PRODUCT DOCUMENTATION

Pivotal™ HAWQ

Version 1.2

Administrator Guide

Rev: A06

Notice

Copyright

Use of Open Source

About Pivotal Software, Inc.

Greenplum transitioned to a new corporate identity (Pivotal, Inc.) in 2013. As a result of this transition, there will be some legacy instances of our former corporate identity (Greenplum) appearing in our products and documentation. If you have any questions or concerns, please do not hesitate to contact us through our web site: *http://support.pivotal.io*.

Published September 2014

Updated November 2014

Contents

Chapter 1

HAWQ Overview

This chapter describes all of the components that comprise a HAWQ system, and how they work together.

About the HAWQ Architecture

HAWQ is designed as a MPP SQL processing engine optimized for analytics with full transaction support. HAWQ breaks complex queries into small tasks and distributes them to MPP query processing units for execution. The legacy query optimizer, dynamic pipeline, leading edge interconnect, and the specific query executor optimization for distributed storage are designed to work seamlessly, to support highest level of performance and scalability.

HAWQ's basic unit of parallelism is the segment instance. Multiple segment instances on commodity servers work together to form a single parallel query processing system. A query submitted to HAWQ is optimized, broken into smaller components, and dispatched to segments that work together to deliver a single result set. All relational operations - such as table scans, joins, aggregations, and sorts - simultaneously execute in parallel across the segments. Data from upstream components in the dynamic pipeline are transmitted to downstream components through the scalable User Datagram Protocol (UDP) interconnect.

Based on Hadoop's distributed storage, HAWQ has no single point of failure and supports fully-automatic online recovery. System states are continuously monitored, therefore if a segment fails, it is automatically removed from the cluster. During this process, the system continues serving customer queries, and the segments can be added back to the system when necessary.

HAWQ Master

The HAWQ *master* is the entry point to the system. It is the database process that accepts client connections and processes the SQL commands issued.

End-users interact with HAWQ through the master and can connect to the database using client programs such as psql or application programming interfaces (APIs) such as JDBC or ODBC.

The master is where the `global system catalog` resides. The global system catalog is the set of system tables that contain metadata about the HAWQ system itself. The master does not contain any user data; data resides only on *HDFS*. The master authenticates client connections, processes incoming SQL commands, distributes workload among segments, coordinates the results returned by each segment, and presents the final results to the client program.

HAWQ Segment

In HAWQ, the *segments* are the units which process the individual data modules simultaneously.

A segment differs from a master by being stateless and because it:

- Does not store the metadata for each database and table
- Does not store data on the local file system.

The master dispatches the SQL request to the segments along with the related metadata information to process. The metadata contains the HDFS url for the required table. The segment accesses the corresponding data using this URL.

HAWQ Storage

HAWQ stores all table data, except the system table, in HDFS. When a user creates a table, the metadata is stored on the master's local file system and the table content is stored in HDFS.

HAWQ also supports the Parquet storage format. For more information, see CREATE TABLE.

HAWQ Interconnect

The *interconnect* is the networking layer of HAWQ. When a user connects to a database and issues a query, processes are created on each segment to handle the query. The *interconnect* refers to the inter-process communication between the segments, as well as the network infrastructure on which this communication relies. The interconnect uses standard Ethernet switching fabric.

By default, the interconnect uses UDP (User Datagram Protocol) to send messages over the network. The HAWQ software performs the additional packet verification beyond what is provided by UDP. This means the reliability is equivalent to Transmission Control Protocol (TCP), and the performance and scalability exceeds that of TCP. If the interconnect used TCP, HAWQ would have a scalability limit of 1000 segment instances. With UDP as the current default protocol for the interconnect, this limit is not applicable.

Redundancy and Failover in HAWQ

HAWQ provides deployment options that protect the system from having a single point of failure. The following sections explain the redundancy components of HAWQ.

- About Master Mirroring
- About Segment Failover
- About Interconnect Redundancy

About Master Mirroring

You can optionally deploy a *backup* or *mirror* of the master instance on a separate host from the master node. A backup master host serves as a *warm standby* in the event that the primary master host becomes inoperable. The standby master is kept up to date by a transaction log replication process. The transaction log replication process runs on the standby master host and synchronizes the data between the primary and standby master hosts.

Since the master does not contain any user data, only the system catalog tables need to be synchronized between the primary and backup copies. When these tables are updated, changes are automatically copied over to the standby master to ensure synchronization with the primary master.

About Segment Failover

In HAWQ, the segments are stateless. This ensures faster recovery and better availability.

When a segment is down, the existing sessions are automatically reassigned to the remaining segments. If a new session is created during segment downtime, it succeeds on the remaining segments.

When the segments are operational again, the Fault Tolerance Service verifies their state, returning the segment number to normal. All the sessions are automatically reconfigured to use the full computing power.

About Interconnect Redundancy

The *interconnect* refers to the inter-process communication between the segments and the network infrastructure on which this communication relies. You can achieve a highly available interconnect by deploying dual Gigabit Ethernet switches on your network and deploying redundant Gigabit connections to the HAWQ host (master and segment) servers.

Chapter 2

HAWQ Query Processing

You can issue queries to HAWQ similarly to any database management system (DBMS). You can connect to the database instance on the HAWQ master host by using a client application such as psql and submitting SQL statements.

HAWQ offers the Pivotal Query Optimizer. This chapter describes how query planning works with the Pivotal Query Optimizer.

Query Planning and Dispatch

This section describes the following:

- Planning and Dispatch
- Pivotal Query Optimizer

Planning and Dispatch

The master receives, parses, and optimizes the query. The resulting query plan is either **parallel** or **targeted**. The master dispatches parallel query plans to all segments, as shown in Figure 2.1. The master dispatches targeted query plans to a single segment, as shown in Figure 2.2.

Each segment is responsible for executing local database operations on its own set of data.

