. Databases are created with the `CREATE DATABASE` SQL command. Unlike most system catalogs, `pg_database` is shared across all databases in the system. There is only one copy of `pg_database` per system, not one per database.

Table 49: pg_catalog.pg_database

column	type	references	description
`datname`	name		Database name.

column	type	references	description
`datdba`	oid	pg_authid.oid	Owner of the database, usually the user who created it.
`encoding`	int4		Character encoding for this database. `pg_encoding_to_char()` can translate this number to the encoding name.
`datistemplate`	boolean		If true then this database can be used in the `TEMPLATE` clause of `CREATE DATABASE` to create a new database as a clone of this one.
`datallowconn`	boolean		If false then no one can connect to this database. This is used to protect the `template0` database from being altered.
`datconnlimit`	int4		Sets the maximum number of concurrent connections that can be made to this database. `-1` means no limit.
`datlastsysoid`	oid		Last system OID in the database.
`datfrozenxid`	xid		All transaction IDs before this one have been replaced with a permanent (frozen) transaction ID in this database. This is used to track whether the database needs to be vacuumed in order to prevent transaction ID wraparound or to allow pg_clog to be shrunk. It is the minimum of the per-table *pg_class.relfrozenxid* values.

column	type	references	description
`dattablespace`	oid	pg_tablespace.oid	The default tablespace for the database. Within this database, all tables for which *pg_class.reltablespace* is zero will be stored in this tablespace. All non-shared system catalogs will also be there.
`datconfig`	text[]		Session defaults for user-settable server configuration parameters.
`datacl`	aclitem[]		Database access privileges as given by `GRANT` and `REVOKE`.

pg_depend

The `pg_depend` system catalog table records the dependency relationships between database objects. This information allows `DROP` commands to find which other objects must be dropped by `DROP CASCADE` or prevent dropping in the `DROP RESTRICT` case. See also *pg_shdepend*, which performs a similar function for dependencies involving objects that are shared across a Greenplum system.

In all cases, a `pg_depend` entry indicates that the referenced object may not be dropped without also dropping the dependent object. However, there are several subflavors identified by `deptype`:

- **DEPENDENCY_NORMAL (n)** — A normal relationship between separately-created objects. The dependent object may be dropped without affecting the referenced object. The referenced object may only be dropped by specifying `CASCADE`, in which case the dependent object is dropped, too. Example: a table column has a normal dependency on its data type.
- **DEPENDENCY_AUTO (a)** — The dependent object can be dropped separately from the referenced object, and should be automatically dropped (regardless of `RESTRICT` or `CASCADE` mode) if the referenced object is dropped. Example: a named constraint on a table is made autodependent on the table, so that it will go away if the table is dropped.
- **DEPENDENCY_INTERNAL (i)** — The dependent object was created as part of creation of the referenced object, and is really just a part of its internal implementation. A `DROP` of the dependent object will be disallowed outright (we'll tell the user to issue a `DROP` against the referenced object, instead). A `DROP` of the referenced object will be propagated through to drop the dependent object whether `CASCADE` is specified or not.
- **DEPENDENCY_PIN (p)** — There is no dependent object; this type of entry is a signal that the system itself depends on the referenced object, and so that object must never be deleted. Entries of this type are created only by system initialization. The columns for the dependent object contain zeroes.

Table 50: pg_catalog.pg_depend

column	type	references	description
`classid`	oid	pg_class.oid	The OID of the system catalog the dependent object is in.
`objid`	oid	any OID column	The OID of the specific dependent object.

column	type	references	description
`objsubid`	int4		For a table column, this is the column number. For all other object types, this column is zero.
`refclassid`	oid	pg_class.oid	The OID of the system catalog the referenced object is in.
`refobjid`	oid	any OID column	The OID of the specific referenced object.
`refobjsubid`	int4		For a table column, this is the referenced column number. For all other object types, this column is zero.
`deptype`	char		A code defining the specific semantics of this dependency relationship.

pg_description

The `pg_description` system catalog table stores optional descriptions (comments) for each database object. Descriptions can be manipulated with the `COMMENT` command and viewed with `psql`'s `\d` meta-commands. Descriptions of many built-in system objects are provided in the initial contents of `pg_description`. See also *pg_shdescription*, which performs a similar function for descriptions involving objects that are shared across a Greenplum system.

Table 51: pg_catalog.pg_description

column	type	references	description
`objoid`	oid	any OID column	The OID of the object this description pertains to.
`classoid`	oid	pg_class.oid	The OID of the system catalog this object appears in
`objsubid`	int4		For a comment on a table column, this is the column number. For all other object types, this column is zero.
`description`	text		Arbitrary text that serves as the description of this object.

pg_exttable

The `pg_exttable` system catalog table is used to track external tables and web tables created by the `CREATE EXTERNAL TABLE` command.

Table 52: pg_catalog.pg_exttable

column	type	references	description
`reloid`	oid	pg_class.oid	The OID of this external table.
`location`	text[]		The URI location(s) of the external table files.
`fmttype`	char		Format of the external table files: `t` for text, or `c` for csv.
`fmtopts`	text		Formatting options of the external table files, such as the field delimiter, null string, escape character, etc.
`command`	text		The OS command to execute when the external table is accessed.
`rejectlimit`	integer		The per segment reject limit for rows with errors, after which the load will fail.
`rejectlimittype`	char		Type of reject limit threshold: `r` for number of rows.
`fmterrtbl`	oid	pg_class.oid	The object id of the error table where format errors will be logged.
`encoding`	text		The client encoding.
`writable`	boolean		`0` for readable external tables, `1` for writable external tables.

pg_filespace

The `pg_filespace` table contains information about the filespaces created in a Greenplum Database system. Every system contains a default filespace, `pg_system`, which is a collection of all the data directory locations created at system initialization time.

A tablespace requires a file system location to store its database files. In Greenplum Database, the master and each segment (primary and mirror) needs its own distinct storage location. This collection of file system locations for all components in a Greenplum system is referred to as a filespace.

Table 53: pg_catalog.pg_filespace

column	type	references	description
`fsname`	name		The name of the filespace.
`fsowner`	oid	pg_roles.oid	The object id of the role that created the filespace.

pg_filespace_entry

A tablespace requires a file system location to store its database files. In Greenplum Database, the master and each segment (primary and mirror) needs its own distinct storage location. This collection of file system locations for all components in a Greenplum system is referred to as a *filespace*. The `pg_filespace_entry` table contains information about the collection of file system locations across a Greenplum Database system that comprise a Greenplum Database filespace.

Table 54: pg_catalog.pg_filespace_entry

column	type	references	description
`fsefsoid`	OID	pg_filespace.oid	Object id of the filespace.
`fsedbid`	integer	gp_segment_configuration.dbid	Segment id.
`fselocation`	text		File system location for this segment id.

pg_index

The `pg_index` system catalog table contains part of the information about indexes. The rest is mostly in *pg_class*.

Table 55: pg_catalog.pg_index

column	type	references	description
`indexrelid`	oid	pg_class.oid	The OID of the pg_class entry for this index.
`indrelid`	oid	pg_class.oid	The OID of the pg_class entry for the table this index is for.
`indnatts`	int2		The number of columns in the index (duplicates pg_class.relnatts).
`indisunique`	boolean		If true, this is a unique index.

column	type	references	description
`indisprimary`	boolean		If true, this index represents the primary key of the table. (indisunique should always be true when this is true.)
`indisclustered`	boolean		If true, the table was last clustered on this index via the `CLUSTER` command.
`indisvalid`	boolean		If true, the index is currently valid for queries. False means the index is possibly incomplete: it must still be modified by `INSERT`/`UPDATE` operations, but it cannot safely be used for queries.
`indkey`	int2vector	pg_attribute.attnum	This is an array of indnatts values that indicate which table columns this index indexes. For example a value of 1 3 would mean that the first and the third table columns make up the index key. A zero in this array indicates that the corresponding index attribute is an expression over the table columns, rather than a simple column reference.
`indclass`	oidvector	pg_opclass.oid	For each column in the index key this contains the OID of the operator class to use.
`indexprs`	text		Expression trees (in `nodeToString()` representation) for index attributes that are not simple column references. This is a list with one element for each zero entry in indkey. NULL if all index attributes are simple references.

column	type	references	description
`indpred`	text		Expression tree (in `nodeToString()` representation) for partial index predicate. NULL if not a partial index.

pg_inherits

The `pg_inherits` system catalog table records information about table inheritance hierarchies. There is one entry for each direct child table in the database. (Indirect inheritance can be determined by following chains of entries.) In Greenplum Database, inheritance relationships are created by both the `INHERITS` clause (standalone inheritance) and the `PARTITION BY` clause (partitioned child table inheritance) of `CREATE TABLE`.

Table 56: pg_catalog.pg_inherits

column	type	references	description
`inhrelid`	oid	pg_class.oid	The OID of the child table.
`inhparent`	oid	pg_class.oid	The OID of the parent table.
`inhseqno`	int4		If there is more than one direct parent for a child table (multiple inheritance), this number tells the order in which the inherited columns are to be arranged. The count starts at 1.

pg_language

The `pg_language` system catalog table registers languages in which you can write functions or stored procedures. It is populated by `CREATE LANGUAGE`.

Table 57: pg_catalog.pg_language

column	type	references	description
`lanname`	name		Name of the language.

column	type	references	description
`lanispl`	boolean		This is false for internal languages (such as SQL) and true for user-defined languages. Currently, `pg_dump` still uses this to determine which languages need to be dumped, but this may be replaced by a different mechanism in the future.
`lanpltrusted`	boolean		True if this is a trusted language, which means that it is believed not to grant access to anything outside the normal SQL execution environment. Only superusers may create functions in untrusted languages.
`lanplcallfoid`	oid	pg_proc.oid	For noninternal languages this references the language handler, which is a special function that is responsible for executing all functions that are written in the particular language.
`lanvalidator`	oid	pg_proc.oid	This references a language validator function that is responsible for checking the syntax and validity of new functions when they are created. Zero if no validator is provided.
`lanacl`	aclitem[]		Access privileges for the language.

pg_largeobject

The `pg_largeobject` system catalog table holds the data making up 'large objects'. A large object is identified by an OID assigned when it is created. Each large object is broken into segments or 'pages' small enough to be conveniently stored as rows in `pg_largeobject`. The amount of data per page is defined to be `LOBLKSIZE` (which is currently `BLCKSZ`/4, or typically 8K).

Each row of `pg_largeobject` holds data for one page of a large object, beginning at byte offset (*pageno* `* LOBLKSIZE`) within the object. The implementation allows sparse storage: pages may be missing, and may be shorter than `LOBLKSIZE` bytes even if they are not the last page of the object. Missing regions within a large object read as zeroes.

Table 58: pg_catalog.pg_largeobject

column	type	references	description
`loid`	oid		Identifier of the large object that includes this page.
`pageno`	int4		Page number of this page within its large object (counting from zero).
`data`	bytea		Actual data stored in the large object. This will never be more than `LOBLKSIZE` bytes and may be less.

pg_listener

The `pg_listener` system catalog table supports the `LISTEN` and `NOTIFY` commands. A listener creates an entry in `pg_listener` for each notification name it is listening for. A notifier scans and updates each matching entry to show that a notification has occurred. The notifier also sends a signal (using the PID recorded in the table) to awaken the listener from sleep.

This table is not currently used in Greenplum Database.

Table 59: pg_catalog.pg_listener

column	type	references	description
`relname`	name		Notify condition name. (The name need not match any actual relation in the database.
`listenerpid`	int4		PID of the server process that created this entry.
`notification`	int4		Zero if no event is pending for this listener. If an event is pending, the PID of the server process that sent the notification.

pg_locks

The `pg_locks` view provides access to information about the locks held by open transactions within Greenplum Database.

`pg_locks` contains one row per active lockable object, requested lock mode, and relevant transaction. Thus, the same lockable object may appear many times, if multiple transactions are holding or waiting for locks on it. However, an object that currently has no locks on it will not appear at all.

There are several distinct types of lockable objects: whole relations (such as tables), individual pages of relations, individual tuples of relations, transaction IDs, and general database objects. Also, the right to extend a relation is represented as a separate lockable object.

Table 60: pg_catalog.pg_locks

column	type	references	description
`locktype`	text		Type of the lockable object: `relation`, `extend`, `page`, `tuple`, `transactionid`, `object`, `userlock`, `resource queue`, or `advisory`
`database`	oid	pg_database.oid	OID of the database in which the object exists, zero if the object is a shared object, or NULL if the object is a transaction ID
`relation`	oid	pg_class.oid	OID of the relation, or NULL if the object is not a relation or part of a relation
`page`	integer		Page number within the relation, or NULL if the object is not a tuple or relation page
`tuple`	smallint		Tuple number within the page, or NULL if the object is not a tuple
`transactionid`	xid		ID of a transaction, or NULL if the object is not a transaction ID
`classid`	oid	pg_class.oid	OID of the system catalog containing the object, or NULL if the object is not a general database object
`objid`	oid	any OID column	OID of the object within its system catalog, or NULL if the object is not a general database object

column	type	references	description
objsubid	smallint		For a table column, this is the column number (the classid and objid refer to the table itself). For all other object types, this column is zero. NULL if the object is not a general database object
transaction	xid		ID of the transaction that is holding or awaiting this lock
pid	integer		Process ID of the server process holding or awaiting this lock. NULL if the lock is held by a prepared transaction
mode	text		Name of the lock mode held or desired by this process
granted	boolean		True if lock is held, false if lock is awaited.
mppsessionid	integer		The id of the client session associated with this lock.
mppiswriter	boolean		Specifies whether the lock is held by a writer process.
gp_segment_id	integer		The Greenplum segment id (dbid) where the lock is held.

pg_namespace

The pg_namespace system catalog table stores namespaces. A namespace is the structure underlying SQL schemas: each namespace can have a separate collection of relations, types, etc. without name conflicts.

Table 61: pg_catalog.pg_namespace

column	type	references	description
nspname	name		Name of the namespace
nspowner	oid	pg_authid.oid	Owner of the namespace
nspacl	aclitem[]		Access privileges as given by GRANT and REVOKE.

pg_opclass

The `pg_opclass` system catalog table defines index access method operator classes. Each operator class defines semantics for index columns of a particular data type and a particular index access method. Note that there can be multiple operator classes for a given data type/access method combination, thus supporting multiple behaviors. The majority of the information defining an operator class is actually not in its `pg_opclass` row, but in the associated rows in *pg_amop* and *pg_amproc*. Those rows are considered to be part of the operator class definition — this is not unlike the way that a relation is defined by a single *pg_class* row plus associated rows in *pg_attribute* and other tables.

Table 62: pg_catalog.pg_opclass

column	type	references	description
`opcamid`	oid	pg_am.oid	Index access method operator class is for.
`opcname`	name		Name of this operator class
`opcnamespace`	oid	pg_namespace.oid	Namespace of this operator class
`opcowner`	oid	pg_authid.oid	Owner of the operator class
`opcintype`	oid	pg_type.oid	Data type that the operator class indexes.
`opcdefault`	boolean		True if this operator class is the default for the data type opcintype.
`opckeytype`	oid	pg_type.oid	Type of data stored in index, or zero if same as opcintype.

pg_operator

The `pg_operator` system catalog table stores information about operators, both built-in and those defined by `CREATE OPERATOR`. Unused column contain zeroes. For example, `oprleft` is zero for a prefix operator.

Table 63: pg_catalog.pg_operator

column	type	references	description
`oprname`	name		Name of the operator.
`oprnamespace`	oid	pg_namespace.oid	The OID of the namespace that contains this operator.
`oprowner`	oid	pg_authid.oid	Owner of the operator.
`oprkind`	char		`b` = infix (both), `l` = prefix (left), `r` = postfix (right)
`oprcanhash`	boolean		This operator supports hash joins.

column	type	references	description
oprleft	oid	pg_type.oid	Type of the left operand.
oprright	oid	pg_type.oid	Type of the right operand.
oprresult	oid	pg_type.oid	Type of the result.
oprcom	oid	pg_operator.oid	Commutator of this operator, if any.
oprnegate		pg_operator.oid	Negator of this operator, if any.
oprlsortop	oid	pg_operator.oid	If this operator supports merge joins, the operator that sorts the type of the left-hand operand (L<L).
oprrsortop	oid	pg_operator.oid	If this operator supports merge joins, the operator that sorts the type of the right-hand operand (R<R).
oprltcmpop	oid	pg_operator.oid	If this operator supports merge joins, the less-than operator that compares the left and right operand types (L<R).
oprgtcmpop	oid	pg_operator.oid	If this operator supports merge joins, the greater-than operator that compares the left and right operand types (L>R).
oprcode	regproc	pg_proc.oid	Function that implements this operator.
oprrest	regproc	pg_proc.oid	Restriction selectivity estimation function for this operator.
oprjoin	regproc	pg_proc.oid	Join selectivity estimation function for this operator.

pg_partition

The pg_partition system catalog table is used to track partitioned tables and their inheritance level relationships. Each row of pg_partition represents either the level of a partitioned table in the partition hierarchy, or a subpartition template description. The value of the attribute paristemplate determines what a particular row represents.

Table 64: pg_catalog.pg_partition

column	type	references	description
`parrelid`	oid	pg_class.oid	The object identifier of the table.
`parkind`	char		The partition type - `R` for range or `L` for list.
`parlevel`	smallint		The partition level of this row: 0 for the top-level parent table, 1 for the first level under the parent table, 2 for the second level, and so on.
`paristemplate`	boolean		Whether or not this row represents a subpartition template definition (true) or an actual partitioning level (false).
`parnatts`	smallint		The number of attributes that define this level.
`paratts`	smallint()		An array of the attribute numbers (as in `pg_attribute.attnum`) of the attributes that participate in defining this level.
`parclass`	oidvector	pg_opclass.oid	The operator class identifier(s) of the partition columns.

pg_partition_columns

The `pg_partition_columns` system view is used to show the partition key columns of a partitioned table.

Table 65: pg_catalog.pg_partition_columns

column	type	references	description
`schemaname`	name		The name of the schema the partitioned table is in.
`tablename`	name		The table name of the top-level parent table.
`columnname`	name		The name of the partition key column.
`partitionlevel`	smallint		The level of this subpartition in the hierarchy.

column	type	references	description
position_in_partition_key	integer		For list partitions you can have a composite (multi-column) partition key. This shows the position of the column in a composite key.

pg_partition_encoding

The pg_partition_encoding system catalog table describes the available column compression options for a partition template.

Table 66: pg_catalog.pg_attribute_encoding

column	type	modifers	storage	description
parencoid	oid	not null	plain	
parencattnum	snallint	not null	plain	
parencattoptions	text [ ]		extended	

pg_partition_rule

The pg_partition_rule system catalog table is used to track partitioned tables, their check constraints, and data containment rules. Each row of pg_partition_rule represents either a leaf partition (the bottom level partitions that contain data), or a branch partition (a top or mid-level partition that is used to define the partition hierarchy, but does not contain any data).

Table 67: pg_catalog.pg_partition_rule

column	type	references	description
paroid	oid	pg_partition.oid	Row identifier of the partitioning level (from *pg_partition*) to which this partition belongs. In the case of a branch partition, the corresponding table (identified by pg_partition_rule) is an empty container table. In case of a leaf partition, the table contains the rows for that partition containment rule.
parchildrelid	oid	pg_class.oid	The table identifier of the partition (child table).
parparentrule	oid	pg_partition_rule.paroid	The row identifier of the rule associated with the parent table of this partition.

column	type	references	description
`parname`	name		The given name of this partition.
`parisdefault`	boolean		Whether or not this partition is a default partition.
`parruleord`	smallint		For range partitioned tables, the rank of this partition on this level of the partition hierarchy.
`parrangestartincl`	boolean		For range partitioned tables, whether or not the starting value is inclusive.
`parrangeendincl`	boolean		For range partitioned tables, whether or not the ending value is inclusive.
`parrangestart`	text		For range partitioned tables, the starting value of the range.
`parrangeend`	text		For range partitioned tables, the ending value of the range.
`parrangeevery`	text		For range partitioned tables, the interval value of the `EVERY` clause.
`parlistvalues`	text		For list partitioned tables, the list of values assigned to this partition.
`parreloptions`	text		An array describing the storage characteristics of the particular partition.

pg_partition_templates

The `pg_partition_templates` system view is used to show the subpartitions that were created using a subpartition template.

Table 68: pg_catalog.pg_partition_templates

column	type	references	description
`schemaname`	name		The name of the schema the partitioned table is in.
`tablename`	name		The table name of the top-level parent table.

column	type	references	description
partitionname	name		The name of the subpartition (this is the name to use if referring to the partition in an ALTER TABLE command). NULL if the partition was not given a name at create time or generated by an EVERY clause.
partitiontype	text		The type of subpartition (range or list).
partitionlevel	smallint		The level of this subpartition in the hierarchy.
partitionrank	bigint		For range partitions, the rank of the partition compared to other partitions of the same level.
partitionposition	smallint		The rule order position of this subpartition.
partitionlistvalues	text		For list partitions, the list value(s) associated with this subpartition.
partitionrangestart	text		For range partitions, the start value of this subpartition.
partitionstartinclusive	boolean		T if the start value is included in this subpartition. F if it is excluded.
partitionrangeend	text		For range partitions, the end value of this subpartition.
partitionendinclusive	boolean		T if the end value is included in this subpartition. F if it is excluded.
partitioneveryclause	text		The EVERY clause (interval) of this subpartition.
partitionisdefault	boolean		T if this is a default subpartition, otherwise F.
partitionboundary	text		The entire partition specification for this subpartition.

pg_partitions

The `pg_partitions` system view is used to show the structure of a partitioned table.

Table 69: pg_catalog.pg_partitions

column	type	references	description
`schemaname`	name		The name of the schema the partitioned table is in.
`tablename`	name		The name of the top-level parent table.
`partitiontablename`	name		The relation name of the partitioned table (this is the table name to use if accessing the partition directly).
`partitionname`	name		The name of the partition (this is the name to use if referring to the partition in an `ALTER TABLE` command). `NULL` if the partition was not given a name at create time or generated by an `EVERY` clause.
`parentpartitiontablename`	name		The relation name of the parent table one level up from this partition.
`parentpartitionname`	name		The given name of the parent table one level up from this partition.
`partitiontype`	text		The type of partition (range or list).
`partitionlevel`	smallint		The level of this partition in the hierarchy.
`partitionrank`	bigint		For range partitions, the rank of the partition compared to other partitions of the same level.
`partitionposition`	smallint		The rule order position of this partition.
`partitionlistvalues`	text		For list partitions, the list value(s) associated with this partition.
`partitionrangestart`	text		For range partitions, the start value of this partition.

column	type	references	description
partitionstartinclusive	boolean		T if the start value is included in this partition. F if it is excluded.
partitionrangeend	text		For range partitions, the end value of this partition.
partitionendinclusive	boolean		T if the end value is included in this partition. F if it is excluded.
partitioneveryclause	text		The EVERY clause (interval) of this partition.
partitionisdefault	boolean		T if this is a default partition, otherwise F.
partitionboundary	text		The entire partition specification for this partition.

pg_pltemplate

The pg_pltemplate system catalog table stores template information for procedural languages. A template for a language allows the language to be created in a particular database by a simple CREATE LANGUAGE command, with no need to specify implementation details. Unlike most system catalogs, pg_pltemplate is shared across all databases of Greenplum system: there is only one copy of pg_pltemplate per system, not one per database. This allows the information to be accessible in each database as it is needed.

There are not currently any commands that manipulate procedural language templates; to change the built-in information, a superuser must modify the table using ordinary INSERT, DELETE, or UPDATE commands.

Table 70: pg_catalog.pg_pltemplate

column	type	references	description
tmplname	name		Name of the language this template is for
tmpltrusted	boolean		True if language is considered trusted
tmplhandler	text		Name of call handler function
tmplvalidator	text		Name of validator function, or NULL if none
tmpllibrary	text		Path of shared library that implements language
tmplacl	aclitem[]		Access privileges for template (not yet implemented).

pg_proc

The pg_proc system catalog table stores information about functions (or procedures), both built-in functions and those defined by CREATE FUNCTION. The table contains data for aggregate and window functions as well as plain functions. If proisagg is true, there should be a matching row in pg_aggregate. If proiswin is true, there should be a matching row in pg_window.

For compiled functions, both built-in and dynamically loaded, prosrc contains the function's C-language name (link symbol). For all other currently-known language types, prosrc contains the function's source text. probin is unused except for dynamically-loaded C functions, for which it gives the name of the shared library file containing the function.

Table 71: pg_catalog.pg_proc

column	type	references	description
proname	name		Name of the function.
pronamespace	oid	pg_namespace.oid	The OID of the namespace that contains this function.
proowner	oid	pg_authid.oid	Owner of the function.
prolang	oid	pg_language.oid	Implementation language or call interface of this function.
proisagg	boolean		Function is an aggregate function.
prosecdef	boolean		Function is a security definer (for example, a 'setuid' function).
proisstrict	boolean		Function returns NULL if any call argument is NULL. In that case the function will not actually be called at all. Functions that are not strict must be prepared to handle NULL inputs.
proretset	boolean		Function returns a set (multiple values of the specified data type).

column	type	references	description
provolatile	char		Tells whether the function's result depends only on its input arguments, or is affected by outside factors. i = *immutable* (always delivers the same result for the same inputs), s = *stable* (results (for fixed inputs) do not change within a scan), or v = *volatile* (results may change at any time or functions with side-effects).
pronargs	int2		Number of arguments.
prorettype	oid	pg_type.oid	Data type of the return value.
proiswin	boolean		Function is neither an aggregate nor a scalar function, but a pure window function.
proargtypes	oidvector	pg_type.oid	An array with the data types of the function arguments. This includes only input arguments (including INOUT arguments), and thus represents the call signature of the function.
proallargtypes	oid[]	pg_type.oid	An array with the data types of the function arguments. This includes all arguments (including OUT and INOUT arguments); however, if all the arguments are IN arguments, this field will be null. Note that subscripting is 1-based, whereas for historical reasons proargtypes is subscripted from 0.

column	type	references	description
`proargmodes`	char[]		An array with the modes of the function arguments: `i` = `IN`, `o` = `OUT` , `b` = `INOUT`. If all the arguments are IN arguments, this field will be null. Note that subscripts correspond to positions of proallargtypes not proargtypes.
`proargnames`	text[]		An array with the names of the function arguments. Arguments without a name are set to empty strings in the array. If none of the arguments have a name, this field will be null. Note that subscripts correspond to positions of proallargtypes not proargtypes.
`prosrc`	text		This tells the function handler how to invoke the function. It might be the actual source code of the function for interpreted languages, a link symbol, a file name, or just about anything else, depending on the implementation language/call convention.
`probin`	bytea		Additional information about how to invoke the function. Again, the interpretation is language-specific.
`proacl`	aclitem[]		Access privileges for the function as given by `GRANT`/`REVOKE`.

pg_resourcetype

The `pg_resourcetype` system catalog table contains information about the extended attributes that can be assigned to Greenplum Database resource queues. Each row details an attribute and inherent qualities such as its default setting, whether it is required, and the value to disable it (when allowed).

This table is populated only on the master. This table is defined in the `pg_global` tablespace, meaning it is globally shared across all databases in the system.

Table 72: pg_catalog.pg_resourcetype

column	type	references	description
`restypid`	smallint		The resource type ID.
`resname`	name		The name of the resource type.
`resrequired`	boolean		Whether the resource type is required for a valid resource queue.
`reshasdefault`	boolean		Whether the resource type has a default value. When true, the default value is specified in reshasdefaultsetting.
`rescandisable`	boolean		Whether the type can be removed or disabled. When true, the default value is specified in resdisabledsetting.
`resdefaultsetting`	text		Default setting for the resource type, when applicable.
`resdisabledsetting`	text		The value that disables this resource type (when allowed).

pg_resqueue

The `pg_resqueue` system catalog table contains information about Greenplum Database resource queues, which are used for the workload management feature. This table is populated only on the master. This table is defined in the `pg_global` tablespace, meaning it is globally shared across all databases in the system.

Table 73: pg_catalog.pg_resqueue

column	type	references	description
`rsqname`	name		The name of the resource queue.
`rsqcountlimit`	real		The active query threshold of the resource queue.
`rsqcostlimit`	real		The query cost threshold of the resource queue.
`rsqovercommit`	boolean		Allows queries that exceed the cost threshold to run when the system is idle.

column	type	references	description
`rsqignorecostlimit`	real		The query cost limit of what is considered a 'small query'. Queries with a cost under this limit will not be queued and run immediately.

pg_resqueue_attributes

The `pg_resqueue_attributes` view allows administrators to see the attributes set for a resource queue, such as its active statement limit, query cost limits, and priority.

Table 74: pg_catalog.pg_resqueue_attributes

column	type	references	description
`rsqname`	name	pg_resqueue.rsqname	The name of the resource queue.
`resname`	text		The name of the resource queue attribute.
`ressetting`	text		The current value of a resource queue attribute.
`restypid`	integer		System assigned resource type id.

pg_resqueuecapability

The `pg_resqueuecapability` system catalog table contains information about the extended attributes, or capabilities, of existing Greenplum Database resource queues. Only resource queues that have been assigned an extended capability, such as a priority setting, are recorded in this table. This table is joined to the *pg_resqueue* table by resource queue object ID, and to the *pg_resourcetype* table by resource type ID (`restypid`).

This table is populated only on the master. This table is defined in the `pg_global` tablespace, meaning it is globally shared across all databases in the system.

Table 75: pg_catalog.pg_resqueuecapability

column	type	references	description
`rsqueueid`	oid	`pg_resqueue.oid`	The object ID of the associated resource queue.
`restypid`	smallint	`pg_resourcetype.restypid`	The resource type, derived from the *pg_resqueuecapability* system table.

column	type	references	description
`ressetting`	opaque type		The specific value set for the capability referenced in this record. Depending on the actual resource type, this value may have different data types.

pg_rewrite

The `pg_rewrite` system catalog table stores rewrite rules for tables and views. `pg_class.relhasrules` must be true if a table has any rules in this catalog.

Table 76: pg_catalog.pg_rewrite

column	type	references	description
`rulename`	name		Rule name.
`ev_class`	oid	pg_class.oid	The table this rule is for.
`ev_attr`	int2		The column this rule is for (currently, always zero to indicate the whole table).
`ev_type`	char		Event type that the rule is for: 1 = SELECT, 2 = UPDATE, 3 = INSERT, 4 = DELETE.
`is_instead`	boolean		True if the rule is an INSTEAD rule.
`ev_qual`	text		Expression tree (in the form of a `nodeToString()` representation) for the rule's qualifying condition.
`ev_action`	text		Query tree (in the form of a `nodeToString()` representation) for the rule's action.

pg_roles

The view `pg_roles` provides access to information about database roles. This is simply a publicly readable view of *pg_authid* that blanks out the password field. This view explicitly exposes the OID column of the underlying table, since that is needed to do joins to other catalogs.

Table 77: pg_catalog.pg_roles

column	type	references	description
`rolname`	name		Role name

column	type	references	description
`rolsuper`	bool		Role has superuser privileges
`rolinherit`	bool		Role automatically inherits privileges of roles it is a member of
`rolcreaterole`	bool		Role may create more roles
`rolcreatedb`	bool		Role may create databases
`rolcatupdate`	bool		Role may update system catalogs directly. (Even a superuser may not do this unless this column is true.)
`rolcanlogin`	bool		Role may log in. That is, this role can be given as the initial session authorization identifier
`rolconnlimit`	int4		For roles that can log in, this sets maximum number of concurrent connections this role can make. -1 means no limit
`rolpassword`	text		Not the password (always reads as ********)
`rolvaliduntil`	timestamptz		Password expiry time (only used for password authentication); NULL if no expiration
`rolconfig`	text[]		Session defaults for run-time configuration variables
`rolresqueue`	oid	pg_resqueue.oid	Object ID of the resource queue this role is assigned to.
`oid`	oid	pg_authid.oid	Object ID of role
`rolcreaterextgpfd`	bool		Role may create readable external tables that use the gpfdist protocol.
`rolcreaterexthttp`	bool		Role may create readable external tables that use the http protocol.

column	type	references	description
`rolcreatewextgpfd`	bool		Role may create writable external tables that use the gpfdist protocol.

pg_shdepend

The `pg_shdepend` system catalog table records the dependency relationships between database objects and shared objects, such as roles. This information allows Greenplum Database to ensure that those objects are unreferenced before attempting to delete them. See also *pg_depend*, which performs a similar function for dependencies involving objects within a single database. Unlike most system catalogs, `pg_shdepend` is shared across all databases of Greenplum system: there is only one copy of `pg_shdepend` per system, not one per database.

In all cases, a `pg_shdepend` entry indicates that the referenced object may not be dropped without also dropping the dependent object. However, there are several subflavors identified by `deptype`:

- **SHARED_DEPENDENCY_OWNER (o)** — The referenced object (which must be a role) is the owner of the dependent object.
- **SHARED_DEPENDENCY_ACL (a)** — The referenced object (which must be a role) is mentioned in the ACL (access control list) of the dependent object.
- **SHARED_DEPENDENCY_PIN (p)** — There is no dependent object; this type of entry is a signal that the system itself depends on the referenced object, and so that object must never be deleted. Entries of this type are created only by system initialization. The columns for the dependent object contain zeroes.

Table 78: pg_catalog.pg_shdepend

column	type	references	description
`dbid`	oid	pg_database.oid	The OID of the database the dependent object is in, or zero for a shared object.
`classid`	oid	pg_class.oid	The OID of the system catalog the dependent object is in.
`objid`	oid	any OID column	The OID of the specific dependent object.
`objsubid`	int4		For a table column, this is the column number. For all other object types, this column is zero.
`refclassid`	oid	pg_class.oid	The OID of the system catalog the referenced object is in (must be a shared catalog).
`refobjid`	oid	any OID column	The OID of the specific referenced object.

column	type	references	description
`refobjsubid`	int4		For a table column, this is the referenced column number. For all other object types, this column is zero.
`deptype`	char		A code defining the specific semantics of this dependency relationship.

pg_shdescription

The `pg_shdescription` system catalog table stores optional descriptions (comments) for shared database objects. Descriptions can be manipulated with the `COMMENT` command and viewed with `psql`'s `\d` meta-commands. See also *pg_description*, which performs a similar function for descriptions involving objects within a single database. Unlike most system catalogs, `pg_shdescription` is shared across all databases of a Greenplum system: there is only one copy of `pg_shdescription` per system, not one per database.

Table 79: pg_catalog.pg_shdescription

column	type	references	description
`objoid`	oid	any OID column	The OID of the object this description pertains to.
`classoid`	oid	pg_class.oid	The OID of the system catalog this object appears in
`description`	text		Arbitrary text that serves as the description of this object.

pg_stat_activity

The view `pg_stat_activity` shows one row per server process and details about it associated user session and query. The columns that report data on the current query are available unless the parameter `stats_command_string` has been turned off. Furthermore, these columns are only visible if the user examining the view is a superuser or the same as the user owning the process being reported on.

The maximum length of the query text sting stored in the column `current_query` can be controlled with the server configuration parameter `pgstat_track_activity_query_size`.

Table 80: pg_catalog.pg_stat_activity

column	type	references	description
`datid`	oid	pg_database.oid	Database OID
`datname`	name		Database name
`procpid`	integer		Process ID of the server process
`sess_id`	integer		Session ID

column	type	references	description
usesysid	oid	pg_authid.oid	Role OID
usename	name		Role name
current_query	text		Current query that process is running
waiting	boolean		True if waiting on a lock, false if not waiting
query_start	timestampz		Time query began execution
backend_start	timestampz		Time backend process was started
client_addr	inet		Client address
client_port	integer		Client port
application_name	text		Client application name
xact_start	timestampz		Transaction start time
waiting_reason	text		Reason the server process is waiting. The value can be: lock replication

pg_stat_last_operation

The pg_stat_last_operation table contains metadata tracking information about database objects (tables, views, etc.).

Table 81: pg_catalog.pg_stat_last_operation

column	type	references	description
classid	oid	pg_class.oid	OID of the system catalog containing the object.
objid	oid	any OID column	OID of the object within its system catalog.
staactionname	name		The action that was taken on the object.
stasysid	oid	pg_authid.oid	A foreign key to pg_ authid.oid.
stausename	name		The name of the role that performed the operation on this object.
stasubtype	text		The type of object operated on or the subclass of operation performed.

column	type	references	description
`statime`	timestamp with timezone		The timestamp of the operation. This is the same timestamp that is written to the Greenplum Database server log files in case you need to look up more detailed information about the operation in the logs.

pg_stat_last_shoperation

The pg_stat_last_shoperation table contains metadata tracking information about global objects (roles, tablespaces, etc.).

Table 82: pg_catalog.pg_stat_last_shoperation

column	type	references	description
classid	oid	pg_class.oid	OID of the system catalog containing the object.
`objid`	oid	any OID column	OID of the object within its system catalog.
`staactionname`	name		The action that was taken on the object.
`stasysid`	oid		
`stausename`	name		The name of the role that performed the operation on this object.
`stasubtype`	text		The type of object operated on or the subclass of operation performed.
`statime`	timestamp with timezone		The timestamp of the operation. This is the same timestamp that is written to the Greenplum Database server log files in case you need to look up more detailed information about the operation in the logs.

pg_stat_operations

The view `pg_stat_operations` shows details about the last operation performed on a database object (such as a table, index, view or database) or a global object (such as a role).

Table 83: pg_catalog.pg_stat_operations

column	type	references	description
classname	text		The name of the system table in the pg_catalog schema where the record about this object is stored (pg_class=relations, pg_database=databases, pg_namespace=schemas, pg_authid=roles)
objname	name		The name of the object.
objid	oid		The OID of the object.
schemaname	name		The name of the schema where the object resides.
usestatus	text		The status of the role who performed the last operation on the object (CURRENT=a currently active role in the system, DROPPED=a role that no longer exists in the system, CHANGED=a role name that exists in the system, but has changed since the last operation was performed).
usename	name		The name of the role that performed the operation on this object.
actionname	name		The action that was taken on the object.
subtype	text		The type of object operated on or the subclass of operation performed.
statime	timestampz		The timestamp of the operation. This is the same timestamp that is written to the Greenplum Database server log files in case you need to look up more detailed information about the operation in the logs.

pg_stat_partition_operations

The pg_stat_partition_operations view shows details about the last operation performed on a partitioned table.

Table 84: pg_catalog.pg_stat_partition_operations

column	type	references	description
classname	text		The name of the system table in the pg_catalog schema where the record about this object is stored (always pg_class for tables and partitions).
objname	name		The name of the object.
objid	oid		The OID of the object.
schemaname	name		The name of the schema where the object resides.
usestatus	text		The status of the role who performed the last operation on the object (CURRENT=a currently active role in the system, DROPPED=a role that no longer exists in the system, CHANGED=a role name that exists in the system, but its definition has changed since the last operation was performed).
usename	name		The name of the role that performed the operation on this object.
actionname	name		The action that was taken on the object.
subtype	text		The type of object operated on or the subclass of operation performed.

column	type	references	description
statime	timestampz		The timestamp of the operation. This is the same timestamp that is written to the Greenplum Database server log files in case you need to look up more detailed information about the operation in the logs.
partitionlevel	smallint		The level of this partition in the hierarchy.
parenttablename	name		The relation name of the parent table one level up from this partition.
parentschemaname	name		The name of the schema where the parent table resides.
parent_relid	oid		The OID of the parent table one level up from this partition.

pg_stat_replication

The `pg_stat_replication` view contains metadata of the `walsender` process that is used for Greenplum Database master mirroring.

Table 85: pg_catalog.pg_stat_replication

column	type	references	description
procpid	integer		Process ID of WAL sender backend process.
usesysid	integer		User system ID that runs the WAL sender backend process
usename	name		User name that runs WAL sender backend process.
application_name	oid		Client application name.
client_addr	name		Client IP address.
client_port	integer		Client port number.
backend_start	timestamp		Operation start timestamp.

column	type	references	description
`state`	text		WAL sender state. The value can be: `startup` `backup` `catchup` `streaming`
`sent_location`	text		WAL sender xlog record sent location.
`write_location`	text		WAL receiver xlog record write location.
`flush_location`	text		WAL receiver xlog record flush location.
`replay_location`	text		Standby xlog record replay location.
`sync_priority`	text		Priorty. the value is `1`.
`sync_state`	text		WAL sender syncronization state. The value is `sync`.

pg_statistic

The `pg_statistic` system catalog table stores statistical data about the contents of the database. Entries are created by `ANALYZE` and subsequently used by the query optimizer. There is one entry for each table column that has been analyzed. Note that all the statistical data is inherently approximate, even assuming that it is up-to-date.

`pg_statistic` also stores statistical data about the values of index expressions. These are described as if they were actual data columns; in particular, `starelid` references the index. No entry is made for an ordinary non-expression index column, however, since it would be redundant with the entry for the underlying table column.

Since different kinds of statistics may be appropriate for different kinds of data, `pg_statistic` is designed not to assume very much about what sort of statistics it stores. Only extremely general statistics (such as nullness) are given dedicated columns in `pg_statistic`. Everything else is stored in slots, which are groups of associated columns whose content is identified by a code number in one of the slot's columns.

`pg_statistic` should not be readable by the public, since even statistical information about a table's contents may be considered sensitive (for example: minimum and maximum values of a salary column). `pg_stats` is a publicly readable view on `pg_statistic` that only exposes information about those tables that are readable by the current user.

Table 86: pg_catalog.pg_statistic

column	type	references	description
`starelid`	oid	pg_class.oid	The table or index that the described column belongs to.

column	type	references	description
`staattnum`	int2	pg_attribute.attnum	The number of the described column.
`stanullfrac`	float4		The fraction of the column's entries that are null.
`stawidth`	int4		The average stored width, in bytes, of nonnull entries.
`stadistinct`	float4		The number of distinct nonnull data values in the column. A value greater than zero is the actual number of distinct values. A value less than zero is the negative of a fraction of the number of rows in the table (for example, a column in which values appear about twice on the average could be represented by `stadistinct` = -0.5). A zero value means the number of distinct values is unknown.
`stakind`*N*	int2		A code number indicating the kind of statistics stored in the `N`th slot of the `pg_statistic` row.
`staop`*N*	oid	pg_operator.oid	An operator used to derive the statistics stored in the `N`th slot. For example, a histogram slot would show the `<` operator that defines the sort order of the data.
`stanumbers`*N*	float4[]		Numerical statistics of the appropriate kind for the `N`th slot, or NULL if the slot kind does not involve numerical values.

column	type	references	description
`stavaluesN`	anyarray		Column data values of the appropriate kind for the `N`th slot, or NULL if the slot kind does not store any data values. Each array's element values are actually of the specific column's data type, so there is no way to define these columns' type more specifically than `anyarray`.

pg_stat_resqueues

The `pg_stat_resqueues` view allows administrators to view metrics about a resource queue's workload over time. To allow statistics to be collected for this view, you must enable the `stats_queue_level` server configuration parameter on the Greenplum Database master instance. Enabling the collection of these metrics does incur a small performance penalty, as each statement submitted through a resource queue must be logged in the system catalog tables.

Table 87: pg_catalog.pg_stat_resqueues

column	type	references	description
`queueoid`	oid		The OID of the resource queue.
`queuename`	name		The name of the resource queue.
`n_queries_exec`	bigint		Number of queries submitted for execution from this resource queue.
`n_queries_wait`	bigint		Number of queries submitted to this resource queue that had to wait before they could execute.
`elapsed_exec`	bigint		Total elapsed execution time for statements submitted through this resource queue.
`elapsed_wait`	bigint		Total elapsed time that statements submitted through this resource queue had to wait before they were executed.

pg_tablespace

The `pg_tablespace` system catalog table stores information about the available tablespaces. Tables can be placed in particular tablespaces to aid administration of disk layout. Unlike most system catalogs, `pg_tablespace` is shared across all databases of a Greenplum system: there is only one copy of `pg_tablespace` per system, not one per database.