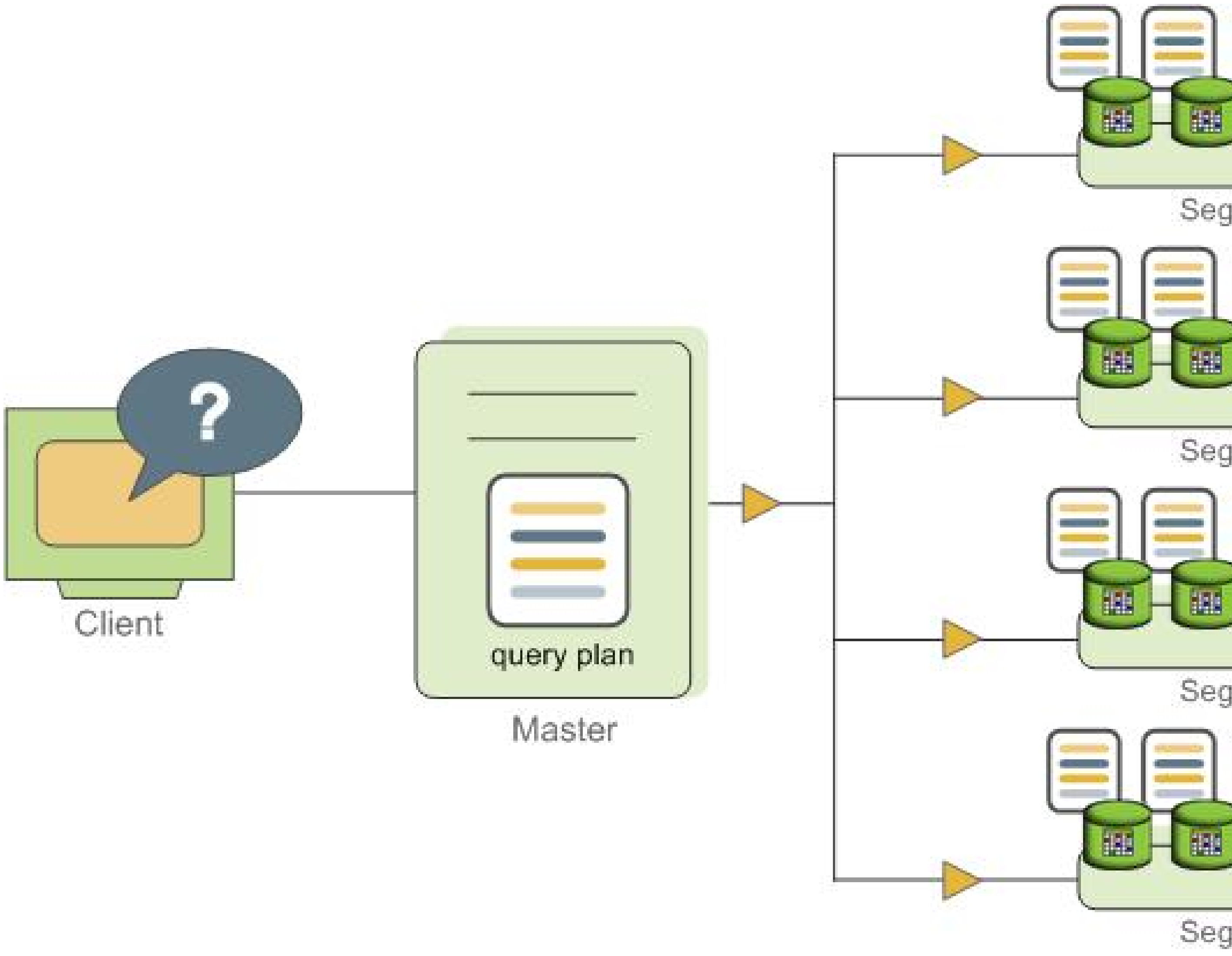


Figure 1: Dispatching Parallel Query Plans>

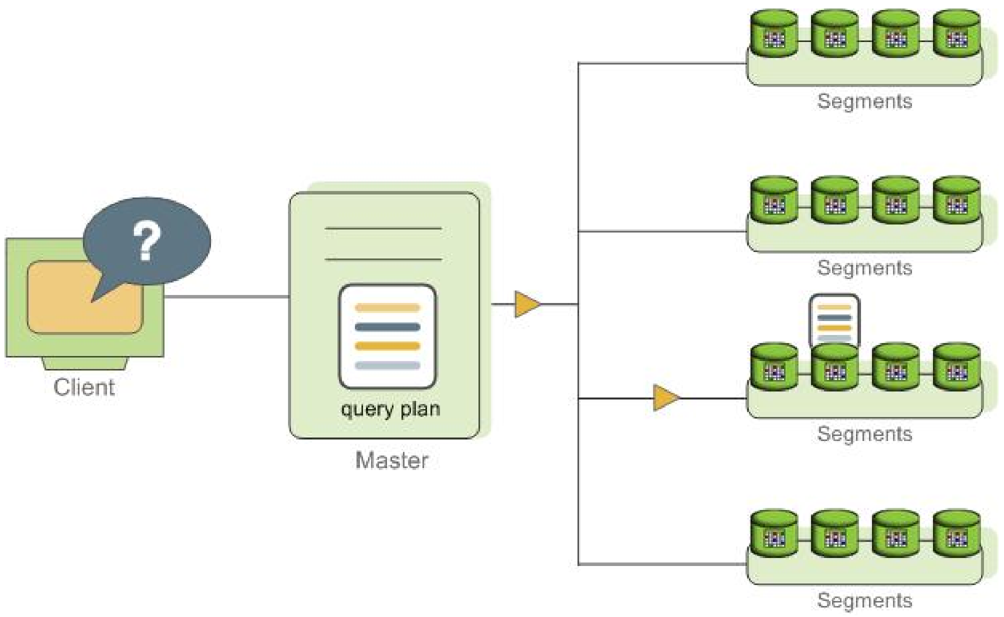


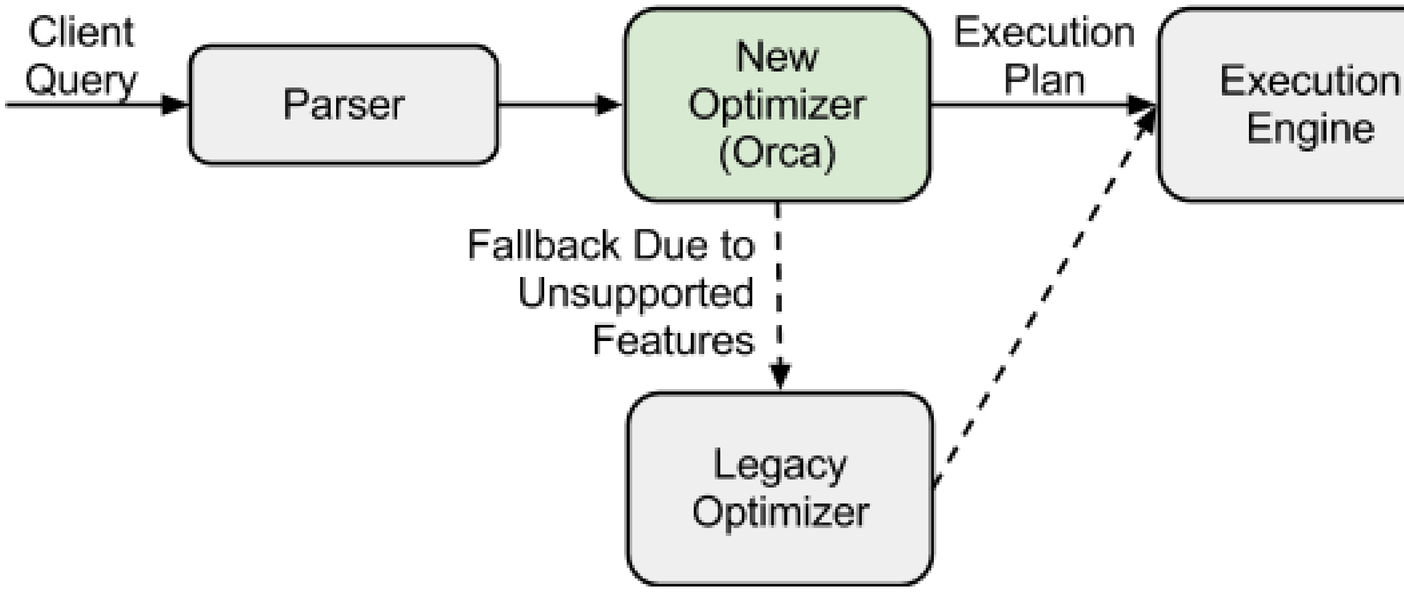
Figure 2: Dispatching Targeted Query Plans

Certain queries may access only data on a single segment, such as single-row INSERT, UPDATE, DELETE, or SELECT operations, or queries that filter on the table distribution key column(s). In queries such as these, the query plan is not dispatched to all segments, but is targeted at the segment that contains the affected or relevant row(s).

Pivotal Query Optimizer

The Pivotal Query Optimizer extends the planning and optimization capabilities of HAWQ. It is extensible, verifiable, and achieves better optimization using multi-core architectures. The Pivotal Query Optimizer also improves performance tuning.

The following flow chart shows how Pivotal Query Optimizer fits into the query planning architecture:



You can inspect the log to determine whether the Pivotal Query Optimizer or the legacy query optimizer produced the plan. The log message, "Optimizer produced plan" indicates that the Pivotal Query

Optimizer generated the plan for your query. If the legacy query optimizer generated the plan, the log message reads "Planner produced plan". To turn off logging, see optimizer_log.

HAWQ Query Plans

A *query plan* is the set of operations HAWQ will perform to produce the answer to a query. Each *node* or step in the plan represents a database operation such as a table scan, join, aggregation, or sort. Plans are read and executed from bottom to top.

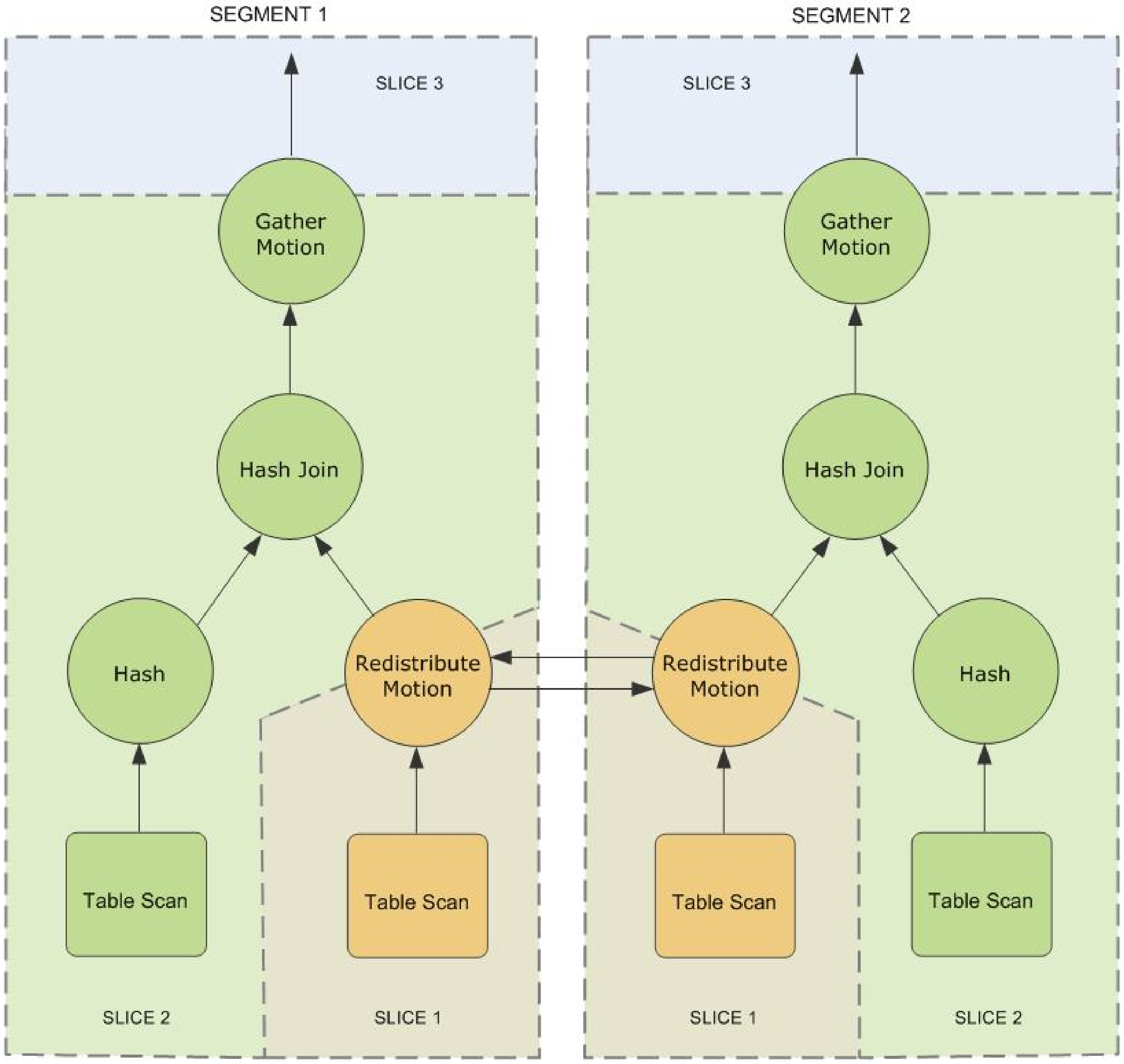
In addition to common database operations such as tables scans, joins, and so on, HAWQ has an additional operation type called *motion*. A motion operation involves moving tuples between the segments during query processing. Note that not every query requires a motion. For example, a targeted query plan does not require data to move across the interconnect.

To achieve maximum parallelism during query execution, HAWQ divides the work of the query plan into **slices**. A slice is a portion of the plan that segments can work on independently. A query plan is sliced wherever a motion operation occurs in the plan, with one slice on each side of the motion.

For example, consider the following simple query involving a join between two tables:

```
SELECT customer, amount
FROM sales JOIN customer USING (cust_id)
WHERE dateCol = '04-30-2008';
```

The following figure shows the query plan. Each segment receives a copy of the query plan and works on it in parallel.



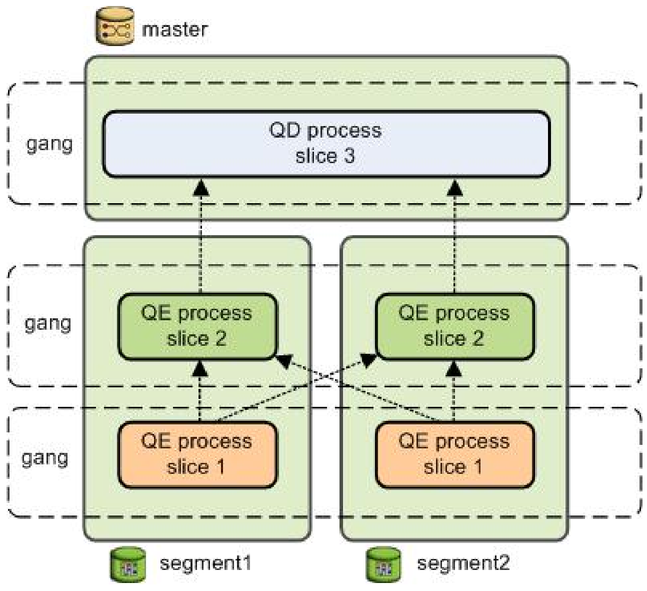
The query plan for this example has a redistribute motion that moves tuples between the segments to complete the join. The redistribute motion is necessary because the customer table is distributed across the segments by *cust_id*, but the sales table is distributed across the segments by *sale_id*. To perform the join, the sales tuples must be redistributed by *cust_id*. The plan is sliced on either side of the redistribute motion, creating slice 1 and slice 2.This query plan has another type of motion operation called a gather motion. A gather motion is when the segments send results back up to the master for presentation to the client. Because a query plan is always sliced wherever a motion occurs, this plan also has an implicit slice at the very top of the plan (slice 3). Not all query plans involve a gather motion. For example, a `CREATE TABLE x AS SELECT...` statement would not have a gather motion because tuples are sent to the newly created table, not to the master.

Parallel Query Execution

HAWQ creates a number of database processes to handle the work of a query. On the master, the query worker process is called the *query dispatcher* (QD). The QD is responsible for creating and dispatching the query plan. It also accumulates and presents the final results. On the segments, a query worker process is called a *query executor* (QE). A QE is responsible for completing its portion of work and communicating its intermediate results to the other worker processes.

There is at least one worker process assigned to each *slice* of the query plan. A worker process works on its assigned portion of the query plan independently. During query execution, each segment will have a number of processes working on the query in parallel.

Related processes that are working on the same slice of the query plan but on different segments are called *gangs*. As a portion of work is completed, tuples flow up the query plan from one gang of processes to the next. This inter-process communication between the segments is referred to as the *interconnect* component of HAWQ. The following figure shows the query worker processes on the master and two segment instances for the query plan.



Chapter 3

Using HAWQ to Query Data

This chapter describes the use of the SQL language in Pivotal HAWQ Database. SQL commands are typically entered using the standard PostgreSQL interactive terminal `psql`, but other programs that have similar functionality can be used as well.

Defining Queries

A query is a SQL command that views, changes or analyzes the data in a database. This section describes how to construct SQL queries in HAWQ Database.

- SQL Lexicon
- SQL Value Expressions

SQL Lexicon

SQL (structured query language) is the language used to access the database. The SQL language has a specific lexicon (words, special characters, etc.) used to construct queries or commands that the database engine can understand.

SQL input consists of a sequence of commands. A command is composed of a sequence of tokens, terminated by a semicolon (;). Which tokens are valid depends on the syntax of the particular command. The syntax rules for each command are described in *SQL Command Reference*.

HAWQ Database is based on PostgreSQL and adheres to the same SQL structure and syntax (with some minor exceptions). In most cases, the syntax is identical to PostgreSQL, however some commands may have additional or restricted syntax in HAWQ Database. For a complete explanation of the SQL rules and concepts, as implemented in PostgreSQL, refer to the section on SQL Syntax in the PostgreSQL documentation.

SQL Value Expressions

Value expressions are used in a variety of contexts, such as in the target list of the SELECT command, as new column values in INSERT or UPDATE, or in search conditions in a number of commands. The result of a value expression is sometimes called a *scalar*, to distinguish it from the result of a table expression (which is a table). Value expressions are therefore also called scalar expressions (or even simply expressions). The expression syntax allows the calculation of values from primitive parts using arithmetic, logical, set, and other operations.

Pivotal HAWQ supports the following value expressions:

- A function call
- An aggregate expression
- A window expression
- A scalar subquery
- Another value expression in parentheses, useful to group subexpressions and override precedence.

In addition to this list, there are a number of constructs that can be classified as expressions, but do not follow any general syntax rules. These generally have the semantics of a function or operator and are explained in *Using Functions and Operators*.

Function Calls

The syntax for a function call is the name of a function (possibly qualified with a schema name), followed by its argument list enclosed in parentheses:

```
function
([expression  [, expression  ... ]] )
```

For example, the following function call computes the square root of 2:

```
sqrt(2)
```

The list of built-in functions is listed in *Built-in Functions and Operators*. Other functions may be added by the user.

Aggregate Expressions

An aggregate expression represents the application of an aggregate function across the rows selected by a query. An aggregate function reduces multiple inputs to a single output value, such as the sum or average of the inputs. The syntax of an aggregate expression is one of the following:

```
aggregate_name (expression [ , ... ] ) [FILTER (WHERE condition)]
aggregate_name (ALL expression [ , ... ] ) [FILTER (WHERE condition)]
aggregate_name (DISTINCT expression [ , ... ] ) [FILTER (WHERE condition)]
aggregate_name ( * ) [FILTER (WHERE condition)]
```

Where *aggregate_name* is a previously defined aggregate (possibly qualified with a schema name), and *expression* is any value expression that does not itself contain an aggregate expression.

The first form of aggregate expression invokes the aggregate across all input rows for which the given expression(s) yield non-null values. The second form is the same as the first, since ALL is the default. The third form invokes the aggregate for all distinct non-null values of the expressions found in the input rows. The last form invokes the aggregate once for each input row regardless of null or non-null values; since no particular input value is specified, it is generally only useful for the count(*) aggregate function.

For example, count(*) yields the total number of input rows; count(f1) yields the number of input rows in which f1 is non-null; count(distinct f1) yields the number of distinct non-null values of f1.

The FILTER clause allows you to specify a condition to limit the input rows to the aggregate function. For example:

```
SELECT count(*) FILTER (WHERE gender='F') FROM employee;
```

The WHERE *condition* of the FILTER clause cannot contain a set returning function, subquery, a window function, or an outer reference. If using a user-defined aggregate function, the state transition function must be declared as STRICT (see CREATE AGGREGATE).

The predefined aggregate functions are described in Aggregate Functions. Other aggregate functions may be added by the user.

An aggregate expression may only appear in the result list or HAVING clause of a SELECT command. It is forbidden in other clauses, such as WHERE, because those clauses are logically evaluated before the results of aggregates are formed.

When an aggregate expression appears in a subquery (see *Scalar Subqueries* and Subquery Expressions in *Table 1: Build-In Functions and Operators*), the aggregate is normally evaluated over the rows of the subquery. But an exception occurs if the aggregate's arguments contain only outer-level variables: the aggregate then belongs to the nearest such outer level, and is evaluated over the rows of that query. The aggregate expression as a whole is then an outer reference for the subquery it appears in, and acts as a constant over any one evaluation of that subquery. The restriction about appearing only in the result list or HAVING clause applies, with respect to the query level of aggregate.

Pivotal HAWQ Database currently does not support DISTINCT with more than one input expression.

Scalar Subqueries

A scalar subquery is an ordinary SELECT query in parentheses that returns exactly one row with one column. The SELECT query is executed and the single returned value is used in the surrounding value expression. It is an error to use a query that returns more than one row or more than one column as a scalar subquery. A scalar subquery is correlated if it contains references to the outer query block.

Using Functions and Operators

- Functions in HAWQ
- User Defined Functions
- Built-in Functions and Operators

Functions in HAWQ

Function Type	Pivotal 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()

Data is divided up across segments — each segment is a distinct HAWQ 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 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();
```

User Defined Functions

HAWQ supports user defined functions. Every kind of function can take base types, composite types, or combinations of these as arguments (parameters). In addition, every kind of function can return a base type or a composite type. Functions can also be defined to return sets of base or composite values. Please see *CREATE FUNCTION* for more information.

Use the CREATE FUNCTION command to register user defined functions. By default, user defined functions are declared as VOLATILE, so if your user-defined function is IMMUTABLE or STABLE, you must specify the correct volatility level when you register your function.

Function Volatility

Every function has a **volatility** classification, with the possibilities being VOLATILE, STABLE, or IMMUTABLE. VOLATILE is the default if the *CREATE FUNCTION* command does not specify a category. The volatility category is a promise to the optimizer about the behavior of the function:

- A VOLATILE function can do anything, including modifying the database. It can return different results on successive calls with the same arguments. The optimizer makes no assumptions about the behavior of such functions. A query using a volatile function will re-evaluate the function at every row where its value is needed.
- A STABLE function cannot modify the database and is guaranteed to return the same results given the same arguments for all rows within a single statement. This category allows the optimizer to optimize multiple calls of the function to a single call. In particular, it is safe to use an expression containing such a function in an index scan condition. (Since an index scan will evaluate the comparison value only once, not once at each row, it is not valid to use a VOLATILE function in an index scan condition.)
- An IMMUTABLE function cannot modify the database and is guaranteed to return the same results given the same arguments forever. This category allows the optimizer to pre-evaluate the function when a query calls it with constant arguments. For example, a query like SELECT ... WHERE x = 2 + 2 can be simplified on sight to SELECT ... WHERE x = 4, because the function underlying the integer addition operator is marked IMMUTABLE.

For best optimization results, you should label your functions with the strictest volatility category that is valid for them.

Any function with side-effects must be labeled VOLATILE, so that calls to it cannot be optimized away. Even a function with no side-effects needs to be labeled VOLATILE if its value can change within a single query; some examples are random(), currval(), timeofday().

Another important example is that the `current_timestamp` family of functions qualify as STABLE, since their values do not change within a transaction.

There is relatively little difference between STABLE and IMMUTABLE categories when considering simple interactive queries that are planned and immediately executed: it doesn't matter a lot whether a function is executed once during planning or once during query execution startup. But there is a big difference if the plan is saved and reused later. Labeling a function IMMUTABLE when it really isn't might allow it to be prematurely folded to a constant during planning, resulting in a stale value being re-used during subsequent uses of the plan. This is a hazard when using prepared statements or when using function languages that cache plans (such as PL/pgSQL).

For functions written in SQL or in any of the standard procedural languages, there is a second important property determined by the volatility category, namely the visibility of any data changes that have been made by the SQL command that is calling the function. A VOLATILE function will see such changes, a STABLE or IMMUTABLE function will not. STABLE and IMMUTABLE functions use a snapshot established as of the start of the calling query, whereas VOLATILE functions obtain a fresh snapshot at the start of each query they execute.

Because of this snapshotting behavior, a function containing only SELECT commands can safely be marked STABLE, even if it selects from tables that might be undergoing modifications by concurrent queries. PostgreSQL will execute all commands of a STABLE function using the snapshot established for the calling query, and so it will see a fixed view of the database throughout that query.

The same snapshotting behavior is used for SELECT commands within IMMUTABLE functions. It is generally unwise to select from database tables within an IMMUTABLEfunction at all, since the immutability will be broken if the table contents ever change. However, PostgreSQL does not enforce that you do not do that.

A common error is to label a function IMMUTABLE when its results depend on a configuration parameter. For example, a function that manipulates timestamps might well have results that depend on the timezone setting. For safety, such functions should be labeled STABLE instead.

When you create user defined functions, avoid using fatal errors or destructive calls. HAWQ may respond to such errors with a sudden shutdown or restart.

In HAWQ, the shared library files for user-created functions must reside in the same library path location on every host in the HAWQ array (masters, segments, and mirrors).

Important: HAWQ does not support the following:

- Enhanced table functions
- PL/Java Type Maps

User Defined Types

HAWQ can be extended to support new data types. This section describes how to define new base types, which are data types defined below the level of the SQL language. Creating a new base type requires implementing functions to operate on the type in a low-level language, usually C.

A user-defined type must always have input and output functions. These functions determine how the type appears in strings (for input by the user and output to the user) and how the type is organized in memory. The input function takes a null-terminated character string as its argument and returns the internal (in memory) representation of the type. The output function takes the internal representation of the type as argument and returns a null-terminated character string. If we want to do anything more with the type than merely store it, we must provide additional functions to implement whatever operations we'd like to have for the type.

You should be careful to make the input and output functions inverses of each other. If you do not, you will have severe problems when you need to dump your data into a file and then read it back in. This is a particularly common problem when floating-point numbers are involved.

Optionally, a user-defined type can provide binary input and output routines. Binary I/O is normally faster but less portable than textual I/O. As with textual I/O, it is up to you to define exactly what the external binary representation is. Most of the built-in data types try to provide a machine-independent binary representation.

Once we have written the I/O functions and compiled them into a shared library, we can define the complex type in SQL. First we declare it as a shell type:

```
CREATE TYPE complex;
```

This serves as a placeholder that allows us to reference the type while defining its I/O functions. Now we can define the I/O functions:

```
CREATE FUNCTION complex_in(cstring)
    RETURNS complex
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
   RETURNS complex
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
   RETURNS bytea
   AS 'filename'
```