Table 88: pg_catalog.pg_tablespace

column	type	references	description
`spcname`	name		Tablespace name.
`spcowner`	oid	pg_authid.oid	Owner of the tablespace, usually the user who created it.
`spclocation`	text[]		Deprecated.
`spcacl`	aclitem[]		Tablespace access privileges.
`spcprilocations`	text[]		Deprecated.
`spcmrilocations`	text[]		Deprecated.
`spcfsoid`	oid	pg_filespace.oid	The object id of the filespace used by this tablespace. A filespace defines directory locations on the primary, mirror and master segments.

pg_trigger

The `pg_trigger` system catalog table stores triggers on tables.

Note: Greenplum Database does not support triggers.

Table 89: pg_catalog.pg_trigger

column	type	references	description
`tgrelid`	oid	*pg_class.oid* Note that Greenplum Database does not enforce referential integrity.	The table this trigger is on.
`tgname`	name		Trigger name (must be unique among triggers of same table).

column	type	references	description
`tgfoid`	oid	*pg_proc.oid* Note that Greenplum Database does not enforce referential integrity.	The function to be called.
`tgtype`	int2		Bit mask identifying trigger conditions.
`tgenabled`	boolean		True if trigger is enabled.
`tgisconstraint`	boolean		True if trigger implements a referential integrity constraint.
`tgconstrname`	name		Referential integrity constraint name.
`tgconstrrelid`	oid	*pg_class.oid* Note that Greenplum Database does not enforce referential integrity.	The table referenced by an referential integrity constraint.
`tgdeferrable`	boolean		True if deferrable.
`tginitdeferred`	boolean		True if initially deferred.
`tgnargs`	int2		Number of argument strings passed to trigger function.
`tgattr`	int2vector		Currently not used.
`tgargs`	bytea		Argument strings to pass to trigger, each NULL-terminated.

pg_type

The `pg_type` system catalog table stores information about data types. Base types (scalar types) are created with `CREATE TYPE`, and domains with `CREATE DOMAIN`. A composite type is automatically created for each table in the database, to represent the row structure of the table. It is also possible to create composite types with `CREATE TYPE AS`.

Table 90: pg_catalog.pg_type

column	type	references	description
`typname`	name		Data type name.
`typnamespace`	oid	pg_namespace.oid	The OID of the namespace that contains this type.
`typowner`	oid	pg_authid.oid	Owner of the type.

column	type	references	description
`typlen`	int2		For a fixed-size type, `typlen` is the number of bytes in the internal representation of the type. But for a variable-length type, `typlen` is negative. `-1` indicates a 'varlena' type (one that has a length word), `-2` indicates a null-terminated C string.
`typbyval`	boolean		Determines whether internal routines pass a value of this type by value or by reference. `typbyval` had better be false if `typlen` is not 1, 2, or 4 (or 8 on machines where Datum is 8 bytes). Variable-length types are always passed by reference. Note that `typbyval` can be false even if the length would allow pass-by-value; this is currently true for type `float4`, for example.
`typtype`	char		`b` for a base type, `c` for a composite type, `d` for a domain, or `p` for a pseudo-type.
`typisdefined`	boolean		True if the type is defined, false if this is a placeholder entry for a not-yet-defined type. When false, nothing except the type name, namespace, and OID can be relied on.
`typdelim`	char		Character that separates two values of this type when parsing array input. Note that the delimiter is associated with the array element data type, not the array data type.

column	type	references	description
typrelid	oid	pg_class.oid	If this is a composite type, then this column points to the pg_class entry that defines the corresponding table. (For a free-standing composite type, the pg_class entry does not really represent a table, but it is needed anyway for the type's pg_attribute entries to link to.) Zero for non-composite types.
typelem	oid	pg_type.oid	If not 0 then it identifies another row in pg_type. The current type can then be subscripted like an array yielding values of type typelem. A true array type is variable length (typlen = -1), but some fixed-length (tylpen > 0) types also have nonzero typelem, for example name and point. If a fixed-length type has a typelem then its internal representation must be some number of values of the typelem data type with no other data. Variable-length array types have a header defined by the array subroutines.
typinput	regproc	pg_proc.oid	Input conversion function (text format).
typoutput	regproc	pg_proc.oid	Output conversion function (text format).
typreceive	regproc	pg_proc.oid	Input conversion function (binary format), or 0 if none.
typsend	regproc	pg_proc.oid	Output conversion function (binary format), or 0 if none.
typanalyze	regproc	pg_proc.oid	Custom ANALYZE function, or 0 to use the standard function.

column	type	references	description
`typalign`	char		The alignment required when storing a value of this type. It applies to storage on disk as well as most representations of the value inside Greenplum Database. When multiple values are stored consecutively, such as in the representation of a complete row on disk, padding is inserted before a datum of this type so that it begins on the specified boundary. The alignment reference is the beginning of the first datum in the sequence. Possible values are: `c` = char alignment (no alignment needed). `s` = short alignment (2 bytes on most machines). `i` = int alignment (4 bytes on most machines). `d` = double alignment (8 bytes on many machines, but not all).

column	type	references	description
`typstorage`	char		For varlena types (those with `typlen` = -1) tells if the type is prepared for toasting and what the default strategy for attributes of this type should be. Possible values are: `p`: Value must always be stored plain. `e`: Value can be stored in a secondary relation (if relation has one, see `pg_class.reltoastrelid`). `m`: Value can be stored compressed inline. `x`: Value can be stored compressed inline or stored in secondary storage. Note that `m` columns can also be moved out to secondary storage, but only as a last resort (`e` and `x` columns are moved first).
`typnotnull`	boolean		Represents a not-null constraint on a type. Used for domains only.
`typbasetype`	oid	pg_type.oid	Identifies the type that a domain is based on. Zero if this type is not a domain.
`typtypmod`	int4		Domains use typtypmod to record the typmod to be applied to their base type (-1 if base type does not use a typmod). -1 if this type is not a domain.

column	type	references	description
`typndims`	int4		The number of array dimensions for a domain that is an array (if `typbasetype` is an array type; the domain's `typelem` will match the base type's `typelem`). Zero for types other than array domains.
`typdefaultbin`	text		If not null, it is the `nodeToString()` representation of a default expression for the type. This is only used for domains.
`typdefault`	text		Null if the type has no associated default value. If not null, typdefault must contain a human-readable version of the default expression represented by typdefaultbin. If typdefaultbin is null and typdefault is not, then typdefault is the external representation of the type's default value, which may be fed to the type's input converter to produce a constant.

pg_type_encoding

The `pg_type_encoding` system catalog table contains the column storage type information.

Table 91: pg_catalog.pg_type_encoding

column	type	modifers	storage	description
`typeid`	oid	not null	plain	Foreign key to *pg_attribute*
`typoptions`	text [ ]		extended	The actual options

pg_user_mapping

The `pg_user_mapping` catalog table stores the mappings from local users to remote users. You must have administrator privileges to view this catalog.

Table 92: pg_catalog.pg_user_mapping

column	type	references	description
`umuser`	oid	pg_authid.oid	OID of the local role being mapped, 0 if the user mapping is public
`umserver`	oid	pg_foreign_server.oid	The OID of the foreign server that contains this mapping
`umoptions`	text [ ]		User mapping specific options, as "keyword= value" strings.

pg_window

The `pg_window` table stores information about window functions. Window functions are often used to compose complex OLAP (online analytical processing) queries. Window functions are applied to partitioned result sets within the scope of a single query expression. A window partition is a subset of rows returned by a query, as defined in a special `OVER()` clause. Typical window functions are `rank`, `dense_rank`, and `row_number`. Each entry in `pg_window` is an extension of an entry in *pg_proc*. The *pg_proc* entry carries the window function's name, input and output data types, and other information that is similar to ordinary functions.

Table 93: pg_catalog.pg_window

column	type	references	description
`winfnoid`	regproc	pg_proc.oid	The OID in `pg_proc` of the window function.
`winrequireorder`	boolean		The window function requires its window specification to have an `ORDER BY` clause.
`winallowframe`	boolean		The window function permits its window specification to have a `ROWS` or `RANGE` framing clause.
`winpeercount`	boolean		The peer group row count is required to compute this window function, so the Window node implementation must 'look ahead' as necessary to make this available in its internal state.
`wincount`	boolean		The partition row count is required to compute this window function.

column	type	references	description
`winfunc`	regproc	pg_proc.oid	The OID in `pg_proc` of a function to compute the value of an immediate-type window function.
`winprefunc`	regproc	pg_proc.oid	The OID in `pg_proc` of a preliminary window function to compute the partial value of a deferred-type window function.
`winpretype`	oid	pg_type.oid	The OID in `pg_type` of the preliminary window function's result type.
winfinfunc	regproc	pg_proc.oid	The OID in `pg_proc` of a function to compute the final value of a deferred-type window function from the partition row count and the result of `winprefunc`.
winkind	char		A character indicating membership of the window function in a class of related functions: `w` - ordinary window functions `n` - NTILE functions `f` - FIRST_VALUE functions `l` - LAST_VALUE functions `g` - LAG functions `d` - LEAD functions

Chapter 6

The gp_toolkit Administrative Schema

Greenplum Database provides an administrative schema called `gp_toolkit` that you can use to query the system catalogs, log files, and operating environment for system status information. The `gp_toolkit` schema contains a number of views that you can access using SQL commands. The `gp_toolkit` schema is accessible to all database users, although some objects may require superuser permissions. For convenience, you may want to add the `gp_toolkit` schema to your schema search path. For example:

```
=> ALTER ROLE myrole SET search_path TO myschema,gp_toolkit;
```

This documentation describes the most useful views in `gp_toolkit`. You may notice other objects (views, functions, and external tables) within the `gp_toolkit` schema that are not described in this documentation (these are supporting objects to the views described in this section).

> **Warning:** Do not change database objects in the gp_toolkit schema. Do not create database objects in the schema. Changes to objects in the schema might affect the accuracy of administrative information returned by schema objects. Any changes made in the gp_toolkit schema are lost when the database is backed up and then restored with the `gpcrondump` and `gpdbrestore` utilities.

These are the categories for views in the `gp_toolkit` schema:

- *Checking for Tables that Need Routine Maintenance*
- *Checking for Locks*
- *Checking Append-Optimized Tables*
- *Viewing Greenplum Database Server Log Files*
- *Checking Server Configuration Files*
- *Checking for Failed Segments*
- *Checking Resource Queue Activity and Status*
- *Checking Query Disk Spill Space Usage*
- *Viewing Users and Groups (Roles)*
- *Checking Database Object Sizes and Disk Space*
- *Checking for Uneven Data Distribution*

Checking for Tables that Need Routine Maintenance

The following views can help identify tables that need routine table maintenance (`VACUUM` and/or `ANALYZE`).

- *gp_bloat_diag*
- *gp_stats_missing*

The `VACUUM` or `VACUUM FULL` command reclaims disk space occupied by deleted or obsolete rows. Because of the MVCC transaction concurrency model used in Greenplum Database, data rows that are deleted or updated still occupy physical space on disk even though they are not visible to any new transactions. Expired rows increase table size on disk and eventually slow down scans of the table.

The `ANALYZE` command collects column-level statistics needed by the query optimizer. Greenplum Database uses a cost-based query optimizer that relies on database statistics. Accurate statistics allow the query optimizer to better estimate selectivity and the number of rows retrieved by a query operation in order to choose the most efficient query plan.

gp_bloat_diag

This view shows regular heap-storage tables that have bloat (the actual number of pages on disk exceeds the expected number of pages given the table statistics). Tables that are bloated require a `VACUUM` or a `VACUUM FULL` in order to reclaim disk space occupied by deleted or obsolete rows. This view is accessible to all users, however non-superusers will only be able to see the tables that they have permission to access.

> **Note:** For diagnostic functions that return append-optimized table information, see *Checking Append-Optimized Tables*.

Table 94: gp_bloat_diag view

Column	Description
bdirelid	Table object id.
bdinspname	Schema name.
bdirelname	Table name.
bdirelpages	Actual number of pages on disk.
bdiexppages	Expected number of pages given the table data.
bdidiag	Bloat diagnostic message.

gp_stats_missing

This view shows tables that do not have statistics and therefore may require an `ANALYZE` be run on the table.

Table 95: gp_stats_missing view

Column	Description
smischema	Schema name.
smitable	Table name.

Column	Description
smisize	Does this table have statistics? False if the table does not have row count and row sizing statistics recorded in the system catalog, which may indicate that the table needs to be analyzed. This will also be false if the table does not contain any rows. For example, the parent tables of partitioned tables are always empty and will always return a false result.
smicols	Number of columns in the table.
smirecs	Number of rows in the table.

Checking for Locks

When a transaction accesses a relation (such as a table), it acquires a lock. Depending on the type of lock acquired, subsequent transactions may have to wait before they can access the same relation. For more information on the types of locks, see "Managing Data" in the *Greenplum Database Administrator Guide*. Greenplum Database resource queues (used for workload management) also use locks to control the admission of queries into the system.

The `gp_locks_*` family of views can help diagnose queries and sessions that are waiting to access an object due to a lock.

- *gp_locks_on_relation*
- *gp_locks_on_resqueue*

gp_locks_on_relation

This view shows any locks currently being held on a relation, and the associated session information about the query associated with the lock. For more information on the types of locks, see "Managing Data" in the *Greenplum Database Administrator Guide*. This view is accessible to all users, however non-superusers will only be able to see the locks for relations that they have permission to access.

Table 96: gp_locks_on_relation view

Column	Description
lorlocktype	Type of the lockable object: `relation`, `extend`, `page`, `tuple`, `transactionid`, `object`, `userlock`, `resource queue`, or `advisory`
lordatabase	Object ID of the database in which the object exists, zero if the object is a shared object.
lorrelname	The name of the relation.
lorrelation	The object ID of the relation.
lortransaction	The transaction ID that is affected by the lock.
lorpid	Process ID of the server process holding or awaiting this lock. NULL if the lock is held by a prepared transaction.
lormode	Name of the lock mode held or desired by this process.
lorgranted	Displays whether the lock is granted (true) or not granted (false).
lorcurrentquery	The current query in the session.

gp_locks_on_resqueue

This view shows any locks currently being held on a resource queue, and the associated session information about the query associated with the lock. This view is accessible to all users, however non-superusers will only be able to see the locks associated with their own sessions.

Table 97: gp_locks_on_resqueue view

Column	Description
lorusename	Name of the user executing the session.
lorrsqname	The resource queue name.
lorlocktype	Type of the lockable object: `resource queue`
lorobjid	The ID of the locked transaction.
lortransaction	The ID of the transaction that is affected by the lock.
lorpid	The process ID of the transaction that is affected by the lock.
lormode	The name of the lock mode held or desired by this process.
lorgranted	Displays whether the lock is granted (true) or not granted (false).
lorwaiting	Displays whether or not the session is waiting.

Checking Append-Optimized Tables

The `gp_toolkit` schema includes a set of diagnostic functions you can use to investigate the state of append-optimized tables.

When an append-optimized table (or column-oriented append-optimized table) is created, another table is implicitly created, containing metadata about the current state of the table. The metadata includes information such as the number of records in each of the table's segments.

Append-optimized tables may have non-visible rows—rows that have been updated or deleted, but remain in storage until the table is compacted using `VACUUM`. The hidden rows are tracked using an auxiliary visibility map table, or visimap.

The following functions let you access the metadata for append-optimized and column-oriented tables and view non-visible rows. Some of the functions have two versions: one that takes the `oid` of the table, and one that takes the name of the table. The latter version has "_name" appended to the function name.

__gp_aovisimap_compaction_info(oid)

This function displays compaction information for an append-optimized table. The information is for the on-disk data files on Greenplum Database segments that store the table data. You can use the information to determine the data files that will be compacted by a `VACUUM` operation on an append-optimized table.

> **Note:** Until a `VACUUM` operation deletes the row from the data file, deleted or updated data rows occupy physical space on disk even though they are hidden to new transactions. The configuration parameter `gp_appendonly_compaction` controls the functionality of the `VACUUM` command.

This table describes the `__gp_aovisimap_compaction_info` function output table.

Table 98: __gp_aovisimap_compaction_info output table

Column	Description
content	Greenplum Database segment ID.
datafile	ID of the data file on the segment.
compaction_possible	The value is either `t` or `f`. The value `t` indicates that the data in data file be compacted when a `VACUUM` operation is performed. The server configuration parameter `gp_appendonly_compaction_threshold` affects this value.
hidden_tupcount	In the data file, the number of hidden (deleted or updated) rows.
total_tupcount	In the data file, the total number of rows.
percent_hidden	In the data file, the ratio (as a percentage) of hidden (deleted or updated) rows to total rows.

> **Note:** If you upgraded your cluster to Greenplum Database 4.3.5.0 or later, you can create the `__gp_aovisimap_compaction_info` function in an existing Greenplum database by running run the

script `$GPHOME/share/postgresql/compaction_info.sql` once for each database. For example, to install the functions in database *testdb*, use this command:

```
$ psql -d testdb -f $GPHOME/share/postgresql/compaction_info.sql
```

If you created the database with Greenplum Database 4.3.5.0 or later, this function is automatically created in the database.

__gp_aoseg_name('table_name')

This function returns metadata information contained in the append-optimized table's on-disk segment file.

Table 99: __gp_aoseg_name output table

Column	Description
segno	The file segment number.
eof	The effective end of file for this file segment.
tupcount	The total number of tuples in the segment, including invisible tuples.
varblockcount	The total number of varblocks in the file segment.
eof_uncompressed	The end of file if the file segment were uncompressed.
modcount	The number of data modification operations.
state	The state of the file segment. Indicates if the segment is active or ready to be dropped after compaction.

__gp_aoseg_history(oid)

This function returns metadata information contained in the append-optimized table's on-disk segment file. It displays all different versions (heap tuples) of the aoseg meta information. The data is complex, but users with a deep understanding of the system may find it usefulfor debugging.

The input argument is the oid of the append-optimized table.

Call __gp_aoseg_history_name('table_name') to get the same result with the table name as an argument.

Table 100: __gp_aoseg_history output table

Column	Description
gp_tid	The id of the tuple.
gp_xmin	The id of the earliest transaction.
gp_xmin_status	Status of the gp_xmin transaction.
gp_xmin_commit_	The commit distribution id of the gp_xmin transaction.
gp_xmax	The id of the latest transaction.
gp_xmax_status	The status of the latest transaction.

Column	Description
gp_xmax_commit_	The commit distribution id of the gp_xmax transaction.
gp_command_id	The id of the query command.
gp_infomask	A bitmap containing state information.
gp_update_tid	The ID of the newer tuple if the row is updated.
gp_visibility	The tuple visibility status.
segno	The number of the segment in the segment file.
tupcount	The number of tuples, including hidden tuples.
eof	The effective end of file for the segment.
eof_uncompressed	The end of file for the segment if data were uncompressed.
modcount	A count of data modifications.
state	The status of the segment.

__gp_aocsseg(oid)

This function returns metadata information contained in a column-oriented append-optimized table's on-disk segment file, excluding non-visible rows. Each row describes a segment for a column in the table.

The input argument is the oid of a column-oriented append-optimized table. Call as __gp_aocsseg_name('table_name') to get the same result with the table name as an argument.

Table 101: __gp_aocsseg(oid) output table

Column	Description
gp_tid	The table id.
segno	The segment number.
column_num	The column number.
physical_segno	The number of the segment in the segment file.
tupcount	The number of rows in the segment, excluding hidden tuples.
eof	The effective end of file for the segment.
eof_uncompressed	The end of file for the segment if the data were uncompressed.
modcount	A count of data modification operations for the segment.
state	The status of the segment.

__gp_aocsseg_history(oid)

This function returns metadata information contained in a column-oriented append-optimized table's on-disk segment file. Each row describes a segment for a column in the table.The data is complex, but users with a deep understanding of the system may find it useful for debugging.

The input argument is the oid of a column-oriented append-optimized table. Call as __gp_aocsseg_history_name('table_name') to get the same result with the table name as argument.

Table 102: __gp_aocsseg_history output table

Column	Description
gp_tid	The oid of the tuple.
gp_xmin	The earliest transaction.
gp_xmin_status	The status of the gp_xmin transaction.
gp_xmin_	Text representation of gp_xmin.
gp_xmax	The latest transaction.
gp_xmax_status	The status of the gp_xmax transaction.
gp_xmax_	Text representation of gp_max.
gp_command_id	ID of the command operating on the tuple.
gp_infomask	A bitmap containing state information.
gp_update_tid	The ID of the newer tuple if the row is updated.
gp_visibility	The tuple visibility status.
segno	The segment number in the segment file.
column_num	The column number.
physical_segno	The segment containing data for the column.
tupcount	The total number of tuples in the segment.
eof	The effective end of file for the segment.
eof_uncompressed	The end of file for the segment if the data were uncompressed.
modcount	A count of the data modification operations.
state	The state of the segment.

__gp_aovisimap(oid)

This function returns the tuple id, the segment file, and the row number of each non-visible tuple according to the visibility map.

The input argument is the oid of an append-optimized table.

Use __gp_aovisimap_name('table_name') to get the same result with the table name as argument.

Column	Description
tid	The tuple id.
segno	The number of the segment file.
row_num	The row number of a row that has been deleted or updated.

__gp_aovisimap_hidden_info(oid)

This function returns the numbers of hidden and visible tuples in the segment files for an append-optimized table.

The input argument is the oid of the append-optimized table.

Call __gp_aovisimap_hidden_info_name('table_name') to get the same result with a table name argument.

Column	Description
segno	The number of the segment file.
hidden_tupcount	The number of hidden tuples in the segment file.
total_tupcount	The total number of tuples in the segment file.

__gp_aovisimap_entry(oid)

This function returns information about each visibility map entry for the table.

The input argument is the oid of an append-optimized table.

Call __gp_aovisimap_entry_name('table_name') to get the same result with a table name argument.

Table 103: __gp_aovisimap_entry output table

Column	Description
segno	Segment number of the visibility map entry.
first_row_num	The first row number of the entry.
hidden_tupcount	The number of hidden tuples in the entry.
bitmap	A text representation of the visibility bitmap.

Viewing Greenplum Database Server Log Files

Each component of a Greenplum Database system (master, standby master, primary segments, and mirror segments) keeps its own server log files. The `gp_log_*` family of views allows you to issue SQL queries against the server log files to find particular entries of interest. The use of these views require superuser permissions.

- *gp_log_command_timings*
- *gp_log_database*
- *gp_log_master_concise*
- *gp_log_system*

gp_log_command_timings

This view uses an external table to read the log files on the master and report the execution time of SQL commands executed in a database session. The use of this view requires superuser permissions.

Table 104: gp_log_command_timings view

Column	Description
logsession	The session identifier (prefixed with "con").
logcmdcount	The command number within a session (prefixed with "cmd").
logdatabase	The name of the database.
loguser	The name of the database user.
logpid	The process id (prefixed with "p").
logtimemin	The time of the first log message for this command.
logtimemax	The time of the last log message for this command.
logduration	Statement duration from start to end time.

gp_log_database

This view uses an external table to read the server log files of the entire Greenplum system (master, segments, and mirrors) and lists log entries associated with the current database. Associated log entries can be identified by the session id (logsession) and command id (logcmdcount). The use of this view requires superuser permissions.

Table 105: gp_log_database view

Column	Description
logtime	The timestamp of the log message.
loguser	The name of the database user.
logdatabase	The name of the database.
logpid	The associated process id (prefixed with "p").
logthread	The associated thread count (prefixed with "th").

Column	Description
loghost	The segment or master host name.
logport	The segment or master port.
logsessiontime	Time session connection was opened.
logtransaction	Global transaction id.
logsession	The session identifier (prefixed with "con").
logcmdcount	The command number within a session (prefixed with "cmd").
logsegment	The segment content identifier (prefixed with "seg" for primary or "mir" for mirror. The master always has a content id of -1).
logslice	The slice id (portion of the query plan being executed).
logdistxact	Distributed transaction id.
loglocalxact	Local transaction id.
logsubxact	Subtransaction id.
logseverity	LOG, ERROR, FATAL, PANIC, DEBUG1 or DEBUG2.
logstate	SQL state code associated with the log message.
logmessage	Log or error message text.
logdetail	Detail message text associated with an error message.
loghint	Hint message text associated with an error message.
logquery	The internally-generated query text.
logquerypos	The cursor index into the internally-generated query text.
logcontext	The context in which this message gets generated.
logdebug	Query string with full detail for debugging.
logcursorpos	The cursor index into the query string.
logfunction	The function in which this message is generated.
logfile	The log file in which this message is generated.
logline	The line in the log file in which this message is generated.
logstack	Full text of the stack trace associated with this message.

gp_log_master_concise

This view uses an external table to read a subset of the log fields from the master log file. The use of this view requires superuser permissions.

Table 106: gp_log_master_concise view

Column	Description
logtime	The timestamp of the log message.
logdatabase	The name of the database.
logsession	The session identifier (prefixed with "con").
logcmdcount	The command number within a session (prefixed with "cmd").
logmessage	Log or error message text.

gp_log_system

This view uses an external table to read the server log files of the entire Greenplum system (master, segments, and mirrors) and lists all log entries. Associated log entries can be identified by the session id (logsession) and command id (logcmdcount). The use of this view requires superuser permissions.

Table 107: gp_log_system view

Column	Description
logtime	The timestamp of the log message.
loguser	The name of the database user.
logdatabase	The name of the database.
logpid	The associated process id (prefixed with "p").
logthread	The associated thread count (prefixed with "th").
loghost	The segment or master host name.
logport	The segment or master port.
logsessiontime	Time session connection was opened.
logtransaction	Global transaction id.
logsession	The session identifier (prefixed with "con").
logcmdcount	The command number within a session (prefixed with "cmd").
logsegment	The segment content identifier (prefixed with "seg" for primary or "mir" for mirror. The master always has a content id of -1).
logslice	The slice id (portion of the query plan being executed).
logdistxact	Distributed transaction id.
loglocalxact	Local transaction id.
logsubxact	Subtransaction id.
logseverity	LOG, ERROR, FATAL, PANIC, DEBUG1 or DEBUG2.
logstate	SQL state code associated with the log message.

Column	Description
logmessage	Log or error message text.
logdetail	Detail message text associated with an error message.
loghint	Hint message text associated with an error message.
logquery	The internally-generated query text.
logquerypos	The cursor index into the internally-generated query text.
logcontext	The context in which this message gets generated.
logdebug	Query string with full detail for debugging.
logcursorpos	The cursor index into the query string.
logfunction	The function in which this message is generated.
logfile	The log file in which this message is generated.
logline	The line in the log file in which this message is generated.
logstack	Full text of the stack trace associated with this message.

Checking Server Configuration Files

Each component of a Greenplum Database system (master, standby master, primary segments, and mirror segments) has its own server configuration file (`postgresql.conf`). The following `gp_toolkit` objects can be used to check parameter settings across all primary `postgresql.conf` files in the system:

- *gp_param_setting('parameter_name')*
- *gp_param_settings_seg_value_diffs*

gp_param_setting('parameter_name')

This function takes the name of a server configuration parameter and returns the `postgresql.conf` value for the master and each active segment. This function is accessible to all users.

Table 108: gp_param_setting('parameter_name') function

Column	Description
paramsegment	The segment content id (only active segments are shown). The master content id is always -1.
paramname	The name of the parameter.
paramvalue	The value of the parameter.

Example:

```
SELECT * FROM gp_param_setting('max_connections');
```

gp_param_settings_seg_value_diffs

Server configuration parameters that are classified as *local* parameters (meaning each segment gets the parameter value from its own `postgresql.conf` file), should be set identically on all segments. This view shows local parameter settings that are inconsistent. Parameters that are supposed to have different values (such as `port`) are not included. This view is accessible to all users.

Table 109: gp_param_settings_seg_value_diffs view

Column	Description
psdname	The name of the parameter.
psdvalue	The value of the parameter.
psdcount	The number of segments that have this value.

Checking for Failed Segments

The *gp_pgdatabase_invalid* view can be used to check for down segments.

gp_pgdatabase_invalid

This view shows information about segments that are marked as down in the system catalog. This view is accessible to all users.

Table 110: gp_pgdatabase_invalid view

Column	Description
pgdbidbid	The segment dbid. Every segment has a unique dbid.
pgdbiisprimary	Is the segment currently acting as the primary (active) segment? (t or f)
pgdbicontent	The content id of this segment. A primary and mirror will have the same content id.
pgdbivalid	Is this segment up and valid? (t or f)
pgdbidefinedprimary	Was this segment assigned the role of primary at system initialization time? (t or f)

Checking Resource Queue Activity and Status

The purpose of resource queues is to limit the number of active queries in the system at any given time in order to avoid exhausting system resources such as memory, CPU, and disk I/O. All database users are assigned to a resource queue, and every statement submitted by a user is first evaluated against the resource queue limits before it can run. The `gp_resq_*` family of views can be used to check the status of statements currently submitted to the system through their respective resource queue. Note that statements issued by superusers are exempt from resource queuing.

- *gp_resq_activity*
- *gp_resq_activity_by_queue*
- *gp_resq_priority_statement*
- *gp_resq_role*
- *gp_resqueue_status*

gp_resq_activity

For the resource queues that have active workload, this view shows one row for each active statement submitted through a resource queue. This view is accessible to all users.

Table 111: gp_resq_activity view

Column	Description
resqprocpid	Process ID assigned to this statement (on the master).
resqrole	User name.
resqoid	Resource queue object id.
resqname	Resource queue name.
resqstart	Time statement was issued to the system.
resqstatus	Status of statement: running, waiting or cancelled.

gp_resq_activity_by_queue

For the resource queues that have active workload, this view shows a summary of queue activity. This view is accessible to all users.

Table 112: gp_resq_activity_by_queue Column

Column	Description
resqoid	Resource queue object id.
resqname	Resource queue name.
resqlast	Time of the last statement issued to the queue.
resqstatus	Status of last statement: running, waiting or cancelled.
resqtotal	Total statements in this queue.

gp_resq_priority_statement

This view shows the resource queue priority, session ID, and other information for all statements currently running in the Greenplum Database system. This view is accessible to all users.

Table 113: gp_resq_priority_statement view

Column	Description
rqpdatname	The database name that the session is connected to.
rqpusename	The user who issued the statement.
rqpsession	The session ID.
rqpcommand	The number of the statement within this session (the command id and session id uniquely identify a statement).
rqppriority	The resource queue priority for this statement (MAX, HIGH, MEDIUM, LOW).
rqpweight	An integer value associated with the priority of this statement.
rqpquery	The query text of the statement.

gp_resq_role

This view shows the resource queues associated with a role. This view is accessible to all users.

Table 114: gp_resq_role view

Column	Description
rrrolname	Role (user) name.
rrrsqname	The resource queue name assigned to this role. If a role has not been explicitly assigned to a resource queue, it will be in the default resource queue (*pg_default*).

gp_resqueue_status

This view allows administrators to see status and activity for a workload management resource queue. It shows how many queries are waiting to run and how many queries are currently active in the system from a particular resource queue.

Table 115: gp_resqueue_status view

Column	Description
`queueid`	The ID of the resource queue.
`rsqname`	The name of the resource queue.
`rsqcountlimit`	The active query threshold of the resource queue. A value of -1 means no limit.

Column	Description
rsqcountvalue	The number of active query slots currently being used in the resource queue.
rsqcostlimit	The query cost threshold of the resource queue. A value of -1 means no limit.
rsqcostvalue	The total cost of all statements currently in the resource queue.
rsqmemorylimit	The memory limit for the resource queue.
rsqmemoryvalue	The total memory used by all statements currently in the resource queue.
rsqwaiters	The number of statements currently waiting in the resource queue.
rsqholders	The number of statements currently running on the system from this resource queue.

Checking Query Disk Spill Space Usage

The *gp_workfile_** views show information about all the queries that are currently using disk spill space. Greenplum Database creates work files on disk if it does not have sufficient memory to execute the query in memory. This information can be used for troubleshooting and tuning queries. The information in the views can also be used to specify the values for the Greenplum Database configuration parameters `gp_workfile_limit_per_query` and `gp_workfile_limit_per_segment`.

- *gp_workfile_entries*
- *gp_workfile_usage_per_query*
- *gp_workfile_usage_per_segment*

gp_workfile_entries

This view contains one row for each operator using disk space for workfiles on a segment at the current time. The view is accessible to all users, however non-superusers only to see information for the databases that they have permission to access.

Table 116: gp_workfile_entries

Column	Type	References	Description
`command_cnt`	integer		Command ID of the query.
`content`	smallint		The content identifier for a segment instance.
`current_query`	text		Current query that the process is running.
`datname`	name		Greenplum database name.
`directory`	text		Path to the work file.
`optype`	text		The query operator type that created the work file.
`procpid`	integer		Process ID of the server process.
`sess_id`	integer		Session ID.
`size`	bigint		The size of the work file in bytes.
`numfiles`	bigint		The number of files created.
`slice`	smallint		The query plan slice. The portion of the query plan that is being executed.

Column	Type	References	Description
`state`	text		The state of the query that created the work file.
`usename`	name		Role name.
`workmem`	integer		The amount of memory allocated to the operator in KB.

gp_workfile_usage_per_query

This view contains one row for each query using disk space for workfiles on a segment at the current time. The view is accessible to all users, however non-superusers only to see information for the databases that they have permission to access.

Table 117: gp_workfile_usage_per_query

Column	Type	References	Description
`command_cnt`	integer		Command ID of the query.
`content`	smallint		The content identifier for a segment instance.
`current_query`	text		Current query that the process is running.
`datname`	name		Greenplum database name.
`procpid`	integer		Process ID of the server process.
`sess_id`	integer		Session ID.
`size`	bigint		The size of the work file in bytes.
`numfiles`	bigint		The number of files created.
`state`	text		The state of the query that created the work file.
`usename`	name		Role name.

gp_workfile_usage_per_segment

This view contains one row for each segment. Each row displays the total amount of disk space used for workfiles on the segment at the current time. The view is accessible to all users, however non-superusers only to see information for the databases that they have permission to access.

Table 118: gp_workfile_usage_per_segment

Column	Type	References	Description
`content`	smallint		The content identifier for a segment instance.
`size`	bigint		The total size of the work files on a segment.
`numfiles`	bigint		The number of files created.

Viewing Users and Groups (Roles)

It is frequently convenient to group users (roles) together to ease management of object privileges: that way, privileges can be granted to, or revoked from, a group as a whole. In Greenplum Database this is done by creating a role that represents the group, and then granting membership in the group role to individual user roles.

The *gp_roles_assigned* view can be used to see all of the roles in the system, and their assigned members (if the role is also a group role).

gp_roles_assigned

This view shows all of the roles in the system, and their assigned members (if the role is also a group role). This view is accessible to all users.

Table 119: gp_roles_assigned view

Column	Description
raroleid	The role object ID. If this role has members (users), it is considered a *group* role.
rarolename	The role (user or group) name.
ramemberid	The role object ID of the role that is a member of this role.
ramembername	Name of the role that is a member of this role.

Checking Database Object Sizes and Disk Space

The `gp_size_*` family of views can be used to determine the disk space usage for a distributed Greenplum Database, schema, table, or index. The following views calculate the total size of an object across all primary segments (mirrors are not included in the size calculations).

- *gp_size_of_all_table_indexes*
- *gp_size_of_database*
- *gp_size_of_index*
- *gp_size_of_partition_and_indexes_disk*
- *gp_size_of_schema_disk*
- *gp_size_of_table_and_indexes_disk*
- *gp_size_of_table_and_indexes_licensing*
- *gp_size_of_table_disk*
- *gp_size_of_table_uncompressed*
- *gp_disk_free*

The table and index sizing views list the relation by object ID (not by name). To check the size of a table or index by name, you must look up the relation name (`relname`) in the `pg_class` table. For example:

```
SELECT relname as name, sotdsize as size, sotdtoastsize as
toast, sotdadditionalsize as other
FROM gp_size_of_table_disk as sotd, pg_class
WHERE sotd.sotdoid=pg_class.oid ORDER BY relname;
```

gp_size_of_all_table_indexes

This view shows the total size of all indexes for a table. This view is accessible to all users, however non-superusers will only be able to see relations that they have permission to access.

Table 120: gp_size_of_all_table_indexes view

Column	Description
soatioid	The object ID of the table
soatisize	The total size of all table indexes in bytes
soatischemaname	The schema name
soatitablename	The table name

gp_size_of_database

This view shows the total size of a database. This view is accessible to all users, however non-superusers will only be able to see databases that they have permission to access.

Table 121: gp_size_of_database view

Column	Description
sodddatname	The name of the database
sodddatsize	The size of the database in bytes

gp_size_of_index

This view shows the total size of an index. This view is accessible to all users, however non-superusers will only be able to see relations that they have permission to access.

Table 122: gp_size_of_index view

Column	Description
soioid	The object ID of the index
soitableoid	The object ID of the table to which the index belongs
soisize	The size of the index in bytes
soiindexschemaname	The name of the index schema
soiindexname	The name of the index
soitableschemaname	The name of the table schema
soitablename	The name of the table

gp_size_of_partition_and_indexes_disk

This view shows the size on disk of partitioned child tables and their indexes. This view is accessible to all users, however non-superusers will only be able to see relations that they have permission to access..

Table 123: gp_size_of_partition_and_indexes_disk view

Column	Description
sopaidparentoid	The object ID of the parent table
sopaidpartitionoid	The object ID of the partition table
sopaidpartitiontablesize	The partition table size in bytes
sopaidpartitionindexessize	The total size of all indexes on this partition
Sopaidparentschemaname	The name of the parent schema
Sopaidparenttablename	The name of the parent table
Sopaidpartitionschemaname	The name of the partition schema
sopaidpartitiontablename	The name of the partition table

gp_size_of_schema_disk

This view shows schema sizes for the public schema and the user-created schemas in the current database. This view is accessible to all users, however non-superusers will be able to see only the schemas that they have permission to access.

Table 124: gp_size_of_schema_disk view

Column	Description
sosdnsp	The name of the schema

Column	Description
sosdschematablesize	The total size of tables in the schema in bytes
sosdschemaidxsize	The total size of indexes in the schema in bytes

gp_size_of_table_and_indexes_disk

This view shows the size on disk of tables and their indexes. This view is accessible to all users, however non-superusers will only be able to see relations that they have permission to access.

Table 125: gp_size_of_table_and_indexes_disk view

Column	Description
sotaidoid	The object ID of the parent table
sotaidtablesize	The disk size of the table
sotaididxsize	The total size of all indexes on the table
sotaidschemaname	The name of the schema
sotaidtablename	The name of the table

gp_size_of_table_and_indexes_licensing

This view shows the total size of tables and their indexes for licensing purposes. The use of this view requires superuser permissions.

Table 126: gp_size_of_table_and_indexes_licensing view

Column	Description
sotailoid	The object ID of the table
sotailtablesizedisk	The total disk size of the table
sotailtablesizeuncompressed	If the table is a compressed append-optimized table, shows the uncompressed table size in bytes.
sotailindexessize	The total size of all indexes in the table
sotailschemaname	The schema name
sotailtablename	The table name

gp_size_of_table_disk

This view shows the size of a table on disk. This view is accessible to all users, however non-superusers will only be able to see tables that they have permission to access

Table 127: gp_size_of_table_disk view

Column	Description
sotdoid	The object ID of the table

Column	Description
sotdsize	The size of the table in bytes. The size is only the main table size. The size does not include auxiliary objects such as oversized (toast) attributes, or additional storage objects for AO tables.
sotdtoastsize	The size of the TOAST table (oversized attribute storage), if there is one.
sotdadditionalsize	Reflects the segment and block directory table sizes for append-optimized (AO) tables.
sotdschemaname	The schema name
sotdtablename	The table name

gp_size_of_table_uncompressed

This view shows the uncompressed table size for append-optimized (AO) tables. Otherwise, the table size on disk is shown. The use of this view requires superuser permissions.

Table 128: gp_size_of_table_uncompressed view

Column	Description
sotuoid	The object ID of the table
sotusize	The uncomresssed size of the table in bytes if it is a compressed AO table. Otherwise, the table size on disk.
sotuschemaname	The schema name
sotutablename	The table name

gp_disk_free

This external table runs the `df` (disk free) command on the active segment hosts and reports back the results. Inactive mirrors are not included in the calculation. The use of this external table requires superuser permissions.

Table 129: gp_disk_free external table

Column	Description
dfsegment	The content id of the segment (only active segments are shown)
dfhostname	The hostname of the segment host
dfdevice	The device name
dfspace	Free disk space in the segment file system in kilobytes

Checking for Uneven Data Distribution

All tables in Greenplum Database are distributed, meaning their data is divided across all of the segments in the system. If the data is not distributed evenly, then query processing performance may suffer. The following views can help diagnose if a table has uneven data distribution:

- *gp_skew_coefficients*
- *gp_skew_idle_fractions*

gp_skew_coefficients

This view shows data distribution skew by calculating the coefficient of variation (CV) for the data stored on each segment. This view is accessible to all users, however non-superusers will only be able to see tables that they have permission to access

Table 130: gp_skew_coefficients view

Column	Description
skcoid	The object id of the table.
skcnamespace	The namespace where the table is defined.
skcrelname	The table name.
skccoeff	The coefficient of variation (CV) is calculated as the standard deviation divided by the average. It takes into account both the average and variability around the average of a data series. The lower the value, the better. Higher values indicate greater data skew.

gp_skew_idle_fractions

This view shows data distribution skew by calculating the percentage of the system that is idle during a table scan, which is an indicator of processing data skew. This view is accessible to all users, however non-superusers will only be able to see tables that they have permission to access

Table 131: gp_skew_idle_fractions view

Column	Description
sifoid	The object id of the table.
sifnamespace	The namespace where the table is defined.
sifrelname	The table name.
siffraction	The percentage of the system that is idle during a table scan, which is an indicator of uneven data distribution or query processing skew. For example, a value of 0.1 indicates 10% skew, a value of 0.5 indicates 50% skew, and so on. Tables that have more than 10% skew should have their distribution policies evaluated.

Chapter 7

Greenplum Database Data Types

Greenplum Database has a rich set of native data types available to users. Users may also define new data types using the `CREATE TYPE` command. This reference shows all of the built-in data types. In addition to the types listed here, there are also some internally used data types, such as *oid* (object identifier), but those are not documented in this guide.

The following data types are specified by SQL: *bit*, *bit varying*, *boolean*, *character varying, varchar*, *character, char*, *date*, *double precision*, *integer*, *interval*, *numeric*, *decimal*, *real*, *smallint*, *time* (with or without time zone), and *timestamp* (with or without time zone).

Each data type has an external representation determined by its input and output functions. Many of the built-in types have obvious external formats. However, several types are either unique to PostgreSQL (and Greenplum Database), such as geometric paths, or have several possibilities for formats, such as the date and time types. Some of the input and output functions are not invertible. That is, the result of an output function may lose accuracy when compared to the original input.