```
    LANGUAGE C IMMUTABLE STRICT;
```

Finally, we can provide the full definition of the data type:

```
CREATE TYPE complex (
   internallength = 16,
   input = complex_in,
   output = complex_out,
   receive = complex_recv,
   send = complex_send,
   alignment = double
);
```

When you define a new base type, HAWQ automatically provides support for arrays of that type. For historical reasons, the array type has the same name as the base type with the underscore character (_) prepended.

Once the data type exists, we can declare additional functions to provide useful operations on the data type. Operators can then be defined atop the functions, and if needed, operator classes can be created to support indexing of the data type.

For further details, see the description of the *CREATE TYPE* command.

User Defined Operators

Every operator is "syntactic sugar" for a call to an underlying function that does the real work; so you must first create the underlying function before you can create the operator. However, an operator is not merely syntactic sugar, because it carries additional information that helps the query planner optimize queries that use the operator. The next section will be devoted to explaining that additional information.

HAWQ supports left unary, right unary, and binary operators. Operators can be overloaded; that is, the same operator name can be used for different operators that have different numbers and types of operands. When a query is executed, the system determines the operator to call from the number and types of the provided operands.

Here is an example of creating an operator for adding two complex numbers. We assume we've already created the definition of type complex. First we need a function that does the work, then we can 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 = +
);
```

Now we could execute a query like this:

```
SELECT (a + b) AS c FROM test_complex;

        c
-----------------
 (5.2,6.05)
 (133.42,144.95)
```

We've shown how to create a binary operator here. To create unary operators, just omit one of leftarg (for left unary) or rightarg (for right unary). The procedure clause and the argument clauses are the only required items in CREATE OPERATOR. The commutator clause shown in the example is an optional hint to the query optimizer. Further details aboutcommutator and other optimizer hints appear in the next section.

Built-in Functions and Operators

The following table lists the categories of built-in functions and operators. All functions and operators are supported in HAWQ (and PostgreSQL). STABLE and VOLATILE functions are subject to the restrictions noted in *User Defined Functions*. For more information about the built-in functions and operators listed in the table, see the PostgreSQL documentation at *http://www.postgresql.org/docs/9.1/static/functions.html*.

Table 1: Build-In Functions and Operators

Operator/Function	Volatile	Stable
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 ad Operators		
Network Address Functions and Operators		
Sequence Manipulation Functions	currval lastval nextval setval	

Operator/Function	**Volatile**	**Stable**
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 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

Operator/Function	Volatile	Stable

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)

Advanced Analytic Functions

HAWQ provides specialized and native built-in advanced analytic **immutable** functions.

Table 2: 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 any element.

Table 3: Advanced Aggregate Functions

Function	Return Type	Full Syntax	Description
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.
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.

Function	Return Type	Full Syntax	Description
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.
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 4

Using Procedural Languages

HAWQ allows user-defined functions to be written in other languages besides SQL and C. These other languages are generically called *procedural languages*(PLs). For a function written in a procedural language, the database server has no built-in knowledge about how to interpret the function's source text. Instead, 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, etc. itself, or it could serve as "glue" between HAWQ 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 three procedural languages available in the HAWQ distribution:

- PL/pgSQL
- PL/R
- PL/Java

This chapter describes the following:

- *Using PL/pgSQL*
- *Supported Argument and Result Data Types*

Using PL/pgSQL

SQL is the language of most other relational databases use as query language. It is portable and easy to learn. But every SQL statement must be executed individually by the database server. PL/pgSQL is a loadable procedural language:

- creates functions and trigger procedures
- adds control structures to theSQLlanguage
- performs complex computations
- inherits all user-defined types, functions, and operators
- is trusted by the server

You can use functions created with PL/pgSQLwith any database that supports built-in functions. For example, it is possible to create complex conditional computation functions and later use them to define operators or use them in index expressions.

Every SQLstatement must be executed individually by the database server. Your client application must send each query to the database server, wait for it to be processed, receive and process the results, do some computation, then send further queries to the server. This requires interprocess communication and incurs network overhead if your client is on a different machine than the database server.

With PL/pgSQL, you can group a block of computation and a series of queries inside the database server, thus having the power of a procedural language and the ease of use of SQL, but with considerable savings of client/server communication overhead.

- Extra round trips between client and server are eliminated
- Intermediate results that the client does not need do not have to be marshaled or transferred between server and client
- Multiple rounds of query parsing can be avoided

This can result in a considerable performance increase as compared to an application that does not use stored functions.

PL/pgSQL supports all the data types, operators, and functions of SQL.

Supported Argument and Result Data Types

Functions written in PL/pgSQL accept as arguments any scalar or array data type supported by the server, and they can return a result containing this data type. They can also accept or return any composite type (row type) specified by name. It is also possible to declare a PL/pgSQL function as returning record, which means that the result is a row type whose columns are determined by specification in the calling query. See *Table Functions*.

PL/pgSQL functions can be declared to accept a variable number of arguments by using the VARIADIC marker. This works exactly the same way as for SQL functions. See *SQL Functions with Variable Numbers of Arguments*.

PL/pgSQLfunctions can also be declared to accept and return the polymorphic typesanyelement,anyarray,anynonarray, andanyenum. The actual data types handled by a polymorphic function can vary from call to call, as discussed in *Section 34.2.5*. An example is shown in *Section 38.3.1*.

PL/pgSQL functions can also be declared to return a "set" (or table) of any data type that can be returned as a single instance. Such a function generates its output by executing RETURN NEXT for each desired element of the result set, or by using RETURN QUERY to output the result of evaluating a query.

Finally, a PL/pgSQL function can be declared to returnvoidif it has no useful return value.

PL/pgSQL functions can also be declared with output parameters in place of an explicit specification of the return type. This does not add any fundamental capability to the language, but it is often convenient, especially for returning multiple values. The RETURNS TABLE notation can also be used in place of RETURNS SETOF .

This topics describes the following PL/pgSQLconcepts:

- Table Functions
- SQL Functions with Variable number of Arguments
- Polymorphic Functions

Table Functions

Table functions are functions that produce a set of rows, made up of either base data types (scalar types) or composite data types (table rows). They are used like a table, view, or subquery in the FROM clause of a query. Columns returned by table functions can be included in SELECT, JOIN, or WHERE clauses in the same manner as a table, view, or subquery column.

If a table function returns a base data type, the single result column name matches the function name. If the function returns a composite type, the result columns get the same names as the individual attributes of the type.

A table function can be aliased in the FROM clause, but it also can be left unaliased. If a function is used in the FROM clause with no alias, the function name is used as the resulting table name.

Some examples:

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo
    WHERE foosubid IN (
                        SELECT foosubid
                        FROM getfoo(foo.fooid) z
                        WHERE z.fooid = foo.fooid
                      );

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);

SELECT * FROM vw_getfoo;
```

In some cases, it is useful to define table functions that can return different column sets depending on how they are invoked. To support this, the table function can be declared as returning the pseudotype record. When such a function is used in a query, the expected row structure must be specified in the query itself, so that the system can know how to parse and plan the query. Consider this example:

```
SELECT *
    FROM dblink('dbname=mydb', 'SELECT proname, prosrc FROM pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

The`dblink` function executes a remote query (see`contrib/dblink`). It is declared to return `record` since it might be used for any kind of query. The actual column set must be specified in the calling query so that the parser knows, for example, what `*` should expand to.

SQL Functions with Variable Numbers of Arguments

SQL functions can be declared to accept variable numbers of arguments, so long as all the "optional" arguments are of the same data type. The optional arguments will be passed to the function as an array. The function is declared by marking the last parameter as VARIADIC; this parameter must be declared as being of an array type. For example:

```
CREATE FUNCTION mleast(VARIADIC numeric[]) RETURNS numeric AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT mleast(10, -1, 5, 4.4);
 mleast
--------
     -1
(1 row)
```

Effectively, all the actual arguments at or beyond the VARIADIC position are gathered up into a one-dimensional array, as if you had written

```
SELECT mleast(ARRAY[10, -1, 5, 4.4]);    -- doesn't work
```

You can't actually write that, though; or at least, it will not match this function definition. A parameter marked VARIADIC matches one or more occurrences of its element type, not of its own type.

Sometimes it is useful to be able to pass an already-constructed array to a variadic function; this is particularly handy when one variadic function wants to pass on its array parameter to another one. You can do that by specifying VARIADIC in the call:

```
SELECT mleast(VARIADIC ARRAY[10, -1, 5, 4.4]);
```

This prevents expansion of the function's variadic parameter into its element type, thereby allowing the array argument value to match normally. VARIADIC can only be attached to the last actual argument of a function call.

Polymorphic Types

Four pseudo-types of special interest are anyelement,anyarray, anynonarray, andanyenum, which are collectively called *polymorphic types*. Any function declared using these types is said to be a*polymorphic function*. A polymorphic function can operate on many different data types, with the specific data type(s) being determined by the data types actually passed to it in a particular call.

Polymorphic arguments and results are tied to each other and are resolved to a specific data type when a query calling a polymorphic function is parsed. Each position (either argument or return value) declared as anyelement is allowed to have any specific actual data type, but in any given call they must all be the sam eactual type. Each position declared as anyarray can have any array data type, but similarly they must all be the same type. If there are positions declared anyarray and others declared anyelement, the actual array type in the anyarray positions must be an array whose elements are the same type appearing in the anyelement positions.anynonarray is treated exactly the same as anyelement, but adds the additional constraint that the actual type must not be an array type. anyenum is treated exactly the same as anyelement, but adds the additional constraint that the actual type must be an enum type.

Thus, when more than one argument position is declared with a polymorphic type, the net effect is that only certain combinations of actual argument types are allowed. For example, a function declared as equal(anyelement, anyelement) will take any two input values, so long as they are of the same data type.

When the return value of a function is declared as a polymorphic type, there must be at least one argument position that is also polymorphic, and the actual data type supplied as the argument determines the actual result type for that call. For example, if there were not already an array subscripting mechanism, one could define a function that implements subscripting `assubscript(anyarray, integer)` returns `anyelement`. This declaration constrains the actual first argument to be an array type, and allows the parser to infer the correct result type from the actual first argument's type. Another example is that a function declared `asf(anyarray)` returns `anyenumwill` only accept arrays of `enum` types.

Note that `anynonarray` and `anyenum` do not represent separate type variables; they are the same type as `anyelement`, just with an additional constraint. For example, declaring a function as `f(anyelement, anyenum)` is equivalent to declaring it as `f(anyenum, anyenum)`; both actual arguments have to be the same enum type.

Variadic functions described in *SQL Functions with Variable Numbers of Arguments*can be polymorphic: this is accomplished by declaring its last parameter as `VARIADICanyarray`. For purposes of argument matching and determining the actual result type, such a function behaves the same as if you had written the appropriate number of `anynonarray` parameters.

Chapter 5

Configuring Client Authentication

When HAWQ is first initialized, the system contains one predefined *superuser* role. This role will have the same name as the operating system user who initialized the HAWQ Database system. This role is referred to as gpadmin. By default, the system is configured to only allow local connections to the database from the gpadmin role. If you want to allow any other roles to connect to the database, or if you want to allow connections from remote hosts, HAWQ must be configured to allow these connections.