Table 132: Greenplum Database Built-in Data Types

Name	Alias	Size	Range	Description
bigint	int8	8 bytes	-9223372036854775808 to 9223372036854775807	large range integer
bigserial	serial8	8 bytes	1 to 9223372036854775807	large autoincrementing integer
bit [(n)]		*n* bits	*bit string constant*	fixed-length bit string
bit varying [(n)]*Footnote.*	varbit	actual number of bits	*bit string constant*	variable-length bit string
boolean	bool	1 byte	true/false, t/f, yes/no, y/n, 1/0	logical boolean (true/false)
box		32 bytes	((x1,y1),(x2,y2))	rectangular box in the plane - not allowed in distribution key columns.
bytea*Footnote.*		1 byte + *binary string*	sequence of *octets*	variable-length binary string
character [(n)]*Footnote.*	char [(n)]	1 byte + *n*	strings up to *n* characters in length	fixed-length, blank padded

[1] For variable length data types, if the data is greater than or equal to 127 bytes, the storage overhead is 4 bytes instead of 1.

Name	Alias	Size	Range	Description
character varying [(n)]*Footnote.*	varchar [(n)]	1 byte + *string size*	strings up to *n* characters in length	variable-length with limit
cidr		12 or 24 bytes		IPv4 and IPv6 networks
circle		24 bytes	<(x,y),r> (center and radius)	circle in the plane - not allowed in distribution key columns.
date		4 bytes	4713 BC - 294,277 AD	calendar date (year, month, day)
decimal [(p, s)]*Footnote.*	numeric [(p, s)]	variable	no limit	user-specified precision, exact
double precision	float8 float	8 bytes	15 decimal digits precision	variable-precision, inexact
inet		12 or 24 bytes		IPv4 and IPv6 hosts and networks
integer	int, int4	4 bytes	-2147483648 to +2147483647	usual choice for integer
interval [(p)]		12 bytes	-178000000 years - 178000000 years	time span
lseg		32 bytes	((x1,y1),(x2,y2))	line segment in the plane - not allowed in distribution key columns.
macaddr		6 bytes		MAC addresses
money		4 bytes	-21474836.48 to +21474836.47	currency amount
path*Footnote.*		16+16n bytes	[(x1,y1),...]	geometric path in the plane - not allowed in distribution key columns.
point		16 bytes	(x,y)	geometric point in the plane - not allowed in distribution key columns.
polygon		40+16n bytes	((x1,y1),...)	closed geometric path in the plane - not allowed in distribution key columns.
real	float4	4 bytes	6 decimal digits precision	variable-precision, inexact

Name	**Alias**	**Size**	**Range**	**Description**
serial	serial4	4 bytes	1 to 2147483647	autoincrementing integer
smallint	int2	2 bytes	-32768 to +32767	small range integer
text*Footnote.*		1 byte + *string size*	strings of any length	variable unlimited length
time [(p)] [without time zone]		8 bytes	00:00:00[.000000] - 24:00:00[.000000]	time of day only
time [(p)] with time zone	timetz	12 bytes	00:00:00+1359 - 24:00:00-1359	time of day only, with time zone
timestamp [(p)] [without time zone]		8 bytes	4713 BC - 294,277 AD	both date and time
timestamp [(p)] with time zone	timestamptz	8 bytes	4713 BC - 294,277 AD	both date and time, with time zone
xml*Footnote.*		1 byte + *xml size*	xml of any length	variable unlimited length

Chapter 8

Character Set Support

The character set support in Greenplum Database allows you to store text in a variety of character sets, including single-byte character sets such as the ISO 8859 series and multiple-byte character sets such as EUC (Extended Unix Code), UTF-8, and Mule internal code. All supported character sets can be used transparently by clients, but a few are not supported for use within the server (that is, as a server-side encoding). The default character set is selected while initializing your Greenplum Database array using `gpinitsystem`. It can be overridden when you create a database, so you can have multiple databases each with a different character set.

Table 133: Greenplum Database Character Sets [2]

Name	Description	Language	Server?	Bytes/Char	Aliases
BIG5	Big Five	Traditional Chinese	No	1-2	WIN950, Windows950
EUC_CN	Extended UNIX Code-CN	Simplified Chinese	Yes	1-3	
EUC_JP	Extended UNIX Code-JP	Japanese	Yes	1-3	
EUC_KR	Extended UNIX Code-KR	Korean	Yes	1-3	
EUC_TW	Extended UNIX Code-TW	Traditional Chinese, Taiwanese	Yes	1-3	
GB18030	National Standard	Chinese	No	1-2	
GBK	Extended National Standard	Simplified Chinese	No	1-2	WIN936, Windows936
ISO_8859_5	ISO 8859-5, ECMA 113	Latin/Cyrillic	Yes	1	
ISO_8859_6	ISO 8859-6, ECMA 114	Latin/Arabic	Yes	1	
ISO_8859_7	ISO 8859-7, ECMA 118	Latin/Greek	Yes	1	
ISO_8859_8	ISO 8859-8, ECMA 121	Latin/Hebrew	Yes	1	
JOHAB	JOHA	Korean (Hangul)	Yes	1-3	

[2] **Not all APIs support all the listed character sets. For example, the JDBC driver does not support MULE_INTERNAL, LATIN6, LATIN8, and LATIN10.**

Name	Description	Language	Server?	Bytes/Char	Aliases
KOI8	KOI8-R(U)	Cyrillic	Yes	1	KOI8R
LATIN1	ISO 8859-1, ECMA 94	Western European	Yes	1	ISO88591
LATIN2	ISO 8859-2, ECMA 94	Central European	Yes	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	South European	Yes	1	ISO88593
LATIN4	ISO 8859-4, ECMA 94	North European	Yes	1	ISO88594
LATIN5	ISO 8859-9, ECMA 128	Turkish	Yes	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	Nordic	Yes	1	ISO885910
LATIN7	ISO 8859-13	Baltic	Yes	1	ISO885913
LATIN8	ISO 8859-14	Celtic	Yes	1	ISO885914
LATIN9	ISO 8859-15	LATIN1 with Euro and accents	Yes	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	Romanian	Yes	1	ISO885916
MULE_ INTERNAL	Mule internal code	Multilingual Emacs	Yes	1-4	
SJIS	Shift JIS	Japanese	No	1-2	Mskanji, ShiftJIS, WIN932, Windows932
SQL_ASCII	unspecified [3]	any	No	1	
UHC	Unified Hangul Code	Korean	No	1-2	WIN949, Windows949
UTF8	Unicode, 8-bit	all	Yes	1-4	Unicode
WIN866	Windows CP866	Cyrillic	Yes	1	ALT
WIN874	Windows CP874	Thai	Yes	1	
WIN1250	Windows CP1250	Central European	Yes	1	

[3] The SQL_ASCII setting behaves considerably differently from the other settings. Byte values 0-127 are interpreted according to the ASCII standard, while byte values 128-255 are taken as uninterpreted characters. If you are working with any non-ASCII data, it is unwise to use the SQL_ASCII setting as a client encoding. SQL_ASCII is not supported as a server encoding.

Name	Description	Language	Server?	Bytes/Char	Aliases
WIN1251	Windows CP1251	Cyrillic	Yes	1	WIN
WIN1252	Windows CP1252	Western European	Yes	1	
WIN1253	Windows CP1253	Greek	Yes	1	
WIN1254	Windows CP1254	Turkish	Yes	1	
WIN1255	Windows CP1255	Hebrew	Yes	1	
WIN1256	Windows CP1256	Arabic	Yes	1	
WIN1257	Windows CP1257	Baltic	Yes	1	
WIN1258	Windows CP1258	Vietnamese	Yes	1	ABC, TCVN, TCVN5712, VSCII

Setting the Character Set

gpinitsystem defines the default character set for a Greenplum Database system by reading the setting of the ENCODING parameter in the gp_init_config file at initialization time. The default character set is UNICODE or UTF8.

You can create a database with a different character set besides what is used as the system-wide default. For example:

```
=> CREATE DATABASE korean WITH ENCODING 'EUC_KR';
```

Important: Although you can specify any encoding you want for a database, it is unwise to choose an encoding that is not what is expected by the locale you have selected. The LC_COLLATE and LC_CTYPE settings imply a particular encoding, and locale-dependent operations (such as sorting) are likely to misinterpret data that is in an incompatible encoding.

Since these locale settings are frozen by gpinitsystem, the apparent flexibility to use different encodings in different databases is more theoretical than real.

One way to use multiple encodings safely is to set the locale to C or POSIX during initialization time, thus disabling any real locale awareness.

Character Set Conversion Between Server and Client

Greenplum Database supports automatic character set conversion between server and client for certain character set combinations. The conversion information is stored in the master *pg_conversion* system catalog table. Greenplum Database comes with some predefined conversions or you can create a new conversion using the SQL command `CREATE CONVERSION`.

Table 134: Client/Server Character Set Conversions

Server Character Set	Available Client Character Sets
BIG5	not supported as a server encoding
EUC_CN	EUC_CN, MULE_INTERNAL, UTF8
EUC_JP	EUC_JP, MULE_INTERNAL, SJIS, UTF8
EUC_KR	EUC_KR, MULE_INTERNAL, UTF8
EUC_TW	EUC_TW, BIG5, MULE_INTERNAL, UTF8
GB18030	not supported as a server encoding
GBK	not supported as a server encoding
ISO_8859_5	ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN866, WIN1251
ISO_8859_6	ISO_8859_6, UTF8
ISO_8859_7	ISO_8859_7, UTF8
ISO_8859_8	ISO_8859_8, UTF8
JOHAB	JOHAB, UTF8
KOI8	KOI8, ISO_8859_5, MULE_INTERNAL, UTF8, WIN866, WIN1251
LATIN1	LATIN1, MULE_INTERNAL, UTF8
LATIN2	LATIN2, MULE_INTERNAL, UTF8, WIN1250
LATIN3	LATIN3, MULE_INTERNAL, UTF8
LATIN4	LATIN4, MULE_INTERNAL, UTF8
LATIN5	LATIN5, UTF8
LATIN6	LATIN6, UTF8
LATIN7	LATIN7, UTF8
LATIN8	LATIN8, UTF8
LATIN9	LATIN9, UTF8
LATIN10	LATIN10, UTF8
MULE_INTERNAL	MULE_INTERNAL, BIG5, EUC_CN, EUC_JP, EUC_KR, EUC_TW, ISO_8859_5, KOI8, LATIN1 to LATIN4, SJIS, WIN866, WIN1250, WIN1251
SJIS	not supported as a server encoding

Server Character Set	Available Client Character Sets
SQL_ASCII	not supported as a server encoding
UHC	not supported as a server encoding
UTF8	all supported encodings
WIN866	WIN866
ISO_8859_5	KOI8, MULE_INTERNAL, UTF8, WIN1251
WIN874	WIN874, UTF8
WIN1250	WIN1250, LATIN2, MULE_INTERNAL, UTF8
WIN1251	WIN1251, ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN866
WIN1252	WIN1252, UTF8
WIN1253	WIN1253, UTF8
WIN1254	WIN1254, UTF8
WIN1255	WIN1255, UTF8
WIN1256	WIN1256, UTF8
WIN1257	WIN1257, UTF8
WIN1258	WIN1258, UTF8

To enable automatic character set conversion, you have to tell Greenplum Database the character set (encoding) you would like to use in the client. There are several ways to accomplish this:

- Using the `\encoding` command in `psql`, which allows you to change client encoding on the fly.
- Using `SETclient_encoding TO`.

  To set the client encoding, use the following SQL command:

  ```
  => SET CLIENT_ENCODING TO 'value';
  ```

  To query the current client encoding:

  ```
  => SHOW client_encoding;
  ```

  To return to the default encoding:

  ```
  => RESET client_encoding;
  ```

- Using the `PGCLIENTENCODING` environment variable. When `PGCLIENTENCODING` is defined in the client's environment, that client encoding is automatically selected when a connection to the server is made. (This can subsequently be overridden using any of the other methods mentioned above.)
- Setting the configuration parameter `client_encoding`. If `client_encoding` is set in the master `postgresql.conf` file, that client encoding is automatically selected when a connection to Greenplum Database is made. (This can subsequently be overridden using any of the other methods mentioned above.)

If the conversion of a particular character is not possible " suppose you chose `EUC_JP` for the server and `LATIN1` for the client, then some Japanese characters do not have a representation in `LATIN1` " then an error is reported.

If the client character set is defined as `SQL_ASCII`, encoding conversion is disabled, regardless of the server's character set. The use of `SQL_ASCII` is unwise unless you are working with all-ASCII data. `SQL_ASCII` is not supported as a server encoding.

Chapter 9

Server Configuration Parameters

There are many Greenplum server configuration parameters that affect the behavior of the Greenplum Database system. Many of these configuration parameters have the same names, settings, and behaviors as in a regular PostgreSQL database system.

- *Parameter Types and Values* describes the parameter data types and values.
- *Setting Parameters* describes limitations on who can change them and where or when they can be set.
- *Parameter Categories* organizes parameters by functionality.
- *Configuration Parameters* lists the parameter descriptions in alphabetic order.

Parameter Types and Values

All parameter names are case-insensitive. Every parameter takes a value of one of four types: `Boolean`, `integer`, `floating point`, or `string`. Boolean values may be written as `ON`, `OFF`, `TRUE`, `FALSE`, `YES`, `NO`, `1`, `0` (all case-insensitive).

Some settings specify a memory size or time value. Each of these has an implicit unit, which is either kilobytes, blocks (typically eight kilobytes), milliseconds, seconds, or minutes. Valid memory size units are `kB` (kilobytes), `MB` (megabytes), and `GB` (gigabytes). Valid time units are `ms` (milliseconds), `s` (seconds), `min` (minutes), `h` (hours), and `d` (days). Note that the multiplier for memory units is 1024, not 1000. A valid time expression contains a number and a unit. When specifying a memory or time unit using the `SET` command, enclose the value in quotes. For example:

```
SET statement_mem TO '200MB';
```

Note: There is no space between the value and the unit names.

Setting Parameters

Many of the configuration parameters have limitations on who can change them and where or when they can be set. For example, to change certain parameters, you must be a Greenplum Database superuser. Other parameters require a restart of the system for the changes to take effect. A parameter that is classified as *session* can be set at the system level (in the `postgresql.conf` file), at the database-level (using `ALTER DATABASE`), at the role-level (using `ALTER ROLE`), or at the session-level (using `SET`). System parameters can only be set in the `postgresql.conf` file.

In Greenplum Database, the master and each segment instance has its own `postgresql.conf` file (located in their respective data directories). Some parameters are considered *local* parameters, meaning that each segment instance looks to its own `postgresql.conf` file to get the value of that parameter. You must set local parameters on every instance in the system (master and segments). Others parameters are considered *master* parameters. Master parameters need only be set at the master instance.

This table describes the values in the Settable Classifications column of the table in the description of a server configuration parameter.

Table 135: Settable Classifications

Set Classification	Description
master or local	A *master* parameter only needs to be set in the `postgresql.conf` file of the Greenplum master instance. The value for this parameter is then either passed to (or ignored by) the segments at run time. A local parameter must be set in the `postgresql.conf` file of the master AND each segment instance. Each segment instance looks to its own configuration to get the value for the parameter. Local parameters always requires a system restart for changes to take effect.
session or system	*Session* parameters can be changed on the fly within a database session, and can have a hierarchy of settings: at the system level (`postgresql.conf`), at the database level (`ALTER DATABASE...SET`), at the role level (`ALTER ROLE...SET`), or at the session level (`SET`). If the parameter is set at multiple levels, then the most granular setting takes precedence (for example, session overrides role, role overrides database, and database overrides system). A *system* parameter can only be changed via the `postgresql.conf` file(s).
restart or reload	When changing parameter values in the postgrsql.conf file(s), some require a *restart* of Greenplum Database for the change to take effect. Other parameter values can be refreshed by just reloading the server configuration file (using `gpstop -u`), and do not require stopping the system.

Set Classification	Description
superuser	These session parameters can only be set by a database superuser. Regular database users cannot set this parameter.
read only	These parameters are not settable by database users or superusers. The current value of the parameter can be shown but not altered.

Configuration Parameter Categories and Listing

Parameter Categories

Configuration parameters affect categories of server behaviors, such as resource consumption, query tuning, and authentication. The following topics describe Greenplum configuration parameter categories.

Connection and Authentication Parameters

These parameters control how clients connect and authenticate to Greenplum Database.

Connection Parameters

gp_connection_send_timeout	*tcp_keepalives_count*
gp_vmem_idle_resource_timeout	*tcp_keepalives_idle*
listen_addresses	*tcp_keepalives_interval*
max_connections	*unix_socket_directory*
max_prepared_transactions	*unix_socket_group*
superuser_reserved_connections	*unix_socket_permissions*

Security and Authentication Parameters

authentication_timeout	*krb_srvname*
db_user_namespace	*password_encryption*
krb_caseins_users	*password_hash_algorithm*
	ssl

krb_server_keyfile

ssl_ciphers

System Resource Consumption Parameters

These parameters set the limits for system resources consumed by Greenplum Database.

Memory Consumption Parameters

These parameters control system memory usage. You can adjust `gp_vmem_protect_limit` to avoid running out of memory at the segment hosts during query processing.

gp_vmem_idle_resource_timeout

gp_vmem_protect_limit

gp_vmem_protect_segworker_cache_limit

gp_workfile_limit_files_per_query

gp_workfile_limit_per_query

gp_workfile_limit_per_segment

max_stack_depth

shared_buffers

temp_buffers

Free Space Map Parameters

These parameters control the sizing of the *free space map, which contains* expired rows. Use `VACUUM` to reclaim the free space map disk space.

max_fsm_pages

max_fsm_relations

OS Resource Parameters

max_files_per_process

shared_preload_libraries

Cost-Based Vacuum Delay Parameters

> **Warning:** Pivotal does not recommend cost-based vacuum delay because it runs asynchronously among the segment instances. The vacuum cost limit and delay is invoked at the segment level without taking into account the state of the entire Greenplum array

You can configure the execution cost of `VACUUM` and `ANALYZE` commands to reduce the I/O impact on concurrent database activity. When the accumulated cost of I/O operations reaches the limit, the process performing the operation sleeps for a while, Then resets the counter and continues execution

vacuum_cost_delay

vacuum_cost_limit

vacuum_cost_page_dirty

vacuum_cost_page_hit

vacuum_cost_page_miss

Transaction ID Management Parameters

xid_stop_limit

xid_warn_limit

Pivotal Query Optimizer Parameters

These parameters control the usage of the Pivotal Query Optimizer by Greenplum Database. For information about the Pivotal Query Optimizer, see "Querying Data" in the *Greenplum Database Administrator Guide*.

optimizer	*optimizer_control*
optimizer_analyze_root_partition	*optimizer_enable_master_only_queries*

Query Tuning Parameters

These parameters control aspects of SQL query processing such as query operators and operator settings and statistics sampling.

Legacy Query Optimizer Operator Control Parameters

The following parameters control the types of plan operations the legacy query optimizer can use. Enable or disable plan operations to force the legacy optimizer to choose a different plan. This is useful for testing and comparing query performance using different plan types.

enable_bitmapscan	*gp_enable_agg_distinct_pruning*
enable_groupagg	*gp_enable_direct_dispatch*
enable_hashagg	*gp_enable_fallback_plan*
enable_hashjoin	*gp_enable_fast_sri*
enable_indexscan	*gp_enable_groupext_distinct_gather*
enable_mergejoin	*gp_enable_groupext_distinct_pruning*
enable_nestloop	*gp_enable_multiphase_agg*
enable_seqscan	*gp_enable_predicate_propagation*
enable_sort	*gp_enable_preunique*
enable_tidscan	*gp_enable_sequential_window_plans*
gp_enable_adaptive_nestloop	*gp_enable_sort_distinct*
gp_enable_agg_distinct	*gp_enable_sort_limit*

Legacy Query Optimizer Costing Parameters

Warning: Pivotal recommends that you do not adjust these query costing parameters. They are tuned to reflect Greenplum Database hardware configurations and typical workloads. All of these parameters are related. Changing one without changing the others can have adverse affects on performance.

cpu_index_tuple_cost	*gp_motion_cost_per_row*
cpu_operator_cost	*gp_segments_for_planner*
cpu_tuple_cost	*random_page_cost*
cursor_tuple_fraction	*seq_page_cost*
effective_cache_size	

Database Statistics Sampling Parameters

These parameters adjust the amount of data sampled by an `ANALYZE` operation. Adjusting these parameters affects statistics collection system-wide. You can configure statistics collection on particular tables and columns by using the `ALTER TABLESET STATISTICS` clause.

default_statistics_target

gp_analyze_relative_error

Sort Operator Configuration Parameters

gp_enable_sort_distinct

gp_enable_sort_limit

Aggregate Operator Configuration Parameters

gp_enable_agg_distinct

gp_enable_agg_distinct_pruning

gp_enable_multiphase_agg

gp_enable_preunique

gp_enable_groupext_distinct_gather

gp_enable_groupext_distinct_pruning

gp_workfile_compress_algorithm

Join Operator Configuration Parameters

join_collapse_limit

gp_adjust_selectivity_for_outerjoins

gp_hashjoin_tuples_per_bucket

gp_statistics_use_fkeys

gp_workfile_compress_algorithm

Other Legacy Query Optimizer Configuration Parameters

from_collapse_limit

gp_enable_predicate_propagation

gp_max_plan_size

gp_statistics_pullup_from_child_partition

Error Reporting and Logging Parameters

These configuration parameters control Greenplum Database logging.

Log Rotation

log_rotation_age

log_rotation_size

log_truncate_on_rotation

When to Log

client_min_messages
log_error_verbosity
log_min_duration_statement
log_min_error_statement
log_min_messages
optimizer_minidump

What to Log

debug_pretty_print
debug_print_parse
debug_print_plan
debug_print_prelim_plan
debug_print_rewritten
debug_print_slice_table
log_autostats
log_connections
log_disconnections
log_dispatch_stats
log_duration
log_executor_stats
log_hostname
log_parser_stats
log_planner_stats
log_statement
log_statement_stats
log_timezone
gp_debug_linger
gp_log_format
gp_max_csv_line_length
gp_reraise_signal

System Monitoring Parameters

These configuration parameters control Greenplum Database data collection and notifications related to database monitoring.

SNMP Alerts

The following parameters send SNMP notifications when events occur.

gp_snmp_community
gp_snmp_monitor_address
gp_snmp_use_inform_or_trap

Email Alerts

The following parameters configure the system to send email alerts for fatal error events, such as a segment going down or a server crash and reset.

gp_email_from
gp_email_smtp_password
gp_email_smtp_server
gp_email_smtp_userid
gp_email_to

Greenplum Command Center Agent

The following parameters configure the data collection agents for Greenplum Command Center.

gp_enable_gpperfmon
gpperfmon_log_alert_level

gp_gpperfmon_send_interval

gpperfmon_port

Runtime Statistics Collection Parameters

These parameters control the server statistics collection feature. When statistics collection is enabled, you can access the statistics data using the *pg_stat* and *pg_statio* family of system catalog views.

stats_queue_level

track_activities

track_counts

update_process_title

Automatic Statistics Collection Parameters

When automatic statistics collection is enabled, you can run `ANALYZE` automatically in the same transaction as an `INSERT`, `UPDATE`, `DELETE`, `COPY` or `CREATE TABLE...AS SELECT` statement when a certain threshold of rows is affected (`on_change`), or when a newly generated table has no statistics (`on_no_stats`). To enable this feature, set the following server configuration parameters in your Greenplum master `postgresql.conf` file and restart Greenplum Database:

gp_autostats_mode

gp_autostats_mode_in_functions

gp_autostats_on_change_threshold

log_autostats

> **Warning:** Depending on the specific nature of your database operations, automatic statistics collection can have a negative performance impact. Carefully evaluate whether the default setting of `on_no_stats` is appropriate for your system.

Client Connection Default Parameters

These configuration parameters set defaults that are used for client connections.

Statement Behavior Parameters

check_function_bodies

default_tablespace

default_transaction_isolation

default_transaction_read_only

search_path

statement_timeout

vacuum_freeze_min_age

Locale and Formatting Parameters

client_encoding

DateStyle

extra_float_digits

IntervalStyle

lc_collate

lc_ctype

lc_messages

lc_monetary

lc_numeric

lc_time

TimeZone

Other Client Default Parameters

dynamic_library_path
explain_pretty_print
local_preload_libraries

Lock Management Parameters

These configuration parameters set limits for locks and deadlocks.

deadlock_timeout
max_locks_per_transaction

Workload Management Parameters

The following configuration parameters configure the Greenplum Database workload management feature (resource queues), query prioritization, memory utilization and concurrency control.

gp_resqueue_priority
gp_resqueue_priority_cpucores_per_segment
gp_resqueue_priority_sweeper_interval
gp_vmem_idle_resource_timeout
gp_vmem_protect_limit
gp_vmem_protect_segworker_cache_limit
max_resource_queues
max_resource_portals_per_transaction
resource_cleanup_gangs_on_wait
resource_select_only
runaway_detector_activation_percent
stats_queue_level

External Table Parameters

The following parameters configure the external tables feature of Greenplum Database.

gp_external_enable_exec
gp_external_grant_privileges
gp_external_max_segs
gp_initial_bad_row_limit
gp_reject_percent_threshold
readable_external_table_timeout
writable_external_table_bufsize

Database Table Parameters

The following parameter configures default option settings for Greenplum Database tables.

gp_create_table_random_default_distribution
gp_default_storage_options
gp_enable_exchange_default_partition

Append-Optimized Table Parameters

The following parameters configure the append-optimized tables feature of Greenplum Database.

max_appendonly_tables

gp_appendonly_compaction

gp_appendonly_compaction_threshhold

Database and Tablespace/Filespace Parameters

The following parameters configure the maximum number of databases, tablespaces, and filespaces allowed in a system.

gp_max_tablespaces

gp_max_filespaces

gp_max_databases

Past PostgreSQL Version Compatibility Parameters

The following parameters provide compatibility with older PostgreSQL versions. You do not need to change these parameters in Greenplum Database.

add_missing_from

array_nulls

backslash_quote

escape_string_warning

regex_flavor

standard_conforming_strings

transform_null_equals

Greenplum Array Configuration Parameters

The parameters in this topic control the configuration of the Greenplum Database array and its components: segments, master, distributed transaction manager, master mirror, and interconnect.

Interconnect Configuration Parameters

gp_interconnect_fc_method

gp_interconnect_hash_multiplier

gp_interconnect_queue_depth

gp_interconnect_snd_queue_depth

gp_interconnect_setup_timeout

gp_interconnect_type

gp_max_packet_size

> **Note:** The Greenplum Database interconnect types TCP and UDP are deprecated. In the next major release, only the UDPIFC interconnect type will be supported by Greenplum Database. The server configuration parameter `gp_interconnect_type` controls the interconnect type.

Dispatch Configuration Parameters

gp_cached_segworkers_threshold

gp_connections_per_thread

gp_enable_direct_dispatch

gp_segment_connect_timeout

gp_set_proc_affinity

Fault Operation Parameters

gp_set_read_only

gp_fts_probe_timeout

gp_fts_probe_interval

gp_fts_probe_retries

gp_fts_probe_threadcount

gp_log_fts

Distributed Transaction Management Parameters

gp_max_local_distributed_cache

Read-Only Parameters

gp_command_count

gp_content

gp_dbid

gp_num_contents_in_cluster

gp_role

gp_session_id

Greenplum Mirroring Parameters for Master and Segments

These parameters control the configuration of the replication between Greenplum Database primary master and standby master.

keep_wal_segments

repl_catchup_within_range

replication_timeout

wal_receiver_status_interval

This parameter controls validation between Greenplum Database primary segment and standby segment during incremental resynchronization.

filerep_mirrorvalidation_during_resync

Greenplum Database Extension Parameters

The parameters in this topic control the configuration of Greenplum Database extensions.

pgcrypto.fips

pljava_classpath

pljava_statement_cache_size

pljava_release_lingering_savepoints

pljava_vmoptions

Configuration Parameters

Descriptions of the Greenplum Database server configuration parameters listed alphabetically.

add_missing_from

Automatically adds missing table references to FROM clauses. Present for compatibility with releases of PostgreSQL prior to 8.1, where this behavior was allowed by default.

Value Range	Default	Set Classifications
Boolean	off	master session reload

application_name

Sets the application name for a client session. For example, if connecting via `psql`, this will be set to `psql`. Setting an application name allows it to be reported in log messages and statistics views.

Value Range	Default	Set Classifications
string		master session reload

array_nulls

This controls whether the array input parser recognizes unquoted NULL as specifying a null array element. By default, this is on, allowing array values containing null values to be entered. Greenplum Database versions before 3.0 did not support null values in arrays, and therefore would treat NULL as specifying a normal array element with the string value 'NULL'.

Value Range	Default	Set Classifications
Boolean	on	master session reload

authentication_timeout

Maximum time to complete client authentication. This prevents hung clients from occupying a connection indefinitely.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	1min	local system restart

backslash_quote

This controls whether a quote mark can be represented by \' in a string literal. The preferred, SQL-standard way to represent a quote mark is by doubling it ('') but PostgreSQL has historically also accepted \'. However, use of \' creates security risks because in some client character set encodings, there are multibyte characters in which the last byte is numerically equivalent to ASCII \.

Value Range	Default	Set Classifications
on (allow \' always) off (reject always) safe_encoding (allow only if client encoding does not allow ASCII \ within a multibyte character)	safe_encoding	master session reload

block_size

Reports the size of a disk block.

Value Range	Default	Set Classifications
number of bytes	32768	read only

bonjour_name

Specifies the Bonjour broadcast name. By default, the computer name is used, specified as an empty string. This option is ignored if the server was not compiled with Bonjour support.

Value Range	Default	Set Classifications
string	unset	master system restart

check_function_bodies

When set to off, disables validation of the function body string during `CREATE FUNCTION`. Disabling validation is occasionally useful to avoid problems such as forward references when restoring function definitions from a dump.

Value Range	Default	Set Classifications
Boolean	on	master session reload

client_encoding

Sets the client-side encoding (character set). The default is to use the same as the database encoding. See *Supported Character Sets* in the PostgreSQL documentation.

Value Range	Default	Set Classifications
character set	UTF8	master session reload

client_min_messages

Controls which message levels are sent to the client. Each level includes all the levels that follow it. The later the level, the fewer messages are sent.

Value Range	Default	Set Classifications
DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1 LOG NOTICE WARNING ERROR FATAL PANIC	NOTICE	master session reload

cpu_index_tuple_cost

For the legacy query optimizer (planner), sets the estimate of the cost of processing each index row during an index scan. This is measured as a fraction of the cost of a sequential page fetch.

Value Range	Default	Set Classifications
floating point	0.005	master session reload

cpu_operator_cost

For the legacy query optimizer (planner), sets the estimate of the cost of processing each operator in a WHERE clause. This is measured as a fraction of the cost of a sequential page fetch.

Value Range	Default	Set Classifications
floating point	0.0025	master session reload

cpu_tuple_cost

For the legacy query optimizer (planner), Sets the estimate of the cost of processing each row during a query. This is measured as a fraction of the cost of a sequential page fetch.

Value Range	Default	Set Classifications
floating point	0.01	master session reload

cursor_tuple_fraction

Tells the legacy query optimizer (planner) how many rows are expected to be fetched in a cursor query, thereby allowing the legacy optimizer to use this information to optimize the query plan. The default of 1 means all rows will be fetched.

Value Range	Default	Set Classifications
integer	1	master session reload

custom_variable_classes

Specifies one or several class names to be used for custom variables. A custom variable is a variable not normally known to the server but used by some add-on modules. Such variables must have names consisting of a class name, a dot, and a variable name.

Value Range	Default	Set Classifications
comma-separated list of class names	unset	local system restart

DateStyle

Sets the display format for date and time values, as well as the rules for interpreting ambiguous date input values. This variable contains two independent components: the output format specification and the input/output specification for year/month/day ordering.

Value Range	Default	Set Classifications
<format>, <date style> where: <format> is ISO, Postgres, SQL, or German <date style> is DMY, MDY, or YMD	ISO, MDY	master session reload

db_user_namespace

This enables per-database user names. If on, you should create users as *username@dbname*. To create ordinary global users, simply append @ when specifying the user name in the client.

Value Range	Default	Set Classifications
Boolean	off	local system restart

deadlock_timeout

The time to wait on a lock before checking to see if there is a deadlock condition. On a heavily loaded server you might want to raise this value. Ideally the setting should exceed your typical transaction time, so as to improve the odds that a lock will be released before the waiter decides to check for deadlock.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	1s	local system restart

debug_assertions

Turns on various assertion checks.

Value Range	Default	Set Classifications
Boolean	off	local system restart

debug_pretty_print

Indents debug output to produce a more readable but much longer output format. *client_min_messages* or *log_min_messages* must be DEBUG1 or lower.

Value Range	Default	Set Classifications
Boolean	off	master session reload

debug_print_parse

For each executed query, prints the resulting parse tree. *client_min_messages* or *log_min_messages* must be DEBUG1 or lower.

Value Range	Default	Set Classifications
Boolean	off	master session reload

debug_print_plan

For each executed query, prints the Greenplum parallel query execution plan. *client_min_messages* or *log_min_messages* must be DEBUG1 or lower.

Value Range	Default	Set Classifications
Boolean	off	master session reload

debug_print_prelim_plan

For each executed query, prints the preliminary query plan. *client_min_messages* or *log_min_messages* must be DEBUG1 or lower.

Value Range	Default	Set Classifications
Boolean	off	master session reload

debug_print_rewritten

For each executed query, prints the query rewriter output. *client_min_messages* or *log_min_messages* must be DEBUG1 or lower.

Value Range	Default	Set Classifications
Boolean	off	master session reload

debug_print_slice_table

For each executed query, prints the Greenplum query slice plan. *client_min_messages* or *log_min_messages* must be DEBUG1 or lower.

Value Range	Default	Set Classifications
Boolean	off	master session reload

default_statistics_target

Sets the default statistics target for table columns that have not had a column-specific target set via `ALTER TABLE SET STATISTICS`. Larger values increase the time needed to do `ANALYZE`, but may improve the quality of the legacy query optimizer (planner) estimates.

Value Range	Default	Set Classifications
integer > 0	25	master session reload

default_tablespace

The default tablespace in which to create objects (tables and indexes) when a `CREATE` command does not explicitly specify a tablespace.

Value Range	Default	Set Classifications
name of a tablespace	unset	master session reload

default_transaction_isolation

Controls the default isolation level of each new transaction.

Value Range	Default	Set Classifications
read committed read uncommitted serializable	read committed	master session reload

default_transaction_read_only

Controls the default read-only status of each new transaction. A read-only SQL transaction cannot alter non-temporary tables.

Value Range	Default	Set Classifications
Boolean	off	master session reload

dynamic_library_path

If a dynamically loadable module needs to be opened and the file name specified in the `CREATE FUNCTION` or `LOAD` command does not have a directory component (i.e. the name does not contain a slash), the system will search this path for the required file. The compiled-in PostgreSQL package library directory is substituted for $libdir. This is where the modules provided by the standard PostgreSQL distribution are installed.

Value Range	Default	Set Classifications
a list of absolute directory paths separated by colons	$libdir	local system restart

effective_cache_size

Sets the assumption about the effective size of the disk cache that is available to a single query for the legacy query optimizer (planner). This is factored into estimates of the cost of using an index; a higher value makes it more likely index scans will be used, a lower value makes it more likely sequential scans will be used. This parameter has no effect on the size of shared memory allocated by a Greenplum server instance, nor does it reserve kernel disk cache; it is used only for estimation purposes.

Value Range	Default	Set Classifications
floating point	512MB	master session reload

enable_bitmapscan

Enables or disables the use of bitmap-scan plan types by the legacy query optimizer (planner). Note that this is different than a Bitmap Index Scan. A Bitmap Scan means that indexes will be dynamically converted to bitmaps in memory when appropriate, giving faster index performance on complex queries against very large tables. It is used when there are multiple predicates on different indexed columns. Each bitmap per column can be compared to create a final list of selected tuples.

Value Range	Default	Set Classifications
Boolean	on	master session reload

enable_groupagg

Enables or disables the use of group aggregation plan types by the legacy query optimizer (planner).

Value Range	Default	Set Classifications
Boolean	on	master session reload

enable_hashagg

Enables or disables the use of hash aggregation plan types by the legacy query optimizer (planner).

Value Range	Default	Set Classifications
Boolean	on	master session reload

enable_hashjoin

Enables or disables the use of hash-join plan types by the legacy query optimizer (planner).

Value Range	Default	Set Classifications
Boolean	on	master session reload

enable_indexscan

Enables or disables the use of index-scan plan types by the legacy query optimizer (planner).

Value Range	Default	Set Classifications
Boolean	on	master session reload

enable_mergejoin

Enables or disables the use of merge-join plan types by the legacy query optimizer (planner). Merge join is based on the idea of sorting the left- and right-hand tables into order and then scanning them in parallel. So, both data types must be capable of being fully ordered, and the join operator must be one that can only succeed for pairs of values that fall at the 'same place' in the sort order. In practice this means that the join operator must behave like equality.

Value Range	Default	Set Classifications
Boolean	off	master session reload

enable_nestloop

Enables or disables the use of nested-loop join plans by the legacy query optimizer (planner). It's not possible to suppress nested-loop joins entirely, but turning this variable off discourages the legacy optimizer from using one if there are other methods available.

Value Range	Default	Set Classifications
Boolean	off	master session reload

enable_seqscan

Enables or disables the use of sequential scan plan types by the legacy query optimizer (planner). It's not possible to suppress sequential scans entirely, but turning this variable off discourages the legacy optimizer from using one if there are other methods available.

Value Range	Default	Set Classifications
Boolean	on	master session reload

enable_sort

Enables or disables the use of explicit sort steps by the legacy query optimizer (planner). It's not possible to suppress explicit sorts entirely, but turning this variable off discourages the legacy optimizer from using one if there are other methods available.

Value Range	Default	Set Classifications
Boolean	on	master session reload

enable_tidscan

Enables or disables the use of tuple identifier (TID) scan plan types by the legacy query optimizer (planner).

Value Range	Default	Set Classifications
Boolean	on	master session reload

escape_string_warning

When on, a warning is issued if a backslash (\) appears in an ordinary string literal ('...' syntax). Escape string syntax (E'...') should be used for escapes, because in future versions, ordinary strings will have the SQL standard-conforming behavior of treating backslashes literally.

Value Range	Default	Set Classifications
Boolean	on	master session reload

explain_pretty_print

Determines whether EXPLAIN VERBOSE uses the indented or non-indented format for displaying detailed query-tree dumps.

Value Range	Default	Set Classifications
Boolean	on	master session reload

extra_float_digits

Adjusts the number of digits displayed for floating-point values, including float4, float8, and geometric data types. The parameter value is added to the standard number of digits. The value can be set as high as 2, to include partially-significant digits; this is especially useful for dumping float data that needs to be restored exactly. Or it can be set negative to suppress unwanted digits.

Value Range	Default	Set Classifications
integer	0	master session reload

filerep_mirrorvalidation_during_resync

The default setting `false` improves Greenplum Database performance during incremental resynchronization of segment mirrors. Setting the value to `true` enables checking for the existence of files for all relations on the segment mirror during incremental resynchronization. Checking for files degrades performance of the incremental resynchronization process but provides a minimal check of database objects.

Value Range	Default	Set Classifications
Boolean	false	master session reload

from_collapse_limit

The legacy query optimizer (planner) will merge sub-queries into upper queries if the resulting FROM list would have no more than this many items. Smaller values reduce planning time but may yield inferior query plans.

Value Range	Default	Set Classifications
1-*n*	20	master session reload

gp_adjust_selectivity_for_outerjoins

Enables the selectivity of NULL tests over outer joins.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_analyze_relative_error

Sets the estimated acceptable error in the cardinality of the table " a value of 0.5 is supposed to be equivalent to an acceptable error of 50% (this is the default value used in PostgreSQL). If the statistics collected during `ANALYZE` are not producing good estimates of cardinality for a particular table attribute, decreasing the relative error fraction (accepting less error) tells the system to sample more rows.

Value Range	Default	Set Classifications
floating point < 1.0	0.25	master session reload

gp_appendonly_compaction

Enables compacting segment files during `VACUUM` commands. When disabled, `VACUUM` only truncates the segment files to the EOF value, as is the current behavior. The administrator may want to disable compaction in high I/O load situations or low space situations.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_appendonly_compaction_threshhold

Specifies the threshold ratio (as a percentage) of hidden rows to total rows that triggers compaction of the segment file when `VACUUM` is run without the `FULL` option (a lazy vacuum). If the ratio of hidden rows in a segment file on a segment is less than this threshold, the segment file is not compacted, and a log message is issued.

Value Range	Default	Set Classifications
integer (%)	10	master session reload

gp_autostats_mode

Specifies the mode for triggering automatic statistics collection with `ANALYZE`. The `on_no_stats` option triggers statistics collection for `CREATE TABLE AS SELECT`, `INSERT`, or `COPY` operations on any table that has no existing statistics.

The `on_change` option triggers statistics collection only when the number of rows affected exceeds the threshold defined by `gp_autostats_on_change_threshold`. Operations that can trigger automatic statistics collection with `on_change` are:

`CREATE TABLE AS SELECT`

`UPDATE`

`DELETE`

`INSERT`

COPY

Default is on_no_stats.

> **Note:** For partitioned tables, automatic statistics collection is not triggered if data is inserted from the top-level parent table of a partitioned table.
>
> Automatic statistics collection is triggered if data is inserted directly in a leaf table (where the data is stored) of the partitioned table. Statistics are collected only on the leaf table.

Value Range	Default	Set Classifications
none on_change on_no_stats	on_no_ stats	master session reload

gp_autostats_mode_in_functions

Specifies the mode for triggering automatic statistics collection with ANALYZE for statements in procedural language functions. The none option disables statistics collection. The on_no_stats option triggers statistics collection for CREATE TABLE AS SELECT, INSERT, or COPY operations that are executed in functions on any table that has no existing statistics.

The on_change option triggers statistics collection only when the number of rows affected exceeds the threshold defined by gp_autostats_on_change_threshold. Operations in functions that can trigger automatic statistics collection with on_change are:

CREATE TABLE AS SELECT

UPDATE

DELETE

INSERT

COPY

Value Range	Default	Set Classifications
none on_change on_no_stats	none	master session reload

gp_autostats_on_change_threshold

Specifies the threshold for automatic statistics collection when gp_autostats_mode is set to on_change. When a triggering table operation affects a number of rows exceeding this threshold, ANALYZE is added and statistics are collected for the table.

Value Range	Default	Set Classifications
integer	2147483647	master session reload

gp_backup_directIO

Direct I/O allows Greenplum Database to bypass the buffering of memory within the file system cache for backup. When Direct I/O is used for a file, data is transferred directly from the disk to the application buffer, without the use of the file buffer cache.

Direct I/O is supported only on Red Hat Enterprise Linux, CentOS, and SUSE.

Value Range	Default	Set Classifications
on, off	off	local session reload

gp_backup_directIO_read_chunk_mb

Sets the chunk size in MB when Direct I/O is enabled with *gp_backup_directIO*. The default chunk size is 20MB.

The default value is the optimal setting. Decreasing it will increase the backup time and increasing it will result in little change to backup time.

Value Range	Default	Set Classifications
1-200	20 MB	local session reload

gp_cached_segworkers_threshold

When a user starts a session with Greenplum Database and issues a query, the system creates groups or 'gangs' of worker processes on each segment to do the work. After the work is done, the segment worker processes are destroyed except for a cached number which is set by this parameter. A lower setting conserves system resources on the segment hosts, but a higher setting may improve performance for power-users that want to issue many complex queries in a row.

Value Range	Default	Set Classifications
integer > 0	5	master session reload

gp_command_count

Shows how many commands the master has received from the client. Note that a single SQLcommand might actually involve more than one command internally, so the counter may increment by more than one for a single query. This counter also is shared by all of the segment processes working on the command.