This chapter explains how to configure client connections and authentication to HAWQ Database.

Allowing Connections to HAWQ

Client access and authentication is controlled by the configuration file, pg_hba.conf. HBA stands for host-based authentication. The `pg_hba.conf` file stored in the master instance controls client access to and authentication for your HAWQ system. HAWQ segments also have `pg_hba.conf` files configured to allow only client connections from the master host and to never accept client connections.

Important: Never alter the `pg_hba.conf` file on your segments.

The general format of the `pg_hba.conf` file is a set of records, one record per line. HAWQ ignores blank lines and any text after the `#` comment character. A record consists of a number of fields, separated by spaces and/or tabs. Fields can contain white space if the field value is quoted. Records cannot be continued across lines.

Each remote client access record has the following format:

```
host database role CIDR-address authentication-method
```

Each UNIX-domain socket access record has the following format:

```
local database role authentication-method
```

The following table describes meaning of each field.

Table 4: pg_hba.conf Fields

Field	Description
local	Matches connection attempts using UNIX-domain sockets. Without a record of this type, UNIX-domain socket connections are disallowed.
host	Matches connection attempts made using TCP/IP. Remote TCP/IP connections will not be possible unless the server is started with an appropriate value for the listen_addresses server configuration parameter.
hostssl	Matches connection attempts made using TCP/IP, but only when the connection is made with SSL encryption. SSL must be enabled at server start time by setting the ssl configuration parameter
hostnossl	Matches connection attempts made over TCP/IP that do not use SSL.
database	Specifies which database names this record matches. Specify "all" to match all databases. Specify multiple database names by separating them with commas. A separate file containing database names can be specified by preceding the file name with @.

Field	Description
role	Specifies which database role names this record matches. The value "all" matches all roles. If the specified role is a group and you want all members of that group to be included, precede the role name with a +. Multiple role names can be supplied by separating them with commas. A separate file containing role names can be specified by preceding the file name with @.
CIDR-address	Specifies the client machine IP address range that this record matches. It contains an IP address in standard dotted decimal notation and a CIDR mask length. IP addresses can only be specified numerically, not as domain or host names. The mask length indicates the number of high-order bits of the client IP address that must match. Bits to the right of this must be zero in the specified IP address. There must not be any white space between the IP address, the /, and the CIDR mask length. Typical examples of a CIDR-address are 172.20. 143.89/32 for a single host, or 172.20.143.0/24 for a small network, or 10.6.0.0/16 for a larger one. To specify a single host, use a CIDR mask of 32 for IPv4 or 128 for IPv6. In a network address, do not omit trailing zeroes.
IP-address IP-mask	These fields can be used as an alternative to the CIDR-address notation. Instead of specifying the mask length, the actual mask is specified in a separate column. For example, 255.0.0.0 represents an IPv4 CIDR mask length of 8, and 255.255.255.255 represents a CIDR mask length of 32. These fields only apply to host, hostssl, and hostnossl records.

Field	Description
authentication method	Specifies the authentication method to use when connecting. The choices are as follows: • **trust**: Allow the connection unconditionally. This method allows anyone that can connect to the HAWQ master to login without the need for a password. • **reject**: Reject the connection unconditionally. This is useful for "filtering out" certain hosts from a group. • **md5**: Require the client to supply an MD5-encrypted password for authentication. • **password**: Require the client to supply an unencrypted password for authentication. Since the password is sent in clear text over the network, this should not be used on untrusted networks. • **gss**: Use GSSAPI to authenticate the user. This is only available for TCP/IP connections. • **krb5**: Use Kerberos to authenticate the user. This is only available for TCP/IP connections. • **ldap**: Authenticate using an LDAP server. • **cert**: Authenticate using SSL client certificates.

The pg_hba.conf File

This example shows how to edit the `pg_hba.conf` file of the master to allow remote client access to all databases from all roles using encrypted password authentication.

Note: For a more secure system, consider removing all connections that use trust authentication from your master `pg_hba.conf`. Trust authentication means the role is granted access without any authentication, therefore bypassing all security. Replace trust entries with ident authentication if your system has an ident service available.

To edit the pg_hba.conf file:

1. Open the file `$MASTER_DATA_DIRECTORY/pg_hba.conf` in a text editor.
2. Add a line to the file for each type of connection you want to allow. Records are read sequentially, so the order of the records is significant. Typically, earlier records will have tight connection match parameters and weaker authentication methods, while later records will have looser match parameters and stronger authentication methods.

   For example:

```
# allow the gpadmin user local access to all databases
# using ident authentication
local all gpadmin ident sameuser
host all gpadmin 127.0.0.1/32 ident
host all gpadmin ::1/128 ident
# allow the 'dba' role access to any database from any
# host with IP address 192.168.x.x and use md5encrypted
# passwords to authenticate the user
# Note that to use SHA-256 encryption, replace md5 with
# password in the line below host all dba192.168.0.0/32 md5
# allow all roles access to any database from any
# host and use ldap to authenticate the user. Greenplum role
# names must match the LDAP common name.
```

```
host all all 192.168.0.0/32 ldap ldapserver=usldap1
ldapport=1389 ldapprefix="cn="
ldapsuffix=",ou=People,dc=company,dc=com"
```

3. When you have completed your entries, save and close the file.
4. Reload the pg_hba.conf configuration file for your changes to take effect:

```
$ gpstop -u
```

Note: You can also control database access by setting object privileges. The `pg_hba.conf` file just controls who can initiate a database session and how those connections are authenticated.

Username Maps

When using an external authentication system like Ident or GSSAPI, the operating system user initiating the connection might want to connect as a different database user. You can set up a user name map that allows a user with an operating system username to connect with a database username. Each connection may require a different mapping using the `pg_hba.conf` file which supports all authentication methods that receive external usernames. Use the map-name parameter in the options field of the `pg_hba.conf` to indicate the map to use for each individual connection.

For example:

```
map=map-name
```

Username maps are defined in the ident map file, which by default is named `pg_ident.conf` and is stored in the cluster's data directory. (It is possible to place the map file elsewhere, however; see the *ident_file* configuration parameter.) The ident map file contains lines of the general form:

```
map-name system-username database-username
```

Comments and whitespace are handled in the same way as in `pg_hba.conf`. The map-name is an arbitrary name that will be used to refer to this mapping in `pg_hba.conf`. The other two fields specify an operating system user name and a matching database user name. The same map-name can be used repeatedly to specify multiple user-mappings within a single map.

There is no restriction regarding how many database users a given operating system user can correspond to, nor vice versa. Thus, entries in a map should be thought of as meaning "this operating system user is allowed to connect as this database user", rather than implying that they are equivalent. The connection will be allowed if there is any map entry that matches the user name obtained from the external authentication system to the database user name that the user has requested to connect as.

If the system-username field starts with a slash (/), the remainder of the field is treated as a regular expression. (See *Section 9.7.3.1* for details of PostgreSQL's regular expression syntax. Regular expressions in username maps are always treated as being "advanced" flavor.) The regular expression can include a single capture, or parenthesized subexpression, which can then be referenced in thedatabase-username field as \1 (backslash-one). This allows the mapping of multiple usernames in a single line, which is particularly useful for simple syntax substitutions.

For example, these entries:

```
mymap   /^(.*)@mydomain\.com$      \1
mymap   /^(.*)@otherdomain\.com$   guest
```

will remove the domain part for users with system usernames that end with @mydomain.com, and allow any user whose system name ends with @otherdomain.com to log in as guest.

Note: Keep in mind that by default, a regular expression can match just part of a string. It's usually wise to use `^` and `$`, as shown in the above example, to force the match to be to the entire system username.

The pg_ident.conf file is read on start-up and when the main server process receives a SIGHUP signal. If you edit the file on an active system, you will need to signal the server (using `pg_ctl reloador kill -HUP`) to make it re-read the file.

A `pg_ident.conf` file that could be used in conjunction with the `pg_hba.conf` file in *Example 19-1* is shown in *Example 19-2*. In this example setup, anyone logged in to a machine on the 192.168 network that does not have the Unix username `bryanh`, `ann`, or `robert` would not be granted access. Unix user `robert` would only be allowed access when he tries to connect as PostgreSQL user `bob`, not as `robert` or anyone else. User `ann` would only be allowed to connect as `ann`. User `bryanh` would be allowed to connect as either `bryanh` himself or as `guest1`.

Example - The pg_ident.conf File

```
# MAPNAME     SYSTEM-USERNAME   PG-USERNAME

omicron       bryanh            bryanh
omicron       ann               ann
# bob has user name robert on these machines
omicron       robert            bob
# bryanh can also connect as guest1
omicron       bryanh            guest1
```

Limiting Concurrent Connections

To limit the number of active concurrent sessions to your HAWQ system, you can configure the `max_connections` server configuration parameter. This is a *local* parameter, meaning that you must set it in the `postgresql.conf` file of the master, the standby master, and each segment instance (primary and mirror). The value of `max_connections` on segments must be 5-10 times the value on the master.

When you set `max_connections`, you must also set the dependent parameter `max_prepared_transactions`. This value must be at least as large as the value of `max_connections` on the master, and segment instances should be set to the same value as the master.

For example:

In `$MASTER_DATA_DIRECTORY/postgresql.conf` (including standby master):

```
max_connections=100
max_prepared_transactions=100
```

In `*SEGMENT_DATA_DIRECTORY*/postgresql.conf` for all segment instances:

```
max_connections=500
max_prepared_transactions=100
```

To change the number of allowed connections:

1. Stop your HAWQ system:

```
$ gpstop
```

2. On your master host, edit `$MASTER_DATA_DIRECTORY/postgresql.conf` and change the following two parameters:

```
max_connections (the number of active user sessions you want to allow plus the number
 of superuser_reserved_connections)
max_prepared_transactions (must be greater than or equal to max_connections)
```

3. On each segment instance, edit `*SEGMENT_DATA_DIRECTORY*/postgresql.conf` and and change the following two parameters:

```
max_connections (must be 5-10 times the value on the master)
max_prepared_transactions (must be equal to the value on the master)
```

4. Restart your HAWQ system:

```
$ gpstart
```

Note: Raising the values of these parameters may cause HAWQ to request more shared memory. To mitigate this effect, consider decreasing other memory-related parameters such as `gp_cached_segworkers_threshold`.

Encrypting Client-Server Connections

HAWQ has native support for SSL connections between the client and the master server. SSL connections prevent third parties from snooping on the packets, and also prevent man-in-the-middle attacks. SSL should be used whenever the client connection goes through an insecure link, and must be used whenever client certificate authentication is used.

To enable SSL requires that OpenSSL be installed on both the client and the master server systems. HAWQ can be started with SSL enabled by setting the server configuration parameter ssl=on in the master postgresql.conf. When starting in SSL mode, the server will look for the files server.key (server private key) and server.crt (server certificate) in the master data directory. These files must be set up correctly before an SSL-enabled HAWQ system can start.

Important: Do not protect the private key with a passphrase. The server does not prompt for a passphrase for the private key, and the database startup fails with an error if one is required.

A self-signed certificate can be used for testing, but a certificate signed by a certificate authority (CA) should be used in production, so the client can verify the identity of the server. Either a global or local CA can be used. If all the clients are local to the organization, a local CA is recommended.

To create a Self-signed Certificate without a Passphrase for Testing Only:

1. Use the following `OpenSSL` command:

```
# openssl req -new -text -out server.req
```

2. Fill out the information that openssl asks for. Be sure to enter the local host name as Common Name. The challenge password can be left blank.

   The program will generate a key that is passphrase protected, and does not accept a passphrase that is less than four characters long.

To use the Self-signed Certificate with HAWQ:

1. Remove the passphrase with the following commands:

```
# openssl rsa -in privkey.pem -out server.key
# rm privkey.pem
```

2. Enter the old passphrase when prompted to unlock the existing key.
3. Enter the following command to change it into a self-signed certificate and to copy it with the key to a location where the server can find them:

```
# openssl req -x509 -in server.req -text -key server.key -out server.crt
```

4. Change the permissions on the key with the following command. The server will reject the file if the permissions are less restrictive than these.

```
# chmod og-rwx server.key
```

For more details on how to create your server private key and certificate, refer to the OpenSSL documentation.

The Password File

The file `.pgpass` in a user's home directory or the file referenced by `PGPASSFILE` can contain passwords to be used if the connection requires a password (and no password has been specified otherwise). On Microsoft Windows, the file is named `%APPDATA%\postgresql\pgpass.conf` (where `%APPDATA%` refers to the Application Data subdirectory in the user's profile).

This file should contain lines of the following format:

```
hostname:port:database:username:password
```

Each of the first four fields can be a literal value, or *, which matches anything. The password field from the first line that matches the current connection parameters will be used. (Therefore, put more-specific entries first when you are using wildcards.) If an entry needs to contain `:` or `\`, escape this character with `\`. A host name of localhost matches both TCP (host name localhost) and Unix domain socket (pghost empty or the default socket directory) connections coming from the local machine.

On Unix systems, the permissions on `.pgpass` must disallow any access to world or group; achieve this by the command `chmod 0600 ~/.pgpass`. If the permissions are less strict than this, the file will be ignored. On Microsoft Windows, it is assumed that the file is stored in a directory that is secure, so no special permissions check is made.

Chapter 6

Configuring Kerberos Authentication

On the versions of Red Hat Enterprise Linux that are supported by HAWQ, you can use a Kerberos authentication system to control access to HAWQ. HAWQ supports GSSAPI with Kerberos authentication. GSSAPI provides automatic authentication (single sign-on) for systems that support it. If Kerberos authentication is not available when a role attempts to log into HAWQ, the login fails.

You specify which HAWQ users require Kerberos authentication in the HAWQ configuration file `pg_hba.conf`. Whether you specify Kerberos authentication or another type of authentication for a HAWQ user, authorization to access HAWQ databases and database objects, such as schemas and tables, is controlled by the settings specified in both the `pg_hba.conf` file and in the the privileges given to HAWQ users and roles within the database.

This chapter describes how to configure a Kerberos authentication system and configure HAWQ to authenticate a HAWQ administrator.

For more information about Kerberos, see *http://web.mit.edu/kerberos/*.

Enabling Kerberos Authentication for HAWQ

Important: HAWQ has two endpoints for Kerberos configuration. One endpoint is configured to allow HAWQ to talk to Kerberized HDFS; in this case HAWQ is acting as a *Kerberos client process*. The other endpoint is configured to allow HAWQ to operate with a Kerberos Key Distribution Center (KDC) in order to require Kerberos authentication for users of the database; in this case HAWQ is acting as a Kerberos server process. It is possible to configure one endpoint without the other endpoint. These instructions are for configuring HAWQ as a Kerberos server process, so that users are required to authenticate via Kerberos to access the database. For instructions on configuring the HAWQ to Kerberized HDFS see *Configuring HAWQ on Secure HDFS*. Note that if HAWQ to Kerberized HDFS has already been configured, or you have an existing KDC to use, you may skip the KDC installation instructions below.

The following tasks are required to use Kerberos with HAWQ:

1. Set up, or identify, a Kerberos Key Distribution Center (KDC) server to use for your authentication. If you have already configured HAWQ to Kerberized HDFS, you will already have a cluster KDC that you can utilize. If necessary, set up a Kerberos realm and principals on the server. For HAWQ, a principal is a HAWQ role that utilizes Kerberos authentication. In the Kerberos database, a realm groups together the Kerberos principals that are the HAWQ roles.
2. Create a Kerberos keytab file for HAWQ (if one does not already exist) and add a `postgres/<hawq_master_fqdn>@REALM` principal key to the file:
   a. To access HAWQ, you create a service key known only by Kerberos and HAWQ. This is of the form `postgres/<hawq_master_fqdn>@REALM`, where `<hawq_master_fqdn>` is the fully qualified domain name of the HAWQ master host and `REALM` is your Kerberos realm.
   b. On the HAWQ master, the service key is stored in key tables, which are files known as keytabs. For a PHD installation, the HAWQ service keys are usually stored in the keytab file called `/etc/security/phd/keytab/hawq-<hawq_master_fqdn>.service.keytab`. This service key is the equivalent of the service's password, and must be kept secure. Data which is meant to be read only by the service is encrypted using this key.

      Note: If you have already configured HAWQ to Kerberized HDFS, this keytab will already exist and you will need to add the `postgres/<hawq_master_fqdn>@REALM` principal to it.
3. Install the Kerberos client packages and the keytab file on the HAWQ master, if they are not already installed. The `/etc/krb5.conf` file for your Kerberos installation needs to be properly configured on the HAWQ master. Again, if you have already configured HAWQ to Kerberized HDFS, this will be done.
4. Create a Kerberos ticket for `gpadmin` on HAWQ master node using the keytab file. The ticket contains the Kerberos authentication credentials that grant access to HAWQ.
5. Create Kerberos principals for any other users of the database; unless you add specific rules to allow non-Kerberos access to specific users, all users will require a Kerberos principal to be able to log into the database, once these steps are completed. Note that the default gpadmin is allowed to log in from the local HAWQ master host without authentication; it is recommended this not be changed, as it allows an admin to access the database to debug authentication issues.

With Kerberos authentication configured on the HAWQ, you can use Kerberos for PSQL and JDBC. For more information, see:

- *Setting up HAWQ with Kerberos for PSQL*
- *Setting up HAWQ with Kerberos for JDBC*

Before Installing Kerberos

Check that you can meet the following requirements before using Kerberos with HAWQ:

- Kerberos Key Distribution Center (KDC) server that uses the `krb5-server` library.
- Kerberos packages for version 5:
  - `krb5-libs`
  - `krb5-workstation`
- HAWQ capable of supporting Kerberos.
- A configuration that allows the Kerberos server and the HAWQ master to communicate with each other.
- Red Hat Enterprise Linux 6.x (requires Java 1.7.0_45 or later).
- Red Hat Enterprise Linux 5.x (requires Java 1.6.0_21 or later).
- Red Hat Enterprise Linux 4.x (requires Java 1.6.0_21 or later).

Note:

- The dates and times on the Kerberos server and clients must be synchronized. Authentication fails if the time difference between the Kerberos server and a client is too large. The maximum time difference is configurable; 5 minutes is the default.
- The Kerberos server and client must be configured, enabling them to ping each other using their host names.
- The Kerberos authentication itself is secure, but the data sent over the database connection is transmitted in clear text, unless SSL is used. Kerberos is for authentication only.
- These instruction consider the simplest most common configuration; it is possible to set up more complex configuration rules. Please see the appropriate documentation if more complex rules are desired.

Installing and Configuring a Kerberos KDC Server

The following steps install and configure a Kerberos Key Distribution Center (KDC) server in the case where one is not already installed:

1. Install the Kerberos packages for the Kerberos server:
   - `krb5-libs`
   - `krb5-server`
   - `krb5-workstation`
2. Edit the `/etc/krb5.conf` configuration file. See the section for the `krb5.conf` *Sample Configuration File* for sample configuration file parameters. When you create a KDC database, the parameters in the `/etc/krb5.conf` file specify that the realm `KRB.GREENPLUM.COM` is created. You use this realm when you create the Kerberos principals that are HAWQ roles. If you have an existing Kerberos server you might need to edit the kdc.conf file. See the Kerberos documentation for information about the `kdc.conf` file.
3. To create a Kerberos KDC database, run the `kdb5_util`. For example:

   ```
   kdb5_util create -s
   ```

   The `create` option creates the database to store keys for the Kerberos realms managed by this KDC server. The `-s` option creates a stash file. Without the stash file, every time the KDC server starts, it requests a password.
4. The Kerberos utility `kadmin` uses Kerberos to authenticate to the server.

   Before using `kadmin`, add an administrative user to KDC database with `kadmin.local`. `kadmin.local` is local to the server and does not use Kerberos authentication. To add the user `gpadmin` as an administrative user to the KDC database, run the following command:

   ```
   kadmin.local -q "addprinc gpadmin/admin"
   ```

   Note: Most users do not need administrative access to the Kerberos server. They can use kadmin to manage their own principals (for example, to change their own password). For information about `kadmin`, see the Kerberos documentation.
5. If needed, edit the `/var/kerberos/krb5kdc/kadm5.acl` file to grant the appropriate permissions to `gpadmin`.
6. Start the Kerberos daemons with the following commands:

   ```
   /sbin/service krb5kdc start
   /sbin/service kadmin start
   ```

   If you want to start Kerberos automatically upon restart, run the following commands:

   ```
   /sbin/chkconfig krb5kdc on
   /sbin/chkconfig kadmin on
   ```

Installing and Configuring the Kerberos Client

If they are not already installed, install the Kerberos client libraries on the HAWQ master and configure the Kerberos client:

1. Install the following Kerberos packages on the HAWQ master.
   - `krb5-libs`
   - `krb5-workstation`
2. Ensure that the `/etc/krb5.conf` file is the same as the one that is on the Kerberos server.

Creating HAWQ Roles in the KDC Database

After you have set up a Kerberos KDC and have created a realm for HAWQ, you add principals to the realm.

1. Create principals in the Kerberos database with `kadmin.local`. Note that the realm `REALM` is an example; replace it with your actual Kerberos realm. Using `kadmin.local` in interactive mode, the following commands add users:

   ```
   addprinc gpadmin@REALM
   addprinc postgres/<hawq_master_fqdn>@REALM
   ```

   The first `addprinc` command creates the HAWQ `gpadmin` user as a principal. See *Setting Up HAWQ with Kerberos for PSQL* for information on modifying the `pg_hba.conf` file so the HAWQ user `gpadmin` uses Kerberos authentication when accessing HAWQ from the client hosts.

   The second `addprinc` command creates the postgres process as principal in the Kerberos KDC. This principal is required when using Kerberos authentication with HAWQ. The syntax for the principal is `postgres/<hawq_master_fqdn>@REALM` where `<hawq_master_fqdn>` is the fully qualified host name of the HAWQ master.

   Note: Principal differences between HAWQ and Postgres

   For a HAWQ to Kerberized HDFS configuration, HAWQ uses the `postgres@REALM` form for a principal, as it is a Kerberos client when talking to HDFS. For the HAWQ service configuration we are concerned with here, it uses the `postgres/<service_host_fqdn>@REALM` form, as it is acting as a Kerberos service. If you are configuring both endpoints, understanding the difference is important.
2. Create a Kerberos keytab file with `kadmin.local` and add the `postgres/<hawq_master_fqdn>@REALM` principal, or if a keytab already exists, add this principal to the existing HAWQ keytab file. The following example creates a keytab file `gpdb-kerberos.keytab` with authentication information for the two principals.