Value Range	Default	Set Classifications
integer > 0	1	read only

gp_connectemc_mode

Controls the ConnectEMC event logging and dial-home capabilities of Greenplum Command Center on the EMC Greenplum Data Computing Appliance (DCA). ConnectEMC must be installed in order to generate events. Allowed values are:

`on` (the default) - log events to the `gpperfmon` database and send dial-home notifications to EMC Support

`off` - turns off ConnectEMC event logging and dial-home capabilities

`local` - log events to the `gpperfmon` database only

`remote` - sends dial-home notifications to Pivotal Support (does not log events to the `gpperfmon` database)

Value Range	Default	Set Classifications
on, off, local, remote	on	master system restart superuser

gp_connection_send_timeout

Timeout for sending data to unresponsive Greenplum Database user clients during query processing. A value of 0 disables the timeout, Greenplum Database waits indefinitely for a client. When the timeout is reached, the query is cancelled with this message:

```
Could not send data to client: Connection timed out.
```

Value Range	Default	Set Classifications
number of seconds	3600 (1 hour)	master system reload

gp_connections_per_thread

A value larger than or equal to the number of primary segments means that each slice in a query plan will get its own thread when dispatching to the segments. A value of 0 indicates that the dispatcher should use a single thread when dispatching all query plan slices to a segment. Lower values will use more threads, which utilizes more resources on the master. Typically, the default does not need to be changed unless there is a known throughput performance problem.

Value Range	Default	Set Classifications
integer	64	master session reload

gp_content

The local content id if a segment.

Value Range	Default	Set Classifications
integer		read only

gp_create_table_random_default_distribution

Controls table creation when a Greenplum Database table is created with a `CREATE TABLE` or `CREATE TABLE AS` command that does not contain a `DISTRIBUTED BY` clause.

For `CREATE TABLE`, if the value of the parameter is `off` (the default), and the table creation command does not contain a `DISTRIBUTED BY` clause, Greenplum Database chooses the table distribution key based on the command. If the `LIKE` or `INHERITS` clause is specified in table creation command, the created table uses the same distribution key as the source or parent table.

If the value of the parameter is set to `on`, Greenplum Database follows these rules to create a table when the `DISTRIBUTED BY` clause is not specified:

- If `PRIMARY KEY` or `UNIQUE` columns are not specified, the distribution of the table is random (`DISTRIBUTED RANDOMLY`). Table distribution is random even if the table creation command contains the `LIKE` or `INHERITS` clause.
- If `PRIMARY KEY` or `UNIQUE` columns are specified, a `DISTRIBUTED BY` clause must also be specified. If a `DISTRIBUTED BY` clause is not specified as part of the table creation command, the command fails.

For a `CREATE TABLE AS` command that does not contain a distribution clause:

- If the legacy query optimizer creates the table, and the value of the parameter is `off`, the table distribution policy is determined based on the command.
- If the legacy query optimizer creates the table, and the value of the parameter is `on`, the table distribution policy is random.
- If the Pivotal Query Optimizer creates the table, the table distribution policy is random. The parameter value has no affect.

For information about the legacy query optimizer and the Pivotal Query Optimizer, see "Querying Data" in the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
boolean	off	master system reload

gp_dbid

The local content dbid if a segment.

Value Range	Default	Set Classifications
integer		read only

gp_debug_linger

Number of seconds for a Greenplum process to linger after a fatal internal error.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	0	master session reload

gp_default_storage_options

Set the default values for the following table storage options when a table is created with the CREATE TABLE command.

- APPENDONLY
- BLOCKSIZE
- CHECKSUM
- COMPRESSTYPE
- COMPRESSLEVEL
- ORIENTATION

Specify multiple storage option values as a comma separated list.

You can set the storage options with this parameter instead of specifying the table storage options in the WITH of the CREATE TABLE command. The table storage options that are specified with the CREATE TABLE command override the values specified by this parameter.

Not all combinations of storage option values are valid. If the specified storage options are not valid, an error is returned. See the CREATE TABLE command for information about table storage options.

The defaults can be set for a database and user. If the server configuration parameter is set at different levels, this the order of precedence, from highest to lowest, of the table storage values when a user logs into a database and creates a table:

1. The values specified in a CREATE TABLE command with the WITH clause or ENCODING clause
2. The value of gp_default_storage_options that set for the user with the ALTER ROLE...SET command
3. The value of gp_default_storage_options that is set for the database with the ALTER DATABASE...SET command
4. The value of gp_default_storage_options that is set for the Greenplum Database system with the gpconfig utility

The parameter value is not cumulative. For example, if the parameter specifies the APPENDONLY and COMPRESSTYPE options for a database and a user logs in and sets the parameter to specify the value for the ORIENTATION option, the APPENDONLY, and COMPRESSTYPE values set at the database level are ignored.

This example ALTER DATABASE command sets the default ORIENTATION and COMPRESSTYPE table storage options for the database mystest.

```
ALTER DATABASE mytest SET gp_default_storage_options = 'orientation=column,
 compresstype=rle_type'
```

To create an append-optimized table in the mytest database with column-oriented table and RLE compression. The user needs to specify only APPENDONLY=TRUE in the WITH clause.

This example gpconfig utility command sets the default storage option for a Greenplum Database system. If you set the defaults for multiple table storage options, the value must be enclosed in single quotes and then in double quotes.

```
gpconfig -c 'gp_default_storage_options' -v "'appendonly=true, orientation=column'"
```

This example `gpconfig` utility command shows the value of the parameter. The parameter value must be consistent across the Greenplum Database master and all segments.

```
gpconfig -s 'gp_default_storage_options'
```

Value Range	Default	Set Classifications [1]

`APPENDONLY`= `TRUE` | `FALSE`
`BLOCKSIZE`= integer between 8192 and 2097152
`CHECKSUM`= `TRUE` | `FALSE`
`COMPRESSTYPE`= `ZLIB` | `QUICKLZ` | `RLE_TYPE` | `NONE`
`COMPRESSLEVEL`= integer between 0 and 9
`ORIENTATION`= `ROW` | `COLUMN`
	`APPENDONLY=FALSE`
	`BLOCKSIZE=32768`
	`CHECKSUM=TRUE`
	`COMPRESSTYPE=none`
	`COMPRESSLEVEL=0`
	`ORIENTATION=ROW`
		master
		session
		reload

Note: [1]The set classification when the parameter is set at the system level with the `gpconfig` utility.

gp_dynamic_partition_pruning

Enables plans that can dynamically eliminate the scanning of partitions.

Value Range	Default	Set Classifications
on/off	on	master session reload

gp_email_from

The email address used to send email alerts, in the format of:

`'username@domain.com'`

or

`'Name <username@domain.com>'`

Value Range	Default	Set Classifications
string		master system reload superuser

gp_email_smtp_password

The password/passphrase used to authenticate with the SMTP server.

Value Range	Default	Set Classifications
string		master system reload superuser

gp_email_smtp_server

The fully qualified domain name or IP address and port of the SMTP server to use to send the email alerts. Must be in the format of:

smtp_servername.domain.com:port

Value Range	Default	Set Classifications
string		master system reload superuser

gp_email_smtp_userid

The user id used to authenticate with the SMTP server.

Value Range	Default	Set Classifications
string		master system reload superuser

gp_email_to

A semi-colon (`;`) separated list of email addresses to receive email alert messages to in the format of: `'username@domain.com'`

or

`'Name <username@domain.com>'`

If this parameter is not set, then email alerts are disabled.

Value Range	Default	Set Classifications
string		master system reload superuser

gp_enable_adaptive_nestloop

Enables the use a new type of join node called "Adaptive Nestloop" at query execution time by the legacy query optimizer (planner). This causes the legacy optimizer to favor a hash-join over a nested-loop join if the number of rows on the outer side of the join exceeds a precalculated threshold. This parameter improves performance of index operations, which previously favored slower nested-loop joins.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_agg_distinct

Enables or disables two-phase aggregation to compute a single distinct-qualified aggregate. This applies only to subqueries that include a single distinct-qualified aggregate function.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_agg_distinct_pruning

Enables or disables three-phase aggregation and join to compute distinct-qualified aggregates. This applies only to subqueries that include one or more distinct-qualified aggregate functions.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_direct_dispatch

Enables or disables the dispatching of targeted query plans for queries that access data on a single segment. When on, queries that target rows on a single segment will only have their query plan dispatched to that segment (rather than to all segments). This significantly reduces the response time of qualifying queries as there is no interconnect setup involved. Direct dispatch does require more CPU utilization on the master.

Value Range	Default	Set Classifications
Boolean	on	master system restart

gp_enable_exchange_default_partition

Controls availability of the `EXCHANGE DEFAULT PARTITION` clause for `ALTER TABLE`. The default value for the parameter is `off`. The clause is not available and Greenplum Database returns an error if the clause is specified in an `ALTER TABLE` command.

If the value is `on`, Greenplum Database returns a warning stating that exchanging the default partition might result in incorrect results due to invalid data in the default partition.

> **Warning:** Before you exchange the default partition, you must ensure the data in the table to be exchanged, the new default partition, is valid for the default partition. For example, the data in the new default partition must not contain data that would be valid in other leaf child partitions of the partitioned table. Otherwise, queries against the partitioned table with the exchanged default partition that are executed by the Pivotal Query Optimizer might return incorrect results.

Value Range	Default	Set Classifications
Boolean	off	master session reload

gp_enable_fallback_plan

Allows use of disabled plan types when a query would not be feasible without them.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_fast_sri

When set to `on`, the legacy query optimizer (planner) plans single row inserts so that they are sent directly to the correct segment instance (no motion operation required). This significantly improves performance of single-row-insert statements.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_gpperfmon

Enables or disables the data collection agents of Greenplum Command Center.

Value Range	Default	Set Classifications
Boolean	off	local system restart

gp_enable_groupext_distinct_gather

Enables or disables gathering data to a single node to compute distinct-qualified aggregates on grouping extension queries. When this parameter and `gp_enable_groupext_distinct_pruning` are both enabled, the legacy query optimizer (planner) uses the cheaper plan.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_groupext_distinct_pruning

Enables or disables three-phase aggregation and join to compute distinct-qualified aggregates on grouping extension queries. Usually, enabling this parameter generates a cheaper query plan that the legacy query optimizer (planner) will use in preference to existing plan.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_multiphase_agg

Enables or disables the use of two or three-stage parallel aggregation plans legacy query optimizer (planner). This approach applies to any subquery with aggregation. If `gp_enable_multiphase_agg` is off, then `gp_enable_agg_distinct` and `gp_enable_agg_distinct_pruning` are disabled.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_predicate_propagation

When enabled, the legacy query optimizer (planner) applies query predicates to both table expressions in cases where the tables are joined on their distribution key column(s). Filtering both tables prior to doing the join (when possible) is more efficient.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_preunique

Enables two-phase duplicate removal for `SELECT DISTINCT` queries (not `SELECT COUNT(DISTINCT)`). When enabled, it adds an extra `SORT DISTINCT` set of plan nodes before motioning. In cases where the distinct operation greatly reduces the number of rows, this extra `SORT DISTINCT` is much cheaper than the cost of sending the rows across the Interconnect.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_sequential_window_plans

If on, enables non-parallel (sequential) query plans for queries containing window function calls. If off, evaluates compatible window functions in parallel and rejoins the results. This is an experimental parameter.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_sort_distinct

Enable duplicates to be removed while sorting.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_enable_sort_limit

Enable `LIMIT` operation to be performed while sorting. Sorts more efficiently when the plan requires the first *limit_number* of rows at most.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_external_enable_exec

Enables or disables the use of external tables that execute OS commands or scripts on the segment hosts (`CREATE EXTERNAL TABLE EXECUTE` syntax). Must be enabled if using the Command Center or MapReduce features.

Value Range	Default	Set Classifications
Boolean	on	master system restart

gp_external_grant_privileges

In releases prior to 4.0, enables or disables non-superusers to issue a `CREATE EXTERNAL [WEB] TABLE` command in cases where the `LOCATION` clause specifies `http` or `gpfdist`. In releases after 4.0, the ability to create an external table can be granted to a role using `CREATE ROLE` or `ALTER ROLE`.

Value Range	Default	Set Classifications
Boolean	off	master system restart

gp_external_max_segs

Sets the number of segments that will scan external table data during an external table operation, the purpose being not to overload the system with scanning data and take away resources from other concurrent operations. This only applies to external tables that use the `gpfdist://` protocol to access external table data.

Value Range	Default	Set Classifications
integer	64	master session reload

gp_filerep_tcp_keepalives_count

How many keepalives may be lost before the connection is considered dead. A value of 0 uses the system default. If TCP_KEEPCNT is not supported, this parameter must be 0.

Use this parameter for all connections that are between a primary and mirror segment. Use `tcp_keepalives_count` for settings that are not between a primary and mirror segment.

Value Range	Default	Set Classifications
number of lost keepalives	2	local system restart

gp_filerep_tcp_keepalives_idle

Number of seconds between sending keepalives on an otherwise idle connection. A value of 0 uses the system default. If TCP_KEEPIDLE is not supported, this parameter must be 0.

Use this parameter for all connections that are between a primary and mirror segment. Use `tcp_keepalives_idle` for settings that are not between a primary and mirror segment.

Value Range	Default	Set Classifications
number of seconds	1 min	local system restart

gp_filerep_tcp_keepalives_interval

How many seconds to wait for a response to a keepalive before retransmitting. A value of 0 uses the system default. If TCP_KEEPINTVL is not supported, this parameter must be 0.

Use this parameter for all connections that are between a primary and mirror segment. Use tcp_keepalives_interval for settings that are not between a primary and mirror segment.

Value Range	Default	Set Classifications
number of seconds	30 sec	local system restart

gp_fts_probe_interval

Specifies the polling interval for the fault detection process (`ftsprobe`). The `ftsprobe` process will take approximately this amount of time to detect a segment failure.

Value Range	Default	Set Classifications
10 seconds or greater	1min	master system restart

gp_fts_probe_retries

Specifies the number of times the fault detection process (`ftsprobe`) attempts to connect to a segment before reporting segment failure.

Value Range	Default	Set Classifications
integer	5	master system restart

gp_fts_probe_threadcount

Specifies the number of `ftsprobe` threads to create. This parameter should be set to a value equal to or greater than the number of segments per host.

Value Range	Default	Set Classifications
1 - 128	16	master system restart

gp_fts_probe_timeout

Specifies the allowed timeout for the fault detection process (`ftsprobe`) to establish a connection to a segment before declaring it down.

Value Range	Default	Set Classifications
10 seconds or greater	20 secs	master system restart

gp_log_fts

Controls the amount of detail the fault detection process (`ftsprobe`) writes to the log file.

Value Range	Default	Set Classifications
OFF TERSE VERBOSE DEBUG	TERSE	master system restart

gp_gpperfmon_send_interval

Sets the frequency that the Greenplum Database server processes send query execution updates to the data collection agent processes used by Command Center. Query operations (iterators) executed during this interval are sent through UDP to the segment monitor agents. If you find that an excessive number of UDP packets are dropped during long-running, complex queries, you may consider increasing this value.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	1sec	master system restart

gpperfmon_log_alert_level

Controls which message levels are written to the gpperfmon log. Each level includes all the levels that follow it. The later the level, the fewer messages are sent to the log.

> **Note:** If the Greenplum Database Command Center is installed and is monitoring the database, the default value is warning.

Value Range	Default	Set Classifications
none warning error fatal panic	none	local system restart

gp_hadoop_home

When using Pivotal HD, specify the installation directory for Hadoop. For example, the default installation directory is `/usr/lib/gphd`.

When using Greenplum HD 1.2 or earlier, specify the same value as the HADOOP_HOME environment variable.

Value Range	Default	Set Classifications
Valid directory name	Value of `HADOOP_HOME`	local session reload

gp_hadoop_target_version

The installed version of Greenplum Hadoop target.

Value Range	Default	Set Classifications
gphd-1.0 gphd-1.1 gphd-1.2 gphd-2.0 gphd-3.0 gpmr-1.0 gpmr-1.2 hadoop2 hdp2 cdh5 cdh3u2 cdh4.1	gphd-1.1	local session reload

gp_hashjoin_tuples_per_bucket

Sets the target density of the hash table used by HashJoin operations. A smaller value will tend to produce larger hash tables, which can increase join performance.

Value Range	Default	Set Classifications
integer	5	master session reload

gp_idf_deduplicate

Changes the strategy to compute and process MEDIAN, and PERCENTILE_DISC.

Value Range	Default	Set Classifications
auto none force	auto	master session reload

gp_initial_bad_row_limit

For the parameter value *n*, Greenplum Database stops processing input rows when you import data with the `COPY` command or from an external table if the first *n* rows processed contain formatting errors. If a valid row is processed within the first *n* rows, Greenplum Database continues processing input rows.

Setting the value to 0 disables this limit.

The `SEGMENT REJECT LIMIT` clause can also be specified for the `COPY` command or the external table definition to limit the number of rejected rows.

`INT_MAX` is the largest value that can be stored as an integer on your system.

Value Range	Default	Set Classifications
integer 0 - `INT_MAX`	1000	master session reload

gp_interconnect_fc_method

Specifies the flow control method used for UDP interconnect when the value of *gp_interconnect_type* is UDPIFC.

For capacity based flow control, senders do not send packets when receivers do not have the capacity.

Loss based flow control is based on capacity based flow control, and also tunes the sending speed according to packet losses.

Value Range	Default	Set Classifications
CAPACITY LOSS	LOSS	master session reload

gp_interconnect_hash_multiplier

Sets the size of the hash table used by the UDP interconnect to track connections. This number is multiplied by the number of segments to determine the number of buckets in the hash table. Increasing the value may increase interconnect performance for complex multi-slice queries (while consuming slightly more memory on the segment hosts).

Value Range	Default	Set Classifications
2-25	2	master session reload

gp_interconnect_queue_depth

Sets the amount of data per-peer to be queued by the UDP interconnect on receivers (when data is received but no space is available to receive it the data will be dropped, and the transmitter will need to resend it). Increasing the depth from its default value will cause the system to use more memory; but may increase performance. It is reasonable for this to be set between 1 and 10. Queries with data skew potentially perform better when this is increased. Increasing this may radically increase the amount of memory used by the system.

Value Range	Default	Set Classifications
1-2048	4	master session reload

gp_interconnect_setup_timeout

When the interconnect type is UDP, the time to wait for the Interconnect to complete setup before it times out.

This parameter is used only when *gp_interconnect_type* is set to UDP.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	2 hours	master session reload

gp_interconnect_snd_queue_depth

Sets the amount of data per-peer to be queued by the UDP interconnect on senders. Increasing the depth from its default value will cause the system to use more memory; but may increase performance. Reasonable values for this parameter are between 1 and 4. Increasing the value might radically increase the amount of memory used by the system.

This parameter is used only when *gp_interconnect_type* is set to UDPIFC.

Value Range	Default	Set Classifications
1 - 4096	2	master session reload

gp_interconnect_type

Sets the networking protocol used for Interconnect traffic. With the TCP protocol, Greenplum Database has an upper limit of 1000 segment instances - less than that if the query workload involves complex, multi-slice queries.

UDP allows for greater interconnect scalability. Note that the Greenplum software does the additional packet verification and checking not performed by UDP, so reliability and performance is equivalent to TCP.

UDPIFC specifies using UDP with flow control for interconnect traffic. Specify the interconnect flow control method with *gp_interconnect_fc_method*.

Note: The Greenplum Database interconnect types TCP and UDP are deprecated. In the next major release, only the UDPIFC interconnect type will be supported by Greenplum Database.

Value Range	Default	Set Classifications
TCP UDP UDPIFC	UDPIFC	local system restart

gp_log_format

Specifies the format of the server log files. If using *gp_toolkit* administrative schema, the log files must be in CSV format.

Value Range	Default	Set Classifications
csv text	csv	local system restart

gp_max_csv_line_length

The maximum length of a line in a CSV formatted file that will be imported into the system. The default is 1MB (1048576 bytes). Maximum allowed is 4MB (4194184 bytes). The default may need to be increased if using the *gp_toolkit* administrative schema to read Greenplum Database log files.

Value Range	Default	Set Classifications
number of bytes	1048576	local system restart

gp_max_databases

The maximum number of databases allowed in a Greenplum Database system.

Value Range	Default	Set Classifications
integer	16	master system restart

gp_max_filespaces

The maximum number of filespaces allowed in a Greenplum Database system.

Value Range	Default	Set Classifications
integer	8	master system restart

gp_max_local_distributed_cache

Sets the number of local to distributed transactions to cache. Higher settings may improve performance.

Value Range	Default	Set Classifications
integer	1024	local system restart

gp_max_packet_size

Sets the size (in bytes) of messages sent by the UDP interconnect, and sets the tuple-serialization chunk size for both the UDP and TCP interconnect.

Value Range	Default	Set Classifications
512-65536	8192	master system restart

gp_max_plan_size

Specifies the total maximum uncompressed size of a query execution plan multiplied by the number of Motion operators (slices) in the plan. If the size of the query plan exceeds the value, the query is cancelled and an error is returned. A value of 0 means that the size of the plan is not monitored.

You can specify a value in KB,MB, or GB. The default unit is KB. For example, a value of 200 is 200KB. A value of 1GB is the same as 1024MB or 1048576KB.

Value Range	Default	Set Classifications
integer	0	master superuser session

gp_max_tablespaces

The maximum number of tablespaces allowed in a Greenplum Database system.

Value Range	Default	Set Classifications
integer	16	master system restart

gp_motion_cost_per_row

Sets the legacy query optimizer (planner) cost estimate for a Motion operator to transfer a row from one segment to another, measured as a fraction of the cost of a sequential page fetch. If 0, then the value used is two times the value of *cpu_tuple_cost*.

Value Range	Default	Set Classifications
floating point	0	master session reload

gp_num_contents_in_cluster

The number of primary segments in the Greenplum Database system.

Value Range	Default	Set Classifications
-	-	read only

gp_reject_percent_threshold

For single row error handling on COPY and external table SELECTs, sets the number of rows processed before SEGMENT REJECT LIMIT *n* PERCENT starts calculating.

Value Range	Default	Set Classifications
1-*n*	300	master session reload

gp_reraise_signal

If enabled, will attempt to dump core if a fatal server error occurs.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_resqueue_memory_policy

Enables Greenplum memory management features. The distribution algorithm `eager_free` takes advantage of the fact that not all operators execute at the same time (in Greenplum Database 4.2 and later). The query plan is divided into stages and Greenplum Database eagerly frees memory allocated to a previous stage at the end of that stage's execution, then allocates the eagerly freed memory to the new stage.

When set to `none`, memory management is the same as in Greenplum Database releases prior to 4.1.

When set to `auto`, query memory usage is controlled by *statement_mem* and resource queue memory limits.

Value Range	Default	Set Classifications
none, auto, eager_free	eager_free	local system restart/reload

gp_resqueue_priority

Enables or disables query prioritization. When this parameter is disabled, existing priority settings are not evaluated at query run time.

Value Range	Default	Set Classifications
Boolean	on	local system restart

gp_resqueue_priority_cpucores_per_segment

Specifies the number of CPU units allocated per segment instance. For example, if a Greenplum Database cluster has 10-core segment hosts that are configured with four segments, set the value for the segment instances to 2.5. For the master instance, the value would be 10. A master host typically has only the master instance running on it, so the value for the master should reflect the usage of all available CPU cores.

Incorrect settings can result in CPU under-utilization or query prioritization not working as designed.

The default values for the Greenplum Data Computing Appliance V2 are 4 for segments and 25 for the master.

Value Range	Default	Set Classifications
0.1 - 512.0	4	local system restart

gp_resqueue_priority_sweeper_interval

Specifies the interval at which the sweeper process evaluates current CPU usage. When a new statement becomes active, its priority is evaluated and its CPU share determined when the next interval is reached.

Value Range	Default	Set Classifications
500 - 15000 ms	1000	local system restart

gp_role

The role of this server process " set to *dispatch* for the master and *execute* for a segment.

Value Range	Default	Set Classifications
dispatch execute utility		read only

gp_safefswritesize

Specifies a minimum size for safe write operations to append-optimized tables in a non-mature file system. When a number of bytes greater than zero is specified, the append-optimized writer adds padding data up to that number in order to prevent data corruption due to file system errors. Each non-mature file system has a known safe write size that must be specified here when using Greenplum Database with that type of file system. This is commonly set to a multiple of the extent size of the file system; for example, Linux ext3 is 4096 bytes, so a value of 32768 is commonly used.

Value Range	Default	Set Classifications
integer	0	local system restart

gp_segment_connect_timeout

Time that the Greenplum interconnect will try to connect to a segment instance over the network before timing out. Controls the network connection timeout between master and primary segments, and primary to mirror segment replication processes.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	10min	local system reload

gp_segments_for_planner

Sets the number of primary segment instances for the legacy query optimizer (planner) to assume in its cost and size estimates. If 0, then the value used is the actual number of primary segments. This variable affects the legacy optimizer's estimates of the number of rows handled by each sending and receiving process in Motion operators.

Value Range	Default	Set Classifications
0-*n*	0	master session reload

gp_session_id

A system assigned ID number for a client session. Starts counting from 1 when the master instance is first started.

Value Range	Default	Set Classifications
1-*n*	14	read only

gp_set_proc_affinity

If enabled, when a Greenplum server process (postmaster) is started it will bind to a CPU.

Value Range	Default	Set Classifications
Boolean	off	master system restart

gp_set_read_only

Set to on to disable writes to the database. Any in progress transactions must finish before read-only mode takes affect.

Value Range	Default	Set Classifications
Boolean	off	master system restart

gp_snmp_community

Set to the community name you specified for your environment.

Value Range	Default	Set Classifications
SNMP community name	public	master system reload

gp_snmp_monitor_address

The hostname:port of your network monitor application. Typically, the port number is 162. If there are multiple monitor addresses, separate them with a comma.

Value Range	Default	Set Classifications
hostname:port		master system reload

gp_snmp_use_inform_or_trap

Trap notifications are SNMP messages sent from one application to another (for example, between Greenplum Database and a network monitoring application). These messages are unacknowledged by the monitoring application, but generate less network overhead.

Inform notifications are the same as trap messages, except that the application sends an acknowledgement to the application that generated the alert.

Value Range	Default	Set Classifications
inform trap	trap	master system reload

gp_statistics_pullup_from_child_partition

Enables the use of statistics from child tables when planning queries on the parent table by the legacy query optimizer (planner).

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_statistics_use_fkeys

When enabled, allows the legacy query optimizer (planner) to use foreign key information stored in the system catalog to optimize joins between foreign keys and primary keys.

Value Range	Default	Set Classifications
Boolean	off	master session reload

gp_vmem_idle_resource_timeout

If a database session is idle for longer than the time specified, the session will free system resources (such as shared memory), but remain connected to the database. This allows more concurrent connections to the database at one time.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	18s	master system restart

gp_vmem_protect_limit

Sets the amount of memory (in number of MBs) that all postgres processes of an active segment instance can consume. If a query causes this limit to be exceeded, memory will not be allocated and the query will fail. Note that this is a local parameter and must be set for every segment in the system (primary and mirrors).

To prevent over-allocation of memory, these calculations can estimate a safe `gp_vmem_protect_limit` value.

First calculate the value *gp_vmem*. This is the Greenplum Database memory available on a host

```
gp_vmem = ((SWAP + RAM) – (7.5GB + 0.05 * RAM)) / 1.7
```

where *SWAP* is the host swap space and *RAM* is the RAM on the host in GB.

Next, calculate the *max_acting_primary_segments*. This is the maximum number of primary segments that can be running on a host when mirror segments are activated due to a failure. With mirrors arranged in a 4-host block with 8 primary segments per host, for example, a single segment host failure would activate two or three mirror segments on each remaining host in the failed host's block. The *max_acting_primary_segments* value for this configuration is 11 (8 primary segments plus 3 mirrors activated on failure).

This is the calculation for `gp_vmem_protect_limit`. The value should be converted to MB.

```
gp_vmem_protect_limit = gp_vmem / acting_primary_segments
```

For scenarios where a large number of workfiles are generated, this is the calculation for *gp_vmem* that accounts for the workfiles.

```
gp_vmem = ((SWAP + RAM) – (7.5GB + 0.05 * RAM - (300KB * total_#_workfiles))) / 1.7
```

For information about monitoring and managing workfile usage, see the *Greenplum Database Administrator Guide*.

Based on the *gp_vmem* value you can calculate the value for the `vm.overcommit_ratio` operating system kernel parameter. This parameter is set when you configure each Greenplum Database host.

```
vm.overcommit_ratio = (RAM - (0.026 * gp_vmem)) / RAM
```

> **Note:** The default value for the kernel parameter `vm.overcommit_ratio` in Red Hat Enterprise Linux is 50.

For information about the kernel parameter, see the *Greenplum Database Installation Guide*.

Value Range	Default	Set Classifications
integer	8192	local system restart

gp_vmem_protect_segworker_cache_limit

If a query executor process consumes more than this configured amount, then the process will not be cached for use in subsequent queries after the process completes. Systems with lots of connections or idle processes may want to reduce this number to free more memory on the segments. Note that this is a local parameter and must be set for every segment.

Value Range	Default	Set Classifications
number of megabytes	500	local system restart

gp_workfile_checksumming

Adds a checksum value to each block of a work file (or spill file) used by `HashAgg` and `HashJoin` query operators. This adds an additional safeguard from faulty OS disk drivers writing corrupted blocks to disk. When a checksum operation fails, the query will cancel and rollback rather than potentially writing bad data to disk.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_workfile_compress_algorithm

When a hash aggregation or hash join operation spills to disk during query processing, specifies the compression algorithm to use on the spill files. If using zlib, it must be in your $PATH on all segments.

If your Greenplum database installation uses serial ATA (SATA) disk drives, setting the value of this parameter to `zlib` might help to avoid overloading the disk subsystem with IO operations.

Value Range	Default	Set Classifications
none zlib	none	master session reload

gp_workfile_limit_files_per_query

Sets the maximum number of temporary spill files (also known as workfiles) allowed per query per segment. Spill files are created when executing a query that requires more memory than it is allocated. The current query is terminated when the limit is exceeded.

Set the value to 0 (zero) to allow an unlimited number of spill files. master session reload

Value Range	Default	Set Classifications
integer	100000	master session reload

gp_workfile_limit_per_query

Sets the maximum disk size an individual query is allowed to use for creating temporary spill files at each segment. The default value is 0, which means a limit is not enforced.

Value Range	Default	Set Classifications
kilobytes	0	master session reload

gp_workfile_limit_per_segment

Sets the maximum total disk size that all running queries are allowed to use for creating temporary spill files at each segment. The default value is 0, which means a limit is not enforced.

Value Range	Default	Set Classifications
kilobytes	0	local system restart

gpperfmon_port

Sets the port on which all data collection agents (for Command Center) communicate with the master.

Value Range	Default	Set Classifications
integer	8888	master system restart

integer_datetimes

Reports whether PostgreSQL was built with support for 64-bit-integer dates and times.

Value Range	Default	Set Classifications
Boolean	on	read only

IntervalStyle

Sets the display format for interval values. The value *sql_standard* produces output matching SQL standard interval literals. The value *postgres* produces output matching PostgreSQL releases prior to 8.4 when the *DateStyle* parameter was set to ISO.

The value *postgres_verbose* produces output matching Greenplum releases prior to 3.3 when the *DateStyle* parameter was set to non-ISO output.

The value *iso_8601* will produce output matching the time interval *format with designators* defined in section 4.4.3.2 of ISO 8601. See the *PostgreSQL 8.4 documentation* for more information.

Value Range	Default	Set Classifications
postgres postgres_verbose sql_standard iso_8601	postgres	master session reload

join_collapse_limit

The legacy query optimizer (planner) will rewrite explicit inner `JOIN` constructs into lists of `FROM` items whenever a list of no more than this many items in total would result. By default, this variable is set the same as *from_collapse_limit*, which is appropriate for most uses. Setting it to 1 prevents any reordering of inner JOINs. Setting this variable to a value between 1 and *from_collapse_limit* might be useful to trade off planning time against the quality of the chosen plan (higher values produce better plans).

Value Range	Default	Set Classifications
1-*n*	20	master session reload

keep_wal_segments

For Greenplum Database master mirroring, sets the maximum number of processed WAL segment files that are saved by the by the active Greenplum Database master if a checkpoint operation occurs.

The segment files are used to sycnronize the active master on the standby master.

Value Range	Default	Set Classifications
integer	5	master system reload superuser

krb_caseins_users

Sets whether Kerberos user names should be treated case-insensitively. The default is case sensitive (off).

Value Range	Default	Set Classifications
Boolean	off	master system restart

krb_server_keyfile

Sets the location of the Kerberos server key file.

Value Range	Default	Set Classifications
path and file name	unset	master system restart

krb_srvname

Sets the Kerberos service name.

Value Range	Default	Set Classifications
service name	postgres	master system restart

lc_collate

Reports the locale in which sorting of textual data is done. The value is determined when the Greenplum Database array is initialized.

Value Range	Default	Set Classifications
<system dependent>		read only

lc_ctype

Reports the locale that determines character classifications. The value is determined when the Greenplum Database array is initialized.

Value Range	Default	Set Classifications
<system dependent>		read only

lc_messages

Sets the language in which messages are displayed. The locales available depends on what was installed with your operating system - use *locale -a* to list available locales. The default value is inherited from the execution environment of the server. On some systems, this locale category does not exist. Setting this variable will still work, but there will be no effect. Also, there is a chance that no translated messages for the desired language exist. In that case you will continue to see the English messages.

Value Range	Default	Set Classifications
<system dependent>		local system restart

lc_monetary

Sets the locale to use for formatting monetary amounts, for example with the *to_char* family of functions. The locales available depends on what was installed with your operating system - use *locale -a* to list available locales. The default value is inherited from the execution environment of the server.

Value Range	Default	Set Classifications
<system dependent>		local system restart

lc_numeric

Sets the locale to use for formatting numbers, for example with the *to_char* family of functions. The locales available depends on what was installed with your operating system - use *locale -a* to list available locales. The default value is inherited from the execution environment of the server.

Value Range	Default	Set Classifications
<system dependent>		local system restart

lc_time

This parameter currently does nothing, but may in the future.

Value Range	Default	Set Classifications
<system dependent>		local system restart

listen_addresses

Specifies the TCP/IP address(es) on which the server is to listen for connections from client applications - a comma-separated list of host names and/or numeric IP addresses. The special entry * corresponds to all available IP interfaces. If the list is empty, only UNIX-domain sockets can connect.

Value Range	Default	Set Classifications
localhost, host names, IP addresses, * (all available IP interfaces)	*	master system restart

local_preload_libraries

Comma separated list of shared library files to preload at the start of a client session.

Value Range	Default	Set Classifications
		local system restart

log_autostats

Logs information about automatic `ANALYZE` operations related to *gp_autostats_mode* and *gp_autostats_on_change_threshold*.

Value Range	Default	Set Classifications
Boolean	off	master session reload superuser

log_connections

This outputs a line to the server log detailing each successful connection. Some client programs, like psql, attempt to connect twice while determining if a password is required, so duplicate "connection received" messages do not always indicate a problem.

Value Range	Default	Set Classifications
Boolean	off	local system restart

log_disconnections

This outputs a line in the server log at termination of a client session, and includes the duration of the session.

Value Range	Default	Set Classifications
Boolean	off	local system restart

log_dispatch_stats

When set to "on," this parameter adds a log message with verbose information about the dispatch of the statement.

Value Range	Default	Set Classifications
Boolean	off	local system restart

log_duration

Causes the duration of every completed statement which satisfies *log_statement* to be logged.

Value Range	Default	Set Classifications
Boolean	off	master session reload superuser

log_error_verbosity

Controls the amount of detail written in the server log for each message that is logged.

Value Range	Default	Set Classifications
TERSE DEFAULT VERBOSE	DEFAULT	master session reload superuser

log_executor_stats

For each query, write performance statistics of the query executor to the server log. This is a crude profiling instrument. Cannot be enabled together with *log_statement_stats*.

Value Range	Default	Set Classifications
Boolean	off	local system restart

log_hostname

By default, connection log messages only show the IP address of the connecting host. Turning on this option causes logging of the IP address and host name of the Greenplum Database master. Note that depending on your host name resolution setup this might impose a non-negligible performance penalty.

Value Range	Default	Set Classifications
Boolean	off	master system restart

log_min_duration_statement

Logs the statement and its duration on a single log line if its duration is greater than or equal to the specified number of milliseconds. Setting this to 0 will print all statements and their durations. -1 disables the feature. For example, if you set it to 250 then all SQL statements that run 250ms or longer will be logged. Enabling this option can be useful in tracking down unoptimized queries in your applications.

Value Range	Default	Set Classifications
number of milliseconds, 0, -1	-1	master session reload superuser

log_min_error_statement

Controls whether or not the SQL statement that causes an error condition will also be recorded in the server log. All SQL statements that cause an error of the specified level or higher are logged. The default is PANIC (effectively turning this feature off for normal use). Enabling this option can be helpful in tracking down the source of any errors that appear in the server log.

Value Range	Default	Set Classifications
DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1 INFO NOTICE WARNING ERROR FATAL PANIC	ERROR	master session reload superuser

log_min_messages

Controls which message levels are written to the server log. Each level includes all the levels that follow it. The later the level, the fewer messages are sent to the log.

Value Range	Default	Set Classifications
DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1 INFO NOTICE WARNING ERROR FATAL PANIC	WARNING	master session reload superuser

log_parser_stats

For each query, write performance statistics of the query parser to the server log. This is a crude profiling instrument. Cannot be enabled together with *log_statement_stats*.

Value Range	Default	Set Classifications
Boolean	off	master session reload superuser

log_planner_stats

For each query, write performance statistics of the legacy query optimizer (planner) to the server log. This is a crude profiling instrument. Cannot be enabled together with *log_statement_stats*.

Value Range	Default	Set Classifications
Boolean	off	master session reload superuser

log_rotation_age

Determines the maximum lifetime of an individual log file. After this time has elapsed, a new log file will be created. Set to zero to disable time-based creation of new log files.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	1d	local system restart

log_rotation_size

Determines the maximum size of an individual log file. After this many kilobytes have been emitted into a log file, a new log file will be created. Set to zero to disable size-based creation of new log files.

The maximum value is INT_MAX/1024. If an invalid value is specified, the default value is used. INT_MAX is the largest value that can be stored as an integer on your system.

Value Range	Default	Set Classifications
number of kilobytes	0	local system restart

log_statement

Controls which SQL statements are logged. DDL logs all data definition commands like CREATE, ALTER, and DROP commands. MOD logs all DDL statements, plus INSERT, UPDATE, DELETE, TRUNCATE, and COPY FROM. PREPARE and EXPLAIN ANALYZE statements are also logged if their contained command is of an appropriate type.

Value Range	Default	Set Classifications
NONE DDL MOD ALL	ALL	master session reload superuser

log_statement_stats

For each query, write total performance statistics of the query parser, planner, and executor to the server log. This is a crude profiling instrument.

Value Range	Default	Set Classifications
Boolean	off	master session reload superuser

log_timezone

Sets the time zone used for timestamps written in the log. Unlike *TimeZone*, this value is system-wide, so that all sessions will report timestamps consistently. The default is `unknown`, which means to use whatever the system environment specifies as the time zone.

Value Range	Default	Set Classifications
string	unknown	local system restart

log_truncate_on_rotation

Truncates (overwrites), rather than appends to, any existing log file of the same name. Truncation will occur only when a new file is being opened due to time-based rotation. For example, using this setting in combination with a log_filename such as `gpseg#-%H.log` would result in generating twenty-four hourly log files and then cyclically overwriting them. When off, pre-existing files will be appended to in all cases.

Value Range	Default	Set Classifications
Boolean	off	local system restart

max_appendonly_tables

Sets the maximum number of append-optimized relations that can be written to or loaded concurrently. Append-optimized table partitions and subpartitions are considered as unique tables against this limit. Increasing the limit will allocate more shared memory at server start.

Value Range	Default	Set Classifications
integer > 0	10000	master system restart

max_connections

The maximum number of concurrent connections to the database server. In a Greenplum Database system, user client connections go through the Greenplum master instance only. Segment instances should allow 5-10 times the amount as the master. When you increase this parameter, *max_prepared_transactions* must be increased as well. For more information about limiting concurrent connections, see "Configuring Client Authentication" in the *Greenplum Database Administrator Guide*.

Increasing this parameter may cause Greenplum Database to request more shared memory.Increasing this parameter might cause Greenplum Database to request more shared memory. See *shared_buffers* for information about Greenplum server instance shared memory buffers.

Value Range	Default	Set Classifications
10-*n*	250 on master 750 on segments	local system restart

max_files_per_process

Sets the maximum number of simultaneously open files allowed to each server subprocess. If the kernel is enforcing a safe per-process limit, you don't need to worry about this setting. Some platforms such as BSD, the kernel will allow individual processes to open many more files than the system can really support.

Value Range	Default	Set Classifications
integer	1000	local system restart

max_fsm_pages

Sets the maximum number of disk pages for which free space will be tracked in the shared free-space map. Six bytes of shared memory are consumed for each page slot.

Value Range	Default	Set Classifications
integer > 16 * *max_fsm_relations*	200000	local system restart

max_fsm_relations

Sets the maximum number of relations for which free space will be tracked in the shared memory free-space map. Should be set to a value larger than the total number of:

tables + indexes + system tables.

It costs about 60 bytes of memory for each relation per segment instance. It is better to allow some room for overhead and set too high rather than too low.

Value Range	Default	Set Classifications
integer	1000	local system restart

max_function_args

Reports the maximum number of function arguments.

Value Range	Default	Set Classifications
integer	100	read only

max_identifier_length

Reports the maximum identifier length.

Value Range	Default	Set Classifications
integer	63	read only

max_index_keys

Reports the maximum number of index keys.

Value Range	Default	Set Classifications
integer	32	read only

max_locks_per_transaction

The shared lock table is created with room to describe locks on *max_locks_per_transaction* * (*max_connections* + *max_prepared_transactions*) objects, so no more than this many distinct objects can be locked at any one time. This is not a hard limit on the number of locks taken by any one transaction, but rather a maximum average value. You might need to raise this value if you have clients that touch many different tables in a single transaction.

Value Range	Default	Set Classifications
integer	128	local system restart

max_prepared_transactions

Sets the maximum number of transactions that can be in the prepared state simultaneously. Greenplum uses prepared transactions internally to ensure data integrity across the segments. This value must be at least as large as the value of *max_connections* on the master. Segment instances should be set to the same value as the master.

Value Range	Default	Set Classifications
integer	250 on master 250 on segments	local system restart

max_resource_portals_per_transaction

Sets the maximum number of simultaneously open user-declared cursors allowed per transaction. Note that an open cursor will hold an active query slot in a resource queue. Used for workload management.

Value Range	Default	Set Classifications
integer	64	master system restart

max_resource_queues

Sets the maximum number of resource queues that can be created in a Greenplum Database system. Note that resource queues are system-wide (as are roles) so they apply to all databases in the system.

Value Range	Default	Set Classifications
integer	9	master system restart

max_stack_depth

Specifies the maximum safe depth of the server's execution stack. The ideal setting for this parameter is the actual stack size limit enforced by the kernel (as set by *ulimit -s* or local equivalent), less a safety margin of a megabyte or so. Setting the parameter higher than the actual kernel limit will mean that a runaway recursive function can crash an individual backend process.

Value Range	Default	Set Classifications
number of kilobytes	2MB	local system restart

max_statement_mem

Sets the maximum memory limit for a query. Helps avoid out-of-memory errors on a segment host during query processing as a result of setting *statement_mem* too high. When *gp_resqueue_memory_policy*=auto, statement_mem and resource queue memory limits control query memory usage. Taking into account the configuration of a single segment host, calculate this setting as follows:

```
(seghost_physical_memory) / (average_number_concurrent_queries)
```

Value Range	Default	Set Classifications
number of kilobytes	2000MB	master session reload superuser

optimizer

Enables the Pivotal Query Optimizer when running SQL queries. The default is `off`, Greenplum Database uses only the legacy query optimizer.

The Pivotal Query Optimizer co-exists with the legacy query optimizer. When the Pivotal Query Optimizer is enabled, Greenplum Database uses the Pivotal Query Optimizer to generate an execution plan for a query when possible. If the Pivotal Query Optimizer cannot be used, the legacy query optimizer is used.

The `optimizer` parameter can be set for a database system, an individual database, or a session or query.

For information about the legacy query optimizer and the Pivotal Query Optimizer, see "Querying Data" in the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
Boolean	off	master session reload

optimizer_analyze_root_partition

For a partitioned table, collects statistics for the root partition when the `ANALYZE` command is run on the table. The Pivotal Query Optimizer uses the root partition statistics when generating a query plan. The legacy query optimizer does not use these statistics. If you set the value of the server configuration parameter *`optimizer`* to on, set the value of this parameter to on and run the command `ANALYZE` or `ANALYZE ROOTPARTITION` on partitioned tables to ensure the proper statistics have been collected.

For information about the legacy query optimizer and the Pivotal Query Optimizer, see "Querying Data" in the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
Boolean	off	master session reload

optimizer_control

Controls whether the server configuration parameter `optimizer` can be changed with `SET`, the `RESET` command, or the Greenplum Database utility `gpconfig`. If the `optimizer_control` parameter value is `on`, users can set the `optimizer` parameter. If the `optimizer_control` parameter value is `off`, the `optimizer` parameter cannot be changed.

Value Range	Default	Set Classifications
Boolean	on	master system restart superuser

optimizer_enable_master_only_queries

When the Pivotal Query Optimizer is enabled (the server configuration parameter `optimizer` is `on`), this parameter allows the Pivotal Query Optimizer to execute catalog queries that run only on the Greenplum Database master. For the default value `off`, only the legacy query optimizer can execute catalog queries that run only on the Greenplum Database master.

The parameter can be set for a database system, an individual database, or a session or query.

> **Note:** Enabling this parameter decreases performance of short running catalog queries. To avoid this issue, set this parameter only for a session or a query.

For information about the Pivotal Query Optimizer, see the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
Boolean	off	master session reload

optimizer_minidump

The Pivotal Query Optimizer generates minidump files to describe the optimization context for a given query. The minidump files are used by Pivotal support to analyze Greenplum Database issues. The information in the file is not in a format that can be easily used by customers for debugging or troubleshooting. The minidump file is located under the master data directory and uses the following naming format:

`Minidump_`*`date_time`*`.mdp`

The minidump file contains this query related information:

- Catalog objects including data types, tables, operators, and statistics required by the Pivotal Query Optimizer
- An internal representation (DXL) of the query
- An internal representation (DXL) of the plan produced by the Pivotal Query Optimizer
- System configuration information passed to the Pivotal Query Optimizer such as server configuration parameters, cost and statistics configuration, and number of segments
- A stack trace of errors generated while optimizing the query

Setting this parameter to `ALWAYS` generates a minidump for all queries. Pivotal recommends that you set this parameter to `ONERROR` in production environments to minimize total optimization time.

For information about the Pivotal Query Optimizer, see "Querying Data" in the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
ONERROR ALWAYS	ONERROR	master session reload

password_encryption

When a password is specified in `CREATE USER` or `ALTER USER` without writing either `ENCRYPTED` or `UNENCRYPTED`, this option determines whether the password is to be encrypted.

Value Range	Default	Set Classifications
Boolean	on	master session reload

password_hash_algorithm

Specifies the cryptographic hash algorithm that is used when storing an encrypted Greenplum Database user password. The default algorithm is MD5.

For information about setting the password hash algorithm to protect user passwords, see "Protecting Passwords in Greenplum Database" in the *Greenplum Database Administrator Guide*.

Value Range	Default	Set Classifications
MD5 SHA-256 SHA-256-FIPS	MD5	master session reload superuser

pgcrypto.fips

Enables support for *Federal Information Processing Standard* (FIPS) 140-2. For information about FIPS, see *http://www.nist.gov/itl/fips.cfm*

To enable FIPS 140-2 support for Greenplum Database, the following are required.

- The Greenplum Database pgcrypto package version 1.2 or later must be installed.
- If the value of `pgcrypto.fips` is set to `on`, the value of the parameter `custom_variable_classes` must contain `pgcrypto`.

When FIPS 140-2 support is enabled, these pgcrypto changes occur:

- FIPS mode is initialized in the OpenSSL library
- The functions `digest()` and `hmac()` allow only the SHA encryption algorithm (MD5 is not allowed)
- The functions for crypt and gen_salt algorithms are disabled
- PGP encryption and decryption functions support only AES and 3DES encryption algorithms (other algorithms such as blowfish are not allowed)
- RAW encryption and decryption functions support only AES and 3DES (other algorithms such as blowfish are not allowed)

These `gpconfig` commands set the parameters to enable FIPS 140-2 support.

```
$ gpconfig -c custom_variable_classes -v pgcrypto --masteronly
$ gpconfig -c pgcrypto.fips -v on --masteronly
```

The value of the `custom_variable_classes` parameter is a comma separated list of classes. For more than one class, the list is inclosed in single quotes. To check the value of the parameter use `gpconfig` with the `-s` option to show the current value.

```
$ gpconfig -s custom_variable_classes
```

If the parameter is already set with custom classes, you can add `pgcrypto`. For example, if the value of `custom_variable_classes` is `plr`, this command adds `pgcrypto`.

```
$ gpconfig -c custom_variable_classes -v \'plr,pgcrypto\' --masteronly
   --skipvalidation
```

In the command, use a backslash (\) to escape the single quotes.

Value Range	Default	Set Classifications
Boolean	off	master system restart

pgstat_track_activity_query_size

Sets the maximum length limit for the query text stored in `current_query` column of the system catalog view pg_stat_activity. The minimum length is 1024 characters.

Value Range	Default	Set Classifications
integer	1024	local system restart

pljava_classpath

A colon (`:`) separated list of jar files or directories containing jar files needed for PL/Java functions. The full path to the jar file or directory must be specified, except the path can be omitted for jar files in the `$GPHOME/lib/postgresql/java` directory. The jar files must be installed in the same locations on all Greenplum hosts and readable by the `gpadmin` user.

The `pljava_classpath` parameter is used to assemble the PL/Java classpath at the beginning of each user session. Jar files added after a session has started are not available to that session.

If the full path to a jar file is specified in `pljava_classpath` it is added to the PL/Java classpath. When a directory is specified, any jar files the directory contains are added to the PL/Java classpath. The search does not descend into subdirectories of the specified directories. If the name of a jar file is included in `pljava_classpath` with no path, the jar file must be in the `$GPHOME/lib/postgresql/java` directory.

> **Note:** Performance can be affected if there are many directories to search or a large number of jar files.

Setting the `pljava_classpath` parameter requires superuser privilege. Setting the classpath in SQL code will fail when the code is executed by a user without superuser privilege. The `pljava_classpath` parameter must have been set previously by a superuser or in the `postgresql.conf` file. Changing the classpath in the `postgresql.conf` file requires a reload (`gpstop -u`).

Value Range	Default	Set Classifications
string		master session reload superuser

pljava_statement_cache_size

Sets the size in KB of the JRE MRU (Most Recently Used) cache for prepared statements.

Value Range	Default	Set Classifications
number of kilobytes	10	master system restart superuser

pljava_release_lingering_savepoints

If true, lingering savepoints used in PL/Java functions will be released on function exit. If false, savepoints will be rolled back.

Value Range	Default	Set Classifications
Boolean	true	master system restart superuser

pljava_vmoptions

Defines the startup options for the Java VM. The default value is an empty string ("").

Value Range	Default	Set Classifications
string		master system restart superuser

port

The database listener port for a Greenplum instance. The master and each segment has its own port. Port numbers for the Greenplum system must also be changed in the `gp_segment_configuration` catalog. You must shut down your Greenplum Database system before changing port numbers.

Value Range	Default	Set Classifications
any valid port number	5432	local system restart

random_page_cost

Sets the estimate of the cost of a nonsequentially fetched disk page for the legacy query optimizer (planner). This is measured as a multiple of the cost of a sequential page fetch. A higher value makes it more likely a sequential scan will be used, a lower value makes it more likely an index scan will be used.

Value Range	Default	Set Classifications
floating point	100	master session reload

readable_external_table_timeout

When an SQL query reads from an external table, the parameter value specifies the amount of time in seconds that Greenplum Database waits before cancelling the query when data stops being returned from the external table.

The default value of 0, specifies no time out. Greenplum Database does not cancel the query.

If queries that use gpfdist run a long time and then return the error "intermittent network connectivity issues",you can specify a value for ternal_table_timeout. If no data is returned by gpfdist for the specified length of time, Greenplum Database cancels the query. master system reload

Value Range	Default	Set Classifications
integer >= 0	0	master system reload

repl_catchup_within_range

For Greenplum Database master mirroring, controls updates to the active master. If the number of WAL segment files that have not been processed by the `walsender` exceeds this value, Greenplum Database updates the active master.

If the number of segment files does not exceed the value, Greenplum Database blocks updates to the to allow the `walsender` process the files. If all WAL segments have been processed, the active master is updated.

Value Range	Default	Set Classifications
0 - 64	1	master system reload superuser

replication_timeout

For Greenplum Database master mirroring, sets the maximum time in milliseconds that the `walsender` process on the active master waits for a status message from the `walreceiver` process on the standby master. If a message is not received, the `walsender` logs an error message.

The *wal_receiver_status_interval* controls the interval between `walreceiver` status messages.

Value Range	Default	Set Classifications
0 - INT_MAX	60000 ms (60 seconds)	master system reload superuser

regex_flavor

The 'extended' setting may be useful for exact backwards compatibility with pre-7.4 releases of PostgreSQL.

Value Range	Default	Set Classifications
advanced extended basic	advanced	master session reload

resource_cleanup_gangs_on_wait

If a statement is submitted through a resource queue, clean up any idle query executor worker processes before taking a lock on the resource queue.

Value Range	Default	Set Classifications
Boolean	on	master system restart

resource_select_only

Sets the types of queries managed by resource queues. If set to on, then SELECT, SELECT INTO, CREATE TABLE AS SELECT, and DECLARE CURSOR commands are evaluated. If set to off INSERT, UPDATE, and DELETE commands will be evaluated as well.

Value Range	Default	Set Classifications
Boolean	off	master system restart

runaway_detector_activation_percent

Sets the percentage of Greenplum Database vmem memory that triggers the termination of queries. If the percentage of vmem memory that is utilized for a Greenplum Database segment exceeds the specified value, Greenplum Database terminates queries based on memory usage, starting with the query consuming the largest amount of memory. Queries are terminated until the percentage of utilized vmem is below the specified percentage.

Specify the maximum vmem value for active Greenplum Database segment instances with the server configuration parameter *gp_vmem_protect_limit*.

For example, if vmem memory is set to 10GB, and the value of `runaway_detector_activation_percent` is 90 (90%), Greenplum Database starts terminating queries when the utilized vmem memory exceeds 9 GB.

A value of 0 disables the automatic termination of queries based on percentage of vmem that is utilized.

Value Range	Default	Set Classifications
percentage (integer)	90	local system restart

search_path

Specifies the order in which schemas are searched when an object is referenced by a simple name with no schema component. When there are objects of identical names in different schemas, the one found first in the search path is used. The system catalog schema, *pg_catalog*, is always searched, whether it is mentioned in the path or not. When objects are created without specifying a particular target schema, they will be placed in the first schema listed in the search path. The current effective value of the search path can be examined via the SQL function *current_schemas()*. *current_schemas()* shows how the requests appearing in *search_path* were resolved.

Value Range	Default	Set Classifications
a comma- separated list of schema names	$user,public	master session reload

seq_page_cost

For the legacy query optimizer (planner), sets the estimate of the cost of a disk page fetch that is part of a series of sequential fetches.

Value Range	Default	Set Classifications
floating point	1	master session reload

server_encoding

Reports the database encoding (character set). It is determined when the Greenplum Database array is initialized. Ordinarily, clients need only be concerned with the value of *client_encoding*.

Value Range	Default	Set Classifications
<system dependent>	UTF8	read only

server_version

Reports the version of PostgreSQL that this release of Greenplum Database is based on.

Value Range	Default	Set Classifications
string	8.2.15	read only

server_version_num

Reports the version of PostgreSQL that this release of Greenplum Database is based on as an integer.

Value Range	Default	Set Classifications
integer	80215	read only

shared_buffers

Sets the amount of memory a Greenplum Database segment instance uses for shared memory buffers. This setting must be at least 128KB and at least 16KB times *max_connections*.

Each Greenplum Database segment instance calculates and attempts to allocate certain amount of shared memory based on the segment configuration. The value of `shared_buffers` is significant portion of this shared memory calculation, but is not all it. When setting `shared_buffers`, the values for the operating system parameters `SHMMAX` or `SHMALL` might also need to be adjusted.

The operating system parameter `SHMMAX` specifies maximum size of a single shared memory allocation. The value of `SHMMAX` must be greater than this value:

```
shared_buffers + other_seg_shmem
```

The value of *other_seg_shmem* is the portion the Greenplum Database shared memory calculation that is not accounted for by the `shared_buffers` value. The *other_seg_shmem* value will vary based on the segment configuration.

With the default Greenplum Database parameter values, the value for *other_seg_shmem* is approximately 111MB for Greenplum Database segments and approximately 79MB for the Greenplum Database master.

The operating system parameter `SHMALL` specifies the maximum amount of shared memory on the host. The value of `SHMALL` must be greater than this value:

```
(num_instances_per_host * ( shared_buffers + other_seg_shmem ))
+ other_app_shared_mem
```

The value of *other_app_shared_mem* is the amount of shared memory that is used by other applications and processes on the host.

When shared memory allocation errors occur, possible ways to resolve shared memory allocation issues are to increase `SHMMAX` or `SHMALL`, or decrease `shared_buffers` or `max_connections`.

See the *Greenplum Database Installation Guide* for information about the Greenplum Database values for the parameters `SHMMAX` and `SHMALL`.

Value Range	Default	Set Classifications
integer > 16K * *max_connections*	125MB	local system restart

shared_preload_libraries

A comma-separated list of shared libraries that are to be preloaded at server start. PostgreSQL procedural language libraries can be preloaded in this way, typically by using the syntax '`$libdir/plXXX`' where *XXX* is pgsql, perl, tcl, or python. By preloading a shared library, the library startup time is avoided when the library is first used. If a specified library is not found, the server will fail to start.

Value Range	Default	Set Classifications
		local system restart

ssl

Enables SSL connections.

Value Range	Default	Set Classifications
Boolean	off	master system restart

ssl_ciphers

Specifies a list of SSL ciphers that are allowed to be used on secure connections. See the openssl manual page for a list of supported ciphers.

Value Range	Default	Set Classifications
string	ALL	master system restart

standard_conforming_strings

Reports whether ordinary string literals ('...') treat backslashes literally, as specified in the SQL standard. The value is currently always off, indicating that backslashes are treated as escapes. It is planned that this will change to on in a future release when string literal syntax changes to meet the standard. Applications may check this parameter to determine how string literals will be processed. The presence of this parameter can also be taken as an indication that the escape string syntax (E'...') is supported.

Value Range	Default	Set Classifications
Boolean	of	read only

statement_mem

Allocates segment host memory per query. The amount of memory allocated with this parameter cannot exceed *max_statement_mem* or the memory limit on the resource queue through which the query was submitted. When *gp_resqueue_memory_policy* =auto, `statement_mem` and resource queue memory limits control query memory usage.

If additional memory is required for a query, temporary spill files on disk are used.

This calculation can be used to estimate a reasonable value for a wide variety of situations.

```
( gp_vmem_protect_limitGB * .9 ) / max_expected_concurrent_queries
```

With the *gp_vmem_protect_limit* set to 8192MB (8GB) and assuming a maximum of 40 concurrent queries with a 10% buffer

```
(8GB * .9) / 40 = .18GB = 184MB
```

Value Range	Default	Set Classifications
number of kilobytes	128MB	master session reload

statement_timeout

Abort any statement that takes over the specified number of milliseconds. 0 turns off the limitation.

Value Range	Default	Set Classifications
number of milliseconds	0	master session reload

stats_queue_level

Collects resource queue statistics on database activity.

Value Range	Default	Set Classifications
Boolean	off	master session reload

superuser_reserved_connections

Determines the number of connection slots that are reserved for Greenplum Database superusers.

Value Range	Default	Set Classifications
integer < *max_connections*	3	local system restart

tcp_keepalives_count

How many keepalives may be lost before the connection is considered dead. A value of 0 uses the system default. If TCP_KEEPCNT is not supported, this parameter must be 0.

Use this parameter for all connections that are not between a primary and mirror segment. Use gp_filerep_tcp_keepalives_count for settings that are between a primary and mirror segment.

Value Range	Default	Set Classifications
number of lost keepalives	0	local system restart

tcp_keepalives_idle

Number of seconds between sending keepalives on an otherwise idle connection. A value of 0 uses the system default. If TCP_KEEPIDLE is not supported, this parameter must be 0.

Use this parameter for all connections that are not between a primary and mirror segment. Use gp_filerep_tcp_keepalives_idle for settings that are between a primary and mirror segment.

Value Range	Default	Set Classifications
number of seconds	0	local system restart

tcp_keepalives_interval

How many seconds to wait for a response to a keepalive before retransmitting. A value of 0 uses the system default. If TCP_KEEPINTVL is not supported, this parameter must be 0.

Use this parameter for all connections that are not between a primary and mirror segment. Use gp_filerep_tcp_keepalives_interval for settings that are between a primary and mirror segment.

Value Range	Default	Set Classifications
number of seconds	0	local system restart

temp_buffers

Sets the maximum number of temporary buffers used by each database session. These are session-local buffers used only for access to temporary tables. The setting can be changed within individual sessions, but only up until the first use of temporary tables within a session. The cost of setting a large value in sessions that do not actually need a lot of temporary buffers is only a buffer descriptor, or about 64 bytes, per increment. However if a buffer is actually used, an additional 8192 bytes will be consumed.

Value Range	Default	Set Classifications
integer	1024	master session reload

TimeZone

Sets the time zone for displaying and interpreting time stamps. The default is to use whatever the system environment specifies as the time zone. See *Date/Time Keywords* in the PostgreSQL documentation.

Value Range	Default	Set Classifications
time zone abbreviation		local restart

timezone_abbreviations

Sets the collection of time zone abbreviations that will be accepted by the server for date time input. The default is `Default`, which is a collection that works in most of the world. `Australia` and `India`, and other collections can be defined for a particular installation. Possible values are names of configuration files stored in `$GPHOME/share/postgresql/timezonesets/`.

To configure Greenplum Database to use a custom collection of timezones, copy the file that contains the timezone definitions to the directory `$GPHOME/share/postgresql/timezonesets/` on the Greenplum Database master and segment hosts. Then set value of the server configuration parameter `timezone_abbreviations` to the file. For example, to use a file `custom` that contains the default timezones and the WIB (Waktu Indonesia Barat) timezone.

1. Copy the file `Default` from the directory `$GPHOME/share/postgresql/timezonesets/` the file `custom`. Add the WIB timezone information from the file `Asia.txt` to the `custom`.
2. Copy the file `custom` to the directory `$GPHOME/share/postgresql/timezonesets/` on the Greenplum Database master and segment hosts.
3. Set value of the server configuration parameter `timezone_abbreviations` to `custom`.
4. Reload the server configuration file (`gpstop -u`).

Value Range	Default	Set Classifications
string	Default	master session reload

track_activities

Enables the collection of statistics on the currently executing command of each session, along with the time at which that command began execution. When enabled, this information is not visible to all users, only to superusers and the user owning the session. This data can be accessed via the *pg_stat_activity* system view.

Value Range	Default	Set Classifications
Boolean	on	master session reload

track_counts

Enables the collection of row and block level statistics on database activity. If enabled, the data that is produced can be accessed via the *pg_stat* and *pg_statio* family of system views.

Value Range	Default	Set Classifications
Boolean	off	local system restart

transaction_isolation

Sets the current transaction's isolation level.

Value Range	Default	Set Classifications
read committed serializable	read committed	master session reload

transaction_read_only

Sets the current transaction's read-only status.

Value Range	Default	Set Classifications
Boolean	off	master session reload

transform_null_equals

When on, expressions of the form expr = NULL (or NULL = expr) are treated as expr IS NULL, that is, they return true if expr evaluates to the null value, and false otherwise. The correct SQL-spec-compliant behavior of expr = NULL is to always return null (unknown).

Value Range	Default	Set Classifications
Boolean	off	master session reload

unix_socket_directory

Specifies the directory of the UNIX-domain socket on which the server is to listen for connections from client applications.

Value Range	Default	Set Classifications
directory path	unset	local system restart

unix_socket_group

Sets the owning group of the UNIX-domain socket. By default this is an empty string, which uses the default group for the current user.

Value Range	Default	Set Classifications
UNIX group name	unset	local system restart

unix_socket_permissions

Sets the access permissions of the UNIX-domain socket. UNIX-domain sockets use the usual UNIX file system permission set. Note that for a UNIX-domain socket, only write permission matters.

Value Range	Default	Set Classifications
numeric UNIX file permission mode (as accepted by the *chmod* or *umask* commands)	511	local system restart

update_process_title

Enables updating of the process title every time a new SQL command is received by the server. The process title is typically viewed by the `ps` command.

Value Range	Default	Set Classifications
Boolean	on	local system restart

vacuum_cost_delay

The length of time that the process will sleep when the cost limit has been exceeded. 0 disables the cost-based vacuum delay feature.

Value Range	Default	Set Classifications
milliseconds < 0 (in multiples of 10)	0	local system restart

vacuum_cost_limit

The accumulated cost that will cause the vacuuming process to sleep.

Value Range	Default	Set Classifications
integer > 0	200	local system restart

vacuum_cost_page_dirty

The estimated cost charged when vacuum modifies a block that was previously clean. It represents the extra I/O required to flush the dirty block out to disk again.

Value Range	Default	Set Classifications
integer > 0	20	local system restart

vacuum_cost_page_hit

The estimated cost for vacuuming a buffer found in the shared buffer cache. It represents the cost to lock the buffer pool, lookup the shared hash table and scan the content of the page.

Value Range	Default	Set Classifications
integer > 0	1	local system restart

vacuum_cost_page_miss

The estimated cost for vacuuming a buffer that has to be read from disk. This represents the effort to lock the buffer pool, lookup the shared hash table, read the desired block in from the disk and scan its content.

Value Range	Default	Set Classifications
integer > 0	10	local system restart

vacuum_freeze_min_age

Specifies the cutoff age (in transactions) that `VACUUM` should use to decide whether to replace transaction IDs with *FrozenXID* while scanning a table.

For information about `VACUUM` and transaction ID management, see "Managing Data" in the *Greenplum Database Administrator Guide* and the *PostgreSQL documentation*.

Value Range	Default	Set Classifications
integer 0-100000000000	100000000	local system restart

wal_receiver_status_interval

For Greenplum Database master mirroring, sets the interval in seconds between `walreceiver` process status messages that are sent to the active master. Under heavy loads, the time might be longer.

The value of *replication_timeout* controls the time that the `walsender` process waits for a `walreceiver` message.

Value Range	Default	Set Classifications
integer 0- INT_MAX/1000	10 sec	master system reload superuser

writable_external_table_bufsize

Size of the buffer (in KB) that Greenplum Database uses for network communication, such as the `gpfdist` utility and web external tables (that use http). Greenplum Database stores data in the buffer before writing the data out. For information about `gpfdist`, see the *Greenplum Database Utility Guide*.

Value Range	Default	Set Classifications
integer 32 - 131072 (32KB - 128MB)	64	local system reload

xid_stop_limit

The number of transaction IDs prior to the ID where transaction ID wraparound occurs. When this limit is reached, Greenplum Database stops creating new transactions to avoid data loss due to transaction ID wraparound.

Value Range	Default	Set Classifications
integer 10000000 - 2000000000	1000000000	local system restart

xid_warn_limit

The number of transaction IDs prior to the limit specified by *xid_stop_limit*. When Greenplum Database reaches this limit, it issues a warning to perform a VACUUM operation to avoid data loss due to transaction ID wraparound.

Value Range	Default	Set Classifications
integer 10000000 - 2000000000	500000000	local system restart

Chapter 10

Summary of Built-in Functions

Greenplum Database supports built-in functions and operators including analytic functions and window functions that can be used in window expressions. For information about using built-in Greenplum Database functions see, "Using Functions and Operators" in the *Greenplum Database Administrator Guide*.

Greenplum Database Function Types

Greenplum Database evaluates functions and operators used in SQL expressions. Some functions and operators are only allowed to execute on the master since they could lead to inconsistencies in Greenplum Database segment instances. This table describes the Greenplum Database Function Types.

Table 136: Functions in Greenplum Database

Function Type	Greenplum Support	Description	Comments
IMMUTABLE	Yes	Relies only on information directly in its argument list. Given the same argument values, always returns the same result.	
STABLE	Yes, in most cases	Within a single table scan, returns the same result for same argument values, but results change across SQL statements.	Results depend on database lookups or parameter values. `current_timestamp` family of functions is `STABLE`; values do not change within an execution.
VOLATILE	Restricted	Function values can change within a single table scan. For example: `random()`, `currval()`, `timeofday()`.	Any function with side effects is volatile, even if its result is predictable. For example: `setval()`.

In Greenplum Database, data is divided up across segments — each segment is a distinct PostgreSQL database. To prevent inconsistent or unexpected results, do not execute functions classified as `VOLATILE` at the segment level if they contain SQL commands or modify the database in any way. For example, functions such as `setval()` are not allowed to execute on distributed data in Greenplum Database because they can cause inconsistent data between segment instances.

To ensure data consistency, you can safely use `VOLATILE` and `STABLE` functions in statements that are evaluated on and run from the master. For example, the following statements run on the master (statements without a `FROM` clause):

```
SELECT setval('myseq', 201);
SELECT foo();
```

If a statement has a `FROM` clause containing a distributed table *and* the function in the `FROM` clause returns a set of rows, the statement can run on the segments:

```
SELECT * from foo();
```

Greenplum Database does not support functions that return a table reference (`rangeFuncs`) or functions that use the `refCursor` datatype.

Built-in Functions and Operators

The following table lists the categories of built-in functions and operators supported by PostgreSQL. All functions and operators are supported in Greenplum Database as in PostgreSQL with the exception of `STABLE` and `VOLATILE` functions, which are subject to the restrictions noted in *Greenplum Database Function Types*. See the *Functions and Operators* section of the PostgreSQL documentation for more information about these built-in functions and operators.

Table 137: Built-in functions and operators

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
Logical Operators			
Comparison Operators			
Mathematical Functions and Operators	random setseed		
String Functions and Operators	*All built-in conversion functions*	convert pg_client_encoding	
Binary String Functions and Operators			
Bit String Functions and Operators			
Pattern Matching			
Data Type Formatting Functions		to_char to_timestamp	
Date/Time Functions and Operators	timeofday	age current_date current_time current_timestamp localtime localtimestamp now	
Geometric Functions and Operators			
Network Address Functions and Operators			

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
Sequence Manipulation Functions	currval lastval nextval setval		
Conditional Expressions			
Array Functions and Operators		*All array functions*	
Aggregate Functions			
Subquery Expressions			
Row and Array Comparisons			
Set Returning Functions	generate_series		
System Information Functions		*All session information functions* *All access privilege inquiry functions* *All schema visibility inquiry functions* *All system catalog information functions* *All comment information functions*	
System Administration Functions	set_config pg_cancel_backend pg_reload_conf pg_rotate_logfile pg_start_backup pg_stop_backup pg_size_pretty pg_ls_dir pg_read_file pg_stat_file	current_setting *All database object size functions*	**Note:** The function `pg_column_size` displays bytes required to store the value, perhaps with TOAST compression.

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
XML Functions		xmlagg(xml) xmlexists(text, xml) xml_is_well_formed(text) xml_is_well_formed_ document(text) xml_is_well_formed_ content(text) xpath(text, xml) xpath(text, xml, text[]) xpath_exists(text, xml) xpath_exists(text, xml, text[]) xml(text) text(xml) xmlcomment(xml) xmlconcat2(xml, xml)	

Window Functions

The following built-in window functions are Greenplum extensions to the PostgreSQL database. All window functions are *immutable*. For more information about window functions, see "Window Expressions" in the *Greenplum Database Administrator Guide*.

Table 138: Window functions

Function	Return Type	Full Syntax	Description
`cume_dist()`	`double precision`	`CUME_DIST() OVER ( [PARTITION BY` expr `] ORDER BY` expr `)`	Calculates the cumulative distribution of a value in a group of values. Rows with equal values always evaluate to the same cumulative distribution value.
`dense_rank()`	`bigint`	`DENSE_RANK () OVER ( [PARTITION BY` expr `] ORDER BY` expr `)`	Computes the rank of a row in an ordered group of rows without skipping rank values. Rows with equal values are given the same rank value.
`first_ value(`*`expr`*`)`	same as input expr type	`FIRST_VALUE(` expr `) OVER ( [PARTITION BY` expr `] ORDER BY` expr `[ROWS|RANGE` frame_expr `] )`	Returns the first value in an ordered set of values.
`lag(`*`expr`* `[,`*`offset`*`] [,`*`default`*`])`	same as input *expr* type	`LAG(` *expr* `[,` *offset* `] [,` *default* `]) OVER ( [PARTITION BY` *expr* `] ORDER BY` *expr* `)`	Provides access to more than one row of the same table without doing a self join. Given a series of rows returned from a query and a position of the cursor, `LAG` provides access to a row at a given physical offset prior to that position. The default `offset` is 1. *default* sets the value that is returned if the offset goes beyond the scope of the window. If *default* is not specified, the default value is null.
`last_value`*`expr`*	same as input *expr* type	`LAST_VALUE(`*`expr`*`) OVER ( [PARTITION BY` *`expr`*`] ORDER BY` *`expr`* `[ROWS|RANGE` *`frame_expr`*`] )`	Returns the last value in an ordered set of values.

Function	Return Type	Full Syntax	Description
lead(*expr* [,*offset*] [,*default*])	same as input *expr* type	LEAD(*expr* [,*offset*] [,*exprdefault*]) OVER ([PARTITION BY *expr*] ORDER BY *expr*)	Provides access to more than one row of the same table without doing a self join. Given a series of rows returned from a query and a position of the cursor, lead provides access to a row at a given physical offset after that position. If *offset* is not specified, the default offset is 1. *default* sets the value that is returned if the offset goes beyond the scope of the window. If *default* is not specified, the default value is null.
ntile(*expr*)	bigint	NTILE(*expr*) OVER ([PARTITION BY *expr*] ORDER BY *expr*)	Divides an ordered data set into a number of buckets (as defined by *expr*) and assigns a bucket number to each row.
percent_rank()	double precision	PERCENT_RANK () OVER ([PARTITION BY *expr*] ORDER BY *expr*)	Calculates the rank of a hypothetical row R minus 1, divided by 1 less than the number of rows being evaluated (within a window partition).
rank()	bigint	RANK () OVER ([PARTITION BY *expr*] ORDER BY *expr*)	Calculates the rank of a row in an ordered group of values. Rows with equal values for the ranking criteria receive the same rank. The number of tied rows are added to the rank number to calculate the next rank value. Ranks may not be consecutive numbers in this case.
row_number()	bigint	ROW_NUMBER () OVER ([PARTITION BY *expr*] ORDER BY *expr*)	Assigns a unique number to each row to which it is applied (either each row in a window partition or each row of the query).

Advanced Analytic Functions

The following built-in advanced analytic functions are Greenplum extensions of the PostgreSQL database. Analytic functions are *immutable*.

Table 139: Advanced Analytic Functions

Function	Return Type	Full Syntax	Description
`matrix_add(array[], array[])`	`smallint[], int[], bigint[], float[]`	`matrix_add( array[[1,1], [2,2]], array[[3,4], [5,6]])`	Adds two two-dimensional matrices. The matrices must be conformable.
`matrix_multiply( array[], array[])`	`smallint[]int[], bigint[], float[]`	`matrix_multiply( array[[2,0,0], [0,2,0],[0,0,2]], array[[3,0,3], [0,3,0],[0,0,3]] )`	Multiplies two, three-dimensional arrays. The matrices must be conformable.
`matrix_multiply( array[], expr)`	`int[], float[]`	`matrix_multiply( array[[1,1,1], [2,2,2], [3,3,3]], 2)`	Multiplies a two-dimensional array and a scalar numeric value.
`matrix_transpose( array[])`	Same as input `array` type.	`matrix_transpose( array [[1,1,1],[2,2,2]])`	Transposes a two-dimensional array.
`pinv(array [])`	`smallint[]int[], bigint[], float[]`	`pinv(array[[2.5,0,0],[0,1,0], [0,0,.5]])`	Calculates the Moore-Penrose pseudoinverse of a matrix.
`unnest (array[])`	set of `anyelement`	`unnest( array['one', 'row', 'per', 'item'])`	Transforms a one dimensional array into rows. Returns a set of `anyelement`, a polymorphic *pseudotype in PostgreSQL*.

Table 140: Advanced Aggregate Functions

Function	Return Type	Full Syntax	Description

`MEDIAN (expr)`	`timestamp, timestampz, interval, float`	`MEDIAN (expression)`
		Example:
		`SELECT department_id, MEDIAN(salary) FROM employees GROUP BY department_id;`
			Can take a two-dimensional array as input. Treats such arrays as matrices.

Function	Return Type	Full Syntax	Description

`PERCENTILE_CONT (`*`expr`*`) WITHIN GROUP (ORDER BY `*`expr`*` [DESC/ASC])`	`timestamp, timestampz, interval, float`	`PERCENTILE_CONT(`*`percentage`*`) WITHIN GROUP (ORDER BY `*`expression`*`)`
		Example:
		`SELECT department_id, PERCENTILE_CONT (0.5) WITHIN GROUP (ORDER BY salary DESC) "Median_cont"; FROM employees GROUP BY department_id;`
			Performs an inverse distirbution function that assumes a continuous distribution model. It takes a percentile value and a sort specification and returns the same datatype as the numeric datatype of the argument. This returned value is a computed result after performing linear interpolation. Null are ignored in this calculation.

`PERCENTILE_DISC (`*`expr`*`) WITHIN GROUP (ORDER BY `*`expr`*` [DESC/ASC])`	`timestamp, timestampz, interval, float`	`PERCENTILE_DISC(`*`percentage`*`) WITHIN GROUP (ORDER BY `*`expression`*`)`
		Example:
		`SELECT department_id, PERCENTILE_DISC (0.5) WITHIN GROUP (ORDER BY salary DESC) "Median_desc"; FROM employees GROUP BY department_id;`
			Performs an inverse distirbution function that assumes a discrete distribution model. It takes a percentile value and a sort specification. This returned value is an element from the set. Null are ignored in this calculation.

`sum(array[])`	`smallint[]int[], bigint[], float[]`	`sum(array[[1,2],[3,4]])`
		Example:
		`CREATE TABLE mymatrix (myvalue int[]);`
		`INSERT INTO mymatrix VALUES (array[[1,2],[3,4]]);`
		`INSERT INTO mymatrix VALUES (array[[0,1],[1,0]]);`
		`SELECT sum(myvalue) FROM mymatrix;`
		`sum`
		`---------------`
		`{{1,3},{4,4}}`
			Performs matrix summation. Can take as input a two-dimensional array that is treated as a matrix.

`pivot_sum (label[], label, expr)`	`int[], bigint[], float[]`	`pivot_sum( array['A1','A2'], attr, value)`	A pivot aggregation using sum to resolve duplicate entries.

Function	Return Type	Full Syntax	Description
mregr_coef(expr, array[])	float[]	mregr_coef(y, array[1, x1, x2])	The four mregr_* aggregates perform linear regressions using the ordinary-least-squares method. mregr_coef calculates the regression coefficients. The size of the return array for mregr_coef is the same as the size of the input array of independent variables, since the return array contains the coefficient for each independent variable.
mregr_r2 (*expr*, array[])	float	mregr_r2(y, array[1, x1, x2])	The four mregr_* aggregates perform linear regressions using the ordinary-least-squares method. mregr_r2 calculates the r-squared error value for the regression.
mregr_pvalues(*expr*, array[])	float[]	mregr_pvalues(y, array[1, x1, x2])	The four mregr_* aggregates perform linear regressions using the ordinary-least-squares method. mregr_pvalues calculates the p-values for the regression.
mregr_tstats(*expr*, array[])	float[]	mregr_tstats(y, array[1, x1, x2])	The four mregr_* aggregates perform linear regressions using the ordinary-least-squares method. mregr_tstats calculates the t-statistics for the regression.
nb_classify(text[], bigint, bigint[], bigint[])	text	nb_classify(classes, attr_count, class_count, class_total)	Classify rows using a Naive Bayes Classifier. This aggregate uses a baseline of training data to predict the classification of new rows and returns the class with the largest likelihood of appearing in the new rows.

Function	Return Type	Full Syntax	Description
`nb_probabilities(text[], bigint, bigint[], bigint[])`	text	`nb_probabilities(classes, attr_count, class_count, class_total)`	Determine probability for each class using a Naive Bayes Classifier. This aggregate uses a baseline of training data to predict the classification of new rows and returns the probabilities that each class will appear in new rows.

Chapter 11

Greenplum MapReduce Specification

This specification describes the document format and schema for defining Greenplum MapReduce jobs.

MapReduce is a programming model developed by Google for processing and generating large data sets on an array of commodity servers. Greenplum MapReduce allows programmers who are familiar with the MapReduce model to write map and reduce functions and submit them to the Greenplum Database parallel engine for processing.

To enable Greenplum to process MapReduce functions, define the functions in a document, then pass the document to the Greenplum MapReduce program, `gpmapreduce`, for execution by the Greenplum Database parallel engine. The Greenplum Database system distributes the input data, executes the program across a set of machines, handles machine failures, and manages the required inter-machine communication.

See the *Greenplum Database Utility Guide* for information about `gpmapreduce`.

Greenplum MapReduce Document Format

This section explains some basics of the Greenplum MapReduce document format to help you get started creating your own Greenplum MapReduce documents. Greenplum uses the *YAML 1.1* document format and then implements its own schema for defining the various steps of a MapReduce job.

All Greenplum MapReduce files must first declare the version of the YAML specification they are using. After that, three dashes (`---`) denote the start of a document, and three dots (`...`) indicate the end of a document without starting a new one. Comment lines are prefixed with a pound symbol (`#`). It is possible to declare multiple Greenplum MapReduce documents in the same file:

```
%YAML 1.1
---
# Begin Document 1
# ...
---
# Begin Document 2
# ...
```

Within a Greenplum MapReduce document, there are three basic types of data structures or *nodes*: *scalars*, *sequences* and *mappings*.

A *scalar* is a basic string of text indented by a space. If you have a scalar input that spans multiple lines, a preceding pipe (`|`) denotes a *literal* style, where all line breaks are significant. Alternatively, a preceding angle bracket (`>`) folds a single line break to a space for subsequent lines that have the same indentation level. If a string contains characters that have reserved meaning, the string must be quoted or the special character must be escaped with a backslash (`\`).

```
# Read each new line literally
somekey: |   this value contains two lines
   and each line is read literally
# Treat each new line as a space
anotherkey: >
   this value contains two lines
   but is treated as one continuous line
# This quoted string contains a special character
ThirdKey: "This is a string: not a mapping"
```

A *sequence* is a list with each entry in the list on its own line denoted by a dash and a space (`- `). Alternatively, you can specify an inline sequence as a comma-separated list within square brackets. A sequence provides a set of data and gives it an order. When you load a list into the Greenplum MapReduce program, the order is kept.

```
# list sequence
- this
- is
- a list
- with
- five scalar values
# inline sequence
[this, is, a list, with, five scalar values]
```

A *mapping* is used to pair up data values with indentifiers called *keys*. Mappings use a colon and space (`: `) for each `key: value` pair, or can also be specified inline as a comma-separated list within curly braces. The *key* is used as an index for retrieving data from a mapping.

```
# a mapping of items
title: War and Peace
author: Leo Tolstoy
date: 1865
# same mapping written inline
```

```
{title: War and Peace, author: Leo Tolstoy, date: 1865}
```

Keys are used to associate meta information with each node and specify the expected node type (*scalar*, *sequence* or *mapping*). See *Greenplum MapReduce Document Schema* for the keys expected by the Greenplum MapReduce program.

The Greenplum MapReduce program processes the nodes of a document in order and uses indentation (spaces) to determine the document hierarchy and the relationships of the nodes to one another. The use of white space is significant. White space should not be used simply for formatting purposes, and tabs should not be used at all.

Greenplum MapReduce Document Schema

Greenplum MapReduce uses the YAML document framework and implements its own YAML schema. The basic structure of a Greenplum MapReduce document is:

```
%YAML 1.1
---
VERSION: 1.0.0.2
DATABASE: dbname
USER: db_username
HOST: master_hostname
PORT: master_port
```

```
DEFINE:
  - INPUT:
     NAME: input_name
     FILE:
      - hostname:/path/to/file
     GPFDIST:
       - hostname:port/file_pattern
     TABLE: table_name
     QUERY: SELECT_statement
     EXEC: command_string
     COLUMNS:
       - field_name data_type
     FORMAT: TEXT | CSV
     DELIMITER: delimiter_character
     ESCAPE: escape_character
     NULL: null_string
     QUOTE: csv_quote_character
     ERROR_LIMIT: integer
     ENCODING: database_encoding
```

```
  - OUTPUT:
     NAME: output_name
     FILE: file_path_on_client
     TABLE: table_name
     KEYS:
       - column_name
     MODE: REPLACE | APPEND
```

```
  - MAP:
     NAME: function_name
     FUNCTION: function_definition
     LANGUAGE: perl | python | c
     LIBRARY: /path/filename.so
     PARAMETERS:
       - nametype
     RETURNS:
       - nametype
     OPTIMIZE: STRICT IMMUTABLE
     MODE: SINGLE | MULTI
```

```
  - TRANSITION | CONSOLIDATE | FINALIZE:
     NAME: function_name
     FUNCTION: function_definition
     LANGUAGE: perl | python | c
     LIBRARY: /path/filename.so
     PARAMETERS:
       - nametype
     RETURNS:
       - nametype
     OPTIMIZE: STRICT IMMUTABLE
```

```
      MODE: SINGLE | MULTI
```

```
   - REDUCE:
      NAME: reduce_job_name
      TRANSITION: transition_function_name
      CONSOLIDATE: consolidate_function_name
      FINALIZE: finalize_function_name
      INITIALIZE: value
      KEYS:
        - key_name
```

```
   - TASK:
      NAME: task_name
      SOURCE: input_name
      MAP: map_function_name
      REDUCE: reduce_function_name
EXECUTE
```

```
   - RUN:
      SOURCE: input_or_task_name
      TARGET: output_name
      MAP: map_function_name
      REDUCE: reduce_function_name...
```

VERSION

Required. The version of the Greenplum MapReduce YAML specification. Current versions are 1.0.0.1.