   ```
   ktadd -norandkey -k hawq-<hawq_master_fqdn>.service.keytab postgres/<hawq_master_
   fqdn>@REALM
   ```

   Place the keytab file in the `/etc/security/phd/keytab` directory on the HAWQ master and set ownership to `gpadmin:gpadmin` and permissions to `r--------`.

   Create a link to the keytab file for clarity and ease of reference. For example:

   ```
   ln -s /etc/security/phd/keytab/hawq-<hawq_master_fqdn>.service.keytab /etc/
   security/phd/keytab/hawq.service.keytab
   ```

   Note:

   - If necessary, create the `/etc/security/phd/keytab` directory and insure that `gpadmin` has access.
   - If you already have a HAWQ keytab file in this directory, you can use `kadmin` from the HAWQ master to add to the keytab.
   - You may choose another location for the keytab file, but there is only one keytab setting in `postgresql.conf` for both endpoints, so you must be consistent.
3. Verify the contents of your keytab.The following is an example `klist` output for a system configured with both endpoints (HAWQ to Kerberized HDFS and HAWQ service configuration to authenticate database users) where the HAWQ master FQDN is set to `centos61-2.localdomain` and the realm is set to `PHD.BIGDATA.COM`:

   ```
   # klist -k -t /etc/security/phd/keytab/hawq-centos61-2.localdomain.service.keytab
   ```

```
Keytab name: WRFILE:/etc/security/phd/keytab/hawq-centos61-2.localdomain.service.
keytab
KVNO Timestamp         Principal
---- ----------------- --------------------------------------------------------
   1 04/30/14 22:45:21 postgres@PHD.BIGDATA.COM
   1 04/30/14 22:45:21 postgres@PHD.BIGDATA.COM
   1 04/30/14 22:45:21 postgres@PHD.BIGDATA.COM
   1 04/30/14 22:45:21 postgres@PHD.BIGDATA.COM
   1 04/30/14 22:45:21 postgres@PHD.BIGDATA.COM
   1 04/30/14 22:45:21 postgres@PHD.BIGDATA.COM
   1 04/30/14 22:45:21 HTTP/centos61-2.localdomain@PHD.BIGDATA.COM
   1 04/30/14 22:45:21 HTTP/centos61-2.localdomain@PHD.BIGDATA.COM
   1 04/30/14 22:45:21 HTTP/centos61-2.localdomain@PHD.BIGDATA.COM
   1 04/30/14 22:45:21 HTTP/centos61-2.localdomain@PHD.BIGDATA.COM
   1 04/30/14 22:45:21 HTTP/centos61-2.localdomain@PHD.BIGDATA.COM
   1 04/30/14 22:45:21 HTTP/centos61-2.localdomain@PHD.BIGDATA.COM
   1 05/01/14 19:41:39 postgres/centos61-2.localdomain@PHD.BIGDATA.COM
   1 05/01/14 19:41:39 postgres/centos61-2.localdomain@PHD.BIGDATA.COM
   1 05/01/14 19:41:39 postgres/centos61-2.localdomain@PHD.BIGDATA.COM
   1 05/01/14 19:41:39 postgres/centos61-2.localdomain@PHD.BIGDATA.COM
   1 05/01/14 19:41:39 postgres/centos61-2.localdomain@PHD.BIGDATA.COM
   1 05/01/14 19:41:39 postgres/centos61-2.localdomain@PHD.BIGDATA.COM
```

Checking Client Operation on the HAWQ Master

On the HAWQ master:

1. Clean up any possible existing tickets:

```
# kdestroy
```

2. Use the Kerberos `kinit` utility to request a ticket using the keytab file on the HAWQ master for `gpadmin@REALM`:

```
# kinit gpadmin
[password prompt will be displayed: enter your gpadmin principal password]
```

3. Use the Kerberos `klist` utility to display the contents of the Kerberos ticket cache on the HAWQ master:

```
# klist
Ticket cache: FILE:/tmp/krb5cc_108061
Default principal: gpadmin@REALM
Valid starting Expires Service principal
03/28/13 14:50:26 03/29/13 14:50:26 krbtgt/REALM@REALM
renew until 03/28/13 14:50:26
```

Setting up HAWQ with Kerberos for PSQL

After you have set up Kerberos on the HAWQ master, you can configure HAWQ to use Kerberos.

1. Check that the `gpadmin` role exists in HAWQ (it should). If not create a HAWQ administrator role in the database `template1` for the Kerberos principal that is used as the database administrator:

```
psql template1 -c 'create role "gpadmin" login superuser;'
```

Note: The role you create in the database `template1` will be available in any new HAWQ database that you create.

Adding this line to the `postgresql.conf` specifies the folder `/home/gpadmin` as the location of the keytab file `gpdb-kerberos.keytab`.

2. If you have not already configured HAWQ to Kerberized HDFS, you will need to modify `postgresql.conf` to specify the location of the keytab file.

For example, adding this line to the `postgresql.conf` specifies the folder `/home/gpadmin` as the location of the keytab file `gpdb-kerberos.keytab`:

```
krb_server_keyfile = '/etc/security/phd/keytab/hawq.service.keytab'
```

Note: The single quote format is important.

3. Modify the HAWQ file `pg_hba.conf` to enable Kerberos support.

   For example, adding the following line to `pg_hba.conf` adds GSSAPI and Kerberos support. The value for `krb_realm` is the Kerberos realm that is used for authentication to HAWQ.

   Add as the last entry in pg_hba.conf:

```
host all all 0.0.0.0/0 gss include_realm=0 krb_realm=REALM
```

   More complex rules are possible. For information about the `pg_hba.conf` file, see "The pga_hba.conf File" section in*Configuring Client Authentication*.
4. Restart HAWQ:

```
# sudo -u gpadmin service hawq stop
# sudo -u gpadmin service hawq start
```

5. As a test, log in into the database from a client node (not the HAWQ master host) as the `gpadmin` role with the Kerberos credentials `gpadmin`:

```
[root@centos61-5 ~]# su - gpadmin
[gpadmin@centos61-5 ~]$ kinit gpadmin
Password for gpadmin@PHD.BIGDATA.COM:
[gpadmin@centos61-5 ~]$ psql -h centos61-2.localdomain
psql (8.4.7, server 8.2.15)
WARNING: psql version 8.4, server version 8.2.
         Some psql features might not work.
Type "help" for help.
gpadmin=#
```

   You can add regular database users by creating a role for the user in the HAWQ database as superuser and creating a corresponding Kerberos principal.

   Note: A username map can be defined in the `pg_ident.conf` file and specified in the `pg_hba.conf` file to simplify logging into HAWQ. For example, this `psql` command logs into the default HAWQ on `mdw.proddb` as the Kerberos principal `adminuser/mdw.proddb`:

```
$ psql -U "adminuser/mdw.proddb" -h mdw.proddb
```

   If the default user is `adminuser`, the `pg_ident.conf` file and the `pg_hba.conf` file can be configured so that `adminuser` can log into the database as the Kerberos principal `adminuser/mdw.proddb` without specifying the `-U` option:

```
$ psql -h mdw.proddb
```

   The following username map is defined in the HAWQ file:

```
$MASTER_DATA_DIRECTORY/pg_ident.conf:
 # MAPNAME SYSTEM-USERNAME GP-USERNAME
mymap /^(.*)mdw\.proddb$ adminuser
```

   The map can be specified in the `pg_hba.conf` file as part of the line that enables Kerberos support:

```
host all all 0.0.0.0/0 krb5 include_realm=0 krb_realm=proddb
map=mymap
```

For more information on specifying username maps, see the Postgres documentation: *http://www.postgresql.org/docs/8.4/static/auth-username-maps.html*.

If a Kerberos principal is not a HAWQ user, a message is similar to the following is displayed from the `psql` command line when the user attempts to log into the database:

```
psql: krb5_sendauth: Bad response
```

The principal must be added as a HAWQ user.

Setting up HAWQ with Kerberos for JDBC

You can configure HAWQ to use Kerberos to run user-defined Java functions:

1. Ensure that Kerberos is installed and configured on the HAWQ master. See *Installing and Configuring the Kerberos Client* for more details.
2. Create the file `.java.login.config` in the folder `/home/gpadmin` and add the following text to the file:

   ```
   pgjdbc {
    com.sun.security.auth.module.Krb5LoginModule required
    doNotPrompt=true
    useTicketCache=true
    debug=true
    client=true;
   };
   ```
3. Create a Java application that connects to HAWQ using Kerberos authentication.

   This example database connection URL uses a PostgreSQL JDBC driver and specifies parameters for Kerberos authentication:

   ```
   jdbc:postgresql://mdw:5432/mytest?kerberosServerName=
   postgres&jaasApplicationName=pgjdbc&user=
   gpadmin/kerberos-gpdb
   ```

   The parameter names and values specified depend on how the Java application performs Kerberos authentication.
4. Test the Kerberos login by running a sample Java application from HAWQ.

Sample Kerberos Configuration File

This sample `krb5.conf` Kerberos configuration file is used in the example that configures HAWQ to use Kerberos authentication:

```
[logging]

  default = FILE:/var/log/krb5libs.log
  kdc = FILE:/var/log/krb5kdc.log
  admin_server = FILE:/var/log/kadmind.log
[libdefaults]
  default_realm = KRB.GREENPLUM.COM
  dns_lookup_realm = false
  dns_lookup_kdc = false
  ticket_lifetime = 24h
  renew_lifetime = 7d
  forwardable = yes
  default_tgs_enctypes = aes128-cts des3-hmac-sha1 des-cbc-crc
des-cbc-md5
default_tkt_enctypes = aes128-cts des3-hmac-sha1 des-cbc-crc
des-cbc-md5
permitted_enctypes = aes128-cts des3-hmac-sha1 des-cbc-crc
des-cbc-md5
realms]
KRB.GREENPLUM.COM = {
  kdc = kerberos-gpdb:88
  admin_server = kerberos-gpdb:749
  default_domain = kerberos-gpdb
}
[domain_realm]
.kerberos-gpdb = KRB.GREENPLUM.COM
kerberos-gpdb = KRB.GREENPLUM.COM
[appdefaults]
pam = {
   debug = false
   ticket_lifetime = 36000
   renew_lifetime = 36000
   forwardable = true
   krb4_convert = false
 }
```

Chapter 7

Configuring LDAP Authentication

This chapter describes how to configure the LDAP authentication system and create a database user.

Before Setting up LDAP Authentication

Check that you can meet the following requirements before setting up LDAP authentication:

- DN (Distinguished Name) format: important because it varies based on company.
- URL of the LDAP server and corresponding firewall permissions.
- LDAP service port — 389 by default.
- Authorized LDAP DN to test the setup.