DATABASE

Optional. Specifies which database in Greenplum to connect to. If not specified, defaults to the default database or `$PGDATABASE` if set.

USER

Optional. Specifies which database role to use to connect. If not specified, defaults to the current user or `$PGUSER` if set. You must be a Greenplum superuser to run functions written in untrusted Python and Perl. Regular database users can run functions written in trusted Perl. You also must be a database superuser to run MapReduce jobs that contain *FILE*, *GPFDIST* and *EXEC* input types.

HOST

Optional. Specifies Greenplum master host name. If not specified, defaults to localhost or `$PGHOST` if set.

PORT

Optional. Specifies Greenplum master port. If not specified, defaults to 5432 or `$PGPORT` if set.

DEFINE

Required. A sequence of definitions for this MapReduce document. The `DEFINE` section must have at least one `INPUT` definition.

INPUT

Required. Defines the input data. Every MapReduce document must have at least one input defined. Multiple input definitions are allowed in a document, but each input definition can specify only one of these access types:a file, a `gpfdist` file distribution program, a table in the database, an SQL command, or an operating system command. See the *Greenplum Database Utility Guide* for information about `gpfdist`.

NAME

A name for this input. Names must be unique with regards to the names of other objects in this MapReduce job (such as map function, task, reduce function and output names). Also, names cannot conflict with existing objects in the database (such as tables, functions or views).

FILE

A sequence of one or more input files in the format: `seghostname:/path/to/filename`. You must be a Greenplum Database superuser to run MapReduce jobs with `FILE` input. The file must reside on a Greenplum segment host.

GPFDIST

A sequence of one or more running `gpfdist` file distribution programs in the format: `hostname[:port]/file_pattern`. You must be a Greenplum Database superuser to run MapReduce jobs with `GPFDIST` input, unless the server configuration parameter *Server Configuration Parameters* is set to `on`.

TABLE

The name of an existing table in the database.

QUERY

A SQL `SELECT` command to run within the database.

EXEC

An operating system command to run on the Greenplum segment hosts. The command is run by all segment instances in the system by default. For example, if you have four segment instances per segment host, the command will be run four times on each host. You must be a Greenplum Database superuser to run MapReduce jobs with `EXEC` input and the server configuration parameter *Server Configuration Parameters* is set to `on`.

COLUMNS

Optional. Columns are specified as: `column_name [data_type]`. If not specified, the default is `value text`. The *DELIMITER* character is what separates two data value fields (columns). A row is determined by a line feed character (`0x0a`).

FORMAT

Optional. Specifies the format of the data - either delimited text (`TEXT`) or comma separated values (`CSV`) format. If the data format is not specified, defaults to `TEXT`.

DELIMITER

Optional for *FILE*, *GPFDIST* and *EXEC* inputs. Specifies a single character that separates data values. The default is a tab character in `TEXT` mode, a comma in `CSV` mode.The delimiter character must only appear between any two data value fields. Do not place a delimiter at the beginning or end of a row.

ESCAPE

Optional for *FILE*, *GPFDIST* and *EXEC* inputs. Specifies the single character that is used for C escape sequences (such as `\n`,`\t`,`\100`, and so on) and for escaping data characters that might otherwise be taken as row or column delimiters. Make sure to choose an escape character that is not used anywhere in your actual column data. The default escape character is a \ (backslash) for text-formatted files and a `"` (double quote) for csv-formatted files, however it is possible to specify another character to represent an escape. It is also possible to disable escaping by specifying the value `'OFF'` as the escape value. This is very useful for data such as text-formatted web log data that has many embedded backslashes that are not intended to be escapes.

NULL

Optional for *FILE*, *GPFDIST* and *EXEC* inputs. Specifies the string that represents a null value. The default is `\N` in `TEXT` format, and an empty value with no quotations in `CSV` format. You might prefer an empty string even in `TEXT` mode for cases where you do not want to distinguish nulls from empty strings. Any input data item that matches this string will be considered a null value.

QUOTE

Optional for *FILE*, *GPFDIST* and *EXEC* inputs. Specifies the quotation character for `CSV` formatted files. The default is a double quote (`"`). In `CSV` formatted files, data value fields must be enclosed in double quotes if they contain any commas or embedded new lines. Fields that contain double quote characters must be surrounded by double quotes, and the embedded double quotes must each be represented by a pair of consecutive double quotes. It is important to always open and close quotes correctly in order for data rows to be parsed correctly.

ERROR_LIMIT

If the input rows have format errors they will be discarded provided that the error limit count is not reached on any Greenplum segment instance during input processing. If the error limit is not reached, all good rows will be processed and any error rows discarded.

ENCODING

Character set encoding to use for the data. Specify a string constant (such as `'SQL_ASCII'`), an integer encoding number, or `DEFAULT` to use the default client encoding. See *Character Set Support* for more information.

OUTPUT

Optional. Defines where to output the formatted data of this MapReduce job. If output is not defined, the default is `STDOUT` (standard output of the client). You can send output to a file on the client host or to an existing table in the database.

NAME

A name for this output. The default output name is `STDOUT`. Names must be unique with regards to the names of other objects in this MapReduce job (such as map function, task, reduce function and input names). Also, names cannot conflict with existing objects in the database (such as tables, functions or views).

FILE

Specifies a file location on the MapReduce client machine to output data in the format: `/path/to/filename`.

TABLE

Specifies the name of a table in the database to output data. If this table does not exist prior to running the MapReduce job, it will be created using the distribution policy specified with *KEYS*.

KEYS

Optional for *TABLE* output. Specifies the column(s) to use as the Greenplum Database distribution key. If the *EXECUTE* task contains a *REDUCE* definition, then the `REDUCE` keys will be used as the table distribution key by default. Otherwise, the first column of the table will be used as the distribution key.

MODE

Optional for *TABLE* output. If not specified, the default is to create the table if it does not already exist, but error out if it does exist. Declaring `APPEND` adds output data to an existing table (provided the table schema matches the output format) without removing any existing data. Declaring `REPLACE` will drop the table if it exists and then recreate it. Both `APPEND` and `REPLACE` will create a new table if one does not exist.

MAP

Required. Each `MAP` function takes data structured in (`key`, `value`) pairs, processes each pair, and generates zero or more output (`key`, `value`) pairs. The Greenplum MapReduce framework then collects all pairs with the same key from all output lists and groups them together. This output is then passed to the *REDUCE* task, which is comprised of *TRANSITION | CONSOLIDATE | FINALIZE* functions. There is one predefined `MAP` function named `IDENTITY` that returns (`key`, `value`) pairs unchanged. Although (`key`, `value`) are the default parameters, you can specify other prototypes as needed.

TRANSITION | CONSOLIDATE | FINALIZE

`TRANSITION`, `CONSOLIDATE` and `FINALIZE` are all component pieces of *REDUCE*. A `TRANSITION` function is required. `CONSOLIDATE` and `FINALIZE` functions are optional. By default, all take `state` as the first of their input *PARAMETERS*, but other prototypes can be defined as well.

A `TRANSITION` function iterates through each value of a given key and accumulates values in a `state` variable. When the transition function is called on the first value of a key, the `state` is set to the value specified by *INITALIZE* of a *REDUCE* job (or the default state value for the data type). A transition takes two arguments as input; the current state of the key reduction, and the next value, which then produces a new `state`.

If a `CONSOLIDATE` function is specified, `TRANSITION` processing is performed at the segment-level before redistributing the keys across the Greenplum interconnect for final aggregation (two-phase aggregation). Only the resulting `state` value for a given key is redistributed, resulting in lower interconnect traffic and greater parallelism. `CONSOLIDATE` is handled like a `TRANSITION`, except that instead of `(state + value) => state`, it is `(state + state) => state`.

If a `FINALIZE` function is specified, it takes the final `state` produced by `CONSOLIDATE` (if present) or `TRANSITION` and does any final processing before emitting the final result. `TRANSITION` and `CONSOLIDATE` functions cannot return a set of values. If you need a `REDUCE` job to return a set, then a `FINALIZE` is necessary to transform the final state into a set of output values.

NAME

Required. A name for the function. Names must be unique with regards to the names of other objects in this MapReduce job (such as function, task, input and output names). You can also specify the name of a function built-in to Greenplum Database. If using a built-in function, do not supply *LANGUAGE* or a *FUNCTION* body.

FUNCTION

Optional. Specifies the full body of the function using the specified *LANGUAGE*. If `FUNCTION` is not specified, then a built-in database function corresponding to *NAME* is used.

LANGUAGE

Required when *FUNCTION* is used. Specifies the implementation language used to interpret the function. This release has language support for `perl`, `python`, and `C`. If calling a built-in database function, `LANGUAGE` should not be specified.

LIBRARY

Required when *LANGUAGE* is C (not allowed for other language functions). To use this attribute, *VERSION* must be 1.0.0.2. The specified library file must be installed prior to running the MapReduce job, and it must exist in the same file system location on all Greenplum hosts (master and segments).

PARAMETERS

Optional. Function input parameters. The default type is `text`.

`MAP` default - `key text`, `value text`

`TRANSITION` default - `state text`, `value text`

`CONSOLIDATE` default - `state1 text`, `state2 text` (must have exactly two input parameters of the same data type)

`FINALIZE` default - `state text` (single parameter only)

RETURNS

Optional. The default return type is `text`.

`MAP` default - `key text`, `value text`

`TRANSITION` default - `state text` (single return value only)

`CONSOLIDATE` default - `state text` (single return value only)

`FINALIZE` default - `value text`

OPTIMIZE

Optional optimization parameters for the function:

`STRICT` - function is not affected by `NULL` values

`IMMUTABLE` - function will always return the same value for a given input

MODE

Optional. Specifies the number of rows returned by the function.

`MULTI` - returns 0 or more rows per input record. The return value of the function must be an array of rows to return, or the function must be written as an iterator using `yield` in Python or `return_next` in Perl. `MULTI` is the default mode for `MAP` and `FINALIZE` functions.

`SINGLE` - returns exactly one row per input record. `SINGLE` is the only mode supported for `TRANSITION` and `CONSOLIDATE` functions. When used with `MAP` and `FINALIZE` functions, `SINGLE` mode can provide modest performance improvement.

REDUCE

Required. A `REDUCE` definition names the *TRANSITION* | *CONSOLIDATE* | *FINALIZE* functions that comprise the reduction of (`key`, `value`) pairs to the final result set. There are also several predefined `REDUCE` jobs you can execute, which all operate over a column named `value`:

`IDENTITY` - returns (key, value) pairs unchanged

`SUM` - calculates the sum of numeric data

`AVG` - calculates the average of numeric data

`COUNT` - calculates the count of input data

`MIN` - calculates minimum value of numeric data

`MAX` - calculates maximum value of numeric data

NAME

Required. The name of this `REDUCE` job. Names must be unique with regards to the names of other objects in this MapReduce job (function, task, input and output names). Also, names cannot conflict with existing objects in the database (such as tables, functions or views).

TRANSITION

Required. The name of the `TRANSITION` function.

CONSOLIDATE

Optional. The name of the `CONSOLIDATE` function.

FINALIZE

Optional. The name of the `FINALIZE` function.

INITIALIZE

Optional for `text` and `float` data types. Required for all other data types. The default value for text is `''` . The default value for float is `0.0` . Sets the initial `state` value of the `TRANSITION` function.

KEYS

Optional. Defaults to `[key, *]`. When using a multi-column reduce it may be necessary to specify which columns are key columns and which columns are value columns. By default, any input columns that are not passed to the `TRANSITION` function are key columns, and a column named `key` is always a key column even if it is passed to the `TRANSITION` function. The special indicator `*` indicates all columns not passed to the `TRANSITION` function. If this indicator is not present in the list of keys then any unmatched columns are discarded.

TASK

Optional. A `TASK` defines a complete end-to-end `INPUT`/`MAP`/`REDUCE` stage within a Greenplum MapReduce job pipeline. It is similar to *EXECUTE* except it is not immediately executed. A task object can be called as *INPUT* to further processing stages.

NAME

Required. The name of this task. Names must be unique with regards to the names of other objects in this MapReduce job (such as map function, reduce function, input and output names). Also, names cannot conflict with existing objects in the database (such as tables, functions or views).

SOURCE

The name of an *INPUT* or another `TASK`.

MAP

Optional. The name of a *MAP* function. If not specified, defaults to `IDENTITY`.

REDUCE

Optional. The name of a *REDUCE* function. If not specified, defaults to `IDENTITY`.

EXECUTE

Required. `EXECUTE` defines the final `INPUT`/`MAP`/`REDUCE` stage within a Greenplum MapReduce job pipeline.

RUN

SOURCE

Required. The name of an *INPUT* or *TASK*.

TARGET

Optional. The name of an *OUTPUT*. The default output is `STDOUT`.

MAP

Optional. The name of a *MAP* function. If not specified, defaults to `IDENTITY`.

REDUCE

Optional. The name of a *REDUCE* function. Defaults to `IDENTITY`.

Example Greenplum MapReduce Document

```
# This example MapReduce job processes documents and looks for keywords in them.
# It takes two database tables as input:
#   - documents (doc_id integer, url text, data text)
# - keywords (keyword_id integer, keyword text)#
# The documents data is searched for occurences of keywords and returns results of
# url, data and keyword (a keyword can be multiple words, such as "high performance #
 computing")
%YAML 1.1
---
VERSION:1.0.0.1

# Connect to Greenplum Database using this database and role
DATABASE:webdata
USER:jsmith

# Begin definition section
DEFINE:

 # Declare the input, which selects all columns and rows from the
 # 'documents' and 'keywords' tables.
- INPUT:
NAME:doc
TABLE:documents
- INPUT:
NAME:kw
TABLE:keywords
# Define the map functions to extract terms from documents and keyword
# This example simply splits on white space, but it would be possible
# to make use of a python library like nltk (the natural language toolkit)
# to perform more complex tokenization and word stemming.
 - MAP:
NAME:doc_map
 LANGUAGE:python
 FUNCTION:|
        i = 0            # the index of a word within the document
terms = {}# a hash of terms and their indexes within the document

# Lower-case and split the text string on space
for term in data.lower().split():
i = i + 1# increment i (the index)

        # Check for the term in the terms list:
        # if stem word already exists, append the i value to the array entry
        # corresponding to the term. This counts multiple occurances of the word.
        # If stem word does not exist, add it to the dictionary with position i.
        # For example:
 # data: "a computer is a machine that manipulates data"
 # "a" [1, 4]
 # "computer" [2]
 # "machine" [3]
 # …
          if term in terms:
            terms[term] += ','+str(i)
          else:
            terms[term] = str(i)

# Return multiple lines for each document. Each line consists of
# the doc_id, a term and the positions in the data where the term appeared.
        # For example:
        #   (doc_id => 100, term => "a", [1,4]
        #   (doc_id => 100, term => "computer", [2]
        #    …
for term in terms:
yield([doc_id, term, terms[term]])
      OPTIMIZE:STRICT IMMUTABLE
```

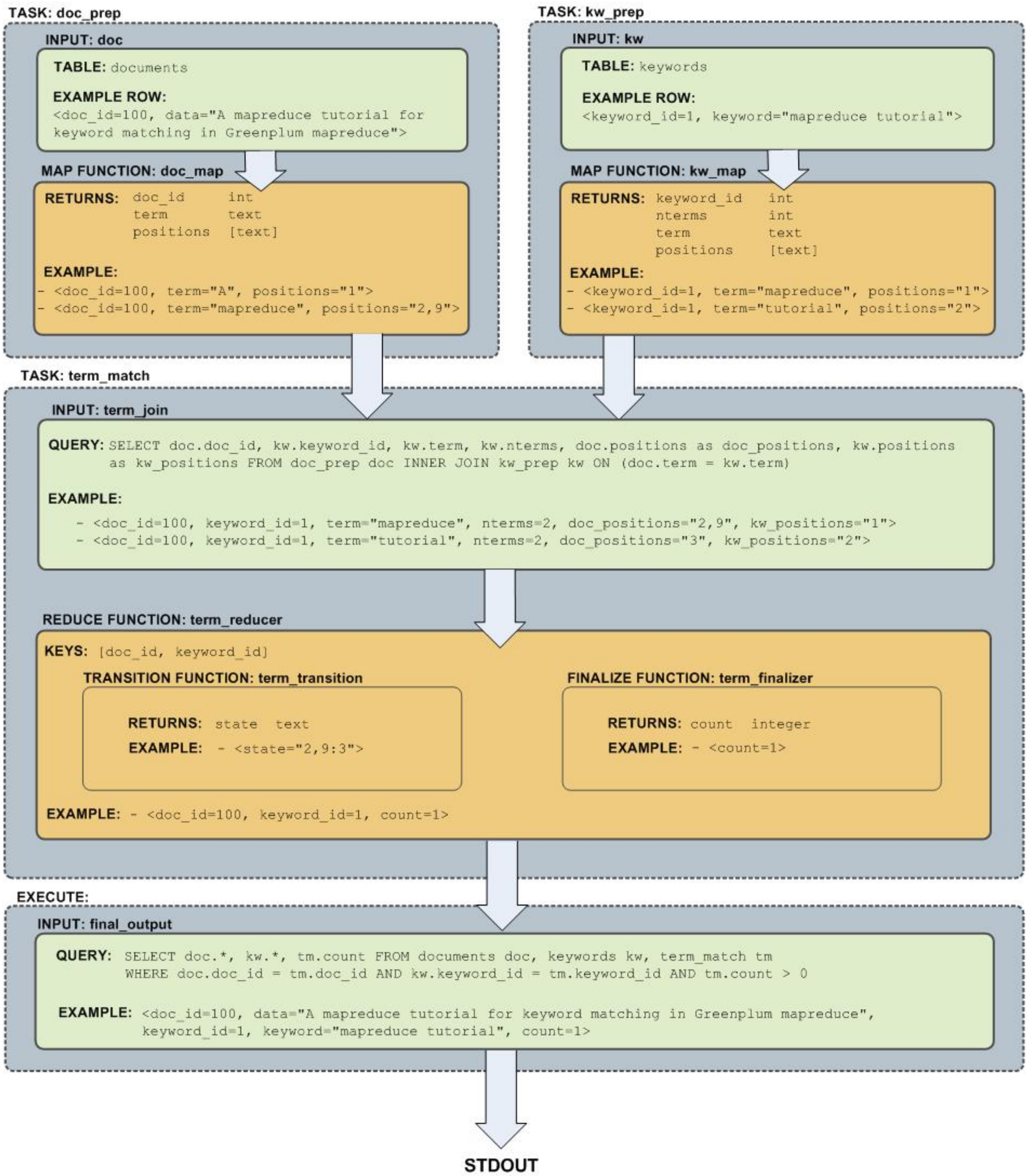
```
      PARAMETERS:
- doc_id integer
        - data text
RETURNS:
- doc_id integer
        - term text
        - positions text

  # The map function for keywords is almost identical to the one for documents
 # but it also counts of the number of terms in the keyword.
 - MAP:
NAME:kw_map
LANGUAGE:python
FUNCTION:|
        i = 0
        terms = {}
        for term in keyword.lower().split():
          i = i + 1
          if term in terms:
            terms[term] += ','+str(i)
          else:
            terms[term] = str(i)

 # output 4 values including i (the total count for term in terms):
yield([keyword_id, i, term, terms[term]])
      OPTIMIZE:STRICT IMMUTABLE
PARAMETERS:
- keyword_id integer
        - keyword text
RETURNS:
- keyword_id integer
        - nterms integer
        - term text
        - positions text

# A TASK is an object that defines an entire INPUT/MAP/REDUCE stage
# within a Greenplum MapReduce pipeline. It is like EXECUTION, but it is
# executed only when called as input to other processing stages.
# Identify a task called 'doc_prep' which takes in the 'doc' INPUT defined earlier
# and runs the 'doc_map' MAP function which returns doc_id, term, [term_position]
- TASK:
NAME:doc_prep
SOURCE:doc
MAP:doc_map

# Identify a task called 'kw_prep' which takes in the 'kw' INPUT defined earlier
# and runs the kw_map MAP function which returns kw_id, term, [term_position]
- TASK:
NAME:kw_prep
SOURCE:kw
MAP:kw_map

# One advantage of Greenplum MapReduce is that MapReduce tasks can be
# used as input to SQL operations and SQL can be used to process a MapReduce task.
# This INPUT defines a SQL query that joins the output of the 'doc_prep'
# TASK to that of the 'kw_prep' TASK. Matching terms are output to the 'candidate'
# list (any keyword that shares at least one term with the document).
- INPUT:
NAME: term_join
QUERY: |
        SELECT doc.doc_id, kw.keyword_id, kw.term, kw.nterms,
               doc.positions as doc_positions,
               kw.positions as kw_positions
          FROM doc_prep doc INNER JOIN kw_prep kw ON (doc.term = kw.term)

# In Greenplum MapReduce, a REDUCE function is comprised of one or more functions.
# A REDUCE has an initial 'state' variable defined for each grouping key. that is
# A TRANSITION function adjusts the state for every value in a key grouping.
# If present, an optional CONSOLIDATE function combines multiple
# 'state' variables. This allows the TRANSITION function to be executed locally at
# the segment-level and only redistribute the accumulated 'state' over
```

```
# the network. If present, an optional FINALIZE function can be used to perform
# final computation on a state and emit one or more rows of output from the state.
#
# This REDUCE function is called 'term_reducer' with a TRANSITION function
# called 'term_transition' and a FINALIZE function called 'term_finalizer'
- REDUCE:
NAME:term_reducer
TRANSITION:term_transition
FINALIZE:term_finalizer

- TRANSITION:
NAME:term_transition
 LANGUAGE:python
 PARAMETERS:
- state text
        - term text
        - nterms integer
        - doc_positions text
        - kw_positions text
FUNCTION: |

 # 'state' has an initial value of '' and is a colon delimited set
        # of keyword positions. keyword positions are comma delimited sets of
        # integers. For example, '1,3,2:4:'
        # If there is an existing state, split it into the set of keyword positions
 # otherwise construct a set of 'nterms' keyword positions - all empty
if state:
          kw_split = state.split(':')
        else:
          kw_split = []
          for i in range(0,nterms):
            kw_split.append('')

        # 'kw_positions' is a comma delimited field of integers indicating what
 # position a single term occurs within a given keyword.
        # Splitting based on ',' converts the string into a python list.
 # add doc_positions for the current term
for kw_p in kw_positions.split(','):
          kw_split[int(kw_p)-1] = doc_positions

        # This section takes each element in the 'kw_split' array and strings
        # them together placing a ':' in between each element from the array.
 # For example: for the keyword "computer software computer hardware",
        # the 'kw_split' array matched up to the document data of
        # "in the business of computer software software engineers"
        # would look like: ['5', '6,7', '5', '']
 # and the outstate would look like: 5:6,7:5:
outstate = kw_split[0]
        for s in kw_split[1:]:
          outstate = outstate + ':' + s
        return outstate

   - FINALIZE:
NAME: term_finalizer
LANGUAGE: python
RETURNS:
        - count integer
      MODE:MULTI
      FUNCTION:|
        if not state:
          return 0
        kw_split = state.split(':')

        # This function does the following:
 # 1) Splits 'kw_split' on ':'
        #    for example, 1,5,7:2,8 creates '1,5,7' and '2,8'
 # 2) For each group of positions in 'kw_split', splits the set on ','
        #    to create ['1','5','7'] from Set 0: 1,5,7 and
        #    eventually ['2', '8'] from Set 1: 2,8
 # 3) Checks for empty strings
 # 4) Adjusts the split sets by subtracting the position of the set
```

```
        #      in the 'kw_split' array
 # ['1','5','7'] - 0 from each element = ['1','5','7']
 # ['2', '8'] - 1 from each element = ['1', '7']
        # 5) Resulting arrays after subtracting the offset in step 4 are
 #    intersected and their overlaping values kept:
        #    ['1','5','7'].intersect['1', '7'] = [1,7]
        # 6) Determines the length of the intersection, which is the number of
 # times that an entire keyword (with all its pieces) matches in the
 #    document data.
previous = None
        for i in range(0,len(kw_split)):
          isplit = kw_split[i].split(',')
          if any(map(lambda(x): x == '', isplit)):
            return 0
          adjusted = set(map(lambda(x): int(x)-i, isplit))
          if (previous):
            previous = adjusted.intersection(previous)
          else:
            previous = adjusted

        # return the final count
if previous:
          return len(previous)

   # Define the 'term_match' task which is then executed as part
   # of the 'final_output' query. It takes the INPUT 'term_join' defined
   # earlier and uses the REDUCE function 'term_reducer' defined earlier
 - TASK:
NAME:term_match
SOURCE:term_join
REDUCE:term_reducer
- INPUT:
NAME:final_output
QUERY:|
        SELECT doc.*, kw.*, tm.count
        FROM documents doc, keywords kw, term_match tm
        WHERE doc.doc_id = tm.doc_id
          AND kw.keyword_id = tm.keyword_id
          AND tm.count > 0

# Execute this MapReduce job and send output to STDOUT
EXECUTE:
 - RUN:
SOURCE:final_output
TARGET:STDOUT
```

Flow Diagram for MapReduce Example

The following diagram shows the job flow of the MapReduce job defined in the example:


TASK: doc_prep
INPUT: doc
TABLE: documents
EXAMPLE ROW:
<doc_id=100, data="A mapreduce tutorial for keyword matching in Greenplum mapreduce">
MAP FUNCTION: doc_map
RETURNS: doc_id int
term text
positions [text]
EXAMPLE:
- <doc_id=100, term="A", positions="1">
- <doc_id=100, term="mapreduce", positions="2,9">
TASK: kw_prep
INPUT: kw
TABLE: keywords
EXAMPLE ROW:
<keyword_id=1, keyword="mapreduce tutorial">
MAP FUNCTION: kw_map
RETURNS: keyword_id int
nterms int
term text
positions [text]
EXAMPLE:
- <keyword_id=1, term="mapreduce", positions="1">
- <keyword_id=1, term="tutorial", positions="2">
TASK: term_match
INPUT: term_join
QUERY: SELECT doc.doc_id, kw.keyword_id, kw.term, kw.nterms, doc.positions as doc_positions, kw.positions as kw_positions FROM doc_prep doc INNER JOIN kw_prep kw ON (doc.term = kw.term)
EXAMPLE:
- <doc_id=100, keyword_id=1, term="mapreduce", nterms=2, doc_positions="2,9", kw_positions="1">
- <doc_id=100, keyword_id=1, term="tutorial", nterms=2, doc_positions="3", kw_positions="2">
REDUCE FUNCTION: term_reducer
KEYS: [doc_id, keyword_id]
TRANSITION FUNCTION: term_transition
RETURNS: state text
EXAMPLE: - <state="2,9:3">
FINALIZE FUNCTION: term_finalizer
RETURNS: count integer
EXAMPLE: - <count=1>
EXAMPLE: - <doc_id=100, keyword_id=1, count=1>
EXECUTE:
INPUT: final_output
QUERY: SELECT doc.*, kw.*, tm.count FROM documents doc, keywords kw, term_match tm WHERE doc.doc_id = tm.doc_id AND kw.keyword_id = tm.keyword_id AND tm.count > 0
EXAMPLE: <doc_id=100, data="A mapreduce tutorial for keyword matching in Greenplum mapreduce", keyword_id=1, keyword="mapreduce tutorial", count=1>
STDOUT

Chapter 12

Greenplum PL/pgSQL Procedural Language

This section contains an overview of the Greenplum Database PL/pgSQL language.

About Greenplum Database PL/pgSQL

Greenplum Database PL/pgSQL is a loadable procedural language that is installed and registered by default with Greenplum Database. You can create user-defined functions using SQL statements, functions, and operators.

With PL/pgSQL you can group a block of computation and a series of SQL queries inside the database server, thus having the power of a procedural language and the ease of use of SQL. Also, with PL/pgSQL you can use all the data types, operators and functions of Greenplum Database SQL.

The PL/pgSQL language is a subset of Oracle PL/SQL. Greenplum Database PL/pgSQL is based on Postgres PL/pgSQL. The Postgres PL/pgSQL documentation is at *http://www.postgresql.org/docs/8.2/static/plpgsql.html*

When using PL/pgSQL functions, function attributes affect how Greenplum Database creates query plans. You can specify the attribute `IMMUTABLE`, `STABLE`, or `VOLATILE` as part of the `LANGUAGE` clause to classify the type of function, For information about the creating functions and function attributes, see the *`CREATE FUNCTION`* command in the *Greenplum Database Reference Guide*.

Greenplum Database SQL Limitations

When using Greenplum Database PL/pgSQL, limitations include

- Triggers are not supported
- Cursors are forward moving only (not scrollable)

For information about Greenplum Database SQL conformance, see *Summary of Greenplum Features* in the *Greenplum Database Reference Guide*.

The PL/pgSQL Language

PL/pgSQL is a block-structured language. The complete text of a function definition must be a block. A block is defined as:

```
[ label ]
[ DECLARE
   declarations ]
BEGIN
   statements
END [ label ];
```

Each declaration and each statement within a block is terminated by a semicolon (;). A block that appears within another block must have a semicolon after `END`, as shown in the previous block. The `END` that concludes a function body does not require a semicolon.

> **Important:** Do not confuse the use of the `BEGIN` and `END` keywords for grouping statements in PL/pgSQL with the database commands for transaction control. The PL/pgSQL `BEGIN` and `END` keywords are only for grouping; they do not start or end a transaction. Functions are always executed within a transaction established by an outer query — they cannot start or commit that transaction, since there would be no context for them to execute in. However, a PL/pgSQL block that contains an `EXCEPTION` clause effectively forms a subtransaction that can be rolled back without affecting the outer transaction. For more about the `EXCEPTION` clause, see the post the Postgres documentation on error trapping at *http://www.postgresql.org/docs/8.2/static/plpgsql-control-structures.html#PLPGSQL-ERROR-TRAPPING*.

All key words and identifiers can be written in mixed upper and lower case. Identifiers are implicitly converted to lowercase unless enclosed in double-quotes (").

You can add comments in PL/pgSQL in the following ways:

- A double dash (--) starts a comment that extends to the end of the line.
- A /* starts a block comment that extends to the next occurrence of */.

  Block comments cannot be nested, but double dash comments can be enclosed into a block comment and a double dash can hide the block comment delimiters /* and */.

Any statement in the statement section of a block can be a subblock. Subblocks can be used for logical grouping or to localize variables to a small group of statements.

The variables declared in the declarations section preceding a block are initialized to their default values every time the block is entered, not only once per function call. For example declares the variable `quantity` several times:

```
CREATE FUNCTION testfunc() RETURNS integer AS $$
DECLARE
   quantity integer := 30;
BEGIN
   RAISE NOTICE 'Quantity here is %', quantity;
   -- Quantity here is 30
   quantity := 50;
   --
   -- Create a subblock
   --
   DECLARE
      quantity integer := 80;
   BEGIN
      RAISE NOTICE 'Quantity here is %', quantity;
      -- Quantity here is 80
   END;
   RAISE NOTICE 'Quantity here is %', quantity;
   -- Quantity here is 50
   RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

Executing SQL Commands

You can execute SQL commands with PL/gpSQL statements such as `EXECUTE`, `PERFORM`, and `SELECT ... INTO`. For information about the PL/gpSQL statements, see *http://www.postgresql.org/docs/8.2/static/plpgsql-statements.html*.

Note: The PL/gpSQL statement `SELECT INTO` is not supported in the `EXECUTE` statement.

PL/pgSQL Examples

The following are examples of PL/pgSQL user-defined functions.

Example: Aliases for Function Parameters

Parameters passed to functions are named with the identifiers such asa `$1`, `$2`. Optionally, aliases can be declared for `$n` parameter names for increased readability. Either the alias or the numeric identifier can then be used to refer to the parameter value.

There are two ways to create an alias. The preferred way is to give a name to the parameter in the `CREATE FUNCTION` command, for example:

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
   RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

You can also explicitly declare an alias, using the declaration syntax:

```
name ALIAS FOR $n;
```

This example, creates the same function with the `DECLARE` syntax.

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Example: Using the Data Type of a Table Column

When declaring a variable, you can use `%TYPE` to specify the data type of a variable or table column. This is the syntax for declaring a variable with the data type of a table column:

```
name table.column_name%TYPE;
```

You can use this to declare variables that will hold database values. For example, if you have a column named `user_id` in your `users` table. To declare the variable `my_userid` with the same data type as the `users.user_id` column:

```
my_userid users.user_id%TYPE;
```

`%TYPE` is particularly valuable in polymorphic functions, since the data types needed for internal variables may change from one call to the next. Appropriate variables can be created by applying `%TYPE` to the function's arguments or result placeholders.

Example: Composite Type Based on a Table Row

The following syntax declares a composite variable based on table row:

```
name table_name%ROWTYPE;
```

Such a row variable can hold a whole row of a `SELECT` or `FOR` query result, so long as that query column set matches the declared type of the variable. The individual fields of the row value are accessed using the usual dot notation, for example `rowvar.column`.

Parameters to a function can be composite types (complete table rows). In that case, the corresponding identifier `$n` will be a row variable, and fields can be selected from it, for `example $1.user_id`.

Only the user-defined columns of a table row are accessible in a row-type variable, not the OID or other system columns. The fields of the row type inherit the table's field size or precision for data types such as `char(n)`.

The next example function uses a row variable composite type. Before creating the function, create the table that is used by the function with this command.

```
CREATE TABLE table1 (
  f1 text,
  f2 numeric,
  f3 integer
) distributed by (f1);
```

This `INSERT` command adds data to the table.

```
INSERT INTO table1 values
 ('test1', 14.1, 3),
 ('test2', 52.5, 2),
 ('test3', 32.22, 6),
 ('test4', 12.1, 4) ;
```

This function uses a variable and `ROWTYPE` composite variable based on `table1`.

```
CREATE OR REPLACE FUNCTION t1_calc( name text) RETURNS integer
AS $$
DECLARE
    t1_row   table1%ROWTYPE;
    calc_int table1.f3%TYPE;
BEGIN
    SELECT * INTO t1_row FROM table1 WHERE table1.f1 = $1 ;
    calc_int = (t1_row.f2 * t1_row.f3)::integer ;
    RETURN calc_int ;
END;
$$ LANGUAGE plpgsql VOLATILE;
```

> **Note:** The previous function is classified as a `VOLATILE` function because function values could change within a single table scan.

The following `SELECT` command uses the function.

```
select t1_calc( 'test1' );
```

> **Note:** The example PL/pgSQL function uses `SELECT` with the `INTO` clause. It is different from the SQL command `SELECT INTO`. If you want to create a table from a `SELECT` result inside a PL/pgSQL function, use the SQL command `CREATE TABLE AS`.

References

The Postgres documentation about PL/pgSQL is at *http://www.postgresql.org/docs/8.2/static/plpgsql.html*

Also, see the `CREATE FUNCTION` command in the *Greenplum Database Reference Guide*.

For a summary of built-in Greenplum Database functions, see *Summary of Built-in Functions* in the *Greenplum Database Reference Guide*. For information about using Greenplum Database functions see "Querying Data" in the *Greenplum Database Administrator Guide*

For information about porting Oracle functions, see *http://www.postgresql.org/docs/8.2/static/plpgsql-porting.html*. For information about installing and using the Oracle compatibility functions with Greenplum Database, see "Oracle Compatibility Functions" in the *Greenplum Database Utility Guide*.

Chapter 13

Greenplum PostGIS Extension

This chapter contains the following information:

About PostGIS

PostGIS is a spatial database extension for PostgreSQL that allows GIS (Geographic Information Systems) objects to be stored in the database. The Greenplum Database PostGIS extension includes support for GiST-based R-Tree spatial indexes and functions for analysis and processing of GIS objects.

Go to *http://postgis.refractions.net/* for more information about PostGIS.

For information about Greenplum Database PostGIS extension support, see *PostGIS Extension Support and Limitations*.

Greenplum PostGIS Extension

The Greenplum Database PostGIS extension is available from *Pivotal Network*. You can install it using the Greenplum Package Manager (`gppkg`). For details, see `gppkg` in the *Greenplum Database Utility Guide*.

- Greenplum Database 4.3 supports PostGIS extension package version 2.0 (PostGIS 2.0.3).
- Greenplum Database 4.2.6 and later supports PostGIS extension package version 1.0 and 2.0 (PostGIS 1.4 and 2.0.3)

  Only one version of the PostGIS extension package, either 1.0 or 2.0, can be installed on an installation of Greenplum Database.
- Greenplum Database prior to 4.2.6 supports PostGIS extension package version 1.0 (PostGIS 1.4).

Table 141: PostGIS Component Version

PostGIS Extension Package	PostGIS	Geos	Proj
2.0	2.0.3	3.3.8	4.8.0
1.0	1.4.2	3.2.2	4.7.0

For the information about supported extension packages and software versions see the *Greenplum Database Release Notes*.

> **Important:** Extension packages for Greenplum Database 4.3.4.x and earlier are not compatible with Greenplum Database 4.3.5.0 and later due to the introduction of Pivotal Query Optimizer. Also, extension packages for Greenplum Database 4.3.5.0 and later are not compatible with Greenplum Database 4.3.4.x and earlier.
>
> To use extension packages with Greenplum Database 4.3.5.0 and later, you must install and use Greenplum Database extension packages (gppkg files and contrib modules) that are built for Greenplum Database 4.3.5.0 and later. For custom modules that were used with Greenplum Database 4.3.4.x and earlier, you must rebuild the modules for use with Greenplum Database 4.3.5.0 and later.

Major enhancements and changes in 2.0.3 from 1.4.2 include:

- Support for geographic coordinates (latitude and longitude) with a GEOGRAPHY type and related functions.
- Input format support for these formats: GML, KML, and JSON
- Unknown SRID changed from -1 to 0
- 3D relationship and measurement support functions
- Making spatial indexes 3D aware
- KNN GiST centroid distance operator
- Many deprecated functions are removed

- Performance improvements

See the PostGIS documentation for a list of changes: *http://postgis.net/docs/manual-2.0/release_notes.html*

> **Warning:** PostGIS 2.0 removed many functions that were deprecated but available in PostGIS 1.4. Functions and applications written with functions that were deprecated in PostGIS 1.4 might need to be rewritten. See the PostGIS documentation for a list of new, enhanced, or changed functions: *http://postgis.net/docs/manual-2.0/PostGIS_Special_Functions_Index.html #NewFunctions*

Greenplum Database PostGIS Limitations

The Greenplum Database PostGIS extension does not support the following features:

- Topology
- Raster
- A small number of user defined functions and aggregates
- PostGIS long transaction support
- Geometry and geography type modifier

For information about Greenplum Database PostGIS support, see *PostGIS Extension Support and Limitations*.

Enabling PostGIS Support

After installing the PostGIS extension package, you enable PostGIS support for each database that requires its use. To enable the support, run enabler SQL scripts that are supplied with the PostGIS package, in your target database.

For PosgGIS 1.4 the enabler script is `postgis.sql`

```
psql -f postgis.sql -d your_database
```

Your database is now spatially enabled.

For PostGIS 2.0.3, you run two SQL scripts `postgis.sql` and `spatial_ref_sys.sql` in your target database.

For example:

```
psql -d mydatabase -f
  $GPHOME/share/postgresql/contrib/postgis-2.0/postgis.sql
psql -d mydatabase -f
  $GPHOME/share/postgresql/contrib/postgis-2.0/spatial_ref_sys.sql
```

> **Note:** `spatial_ref_sys.sql` populates the `spatial_ref_sys` table with EPSG coordinate system definition identifiers. If you have overridden standard entries and want to use those overrides, do not load the `spatial_ref_sys.sql` file when creating the new database.

Your database is now spatially enabled.

Upgrading the Greenplum PostGIS Extension

If you upgrade from PostGIS extension package version 2.0 (pv2.0) or later, you must run `postgis_upgrade_20_minor.sql` in your target database. This example upgrades the PostGIS extension package and runs the script:

```
gppkg -u postgis-ossv2.0.3_pv2.0.1_gpdb4.3-rhel5-x86_64.gppkg

psql -d mydatabase -f $GPHOME/share/postgresql/contrib/postgis-2.0/
postgis_upgrade_20_minor.sql
```

Migrating from PostGIS 1.4 to 2.0

To migrate a PostGIS-enabled database from 1.4 to 2.0 you must perform a PostGIS HARD UPGRADE. A HARD UPGRADE consists of dumping a database that is enabled with PostGIS 1.4 and loading the database the data to a new database that is enabled with PostGIS 2.0.

For information about a PostGIS HARD UPGRADE procedure, see the PostGIS documentation: *http://postgis.net/docs/manual-2.0/postgis_installation.html#hard_upgrade*

Usage

The following example SQL statements create non-OpenGIS tables and geometries.

```
CREATE TABLE geom_test ( gid int4, geom geometry,
  name varchar(25) );
INSERT INTO geom_test ( gid, geom, name )
  VALUES ( 1, 'POLYGON((0 0 0,0 5 0,5 5 0,5 0 0,0 0 0))', '3D Square');
INSERT INTO geom_test ( gid, geom, name )
  VALUES ( 2, 'LINESTRING(1 1 1,5 5 5,7 7 5)', '3D Line' );
INSERT INTO geom_test ( gid, geom, name )
  VALUES ( 3, 'MULTIPOINT(3 4,8 9)', '2D Aggregate Point' );
SELECT * from geom_test WHERE geom &&
  Box3D(ST_GeomFromEWKT('LINESTRING(2 2 0, 3 3 0)'));
```

The following example SQL statements create a table, adds a geometry column to the table with a SRID integer value that references an entry in the SPATIAL_REF_SYS table. The `INSERT` statements add to geopoints to the table.

```
CREATE TABLE geotest (id INT4, name VARCHAR(32) );
SELECT AddGeometryColumn('geotest','geopoint', 4326,'POINT',2);
INSERT INTO geotest (id, name, geopoint)
  VALUES (1, 'Olympia', ST_GeometryFromText('POINT(-122.90 46.97)', 4326));
INSERT INTO geotest (id, name, geopoint)|
  VALUES (2, 'Renton', ST_GeometryFromText('POINT(-122.22 47.50)', 4326));
SELECT name,ST_AsText(geopoint) FROM geotest;
```

Spatial Indexes

PostgreSQL provides support for GiST spatial indexing. The GiST scheme offers indexing even on large objects. It uses a system of lossy indexing in which smaller objects act as proxies for larger ones in the index. In the PostGIS indexing system, all objects use their bounding boxes as proxies in the index.

Building a Spatial Index

You can build a GiST index as follows:

```
CREATE INDEX indexname
ON tablename
USING GIST ( geometryfield );
```

PostGIS Extension Support and Limitations

This section describes Greenplum PostGIS extension feature support and limitations.

- *Supported PostGIS Data Types*
- *Supported PostGIS Index*
- *PostGIS Extension Limitations*

The Greenplum Database PostGIS extension does not support the following features:

- Topology
- Raster

Supported PostGIS Data Types

Greenplum Database PostGIS extension supports these PostGIS data types:

- box2d
- box3d
- geometry
- geography
- spheroid

Supported PostGIS Index

Greenplum Database PostGIS extension supports the GiST (Generalized Search Tree) index.

PostGIS Extension Limitations

This section lists the Greenplum Database PostGIS extension limitations for user defined functions (UDFs), data types and aggregates.

- Data types and functions related to PostGIS topology or raster functionality, such as `TopoGeometry` and `ST_AsRaster` are not supported by Greenplum Database.
- `ST_Estimated_Extent` function is not supported. The function requires table column statistics for user defined data types that are not available with Greenplum Database.
- `ST_GeomFronGeoJSON` function is not supported. The function requires JSON support. JSON is not supported in Greenplum Database.
- These PostGIS aggregates are not supported by Greenplum Database:
    - `ST_MemCollect`
    - `ST_MakeLine`

  On a Greenplum Database with multiple segments, the aggregate might return different answers if it is called several times repeatedly.
- Greenplum Database does not support PostGIS long transactions.

  PostGIS relies on triggers and the PostGIS table *public.authorization_table* for long transaction support. When PostGIS attempts to acquire locks for long transactions, Greenplum Database reports errors citing that the function cannot access the relation, *authorization_table*.
- Greenplum Database does not support type modifiers for user defined types.

The work around is to use the `AddGeometryColumn` function for PostGIS geometry. For example, a table with PostGIS geometry cannot be created with the following SQL command:

```
CREATE TABLE geometries(id INTEGER, geom geometry(LINESTRING));
```

Use the `AddGeometryColumn` function to add PostGIS geometry to a table. For example, these following SQL statements create a table and add PostGIS geometry to the table:

```
CREATE TABLE geometries(id INTEGER);
SELECT AddGeometryColumn('public', 'geometries', 'geom', 0, 'LINESTRING', 2);
```

Chapter 14

Greenplum PL/R Language Extension

This chapter contains the following information:

About Greenplum Database PL/R

PL/R is a procedural language. With the Greenplum Database PL/R extension you can write database functions in the R programming language and use R packages that contain R functions and data sets.

> **Important:** Extension packages for Greenplum Database 4.3.4.x and earlier are not compatible with Greenplum Database 4.3.5.0 due to the introduction of Pivotal Query Optimizer. Also, extension packages for Greenplum Database 4.3.5.0 and later are not compatible with Greenplum Database 4.3.4.x and earlier.
>
> To use extension packages with Greenplum Database 4.3.5.0 and later, you must install and use Greenplum Database extension packages (gppkg files and contrib modules) that are built for Greenplum Database 4.3.5.0 and later. For custom modules that were used with Greenplum Database 4.3.4.x and earlier, you must rebuild the modules for use with Greenplum Database 4.3.5.0 and later.

For information about supported PL/R versions, see the *Greenplum Database Release Notes*.

Installing PL/R

For Greenplum Database version 4.3 and later, the PL/R extension is available as a package. Download the package from *Pivotal Network* and install it with the Greenplum Package Manager (`gppkg`).

The `gppkg` utility installs Greenplum Database extensions, along with any dependencies, on all hosts across a cluster. It also automatically installs extensions on new hosts in the case of system expansion and segment recovery.

For information about `gppkg`, see the *Greenplum Database Utility Guide*.

Installing the Extension Package

Before you install the PL/R extension, make sure that your Greenplum Database is running, you have sourced `greenplum_path.sh`, and that the `$MASTER_DATA_DIRECTORY` and `$GPHOME` variables are set.

1. Download the PL/R extension package from *Pivotal Network*, then copy it to the master host.
2. Install the software extension package by running the `gppkg` command. This example installs the PL/R extension on a Linux system:

   ```
   $ gppkg -i plr-ossv8.3.0.12_pv2.0_gpdb4.3-rhel5-x86_64.gppkg
   ```
3. Restart the database.

   ```
   $ gpstop -r
   ```
4. Source the file `$GPHOME/greenplum_path.sh`.

The extension and the R environment is installed in this directory:

```
$GPHOME/ext/R-2.12.0/
```

> **Note:** The version of some shared libraries installed with the operating system might not be compatible with the Greenplum Database PL/R extension.
>
> If a shared library is not compatible, edit the file `$GPHOME/greenplum_path.sh` in all Greenplum Database master and segment hosts and set environment variable `LD_LIBRARY_PATH` to specify the location that is installed with the PL/R extension.
>
> ```
> export LD_LIBRARY_PATH=
>    $GPHOME/ext/R-2.12.0/lib64/R/lib:$LD_LIBRARY_PATH
> ```

Enabling PL/R Language Support

For each database that requires its use, register the PL/R language with the SQL command `CREATE LANGUAGE` or the utility `createlang`. For example, this command registers the language for the database *testdb*:

```
$ createlang plr -d testdb
```

PL/R is registered as an untrusted language.

Uninstalling PL/R

- *Remove PL/R Support for a Database*
- *Uninstall the Extension Package*

When you remove PL/R language support from a database, the PL/R routines that you created in the database will no longer work.

Remove PL/R Support for a Database

For a database that no long requires the PL/R language, remove support for PL/R with the SQL command `DROP LANGUAGE` or the Greenplum Database `droplang` utility. For example, running this command run as the gpadmin user removes support for PL/R from the database *testdb*:

```
$ droplang plr -d testdb
```

Uninstall the Extension Package

If no databases have PL/R as a registered language, uninstall the Greenplum PL/R extension with the `gppkg` utility. This example uninstalls PL/R package version 1.0

```
$ gppkg -r plr-1.0
```

You can run the `gppkg` utility with the options `-q --all` to list the installed extensions and their versions.

Restart the database.

```
$ gpstop -r
```

Migrating from PL/R from R 2.12.0 to R 3.1.0

The PL/R extension package for Greenplum Database 4.3.3.0 and later supports R 3.1.0.

To migrate Greenplum Database PL/R extension to the PL/R extension that supports R 3.1.0, Pivotal recommends uninstalling the old PL/R extension package and installing the extension package supports R 3.1.0.

See *Uninstall the Extension Package* and *Installing the Extension Package* for information about uninstalling and installing the PL/R extension package.

After installing PL/R extension package that supports R 3.1.0, you must re-install R packages.

This page on the R web site, describes the changes to R:

http://cran.r-project.org/src/base/NEWS.html

See the information in section *CHANGES IN R 3.1.0* and earlier sections for information about changes in R3.1.0 and earlier.

Examples

The following are simple PL/R examples.

Example 1: Using PL/R for single row operators

This function generates an array of numbers with a normal distribution using the R function `rnorm()`.

```
CREATE OR REPLACE FUNCTION r_norm(n integer, mean float8,
  std_dev float8) RETURNS float8[ ] AS
$$
  x<-rnorm(n,mean,std_dev)
  return(x)
$$
LANGUAGE 'plr';
```

The following CREATE TABLE command uses the r_norm function to populate the table. The r_norm function creates an array of 10 numbers.

```
CREATE TABLE test_norm_var
  AS SELECT id, r_norm(10,0,1) as x
  FROM (SELECT generate_series(1,30:: bigint) AS ID) foo
  DISTRIBUTED BY (id);
```

Example 2: Returning PL/R data.frames in Tabular Form

Assuming your PL/R function returns an R `data.frame` as its output, unless you want to use arrays of arrays, some work is required to see your `data.frame` from PL/R as a simple SQL table:

- Create a `TYPE` in a Greenplum database with the same dimensions as your R `data.frame:`

  ```
  CREATE TYPE t1 AS ...
  ```

- Use this `TYPE` when defining your PL/R function

  ```
  ... RETURNS SET OF t1 AS ...
  ```

Sample SQL for this is given in the next example.

Example 3: Hierarchical Regression using PL/R

The SQL below defines a `TYPE` and runs hierarchical regression using PL/R:

```
--Create TYPE to store model results
DROP TYPE IF EXISTS wj_model_results CASCADE;
CREATE TYPE wj_model_results AS (
  cs text, coefext float, ci_95_lower float, ci_95_upper float,
  ci_90_lower, float, ci_90_upper float, ci_80_lower,
  float, ci_80_upper float);

--Create PL/R function to run model in R
DROP FUNCTION wj.plr.RE(response float [ ], cs text [ ])
RETURNS SETOF wj_model_results AS
$$
  library(arm)
  y<- log(response)
  cs<- cs
  d_temp<- data.frame(y,cs)
  m0 <- lmer (y ~ 1 + (1 | cs), data=d_temp)
  cs_unique<- sort(unique(cs))
  n_cs_unique<- length(cs_unique)
  temp_m0<- data.frame(matrix0,n_cs_unique, 7))
```

```
  for (i in 1:n_cs_unique){temp_m0[i,]<-
    c(exp(coef(m0)$cs[i,1] + c(0,-1.96,1.96,-1.65,1.65
      -1.28,1.28)*se.ranef(m0)$cs[i])))}
  names(temp_m0)<- c("Coefest", "CI_95_Lower",
    "CI_95_Upper", "CI_90_Lower", "CI_90_Upper",
   "CI_80_Lower", "CI_80_Upper")
  temp_m0_v2<- data.frames(cs_unique, temp_m0)
  return(temp_m0_v2)
$$
LANGUAGE 'plr';

--Run modeling plr function and store model results in a
--table
DROP TABLE IF EXISTS wj_model_results_roi;
CREATE TABLE wj_model_results_roi AS SELECT *
  FROM wj.plr_RE((SELECT wj.droi2_array),
  (SELECT cs FROM wj.droi2_array));
```

Downloading and Installing R Packages

R packages are modules that contain R functions and data sets. You can install R packages to extend R and PL/R functionality in Greenplum Database.

Note: If you expand Greenplum Database and add segment hosts, you must install the R packages in the R installation of the new hosts.

1. For an R package, identify all dependent R packages and each package web URL. The information can be found by selecting the given package from the following navigation page:

   http://cran.r-project.org/web/packages/available_packages_by_name.html

   As an example, the page for the R package arm indicates that the package requires the following R libraries: Matrix, lattice, lme4, R2WinBUGS, coda, abind, foreign, and MASS.

   You can also try installing the package with `R CMD INSTALL` command to determine the dependent packages.

   For the R installation included with the Greenplum Database PL/R extension, the required R packages are installed with the PL/R extension. However, the Matrix package requires a newer version.
2. From the command line, use the `wget` utility to download the `tar.gz` files for the arm package to the Greenplum Database master host:

   ```
   wget http://cran.r-project.org/src/contrib/Archive/arm/arm_1.5-03.tar.gz
   ```

   ```
   wget http://cran.r-project.org/src/contrib/Archive/Matrix/
   Matrix_0.9996875-1.tar.gz
   ```
3. Use the `gpscp` utility and the hosts_all file to copy the `tar.gz` files to the same directory on all nodes of the Greenplum cluster. The `hosts_all` file contains a list of all the Greenplum Database segment hosts. You might require root access to do this.

   ```
   gpscp -f hosts_all Matrix_0.9996875-1.tar.gz =:/home/gpadmin
   ```

   ```
   gpscp -f /hosts_all arm_1.5-03.tar.gz =:/home/gpadmin
   ```
4. Use the `gpssh` utility in interactive mode to log into each Greenplum Database segment host (`gpssh -f all_hosts`). Install the packages from the command prompt using the `R CMD INSTALL` command. Note that this may require root access. For example, this R install command installs the packages for the arm package.

   ```
   $R_HOME/bin/R CMD INSTALL Matrix_0.9996875-1.tar.gz   arm_1.5-03.tar.gz
   ```

5. Ensure that the package is installed in the `$R_HOME/library` directory on all the segments (the `gpssh` can be use to install the package). For example, this `gpssh` command list the contents of the R library directory.

```
gpssh -f all_hosts "ls $R_HOME/library"
```

6. Test if the R package can be loaded.

   This function performs a simple test to if an R package can be loaded:

```
CREATE OR REPLACE FUNCTION R_test_require(fname text)
RETURNS boolean AS
$BODY$
    return(require(fname,character.only=T))
$BODY$
LANGUAGE 'plr';
```

   This SQL command checks if the R package arm can be loaded:

```
SELECT R_test_require('arm');
```

Displaying R Library Information

You can use the R command line to display information about the installed libraries and functions on the Greenplum Database host. You can also add and remove libraries from the R installation. To start the R command line on the host, log into the host as the gadmin user and run the script R from the directory `$GPHOME/ext/R-2.12.0/bin`.

This R function lists the available R packages from the R command line:

```
> library()
```

Display the documentation for a particular R package

```
> library(help="package_name")
> help(package="package_name")
```

Display the help file for an R function:

```
> help("function_name")
> ?function_name
```

To see what packages are installed, use the R command `installed.packages()`. This will return a matrix with a row for each package that has been installed. Below, we look at the first 5 rows of this matrix.

```
> installed.packages()
```

Any package that does not appear in the installed packages matrix must be installed and loaded before its functions can be used.

An R package can be installed with `install.packages()`:

```
> install.packages("package_name")
> install.packages("mypkg", dependencies = TRUE, type="source")
```

Load a package from the R command line.

```
> library(" package_name ")
```

An R package can be removed with `remove.packages`

```
> remove.packages("package_name")
```

You can use the R command `-e` option to run functions from the command line. For example, this command displays help on the R package MASS.

```
$ R -e 'help("MASS")'
```

References

http://www.r-project.org/ - The R Project home page

https://github.com/pivotalsoftware/gp-r - GitHub repository that contains information about using R with Pivotal Data Fabric, including Pivotal Greenplum Database.

https://github.com/pivotalsoftware/PivotalR - GitHub repository for PivotalR, a package that provides an R interface to operate on Greenplum Database tables and views that is similar to the R `data.frame`. PivotalR also supports using the machine learning package *MADlib* directly from R.

R documentation is installed with the Greenplum R package:

`$GPHOME/ext/R-2.12.0/lib64/R/doc`

R Functions and Arguments

- See *http://www.joeconway.com/plr/doc/plr-funcs.html*

Passing Data Values in R

- See *http://www.joeconway.com/plr/doc/plr-data.html*

Aggregate Functions in R

- See *http://www.joeconway.com/plr/doc/plr-aggregate-funcs.html*

Chapter 15

Greenplum PL/Python Language Extension

This section contains an overview of the Greenplum Database PL/Python Language.

About Greenplum PL/Python

PL/Python is a loadable procedural language. With the Greenplum Database PL/Python extension, you can write a Greenplum Database user-defined functions in Python that take advantage of Python features and modules to quickly build robust database applications.

The Greenplum Database PL/Python extension is installed by default with Greenplum Database. Greenplum Database installs a version of Python and PL/Python. This is location of the Python installation that Greenplum Database uses:

```
$GPHOME/ext/python/
```

Greenplum Database PL/Python Limitations

- Greenplum Database does not support PL/Python triggers.
- PL/Python is available only as a Greenplum Database untrusted language.

Enabling and Removing PL/Python support

The PL/Python language is installed with Greenplum Database. To create and run a PL\Python user-defined function (UDF) in a database, you must register the PL\Python language with the database.

Enabling PL/Python Support

For each database that requires its use, register the PL/Python language with the SQL command `CREATE LANGUAGE` or the Greenplum Database utility `createlang`. For example, running this command as the `gpadmin` system user registers PL/Python for the database `testdb`:

```
$ createlang plpythonu -d testdb
```

PL/Python is registered as an untrusted language.

Removing PL/Python Support

For a database that no long requires the PL/Python language, remove support for PL/Python with the SQL command `DROP LANGUAGE` or the Greenplum Database `droplang` utility. For example, running this command run as the `gpadmin` system user removes support for PL/Python from the database `testdb`:

```
$ droplang plpythonu -d testdb
```

When you remove support for PL/Python, the PL/Python user-defined functions that you created in the database will no longer work.

Developing Functions with PL/Python

The body of a PL/Python user-defined function is a Python script. When the function is called, its arguments are passed as elements of the array `args[]`. Named arguments are also passed as ordinary variables to the Python script. The result is returned from the PL/Python function with `return` statement, or `yield` statement in case of a result-set statement.

The Greenplum Database PL/Python language module imports the Python module `plpy`. The module `plpy` implements these functions:

- Functions to execute SQL queries and prepare execution plans for queries.

  `plpy.execute`

  `plpy.prepare`
- Functions to manage errors and messages.

  `plpy.debug`

  `plpy.log`

  `plpy.info`

  `plpy.notice`

  `plpy.warning`

  `plpy.error`

  `plpy.fatal`

  `plpy.debug`

Executing and Preparing SQL Queries

The PL/Python `plpy` module provides two Python functions to execute an SQL query and prepare an execution plan for a query, `plpy.execute` and `plpy.prepare`. Preparing the execution plan for a query is useful if you run the query from multiple Python function.

plpy.execute

Calling `plpy.execute` with a query string and an optional limit argument causes the query to be run and the result to be returned in a Python result object. The result object emulates a list or dictionary object. The rows returned in the result object can be accessed by row number and column name. The result set row numbering starts with 0 (zero). The result object can be modified. The result object has these additional methods:

- `nrows` that returns the number of rows returned by the query.
- `status` which is the `SPI_execute()` return value.

For example, this Python statement in a PL/Python user-defined function executes a query.

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

The `plpy.execute` function returns up to 5 rows from `my_table`. The result set is stored in the `rv` object. If `my_table` has a column `my_column`, it would be accessed as:

```
my_col_data = rv[i]["my_column"]
```

Since the function returns a maximum of 5 rows, the index *i* can be an integer between 0 and 4.

plpy.prepare

The function `plpy.prepare` prepares the execution plan for a query. It is called with a query string and a list of parameter types, if you have parameter references in the query. For example, this statement can be in a PL/Python user-defined function:

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE
  first_name = $1", [ "text" ])
```

The string `text` is the data type of the variable that is passed for the variable *$1*. After preparing a statement, you use the function `plpy.execute` to run it:

```
rv = plpy.execute(plan, [ "Fred" ], 5)
```

The third argument is the limit for the number of rows returned and is optional.

When you prepare an execution plan using the PL/Python module the plan is automatically saved. See the Postgres Server Programming Interface (SPI) documentation for information about the execution plans *http://www.postgresql.org/docs/8.2/static/spi.html*.

To make effective use of saved plans across function calls you use one of the Python persistent storage dictionaries SD or GD.

The global dictionary SD is available to store data between function calls. This variable is private static data. The global dictionary GD is public data, available to all Python functions within a session. Use GD with care.

Each function gets its own execution environment in the Python interpreter, so that global data and function arguments from `myfunc` are not available to `myfunc2`. The exception is the data in the GD dictionary, as mentioned previously.

This example uses the SD dictionary:

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
  if SD.has_key("plan"):
    plan = SD["plan"]
  else:
    plan = plpy.prepare("SELECT 1")
    SD["plan"] = plan

  # rest of function

$$ LANGUAGE plpythonu;
```

Handling Python Errors and Messages

The message functions `plpy.error` and `plpy.fatal` raise a Python exception which, if uncaught, propagates out to the calling query, causing the current transaction or subtransaction to be aborted. The functions `raise plpy.ERROR(msg)` and `raise plpy.FATAL(msg)` are equivalent to calling `plpy.error` and `plpy.fatal`, respectively. The other message functions only generate messages of different priority levels.

Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the Greenplum Database server configuration parameters `log_min_messages` and `client_min_messages`. For information about the parameters see the *Greenplum Database Reference Guide*.

Using the dictionary GD To Improve PL/Python Performance

In terms of performance, importing a Python module is an expensive operation and can affect performance. If you are importing the same module frequently, you can use Python global variables to load the module on the first invocation and not require importing the module on subsequent calls. The following PL/Python function uses the GD persistent storage dictionary to avoid importing a module if it has already been imported and is in the GD.

```
psql=#
   CREATE FUNCTION pytest() returns text as $$
      if 'mymodule' not in GD:
        import mymodule
        GD['mymodule'] = mymodule
    return GD['mymodule'].sumd([1,2,3])
$$;
```

Installing Python Modules

When you install a Python module on Greenplum Database, the Greenplum Database Python environment must have the module added to it across all segment hosts and mirror hosts in the cluster. When expanding Greenplum Database, you must add the Python modules to the new segment hosts. You can use the Greenplum Database utilities `gpssh` and `gpscp` run commands on Greenplum Database hosts and copy files to the hosts. For information about the utilities, see the *Greenplum Database Utility Guide*.

As part of the Greenplum Database installation, the `gpadmin` user environment is configured to use Python that is installed with Greenplum Database.

To check the Python environment, you can use the `which` command:

```
which python
```

The command returns the location of the Python installation. The Python installed with Greenplum Database is in the Greenplum Database `ext/python` directory.

```
/path_to_greenplum-db/ext/python/bin/python
```

If you are building a Python module, you must ensure that the build creates the correct executable. For example on a Linux system, the build should create a 64-bit executable.

Before building a Python module to be installed, ensure that the appropriate software to build the module is installed and properly configured. The build environment is required only on the host where you build the module.

These are examples of installing and testing Python modules:

- *Simple Python Module Installation Example (setuptools)*
- *Complex Python Installation Example (NumPy)*
- *Testing Installed Python Modules*

Simple Python Module Installation Example (setuptools)

This example manually installs the Python `setuptools` module from the Python Package Index repository. The module lets you easily download, build, install, upgrade, and uninstall Python packages.

This example first builds the module from a package and installs the module on a single host. Then the module is built and installed on segment hosts.

1. Get the module package from the Python Package Index site. For example, run this `wget` command on a Greenplum Database host as the `gpadmin` user to get the tar file.

   ```
   wget --no-check-certificate https://pypi.python.org/packages/source/s/setuptools/
   setuptools-18.4.tar.gz
   ```
2. Extract the files from the tar file.

   ```
   tar -xzvf distribute-0.6.21.tar.gz
   ```
3. Go to the directory that contains the package files and run the Python scripts to build and install the Python package.

   ```
   cd setuptools-18.4
   python setup.py build && python setup.py install
   ```

4. The following Python command returns no errors if the module is available to Python.

```
python -c "import setuptools"
```

5. Copy the package to the Greenplum Database hosts with the `gpscp` utility. For example, this command copies the tar file from the current host to the host systems listed in the file `remote-hosts`.

```
gpscp -f remote-hosts setuptools-18.4.tar.gz =:/home/gpadmin
```

6. Run the commands to build, install, and test the package with `gpssh` utility on the hosts listed in the file `remote-hosts`. The file `remote-hosts` lists all the remote Greenplum Database segment hosts:

```
gpssh -f remote_hosts
>>> tar -xzvf distribute-0.6.21.tar.gz
>>> cd setuptools-18.4
>>> python setup.py build && python setup.py install
>>> python -c "import setuptools"
>>> exit
```

The `setuptools` package installs the `easy_install` utility that lets you install Python packages from the Python Package Index repository. For example, this command installs Python `PIP` utility from the Python Package Index site.

```
easy_install pip
```

You can use the `gpssh` utility to run the `easy-install` command on all the Greenplum Database segment hosts.

Complex Python Installation Example (NumPy)

This example builds and installs the Python module NumPy. NumPy is a module for scientific computing with Python. For information about NumPy, see *http://www.numpy.org/*.

Building the NumPy package requires this software:

- OpenBLAS libraries, an open source implementation of BLAS (Basic Linear Algebra Subprograms).
- The gcc compilers: gcc, gcc-gfortran, and gcc-c++. The compilers are required to build the OpenBLAS libraries. See *OpenBLAS Prerequisites*

This example process assumes `yum` is installed on the Greenplum Database segment hosts and the `gpadmin` user is a member of `sudoers` with root privileges on the hosts.

Download the OpenBLAS and NumPy source files. For example, these `wget` commands download tar files into the directory `packages`:

```
wget --directory-prefix=packages http://github.com/xianyi/OpenBLAS/tarball/v0.2.8
wget --directory-prefix=packages http://sourceforge.net/projects/numpy/files/
NumPy/1.8.0/numpy-1.8.0.tar.gz/download
```

Distribute the software to the Geeenplum Database hosts. For example, if you download the software to `/home/gpadmin/packages` these commands create the directory on the hosts and copies the software to hosts for the hosts listed in the `gpdb_remotes` file.

```
gpssh -f gpdb_remotes mkdir packages
gpscp -f gpdb_remotes packages/* =:/home/gpadmin/packages
```

OpenBLAS Prerequisites

If needed, use `yum` to install gcc compilers from system repositories. The compilers are required on all hosts where you compile OpenBLAS:

```
sudo yum -y install gcc gcc-gfortran gcc-c++
```

> **Note:** If you cannot install the correct compiler versions with yum, you can download the gcc compilers, including gfortran, from source and install them. These two commands download and install the compilers:
>
> ```
> wget http://gfortran.com/download/x86_64/snapshots/gcc-4.4.tar.xz
> tar xf gcc-4.4.tar.xz -C /usr/local/
> ```
>
> If you installed gcc manually from a tar file, add the new gcc binaries to `PATH` and `LD_LIBRARY_PATH`:
>
> ```
> export PATH=$PATH:/usr/local/gcc-4.4/bin
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/gcc-4.4/lib
> ```

Create a symbolic link to g++ and call it gxx

```
sudo ln -s /usr/bin/g++ /usr/bin/gxx
```

You might also need to create symbolic links to any libraries that have different versions available for example `libppl_c.so.4` to `libppl_c.so.2`.

If needed, you can use the `gpscp` utility to copy files to Greenplum Database hosts and the `gpssh` utility to run commands on the hosts.

Build and Install OpenBLAS Libraries

Before build and install the NumPy module, you install the OpenBLAS libraries. This section describes how to build and install the libraries on a single host.

1. Extract the OpenBLAS files from the file. These commands extract the files from the OpenBLAS tar file and simplify the directory name that contains the OpenBLAS files.

   ```
   tar -xzf packages/v0.2.8 -C /home/gpadmin/packages
   mv /home/gpadmin/packages/xianyi-OpenBLAS-9c51cdf /home/gpadmin/packages/OpenBLAS
   ```

2. Compile OpenBLAS. These commands set the `LIBRARY_PATH` environment variable and run the `make` command to build OpenBLAS libraries.

   ```
   cd /home/gpadmin/packages/OpenBLAS
   export LIBRARY_PATH=$LD_LIBRARY_PATH
   make FC=gfortran USE_THREAD=0
   ```

3. These commands install the OpenBLAS libraries in `/usr/local` as root and change the owner of the files to `gpadmin`.

   ```
   cd /home/gpadmin/packages/OpenBLAS/
   sudo make PREFIX=/usr/local install
   sudo ldconfig
   sudo chown -R gpadmin /usr/local/lib
   ```

These are the libraries that are installed and symbolic links that are created:

```
libopenblas.a -> libopenblas_sandybridge-r0.2.8.a
libopenblas_sandybridge-r0.2.8.a
libopenblas_sandybridge-r0.2.8.so
```

```
libopenblas.so -> libopenblas_sandybridge-r0.2.8.so
libopenblas.so.0 -> libopenblas_sandybridge-r0.2.8.so
```

You can use the `gpssh` utility to build and install the OpenBLAS libraries on multiple hosts.

All the Greenplum Database hosts (master and segment hosts) have identical configurations. You can copy the OpenBLAS libraries from the system where they were built instead of building the OpenBlas libraries on all the hosts. For example, these `gpssh` and `gpscp` commands copy and install the OpenBLAS libraries on the hosts listed in the `gpdb_remotes` file.

```
gpssh -f gpdb_remotes -e 'sudo yum -y install gcc gcc-gfortran gcc-c++'
gpssh -f gpdb_remotes -e 'ln -s /usr/bin/g++ /usr/bin/gxx'
gpssh -f gpdb_remotes -e sudo chown gpadmin /usr/local/lib
gpscp -f gpdb_remotes /usr/local/lib/libopen*sandy* =:/usr/local/lib

gpssh -f gpdb_remotes
>>> cd /usr/local/lib
>>> ln -s libopenblas_sandybridge-r0.2.8.a libopenblas.a
>>> ln -s libopenblas_sandybridge-r0.2.8.so libopenblas.so
>>> ln -s libopenblas_sandybridge-r0.2.8.so libopenblas.so.0
>>> sudo ldconfig
```

Build and Install NumPy

After you have installed the OpenBLAS libraries, you can build and install NumPy module. These steps install the NumPy module on a single host. You can use the `gpssh` utility to build and install the NumPy module on multiple hosts.

1. Go to the `packages` subdirectory and get the NumPy module source and extract the files.

```
cd /home/gpadmin/packages
tar -xzf numpy-1.8.0.tar.gz
```

2. Set up the environment for building and installing NumPy.

```
export BLAS=/usr/local/lib/libopenblas.a
export LAPACK=/usr/local/lib/libopenblas.a
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib/
export LIBRARY_PATH=$LD_LIBRARY_PATH
```

3. Go to the NumPy directory and build and install NumPy. Building the NumPy package might take some time.

```
cd numpy-1.8.0
python setup.py build
python setup.py install
```

Note: If the NumPy module did not successfully build, the NumPy build process might need a `site.cfg` that specifies the location of the OpenBLAS libraries. Create the file `site.cfg` in the NumPy package directory:

```
cd ~/packages/numpy-1.8.0
touch site.cfg
```

Add the following to the `site.cfg` file and run the NumPy build command again:

```
[default]
library_dirs = /usr/local/lib

[atlas]
atlas_libs = openblas
library_dirs = /usr/local/lib

[lapack]
```

```
lapack_libs = openblas
library_dirs = /usr/local/lib

# added for scikit-learn
[openblas]
libraries = openblas
library_dirs = /usr/local/lib
include_dirs = /usr/local/include
```

4. The following Python command ensures that the module is available for import by Python on a host system.

```
python -c "import numpy"
```

As in the simple module installation, you can use the `gpssh` utility to build, install, and test the module on Greenplum Database segment hosts.

The environment variables that are require to build the NumPy module are also required in the `gpadmin` user environment when running Python NumPy functions. You can use the `gpssh` utility with the `echo` command to add the environment variables to the `.bashrc` file. For example, these `echo` commands add the environment variables to the `.bashrc` file in the user home directory.

```
echo -e '\n#Needed for NumPy' >> ~/.bashrc
echo -e 'export BLAS=/usr/local/lib/libopenblas.a' >> ~/.bashrc
echo -e 'export LAPACK=/usr/local/lib/libopenblas.a' >> ~/.bashrc
echo -e 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib' >> ~/.bashrc
echo -e 'export LIBRARY_PATH=$LD_LIBRARY_PATH' >> ~/.bashrc
```

Testing Installed Python Modules

You can create a simple PL/Python user-defined function (UDF) to validate that Python a module is available in the Greenplum Database. This example tests the NumPy module.

This PL/Python UDF imports the NumPy module. The function returns `SUCCESS` if the module is imported, and `FAILURE` if an import error occurs.

```
CREATE OR REPLACE FUNCTION plpy_test(x int)
returns text
as $$
  try:
      from numpy import *
      return 'SUCCESS'
  except ImportError, e:
      return 'FAILURE'
$$ language plpythonu;
```

Create a table that contains data on each Greenplum Database segment instance. Depending on the size of your Greenplum Database installation, you might need to generate more data to ensure data is distributed to all segment instances.

```
CREATE TABLE DIST AS (SELECT x FROM generate_series(1,50) x ) DISTRIBUTED RANDOMLY ;
```

This `SELECT` command runs the UDF on the segment hosts where data is stored in the primary segment instances.

```
SELECT gp_segment_id, plpy_test(x) AS status
  FROM dist
  GROUP BY gp_segment_id, status
  ORDER BY gp_segment_id, status;
```

The `SELECT` command returns `SUCCESS` if the UDF imported the Python module on the Greenplum Database segment instance. If the `SELECT` command returns `FAILURE`, you can find the segment host of the segment instance host. The Greenplum Database system table `gp_segment_configuration` contains information about mirroring and segment configuration. This command returns the host name for a segment ID.

```
SELECT hostname, content AS seg_ID FROM gp_segment_configuration
  WHERE content = seg_id ;
```

If `FAILURE` is returned, these are some possible causes:

- A problem accessing required libraries. For the NumPy example, a Greenplum Database might have a problem accessing the OpenBLAS libraries or the Python libraries on a segment host.

  Make sure you get no errors when running command on the segment host as the `gpadmin` user. This `gpssh` command tests importing the numpy module on the segment host `mdw1`.

  ```
  gpssh -h mdw1 python -c "import numpy"
  ```

- If the Python `import` command does not return an error, environment variables might not be configured in the Greenplum Database environment. For example, the variables are not in the `.bashrc` file, or Greenplum Database might not have been restarted after adding the environment variables to the `.bashrc` file.

  Ensure sure that the environment variables are properly set and then restart the Greenplum Database. For the NumPy example, ensure the environment variables listed at the end of the section *Build and Install NumPy* are defined in the `.bashrc` file for the `gpadmin` user on the master and segment hosts.

  Note: On the Greenplum Database master and segment hosts, the `.bashrc` file for the `gpadmin` user must source the file `$GPHOME/greenplum_path.sh`.

Examples

This PL/Python UDF returns the maximum of two integers:

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
      return None
  if a > b:
     return a
  return b
$$ LANGUAGE plpythonu;
```

You can use the `STRICT` property to perform the null handling instead of using the two conditional statements.

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer AS $$
return max(a,b)
$$ LANGUAGE plpythonu STRICT ;
```

You can run the user-defined function `pymax` with `SELECT` command. This example runs the UDF and shows the output.

```
SELECT ( pymax(123, 43));
column1
---------
     123
(1 row)
```

This example that returns data from an SQL query that is run against a table. These two commands create a simple table and add data to the table.

```
CREATE TABLE sales (id int, year int, qtr int, day int, region text)
  DISTRIBUTED BY (id) ;

INSERT INTO sales VALUES
 (1, 2014, 1,1, 'usa'),
 (2, 2002, 2,2, 'europe'),
 (3, 2014, 3,3, 'asia'),
 (4, 2014, 4,4, 'usa'),
 (5, 2014, 1,5, 'europe'),
 (6, 2014, 2,6, 'asia'),
 (7, 2002, 3,7, 'usa') ;
```

This PL/Python UDF executes a `SELECT` command that returns 5 rows from the table. The Python function returns the `REGION` value from the row specified by the input value. In the Python function, the row numbering starts from 0. Valid input for the function is an integer between 0 and 4.

```
CREATE OR REPLACE FUNCTION mypytest(a integer)
  RETURNS text
AS $$
  rv = plpy.execute("SELECT * FROM sales ORDER BY id", 5)
  region = rv[a]["region"]
  return region
$$ language plpythonu;
```

Running this `SELECT` statement returns the `REGION` column value from the third row of the result set.

```
SELECT mypytest(2) ;
```

This command deletes the UDF from the database.

```
DROP FUNCTION mypytest(integer) ;
```

References

Technical References

For information about PL/Python see the PostgreSQL documentation at *http://www.postgresql.org/docs/8.2/static/plpython.html*.

For information about Python Package Index (PyPI), see *https://pypi.python.org/pypi*.

These are some Python modules that can be downloaded:

- SciPy library provides user-friendly and efficient numerical routines such as routines for numerical integration and optimization *http://www.scipy.org/scipylib/index.html*. This `wget` command downloads the SciPy package tar file.

```
wget http://sourceforge.net/projects/scipy/files/scipy/0.10.1/scipy-0.10.1.tar.gz/
download
```

- Natural Language Toolkit (nltk) is a platform for building Python programs to work with human language data. *http://www.nltk.org/*. This `wget` command downloads the nltk package tar file.

```
wget http://pypi.python.org/packages/source/n/nltk/
nltk-2.0.2.tar.gz#md5=6e714ff74c3398e88be084748df4e657
```

Note: The Python package Distribute *https://pypi.python.org/pypi/distribute* is required for nltk. The Distribute module should be installed the ntlk package. This `wget` command downloads the Distribute package tar file.

```
wget http://pypi.python.org/packages/source/d/distribute/
distribute-0.6.21.tar.gz
```

Useful Reading

For information about the Python language, see *http://www.python.org/*.

A set of slides that were used in a talk about how the Pivotal data science team uses the PyData stack in the Pivotal MPP databases and on Pivotal Cloud Foundry *http://www.slideshare.net/SrivatsanRamanujam/all-thingspythonpivotal*.

Chapter 16

Greenplum PL/Java Language Extension

This section contains an overview of the Greenplum Database PL/Java language.

About PL/Java

With Greenplum Database PL/Java extension, you can write Java methods using your favorite Java IDE and install the JAR files that contain the methods into Greenplum Database.

Greenplum Database PL/Java package is based on the open source PL/Java 1.4.0. Greenplum Database PL/Java provides the following features.

- Ability to execute PL/Java functions with Java 1.6 or higher.
- Standardized utilities (modeled after the SQL 2003 proposal) to install and maintain Java code in the database.
- Standardized mappings of parameters and result. Complex types as well as sets are supported.
- An embedded, high performance, JDBC driver utilizing the internal Greenplum Database SPI routines.
- Metadata support for the JDBC driver. Both `DatabaseMetaData` and `ResultSetMetaData` are included.
- The ability to return a ResultSet from a query as an alternative to building a ResultSet row by row.
- Full support for savepoints and exception handling.
- The ability to use IN, INOUT, and OUT parameters.
- Two separate Greenplum Database languages:
    - pljava, TRUSTED PL/Java language
    - pljavau, UNTRUSTED PL/Java language
- Transaction and Savepoint listeners enabling code execution when a transaction or savepoint is committed or rolled back.
- Integration with GNU GCJ on selected platforms.

The Greenplum Database PL/Java extension package ships with embedded Java Runtime 6u32 that is installed in the directory `$GPHOME/ext/jre-1.6.0_32` on each of the Greenplum Database hosts.

A function in SQL will appoint a static method in a Java class. In order for the function to execute, the appointed class must available on the class path specified by the Greenplum Database sever configuration parameter `pljava_classpath`. The PL/Java extension adds a set of functions that helps installing and maintaining the java classes. Classes are stored in normal Java archives, JAR files. A JAR file can optionally contain a deployment descriptor that in turn contains SQL commands to be executed when the JAR is deployed or undeployed. The functions are modeled after the standards proposed for SQL 2003.

PL/Java implements a standardized way of passing parameters and return values. Complex types and sets are passed using the standard JDBC ResultSet class.

A JDBC driver is included in PL/Java. This driver calls Greenplum Database internal SPI routines. The driver is essential since it is common for functions to make calls back to the database to fetch data. When PL/Java functions fetch data, they must use the same transactional boundaries that are used by the main function that entered PL/Java execution context.

PL/Java is optimized for performance. The Java virtual machine executes within the same process as the backend to minimize call overhead. PL/Java is designed with the objective to enable the power of Java to the database itself so that database intensive business logic can execute as close to the actual data as possible.

The standard Java Native Interface (JNI) is used when bridging calls between the backend and the Java VM.

Installing PL/Java

For Greenplum Database, the PL/Java extension is available as a package. Download the package from *Pivotal Network* and then install it with the Greenplum Package Manager (`gppkg`).

The `gppkg` utility installs Greenplum Database extensions, along with any dependencies, on all hosts across a cluster. It also automatically installs extensions on new hosts in the case of system expansion and segment recovery.

For information about `gppkg`, see the *Greenplum Database Utility Guide*.

To install and use PL/Java:

1. Install the Greenplum Database PL/Java extension.
2. *Optional*. Change the Java version used by PL/Java.
3. Enable the language for each database where you intend to use PL/Java.
4. Install user-created JAR files containing Java methods into the same directory on all Greenplum Database hosts.
5. Add the name of the JAR file to the Greenplum Database server configuration parameter `pljava_classpath`. The parameter lists the installed JAR files. For information about the parameter, see the *Greenplum Database Reference Guide*.

Installing the Greenplum PL/Java Extension

Before you install the PL/Java extension, make sure that your Greenplum database is running, you have sourced `greenplum_path.sh`, and that the `$MASTER_DATA_DIRECTORY` and `$GPHOME` variables are set.

1. Download the PL/Java extension package from *Pivotal Network* then copy it to the master host.
2. Install the software extension package by running the `gppkg` command. This example installs the PL/Java extension package on a Linux system:

```
$ gppkg -i pljava-ossv1.4.0_pv1.2_gpdb4.3orca-rhel5-x86_64.gppkg
```

3. Reload `greenplum_path.sh`.

```
$ source $GPHOME/greenplum_path.sh
```

4. Restart Greenplum Database.

```
$ gpstop -r
```

Changing Java version used by PL/Java (Optional)

The PL/Java extension package ships with Java JRE 6u32. When the package is installed, the JRE is installed on each host of your Greenplum Database cluster. To use newer version of Java with PL/Java, perform these steps to specify the location of the Java version. For information about supported Java versions, see the *Greenplum Database Release Notes* for your release.

> **Note:** The newer Java must be installed in the same location on all Greenplum Database hosts and must be accessible to the system user of the Greenplum Database administrator (`gpadmin`).

1. In the `$GPHOME/greenplum_path.sh` file, modify the `JAVA_HOME` and `LD_LIBRARY_PATH` environment variables.

- Set the `JAVA_HOME` variable to the directory where your Java Runtime is installed. For example, for Oracle JRE this directory would be `/usr/java/latest`. For OpenJDK, the directory is `/usr/lib/jvm/jre`. This example changes the environment variable to use `/usr/java/latest`.

  ```
  JAVA_HOME=/usr/java/latest
  ```
- Set the `LD_LIBRARY_PATH` to include the directory with Java server runtime libraries. PL/Java depends on `libjvm.so` and the shared object should be in your `LD_LIBRARY_PATH`. By default, `libjvm.so` is available in `$JAVA_HOME/lib/amd64/server`. This example adds the directory to the environment variable.

  ```
  LD_LIBRARY_PATH=$GPHOME/lib:$GPHOME/ext/python/lib:$JAVA_HOME/lib/amd64/server:
  $LD_LIBRARY_PATH
  ```

2. Copy the uppdated `greenplum_path.sh` file to all the Greenplum Database hosts. This example `gpscp` command copies the file to all hosts specified in the file `gphosts_file`.

   ```
   $ gpscp -f gphosts_file $GPHOME/greenplum_path.sh
   =:$GPHOME/greenplum_path.sh
   ```
3. Reload `greenplum_path.sh`.

   ```
   $ source $GPHOME/greenplum_path.sh
   ```
4. Restart Greenplum Database.

   ```
   $ gpstop -r
   ```

Enabling PL/Java and Installing JAR Files

Perform the following steps as the Greenplum Database administrator `gpadmin`.

1. Enable PL/Java by running the SQL script `$GPHOME/share/postgresql/pljava/install.sql` in the databases that will use PL/Java. For example, this example enables PL/Java on the database `mytestdb`:

   ```
   $ psql -d mytestdb
     -f $GPHOME/share/postgresql/pljava/install.sql
   ```

   The script `install.sql` registers both the trusted and untrusted PL/Java language.
2. Copy your Java archives (JAR files) to the same directory on all Greenplum Database hosts. This example uses the Greenplum Database `gpscp` utility to copy the file `myclasses.jar` to the directory `$GPHOME/lib/postgresql/java/`:

   ```
   $ gpscp -f gphosts_file myclasses.jar
   =:/usr/local/greenplum-db/lib/postgresql/java/
   ```

   The file `gphosts_file` contains a list of the Greenplum Database hosts.
3. Set the `pljava_classpath` server configuration parameter in the master `postgresql.conf` file. For this example, the parameter value is a colon (:) separated list of the JAR files. For example:

   ```
   $ gpconfig -c pljava_classpath
   -v \'examples.jar:myclasses.jar\'
   ```

   The file `examples.jar` is installed when you install the PL/Java extension package with the `gppkg` utility.

   > **Note:** If you install JAR files in a directory other than `$GPHOME/lib/postgresql/java/`, you must specify the absolute path to the JAR file. Each JAR file must be in the same location on all Greenplum Database hosts. For more information about specifying the location of JAR files, see

the information about the `pljava_classpath` server configuration parameter in the *Greenplum Database Reference Guide*.

4. Reload the `postgresql.conf` file.

```
$ gpstop -u
```

5. (optional) Greenplum provides an `examples.sql` file containing sample PL/Java functions that you can use for testing. Run the commands in this file to create the test functions (which use the Java classes in `examples.jar`).

```
$ psql -f $GPHOME/share/postgresql/pljava/examples.sql
```

Uninstalling PL/Java

Remove PL/Java Support for a Database

For a database that no long requires the PL/Java language, remove support for PL/Java. Run the `uninstall.sql` file as the gpadmin user. For example, this command disables the PL/Java language in the specified database.

```
$ psql -d mydatabase
  -f $GPHOME/share/postgresql/pljava/uninstall.sql
```

Uninstall the Java JAR files and Software Package

If no databases have PL/Java as a registered language, remove the Java JAR files and uninstall the Greenplum PL/Java extension with the `gppkg` utility.

1. Remove the `pljava_classpath` server configuration parameter from the `postgresql.conf` file on all Greenplum Database hosts. For example:

   ```
   $ gpconfig -r pljava_classpath
   ```
2. Remove the JAR files from the directories where they were installed on all Greenplum Database hosts. For information about JAR file installation directories, see *Enabling PL/Java and Installing JAR Files*.
3. Use the Greenplum `gppkg` utility with the `-r` option to uninstall the PL/Java extension. This example uninstalls the PL/Java extension on a Linux system:

   ```
   $ gppkg -r pljava-ossv1.4.0_pv1.3_gpdb4.3orca
   ```

   You can run the `gppkg` utility with the options `-q --all` to list the installed extensions and their versions.
4. Reload `greenplum_path.sh`.

   ```
   $ source $GPHOME/greenplum_path.sh
   ```
5. Restart the database.

   ```
   $ gpstop -r
   ```

About Greenplum Database PL/Java

There are a few key differences between the implementation of PL/Java in standard PostgreSQL and Greenplum Database.

Functions

The following functions are not supported in Greenplum Database. The classpath is handled differently in a distributed Greenplum Database environment than in the PostgreSQL environment.

```
sqlj.install_jar
sqlj.install_jar
sqlj.replace_jar
sqlj.remove_jar
sqlj.get_classpath
sqlj.set_classpath
```

Greenplum Database uses the `pljava_classpath` server configuration parameter in place of the `sqlj.set_classpath` function.

Server Configuration Parameters

The following server configuration parameters are used by PL/Java in Greenplum Database. These parameters replace the `pljava.*` parameters that are used in the standard PostgreSQL PL/Java implementation:

- `pljava_classpath`

  A colon (`:`) separated list of the jar files containing the Java classes used in any PL/Java functions. The jar files must be installed in the same locations on all Greenplum Database hosts. With the trusted PL/Java language handler, jar file paths must be relative to the `$GPHOME/lib/postgresql/java/` directory. With the untrusted language handler (javaU language tag), paths may be relative to `$GPHOME/lib/postgresql/java/` or absolute.
- `pljava_statement_cache_size`

  Sets the size in KB of the Most Recently Used (MRU) cache for prepared statements.
- `pljava_release_lingering_savepoints`

  If `TRUE`, lingering savepoints will be released on function exit. If `FALSE`, they will be rolled back.
- `pljava_vmoptions`

  Defines the start up options for the Greenplum Database Java VM.

See the *Greenplum Database Reference Guide* for information about the Greenplum Database parameters.

Writing PL/Java functions

Information about writing functions with PL/Java.

SQL Declaration

A Java function is declared with the name of a class and a static method on that class. The class will be resolved using the classpath that has been defined for the schema where the function is declared. If no classpath has been defined for that schema, the public schema is used. If no classpath is found there either, the class is resolved using the system classloader.

The following function can be declared to access the static method `getProperty` on `java.lang.System` class:

```
CREATE FUNCTION getsysprop(VARCHAR)
  RETURNS VARCHAR
  AS 'java.lang.System.getProperty'
  LANGUAGE java;
```

Run the following command to return the Java `user.home` property:

```
SELECT getsysprop('user.home');
```

Type Mapping

Scalar types are mapped in a straight forward way. This table lists the current mappings.

Table 142: PL/Java data type mapping

PostgreSQL	Java
bool	boolean
char	byte
int2	short
int4	int
int8	long
varchar	java.lang.String
text	java.lang.String
bytea	byte[ ]

PostgreSQL	Java
date	java.sql.Date
time	java.sql.Time (stored value treated as local time)
timetz	java.sql.Time
timestamp	java.sql.Timestamp (stored value treated as local time)
timestampz	java.sql.Timestamp
complex	java.sql.ResultSet
setof complex	java.sql.ResultSet

All other types are mapped to java.lang.String and will utilize the standard `textin`/`textout` routines registered for respective type.

NULL Handling

The scalar types that map to Java primitives can not be passed as `NULL` values. To pass `NULL` values, those types can have an alternative mapping. You enable this mapping by explicitly denoting it in the method reference.

```
CREATE FUNCTION trueIfEvenOrNull(integer)
  RETURNS bool
  AS 'foo.fee.Fum.trueIfEvenOrNull(java.lang.Integer)'
  LANGUAGE java;
```

The Java code would be similar to this:

```
package foo.fee;
public class Fum
{
  static boolean trueIfEvenOrNull(Integer value)
  {
    return (value == null)
      ? true
      : (value.intValue() % 1) == 0;
  }
}
```

The following two statements both yield true:

```
SELECT trueIfEvenOrNull(NULL);
SELECT trueIfEvenOrNull(4);
```

In order to return `NULL` values from a Java method, you use the object type that corresponds to the primitive (for example, you return `java.lang.Integer` instead of `int`). The PL/Java resolve mechanism finds the method regardless. Since Java cannot have different return types for methods with the same name, this does not introduce any ambiguity.

Complex Types

A complex type will always be passed as a read-only `java.sql.ResultSet` with exactly one row. The ResultSet is positioned on its row so a call to `next()` should not be made. The values of the complex type are retrieved using the standard getter methods of the ResultSet.

Example:

```
CREATE TYPE complexTest
  AS(base integer, incbase integer, ctime timestamptz);
CREATE FUNCTION useComplexTest(complexTest)
  RETURNS VARCHAR
  AS 'foo.fee.Fum.useComplexTest'
  IMMUTABLE LANGUAGE java;
```

In the Java class `Fum`, we add the following static method:

```
public static String useComplexTest(ResultSet complexTest)
throws SQLException
{
  int base = complexTest.getInt(1);
  int incbase = complexTest.getInt(2);
  Timestamp ctime = complexTest.getTimestamp(3);
  return "Base = \"" + base +
    "\", incbase = \"" + incbase +
    "\", ctime = \"" + ctime + "\"";
}
```

Returning Complex Types

Java does not stipulate any way to create a ResultSet. Hence, returning a ResultSet is not an option. The SQL-2003 draft suggests that a complex return value should be handled as an IN/OUT parameter. PL/Java implements a ResultSet that way. If you declare a function that returns a complex type, you will need to use a Java method with boolean return type with a last parameter of type `java.sql.ResultSet`. The parameter will be initialized to an empty updateable ResultSet that contains exactly one row.

Assume that the `complexTest` type in previous section has been created.

```
CREATE FUNCTION createComplexTest(int, int)
  RETURNS complexTest
  AS 'foo.fee.Fum.createComplexTest'
  IMMUTABLE LANGUAGE java;
```

The PL/Java method resolve will now find the following method in the `Fum` class:

```
public static boolean complexReturn(int base, int increment,
  ResultSet receiver)
throws SQLException
{
  receiver.updateInt(1, base);
  receiver.updateInt(2, base + increment);
  receiver.updateTimestamp(3, new
    Timestamp(System.currentTimeMillis()));
  return true;
}
```

The return value denotes if the receiver should be considered as a valid tuple (true) or NULL (false).

Functions That Return Sets

When returning result set, you should not build a result set before returning it, because building a large result set would consume a large amount of resources. It is better to produce one row at a time. Incidentally, that is what the Greenplum Database backend expects a function with SETOF return to do. You can return a SETOF a scalar type such as an `int`, `float` or `varchar`, or you can return a SETOF a complex type.

Returning a SETOF <scalar type>

In order to return a set of a scalar type, you need create a Java method that returns something that implements the `java.util.Iterator` interface. Here is an example of a method that returns a SETOF `varchar`:

```
CREATE FUNCTION javatest.getSystemProperties()
  RETURNS SETOF varchar
  AS 'foo.fee.Bar.getNames'
  IMMUTABLE LANGUAGE java;
```

This simple Java method returns an iterator:

```
package foo.fee;
import java.util.Iterator;

public class Bar
{
    public static Iterator getNames()
    {
        ArrayList names = new ArrayList();
        names.add("Lisa");
        names.add("Bob");
        names.add("Bill");
        names.add("Sally");
        return names.iterator();
    }
}
```

Returning a SETOF <complex type>

A method returning a SETOF <complex type> must use either the interface `org.postgresql.pljava.ResultSetProvider` or `org.postgresql.pljava.ResultSetHandle`. The reason for having two interfaces is that they cater for optimal handling of two distinct use cases. The former is for cases when you want to dynamically create each row that is to be returned from the SETOF function. The latter makes is in cases where you want to return the result of an executed query.

Using the ResultSetProvider Interface

This interface has two methods. The boolean `assignRowValues(java.sql.ResultSet tupleBuilder, int rowNumber)` and the `void close()` method. The Greenplum Database query evaluator will call the `assignRowValues` repeatedly until it returns false or until the evaluator decides that it does not need any more rows. Then it calls close.

You can use this interface the following way:

```
CREATE FUNCTION javatest.listComplexTests(int, int)
  RETURNS SETOF complexTest
  AS 'foo.fee.Fum.listComplexTest'
  IMMUTABLE LANGUAGE java;
```

The function maps to a static java method that returns an instance that implements the `ResultSetProvider` interface.

```
public class Fum implements ResultSetProvider
{
  private final int m_base;
  private final int m_increment;
  public Fum(int base, int increment)
  {
    m_base = base;
```

```
    m_increment = increment;
  }
  public boolean assignRowValues(ResultSet receiver, int
currentRow)
  throws SQLException
  {
    // Stop when we reach 12 rows.
    //
    if(currentRow >= 12)
      return false;
    receiver.updateInt(1, m_base);
    receiver.updateInt(2, m_base + m_increment * currentRow);
    receiver.updateTimestamp(3, new
Timestamp(System.currentTimeMillis()));
    return true;
  }
  public void close()
  {
   // Nothing needed in this example
  }
  public static ResultSetProvider listComplexTests(int base,
int increment)
  throws SQLException
  {
    return new Fum(base, increment);
  }
}
```

The `listComplextTests` method is called once. It may return `NULL` if no results are available or an instance of the `ResultSetProvider`. Here the Java class `Fum` implements this interface so it returns an instance of itself. The method `assignRowValues` will then be called repeatedly until it returns false. At that time, close will be called

Using the ResultSetHandle Interface

This interface is similar to the `ResultSetProvider` interface in that it has a `close()` method that will be called at the end. But instead of having the evaluator call a method that builds one row at a time, this method has a method that returns a ResultSet. The query evaluator will iterate over this set and deliver the RestulSet contents, one tuple at a time, to the caller until a call to `next()` returns false or the evaluator decides that no more rows are needed.

Here is an example that executes a query using a statement that it obtained using the default connection. The SQL suitable for the deployment descriptor looks like this:

```
CREATE FUNCTION javatest.listSupers()
  RETURNS SETOF pg_user
  AS 'org.postgresql.pljava.example.Users.listSupers'
  LANGUAGE java;
CREATE FUNCTION javatest.listNonSupers()
  RETURNS SETOF pg_user
  AS 'org.postgresql.pljava.example.Users.listNonSupers'
  LANGUAGE java;
```

And in the Java package `org.postgresql.pljava.example` a class `Users` is added:

```
public class Users implements ResultSetHandle
{
  private final String m_filter;
  private Statement m_statement;
  public Users(String filter)
  {
    m_filter = filter;
  }
  public ResultSet getResultSet()
  throws SQLException
  {
```

```
    m_statement =
      DriverManager.getConnection("jdbc:default:connection").cr
eateStatement();
    return m_statement.executeQuery("SELECT * FROM pg_user
       WHERE " + m_filter);
  }

  public void close()
  throws SQLException
  {
    m_statement.close();
  }

  public static ResultSetHandle listSupers()
  {
    return new Users("usesuper = true");
  }

  public static ResultSetHandle listNonSupers()
  {
    return new Users("usesuper = false");
  }
}
```

Using JDBC

PL/Java contains a JDBC driver that maps to the PostgreSQL SPI functions. A connection that maps to the current transaction can be obtained using the following statement:

```
Connection conn =
  DriverManager.getConnection("jdbc:default:connection");
```

After obtaining a connection, you can prepare and execute statements similar to other JDBC connections. These are limitations for the PL/Java JDBC driver:

- The transaction cannot be managed in any way. Thus, you cannot use methods on the connection such as:
  - `commit()`
  - `rollback()`
  - `setAutoCommit()`
  - `setTransactionIsolation()`
- Savepoints are available with some restrictions. A savepoint cannot outlive the function in which it was set and it must be rolled back or released by that same function.
- A ResultSet returned from `executeQuery()` are always `FETCH_FORWARD` and `CONCUR_READ_ONLY`.
- Meta-data is only available in PL/Java 1.1 or higher.
- `CallableStatement` (for stored procedures) is not implemented.
- The types `Clob` or `Blob` are not completely implemented, they need more work. The types `byte[]` and `String` can be used for `bytea` and `text` respectively.

Exception Handling

You can catch and handle an exception in the Greenplum Database backend just like any other exception. The backend ErrorData structure is exposed as a property in a class called `org.postgresql.pljava.ServerException` (derived from `java.sql.SQLException`) and the Java try/catch mechanism is synchronized with the backend mechanism.

Important: You will not be able to continue executing backend functions until your function has returned and the error has been propagated when the backend has generated an exception unless you have used a savepoint. When a savepoint is rolled back, the exceptional condition is reset and you can continue your execution.

Savepoints

Greenplum Database savepoints are exposed using the java.sql.Connection interface. Two restrictions apply.

- A savepoint must be rolled back or released in the function where it was set.
- A savepoint must not outlive the function where it was set

Logging

PL/Java uses the standard Java Logger. Hence, you can write things like:

```
Logger.getAnonymousLogger().info( "Time is " + new
Date(System.currentTimeMillis()));
```

At present, the logger uses a handler that maps the current state of the Greenplum Database configuration setting `log_min_messages` to a valid Logger level and that outputs all messages using the Greenplum Database backend function `elog()`.

Note: The `log_min_messages` setting is read from the database the first time a PL/Java function in a session is executed. On the Java side, the setting does not change after the first PL/Java function execution in a specific session until the Greenplum Database session that is working with PL/Java is restarted.

The following mapping apply between the Logger levels and the Greenplum Database backend levels.

Table 143: PL/Java Logging Levels

java.util.logging.Level	Greenplum Database Level
SEVERE ERROR	ERROR
WARNING	WARNING
CONFIG	LOG
INFO	INFO
FINE	DEBUG1
FINER	DEBUG2
FINEST	DEBUG3

Security

Installation

Only a database super user can install PL/Java. The PL/Java utility functions are installed using SECURITY DEFINER so that they execute with the access permissions that where granted to the creator of the functions.

Trusted Language

PL/Java is a *trusted* language. The trusted PL/Java language has no access to the file system as stipulated by PostgreSQL definition of a trusted language. Any database user can create and access functions in a trusted language.

PL/Java also installs a language handler for the language `javau`. This version is *not trusted* and only a superuser can create new functions that use it. Any user can call the functions.

Some PL/Java Issues and Solutions

When writing the PL/Java, mapping the JVM into the same process-space as the Greenplum Database backend code, some concerns have been raised regarding multiple threads, exception handling, and memory management. Here are brief descriptions explaining how these issues where resolved.

- *Multi-threading*
- *Exception Handling*
- *Java Garbage Collector Versus palloc() and Stack Allocation*

Multi-threading

Java is inherently multi-threaded. The Greenplum Database backend is not. There is nothing stopping a developer from utilizing multiple Threads class in the Java code. Finalizers that call out to the backend might have been spawned from a background Garbage Collection thread. Several third party Java-packages that are likely to be used make use of multiple threads. How can this model coexist with the Greenplum Database backend in the same process?

Solution

The solution is simple. PL/Java defines a special object called the `Backend.THREADLOCK`. When PL/Java is initialized, the backend immediately grabs this objects monitor (i.e. it will synchronize on this object). When the backend calls a Java function, the monitor is released and then immediately regained when the call returns. All calls from Java out to backend code are synchronized on the same lock. This ensures that only one thread at a time can call the backend from Java, and only at a time when the backend is awaiting the return of a Java function call.

Exception Handling

Java makes frequent use of try/catch/finally blocks. Greenplum Database sometimes use an exception mechanism that calls `longjmp` to transfer control to a known state. Such a jump would normally effectively bypass the JVM.

Solution

The backend now allows errors to be caught using the macros `PG_TRY/PG_CATCH`/`PG_END_TRY` and in the catch block, the error can be examined using the ErrorData structure. PL/Java implements a `java.sql.SQLException` subclass called `org.postgresql.pljava.ServerException`. The ErrorData can be retrieved and examined from that exception. A catch handler is allowed to issue a rollback to a savepoint. After a successful rollback, execution can continue.

Java Garbage Collector Versus palloc() and Stack Allocation

Primitive types are always be passed by value. This includes the `String` type (this is a must since Java uses double byte characters). Complex types are often wrapped in Java objects and passed by reference. For example, a Java object can contain a pointer to a palloc'ed or stack allocated memory and use native JNI calls to extract and manipulate data. Such data will become stale once a call has ended. Further attempts to access such data will at best give very unpredictable results but more likely cause a memory fault and a crash.

Solution

The PL/Java contains code that ensures that stale pointers are cleared when the MemoryContext or stack where they where allocated goes out of scope. The Java wrapper objects might live on but any attempt to use them will result in a stale native handle exception.

Example

The following simple Java example creates a JAR file that contains a single method and runs the method.

Note: The example requires Java SDK to compile the Java file.

The following method returns a substring.

```
{
public static String substring(String text, int beginIndex,
  int endIndex)
    {
    return text.substring(beginIndex, endIndex);
    }
}
```

Enter the java code in a text file `example.class`.

Contents of the file `manifest.txt`:

```
Manifest-Version: 1.0
Main-Class: Example
Specification-Title: "Example"
Specification-Version: "1.0"
Created-By: 1.6.0_35-b10-428-11M3811
Build-Date: 01/20/2013 10:09 AM
```

Compile the java code:

```
javac *.java
```

Create a JAR archive named analytics.jar that contains the class file and the manifest file MANIFEST file in the JAR.

```
jar cfm analytics.jar manifest.txt *.class
```

Upload the jar file to the Greenplum master host.

Run the `gpscp` utility to copy the jar file to the Greenplum Java directory. Use the `-f` option to specify the file that contains a list of the master and segment hosts.

```
gpscp -f gphosts_file analytics.jar
=:/usr/local/greenplum-db/lib/postgresql/java/
```

Use the `gpconfig` utility to set the Greenplum `pljava_classpath` server configuration parameter. The parameter lists the installed jar files.

```
gpconfig -c pljava_classpath -v \'analytics.jar\'
```

Run the `gpstop` utility with the `-u` option to reload the configuration files.

```
gpstop -u
```

From the `psql` command line, run the following command to show the installed jar files.

```
show pljava_classpath
```

The following SQL commands create a table and define a Java function to test the method in the jar file:

```
create table temp (a varchar) distributed randomly;
```

```
insert into temp values ('my string');
--Example function
create or replace function java_substring(varchar, int, int)
returns varchar as 'Example.substring' language java;
--Example execution
select java_substring(a, 1, 5) from temp;
```

You can place the contents in a file, `mysample.sql` and run the command from a `psql` command line:

```
> \i mysample.sql
```

The output is similar to this:

```
java_substring
----------------
 y st
(1 row)
```

References

The PL/Java Github wiki page - *https://github.com/tada/pljava/wiki*.

PL/Java 1.4.0 release - *https://github.com/tada/pljava/tree/B1_4*.

Chapter 17

Greenplum Fuzzy String Match Extension

The Greenplum Database Fuzzy String Match extension provides functions to determine similarities and distance between strings based on various algorithms.

- *Soundex Functions*
- *Levenshtein Functions*
- *Metaphone Functions*
- *Double Metaphone Functions*
- *Installing and Uninstalling the Fuzzy String Match Functions*

The Greenplum Database installation contains the files required for the functions in this extension module and SQL scripts to define the extension functions in a database and remove the functions from a database.

Warning: The functions `soundex`, `metaphone`, `dmetaphone`, and `dmetaphone_alt` do not work well with multibyte encodings (such as UTF-8).

The Greenplum Database Fuzzy String Match extension is based on the PostgreSQL fuzzystrmatch module.

Soundex Functions

The Soundex system is a method of matching similar-sounding (similar phonemes) names by converting them to the same code.

Note: Soundex is most useful for English names.

These functions work with Soundex codes:

```
soundex(text string1) returns text
difference(text string1, text string2) returns int
```

The `soundex` function converts a string to its Soundex code. Soundex codes consist of four characters.

The `difference` function converts two strings to their Soundex codes and then reports the number of matching code positions. The result ranges from zero to four, zero being no match and four being an exact match. These are some examples:

```
SELECT soundex('hello world!');
SELECT soundex('Anne'), soundex('Ann'), difference('Anne', 'Ann');
SELECT soundex('Anne'), soundex('Andrew'), difference('Anne', 'Andrew');
SELECT soundex('Anne'), soundex('Margaret'), difference('Anne', 'Margaret');

CREATE TABLE s (nm text);

INSERT INTO s VALUES ('john');
INSERT INTO s VALUES ('joan');
INSERT INTO s VALUES ('wobbly');
INSERT INTO s VALUES ('jack');

SELECT * FROM s WHERE soundex(nm) = soundex('john');

SELECT * FROM s WHERE difference(s.nm, 'john') > 2;
```

For information about the Soundex indexing system see *http://www.archives.gov/research/census/soundex.html*.

Levenshtein Functions

These functions calculate the Levenshtein distance between two strings:

```
levenshtein(text source, text target, int ins_cost, int del_cost, int sub_cost)
 returns int
levenshtein(text source, text target) returns int
levenshtein_less_equal(text source, text target, int ins_cost, int del_cost,
 int sub_cost, int max_d) returns int
levenshtein_less_equal(text source, text target, int max_d) returns int
```

Both the `source` and `target` parameters can be any non-null string, with a maximum of 255 bytes. The cost parameters `ins_cost`, `del_cost`, and `sub_cost` specify cost of a character insertion, deletion, or substitution, respectively. You can omit the cost parameters, as in the second version of the function; in that case the cost parameters default to 1.

`levenshtein_less_equal` is accelerated version of `levenshtein` function for low values of distance. If actual distance is less or equal then `max_d`, then `levenshtein_less_equal` returns an accurate value of the distance. Otherwise, this function returns value which is greater than `max_d`. Examples:

```
test=# SELECT levenshtein('GUMBO', 'GAMBOL');
 levenshtein
-------------
           2
(1 row)

test=# SELECT levenshtein('GUMBO', 'GAMBOL', 2,1,1);
 levenshtein
-------------
           3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive',2);
 levenshtein_less_equal
------------------------
                      3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive',4);
 levenshtein_less_equal
------------------------
                      4
(1 row)
```

For information about the Levenshtein algorithm, see *http://www.levenshtein.net/*.

Metaphone Functions

Metaphone, like Soundex, is based on the idea of constructing a representative code for an input string. Two strings are then deemed similar if they have the same codes. This function calculates the metaphone code of an input string :

```
metaphone(text source, int max_output_length) returns text
```

The `source` parameter must be a non-null string with a maximum of 255 characters. The `max_output_length` parameter sets the maximum length of the output metaphone code; if longer, the output is truncated to this length. Example:

```
test=# SELECT metaphone('GUMBO', 4);
 metaphone
-----------
 KM
(1 row)
```

For information about the Metaphone algorithm, see *http://en.wikipedia.org/wiki/Metaphone*.

Double Metaphone Functions

The Double Metaphone system computes two "sounds like" strings for a given input string - a "primary" and an "alternate". In most cases they are the same, but for non-English names especially they can be a bit different, depending on pronunciation. These functions compute the primary and alternate codes:

```
dmetaphone(text source) returns text
dmetaphone_alt(text source) returns text
```

There is no length limit on the input strings. Example:

```
test=# select dmetaphone('gumbo');
 dmetaphone
------------
 KMP
(1 row)
```

For information about the Double Metaphone algorithm, see *http://en.wikipedia.org/wiki/Metaphone#Double_Metaphone*.

Installing and Uninstalling the Fuzzy String Match Functions

Greenplum Database supplies SQL scripts to install and uninstall the Fuzzy String Match extension functions.

To install the functions in a database, run the following SQL script:

```
psql -f $GPHOME/share/postgresql/contrib/fuzzystrmatch.sql
```

To uninstall the functions, run the following SQL script:

```
psql -f $GPHOME/share/postgresql/contrib/uninstall_fuzzystrmatch.sql
```

Note: When you uninstall the Fuzzy String Match functions from a database, routines that you created in the database that use the functions will no longer work.

Chapter 18

Summary of Greenplum Features

This section provides a high-level overview of the system requirements and feature set of Greenplum Database. It contains the following topics:

- *Greenplum SQL Standard Conformance*
- *Greenplum and PostgreSQL Compatibility*

Greenplum SQL Standard Conformance

The SQL language was first formally standardized in 1986 by the American National Standards Institute (ANSI) as SQL 1986. Subsequent versions of the SQL standard have been released by ANSI and as International Organization for Standardization (ISO) standards: SQL 1989, SQL 1992, SQL 1999, SQL 2003, SQL 2006, and finally SQL 2008, which is the current SQL standard. The official name of the standard is ISO/IEC 9075-14:2008. In general, each new version adds more features, although occasionally features are deprecated or removed.

It is important to note that there are no commercial database systems that are fully compliant with the SQL standard. Greenplum Database is almost fully compliant with the SQL 1992 standard, with most of the features from SQL 1999. Several features from SQL 2003 have also been implemented (most notably the SQL OLAP features).

This section addresses the important conformance issues of Greenplum Database as they relate to the SQL standards. For a feature-by-feature list of Greenplum's support of the latest SQL standard, see *SQL 2008 Optional Feature Compliance*.

Core SQL Conformance

In the process of building a parallel, shared-nothing database system and query optimizer, certain common SQL constructs are not currently implemented in Greenplum Database. The following SQL constructs are not supported:

1. Some set returning subqueries in `EXISTS` or `NOT EXISTS` clauses that Greenplum's parallel optimizer cannot rewrite into joins.
2. `UNION ALL` of joined tables with subqueries.
3. Set-returning functions in the `FROM` clause of a subquery.
4. Backwards scrolling cursors, including the use of `FETCH PRIOR`, `FETCH FIRST`, `FETCH ABOLUTE`, and `FETCH RELATIVE`.
5. In `CREATE TABLE` statements (on hash-distributed tables): a `UNIQUE` or `PRIMARY KEY` clause must include all of (or a superset of) the distribution key columns. Because of this restriction, only one `UNIQUE` clause or `PRIMARY KEY` clause is allowed in a `CREATE TABLE` statement. `UNIQUE` or `PRIMARY KEY` clauses are not allowed on randomly-distributed tables.
6. `CREATE UNIQUE INDEX` statements that do not contain all of (or a superset of) the distribution key columns. `CREATE UNIQUE INDEX` is not allowed on randomly-distributed tables.

   Note that `UNIQUE INDEXES` (but not `UNIQUE CONSTRAINTS`) are enforced on a part basis within a partitioned table. They guarantee the uniqueness of the key within each part or sub-part.
7. `VOLATILE` or `STABLE` functions cannot execute on the segments, and so are generally limited to being passed literal values as the arguments to their parameters.
8. Triggers are not supported since they typically rely on the use of `VOLATILE` functions.
9. Referential integrity constraints (foreign keys) are not enforced in Greenplum Database. Users can declare foreign keys and this information is kept in the system catalog, however.
10. Sequence manipulation functions `CURRVAL` and `LASTVAL`.
11. `DELETE WHERE CURRENT OF` and `UPDATE WHERE CURRENT OF` (positioned delete and positioned update operations).

SQL 1992 Conformance

The following features of SQL 1992 are not supported in Greenplum Database:

1. `NATIONAL CHARACTER` (`NCHAR`) and `NATIONAL CHARACTER VARYING` (`NVARCHAR`). Users can declare the `NCHAR` and `NVARCHAR` types, however they are just synonyms for `CHAR` and `VARCHAR` in Greenplum Database.

2. `CREATE ASSERTION` statement.
3. `INTERVAL` literals are supported in Greenplum Database, but do not conform to the standard.
4. `GET DIAGNOSTICS` statement.
5. `GRANT INSERT` or `UPDATE` privileges on columns. Privileges can only be granted on tables in Greenplum Database.
6. `GLOBAL TEMPORARY TABLE`s and `LOCAL TEMPORARY TABLE`s. Greenplum `TEMPORARY TABLE`s do not conform to the SQL standard, but many commercial database systems have implemented temporary tables in the same way. Greenplum temporary tables are the same as `VOLATILE TABLE`s in Teradata.
7. `UNIQUE` predicate.
8. `MATCH PARTIAL` for referential integrity checks (most likely will not be implemented in Greenplum Database).

SQL 1999 Conformance

The following features of SQL 1999 are not supported in Greenplum Database:

1. Large Object data types: `BLOB`, `CLOB`, `NCLOB`. However, the `BYTEA` and `TEXT` columns can store very large amounts of data in Greenplum Database (hundreds of megabytes).
2. `MODULE` (SQL client modules).
3. `CREATE PROCEDURE` (`SQL/PSM`). This can be worked around in Greenplum Database by creating a `FUNCTION` that returns `void`, and invoking the function as follows:

```
SELECT myfunc(args);
```

4. The PostgreSQL/Greenplum function definition language (`PL/PGSQL`) is a subset of Oracle's `PL/SQL`, rather than being compatible with the `SQL/PSM` function definition language. Greenplum Database also supports function definitions written in Python, Perl, Java, and R.
5. `BIT` and `BIT VARYING` data types (intentionally omitted). These were deprecated in SQL 2003, and replaced in SQL 2008.
6. Greenplum supports identifiers up to 63 characters long. The SQL standard requires support for identifiers up to 128 characters long.
7. Prepared transactions (`PREPARE TRANSACTION`, `COMMIT PREPARED`, `ROLLBACK PREPARED`). This also means Greenplum does not support `XA` Transactions (2 phase commit coordination of database transactions with external transactions).
8. `CHARACTER SET` option on the definition of `CHAR()` or `VARCHAR()` columns.
9. Specification of `CHARACTERS` or `OCTETS` (`BYTES`) on the length of a `CHAR()` or `VARCHAR()` column. For example, `VARCHAR(15 CHARACTERS)` or `VARCHAR(15 OCTETS)` or `VARCHAR(15 BYTES)`.
10. `CURRENT_SCHEMA` function.
11. `CREATE DISTINCT TYPE` statement. `CREATE DOMAIN` can be used as a work-around in Greenplum.
12. The *explicit table* construct.

SQL 2003 Conformance

The following features of SQL 2003 are not supported in Greenplum Database:

1. `MERGE` statements.
2. `IDENTITY` columns and the associated `GENERATED ALWAYS/GENERATED BY DEFAULT` clause. The `SERIAL` or `BIGSERIAL` data types are very similar to `INT` or `BIGINT GENERATED BY DEFAULT AS IDENTITY`.
3. `MULTISET` modifiers on data types.
4. `ROW` data type.
5. Greenplum Database syntax for using sequences is non-standard. For example, `nextval('seq')` is used in Greenplum instead of the standard `NEXT VALUE FOR seq`.
6. `GENERATED ALWAYS AS` columns. Views can be used as a work-around.

7. The sample clause (`TABLESAMPLE`) on `SELECT` statements. The `random()` function can be used as a work-around to get random samples from tables.
8. `NULLS FIRST/NULLS LAST` clause on `SELECT` statements and subqueries (nulls are always last in Greenplum Database).
9. The *partitioned join tables* construct (`PARTITION BY` in a join).
10. `GRANT SELECT` privileges on columns. Privileges can only be granted on tables in Greenplum Database. Views can be used as a work-around.
11. For `CREATE TABLE x (LIKE(y))` statements, Greenplum does not support the `[INCLUDING|EXCLUDING][DEFAULTS|CONSTRAINTS|INDEXES]` clauses.
12. Greenplum array data types are almost SQL standard compliant with some exceptions. Generally customers should not encounter any problems using them.

SQL 2008 Conformance

The following features of SQL 2008 are not supported in Greenplum Database:

1. `BINARY` and `VARBINARY` data types. `BYTEA` can be used in place of `VARBINARY` in Greenplum Database.
2. `FETCH FIRST` or `FETCH NEXT` clause for `SELECT`, for example:

   ```
   SELECT id, name FROM tab1 ORDER BY id OFFSET 20 ROWS FETCH
   NEXT 10 ROWS ONLY;
   ```

   Greenplum has `LIMIT` and `LIMIT OFFSET` clauses instead.
3. The `ORDER BY` clause is ignored in views and subqueries unless a `LIMIT` clause is also used. This is intentional, as the Greenplum optimizer cannot determine when it is safe to avoid the sort, causing an unexpected performance impact for such `ORDER BY` clauses. To work around, you can specify a really large `LIMIT`. For example: `SELECT * FROM mytable ORDER BY 1 LIMIT 9999999999`
4. The *row subquery* construct is not supported.
5. `TRUNCATE TABLE` does not accept the `CONTINUE IDENTITY` and `RESTART IDENTITY` clauses.

Greenplum and PostgreSQL Compatibility

Greenplum Database is based on PostgreSQL 8.2 with a few features added in from the 8.3 release. To support the distributed nature and typical workload of a Greenplum Database system, some SQL commands have been added or modified, and there are a few PostgreSQL features that are not supported. Greenplum has also added features not found in PostgreSQL, such as physical data distribution, parallel query optimization, external tables, resource queues for workload management and enhanced table partitioning. For full SQL syntax and references, see the *SQL Command Reference*.

Table 144: SQL Support in Greenplum Database

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
`ALTER AGGREGATE`	YES	
`ALTER CONVERSION`	YES	
`ALTER DATABASE`	YES	
`ALTER DOMAIN`	YES	
`ALTER FILESPACE`	YES	Greenplum Database parallel tablespace feature - not in PostgreSQL 8.2.15.
`ALTER FUNCTION`	YES	
`ALTER GROUP`	YES	An alias for *ALTER ROLE*
`ALTER INDEX`	YES	
`ALTER LANGUAGE`	YES	
`ALTER OPERATOR`	YES	
`ALTER OPERATOR CLASS`	**NO**	
`ALTER RESOURCE QUEUE`	YES	Greenplum Database workload management feature - not in PostgreSQL.
`ALTER ROLE`	YES	**Greenplum Database Clauses:** `RESOURCE QUEUE` *queue_name* `| none`
`ALTER SCHEMA`	YES	
`ALTER SEQUENCE`	YES	

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions

`ALTER TABLE`	YES	**Unsupported Clauses / Options:**
		`CLUSTER ON`
		`ENABLE/DISABLE TRIGGER`
		Greenplum Database Clauses:
		`ADD | DROP | RENAME | SPLIT | EXCHANGE PARTITION | SET SUBPARTITION TEMPLATE | SET WITH (REORGANIZE=true | false) | SET DISTRIBUTED BY`

`ALTER TABLESPACE`	YES	
`ALTER TRIGGER`	**NO**	
`ALTER TYPE`	YES	
`ALTER USER`	YES	An alias for *ALTER ROLE*
`ANALYZE`	YES	
`BEGIN`	YES	
`CHECKPOINT`	YES	
`CLOSE`	YES	
`CLUSTER`	YES	
`COMMENT`	YES	
`COMMIT`	YES	
`COMMIT PREPARED`	**NO**	

`COPY`	YES	**Modified Clauses:**
		`ESCAPE [ AS ] '`*escape*`' | 'OFF'`
		Greenplum Database Clauses:
		`[LOG ERRORS INTO` *error_table*`] SEGMENT REJECT LIMIT` *count* `[ROWS|PERCENT]`

`CREATE AGGREGATE`	YES	**Unsupported Clauses / Options:**
		`[ , SORTOP =` *sort_operator* `]`
		Greenplum Database Clauses:
		`[ , PREFUNC =` *prefunc* `]`
		Limitations:
		The functions used to implement the aggregate must be `IMMUTABLE` functions.

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
`CREATE CAST`	YES	
`CREATE CONSTRAINT TRIGGER`	**NO**	
`CREATE CONVERSION`	YES	
`CREATE DATABASE`	YES	
`CREATE DOMAIN`	YES	
`CREATE EXTERNAL TABLE`	YES	Greenplum Database parallel ETL feature - not in PostgreSQL 8.2.15.
`CREATE FUNCTION`	YES	**Limitations:** Functions defined as `STABLE` or `VOLATILE` can be executed in Greenplum Database provided that they are executed on the master only. `STABLE` and `VOLATILE` functions cannot be used in statements that execute at the segment level.
`CREATE GROUP`	YES	An alias for *CREATE ROLE*
`CREATE INDEX`	YES	**Greenplum Database Clauses:** `USING bitmap` (bitmap indexes) **Limitations:** `UNIQUE` indexes are allowed only if they contain all of (or a superset of) the Greenplum distribution key columns. On partitioned tables, a unique index is only supported within an individual partition - not across all partitions. `CONCURRENTLY` keyword not supported in Greenplum.
`CREATE LANGUAGE`	YES	
`CREATE OPERATOR`	YES	**Limitations:** The function used to implement the operator must be an `IMMUTABLE` function.
`CREATE OPERATOR CLASS`	**NO**	
`CREATE OPERATOR FAMILY`	**NO**	
`CREATE RESOURCE QUEUE`	YES	Greenplum Database workload management feature - not in PostgreSQL 8.2.15.

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
`CREATE ROLE`	YES	**Greenplum Database Clauses:** `RESOURCE QUEUE` *queue_name* `| none`
`CREATE RULE`	YES	
`CREATE SCHEMA`	YES	
`CREATE SEQUENCE`	YES	**Limitations:** The `lastval` and `currval` functions are not supported. The `setval` function is only allowed in queries that do not operate on distributed data.

`CREATE TABLE`	YES	**Unsupported Clauses / Options:**
		`[GLOBAL | LOCAL]`
		`REFERENCES`
		`FOREIGN KEY`
		`[DEFERRABLE | NOT DEFERRABLE]`
		Limited Clauses:
		`UNIQUE` or `PRIMARY KEY` constraints are only allowed on hash-distributed tables (`DISTRIBUTED BY`), and the constraint columns must be the same as or a superset of the distribution key columns of the table and must include all the distribution key columns of the partitioning key.
		Greenplum Database Clauses:
		`DISTRIBUTED BY (`*column*`, [ ... ] ) |`
		`DISTRIBUTED RANDOMLY`
		`PARTITION BY` *type* `(`*column* `[, ...])    (` *partition_specification*`, [...] )`
		`WITH (appendonly=true [,compresslevel=value,blocksize=value] )`

`CREATE TABLE AS`	YES	See *CREATE TABLE*

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
`CREATE TABLESPACE`	**NO**	**Greenplum Database Clauses:** `FILESPACE` *filespace_name*
`CREATE TRIGGER`	**NO**	
`CREATE TYPE`	YES	**Limitations:** The functions used to implement a new base type must be `IMMUTABLE` functions.
`CREATE USER`	YES	An alias for *CREATE ROLE*
`CREATE VIEW`	YES	
`DEALLOCATE`	YES	

`DECLARE`	YES	**Unsupported Clauses / Options:**
		`SCROLL`
		`FOR UPDATE [ OF column [, ...] ]`
		Limitations:
		Cursors are non-updatable, and cannot be backward-scrolled. Forward scrolling is supported.

`DELETE`	YES	**Unsupported Clauses / Options:** `RETURNING`
`DROP AGGREGATE`	YES	
`DROP CAST`	YES	
`DROP CONVERSION`	YES	
`DROP DATABASE`	YES	
`DROP DOMAIN`	YES	
`DROP EXTERNAL TABLE`	YES	Greenplum Database parallel ETL feature - not in PostgreSQL 8.2.15.
`DROP FILESPACE`	YES	Greenplum Database parallel tablespace feature - not in PostgreSQL 8.2.15.
`DROP FUNCTION`	YES	
`DROP GROUP`	YES	An alias for *DROP ROLE*
`DROP INDEX`	YES	
`DROP LANGUAGE`	YES	

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
`DROP OPERATOR`	YES	
`DROP OPERATOR CLASS`	**NO**	
`DROP OWNED`	**NO**	
`DROP RESOURCE QUEUE`	YES	Greenplum Database workload management feature - not in PostgreSQL 8.2.15.
`DROP ROLE`	YES	
`DROP RULE`	YES	
`DROP SCHEMA`	YES	
`DROP SEQUENCE`	YES	
`DROP TABLE`	YES	
`DROP TABLESPACE`	**NO**	
`DROP TRIGGER`	**NO**	
`DROP TYPE`	YES	
`DROP USER`	YES	An alias for *DROP ROLE*
`DROP VIEW`	YES	
`END`	YES	
`EXECUTE`	YES	
`EXPLAIN`	YES	

`FETCH`	YES	**Unsupported Clauses / Options:**
		`LAST`
		`PRIOR`
		`BACKWARD`
		`BACKWARD ALL`
		Limitations:
		Cannot fetch rows in a nonsequential fashion; backward scan is not supported.

`GRANT`	YES	
`INSERT`	YES	**Unsupported Clauses / Options:** `RETURNING`
`LISTEN`	**NO**	
`LOAD`	YES	
`LOCK`	YES	

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
`MOVE`	YES	See *FETCH*
`NOTIFY`	**NO**	
`PREPARE`	YES	
`PREPARE TRANSACTION`	**NO**	
`REASSIGN OWNED`	YES	
`REINDEX`	YES	
`RELEASE SAVEPOINT`	YES	
`RESET`	YES	
`REVOKE`	YES	
`ROLLBACK`	YES	
`ROLLBACK PREPARED`	**NO**	
`ROLLBACK TO SAVEPOINT`	YES	
`SAVEPOINT`	YES	

`SELECT`	YES	**Limitations:**
		Limited use of `VOLATILE` and `STABLE` functions in `FROM` or `WHERE` clauses
		Text search (`Tsearch2`) is not supported
		`FETCH FIRST` or `FETCH NEXT` clauses not supported
		Greenplum Database Clauses (OLAP):
		`[GROUP BY` *grouping_element* `[, ...]]`
		`[WINDOW` *window_name* `AS (`*window_specification*`)]`
		`[FILTER (WHERE` *condition*`)]` applied to an aggregate function in the `SELECT` list

`SELECT INTO`	YES	See *SELECT*
`SET`	YES	
`SET CONSTRAINTS`	**NO**	In PostgreSQL, this only applies to foreign key constraints, which are currently not enforced in Greenplum Database.
`SET ROLE`	YES	
`SET SESSION AUTHORIZATION`	YES	Deprecated as of PostgreSQL 8.1 - see *SET ROLE*

SQL Command	Supported in Greenplum	Modifications, Limitations, Exceptions
`SET TRANSACTION`	YES	
`SHOW`	YES	
`START TRANSACTION`	YES	
`TRUNCATE`	YES	
`UNLISTEN`	**NO**	
`UPDATE`	YES	**Unsupported Clauses:** `RETURNING` **Limitations:** `SET` not allowed for Greenplum distribution key columns.
`VACUUM`	YES	**Limitations:** `VACUUM FULL` is not recommended in Greenplum Database.
`VALUES`	YES	