Verifying the LDAP server and the LDAP DN

Note:

- DN format shown in the current example is `"uid=username,ou=DBUser,dc=hawq-pivotal,dc=com"`
- LDAP server is `"mdw"`

1. Execute the following command to make sure HAWQ master can connect to LDAP server:

```
$ nc -v mdw 389
Connection to mdw 389 port [tcp/ldap] succeeded
```

2. Execute the following command to make sure LDAP DN is valid.:

```
$ ldapsearch -W -h mdw -D "uid=john,ou=DBUser,dc=hawq-pivotal,dc=com" -b "dc=hawq-
pivotal,dc=com" cn
Enter LDAP Password:
```

If the bind succeeds, you will see the following output. The search result should show ‘0 Success’:

```
# extended LDIF
#
# LDAPv3
# base <uid=john,ou=DBUser,dc=hawq-pivotal,dc=com> with scope subtree
# filter: (objectclass=*)
# requesting: cn
#

# john, DBUser, hawq-pivotal.com
dn: uid=john,ou=DBUser,dc=hawq-pivotal,dc=com
cn: John Doe

# search result
search: 2
result: 0 Success

# numResponses: 2
# numEntries: 1
```

Editing the pg_hba.conf file

Add the following line to the `pg_hba.conf` file:

```
host all john 0.0.0.0/0  ldap ldapserver=mdw ldapprefix="uid=" ldapsuffix=",ou=
DBUser,dc=hawq-pivotal,dc=com"
```

For more information about the `pg_hba.conf` file, see *The pg_hba.conf File* for more information.

Creating the Database User

Execute the following command to create the database user `john`:

```
demo=# create user john;
CREATE ROLE
```

You can now log in to the HAWQ with user `john` using LDAP:

```
$ psql -h 127.0.0.1 -U john
# Password for user john:
# psql (8.2.15)
# Type "help" for help.
```

Chapter 8

Backing Up and Restoring HAWQ Databases

This chapter provides information on backing up and restoring databases in HAWQ system.

As an administrator, you will need to back up and restore your database. HAWQ provides three utilities to help you back up your data:

- `gpfdist`
- PXF
- `pg_dump`

`gpfdist` and PXF are parallel loading and unloading tools that provide the best performance. You can use `pg_dump`, a non-parallel utility, to migrate from PostgreSQL to HAWQ.

In addition, in some situations, Pivotal recommends backing up your raw data from ETL processes.

This section describes these three utilities, as well as raw data backup, to help you decide what fits your needs.

About gpfdist and PXF

You can perform a parallel backup in HAWQ using `gpfdist` or PXF to unload all data to external tables. Backup files can reside on a local file system or HDFS. To recover tables, you can load data back from external tables to the database.

Performing a Parallel Backup

1. Check the database size to ensure that the file system has enough space to save the backed up files.
2. Use the `pg_dump` utility to dump the schema of the target database.
3. Create a writable external table for each table to back up to that database.
4. Load table data into the newly created external tables.

    Note: Pivotal recommends that you put the insert statements in a single transaction to prevent problems if you perform any update operations during the backup.

Restoring from a Backup

1. Create a database to recover to.
2. Recreate the schema from the schema file (created during the `pg_dump` process).
3. Create a readable external table for each table in the database.
4. Load data from the external table to the actual table.
5. Run the `ANALYZE` command once loading is complete. This ensures that the query planner generates optimal plan based on up-to-date table statistics.

Differences between gpfdist and PXF

`gpfdist` and PXF differ in the following ways:

- `gpfdist` stores backup files on local file system, while PXF stores files on HDFS.
- `gpfdist` only supports plain text format, while PXF also supports binary format like AVRO and customized format.
- `gpfdist` doesn't support generating compressed file, while PXF supports compression (you can specify compression codec used in Hadoop like `org.apache.hadoop.io.compress.GzipCodec`).
- Both `gpfdist` and PXF both have fast loading performance, but `gpfdist` is much faster than PXF.

About pg_dump and pg_restore

HAWQ supports the PostgreSQL backup and restore utilities, `pg_dump` and `pg_restore`. The `pg_dump` utility creates a single, large dump file in the master host containing the data from all active segments. The `pg_restore` utility restores a HAWQ database from the archive created by `pg_dump`. In most cases, this is probably not practical, as there is most likely not enough disk space in the master host for creating a single backup file of an entire distributed database. HAWQ supports these utilities in case you are migrating data from PostgreSQL to HAWQ.

To create a backup archive for database `mydb`:

```
$ pg_dump -Ft -f mydb.tar mydb
```

To create a compressed backup using custom format and compression level 3:

```
$ pg_dump -Fc -Z3 -f mydb.dump mydb
```

To restore from an archive using `pg_restore`:

```
$ pg_restore -d new_db mydb.dump
```

About Backing Up Raw Data

Parallel backup using `gpfdist` or PXF works fine in most cases. There are a couple of situations where you cannot perform parallel backup and restore operations:

- Performing periodically incremental backups.
- Dumping a large data volume to external tables - this process takes a long time.

In such situations, you can back up raw data generated during ETL processes and reload it into HAWQ. Pivotal recommends this approach due to its incremental nature and the flexibility to choose where you store backup files.

Selecting a Backup Strategy/Utility

The table below summaries the differences between the four approaches we discussed above.

	`gpfdist`	**PXF**	`pg_dump`	**Raw Data Backup**
Parallel	Yes	Yes	No	No
Incremental Backup	No	No	No	Yes
Backup Location	Local FS	HDFS	Local FS	Local FS, HDFS
Format	Text, CSV	Text, CSV, Custom	Text, Tar, Custom	Depends on format of row data
Compression	No	Yes	Only support custom format	Optional
Scalability	Good	Good	---	Good
Performance	Fast loading, Fast unloading	Fast loading, Normal unloading	---	Fast (Just file copy)

Estimating Space Requirements

Before you back up your database, ensure that you have enough space to store backup files. This section describes how to get the database size and estimate space requirements.

- Use `hawq_toolkit` to query size of the database you want to backup.

  ```
  mydb=# SELECT sodddatsize FROM hawq_toolkit.hawq_size_of_database WHERE
   sodddatname=’mydb’;
  ```

  If tables in your database are compressed, this query shows the compressed size of the database.
- Estimate the total size of the backup files.
  - If your database tables and backup files are both compressed, you can use the value `soddatsize` as an estimate value.
  - If your database tables are compressed and backup files are not, you need to multiply `soddatsize` by the compression ratio. Although this depends on the compression algorithms, Pivotal recommends that you can use an empirical value such as 300%.
  - If your back files are compressed and database tables are not, you need to divide `soddatsize` by the compression ratio.
- Get space requirement.
  - If you use HDFS with PXF, the space requirement is `size_of_backup_files * replication_factor`.
  - If you use gpfdist, the space requirement for each gpfdist instance is `size_of_backup_files / num_gpfdist_instances` since table data will be evenly distributed to all `gpfdist` instances.

Using gpfdist

This section discusses `gpfdist` and shows an example of how to backup and restore HAWQ database.

`gpfdist` is HAWQ's parallel file distribution program. It is used by readable external tables and `gpload` to serve external table files to all HAWQ segments in parallel. It is used by writable external tables to accept output streams from HAWQ segments in parallel and write them out to a file.

To use `gpfdist`, start the `gpfdist` server program on the host where you want to store backup files. You can start multiple `gpfdist` instances on the same host or on different hosts. For each `gpfdist` instance, you specify a directory from which `gpfdist` will serve files for readable external tables or create output files for writable external tables. For example, if you have a dedicated machine for backup with two disks, you can start two `gpfdist` instances, each using one disk:

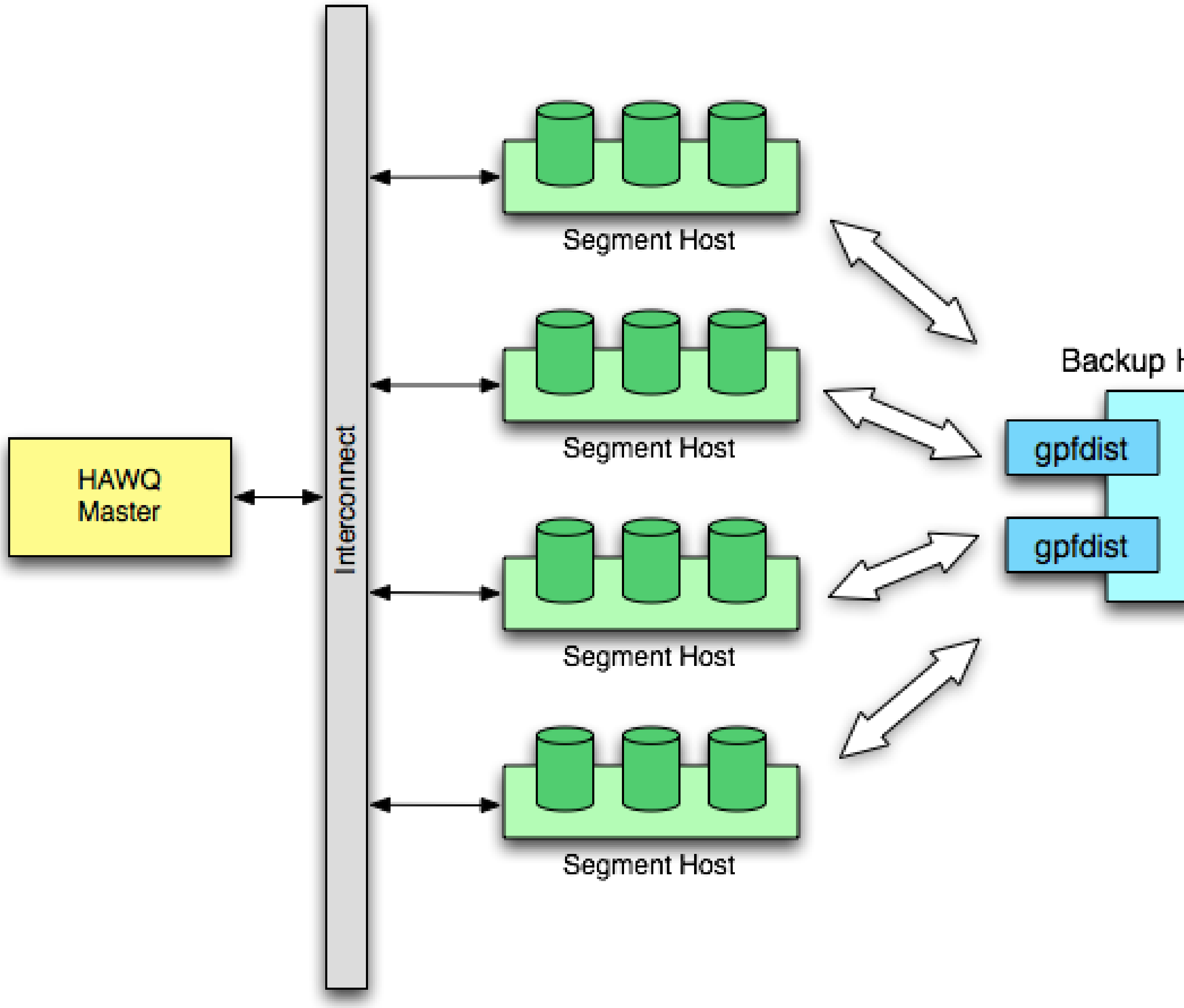


Figure 3: Deploying multiple gpfdist instances on a backup host

You can also run `gpfdist` instances on each segment host. During backup, table data will be evenly distributed to all `gpfdist` instances specified in the `LOCATION` clause in the `CREATE EXTERNAL TABLE` definition.

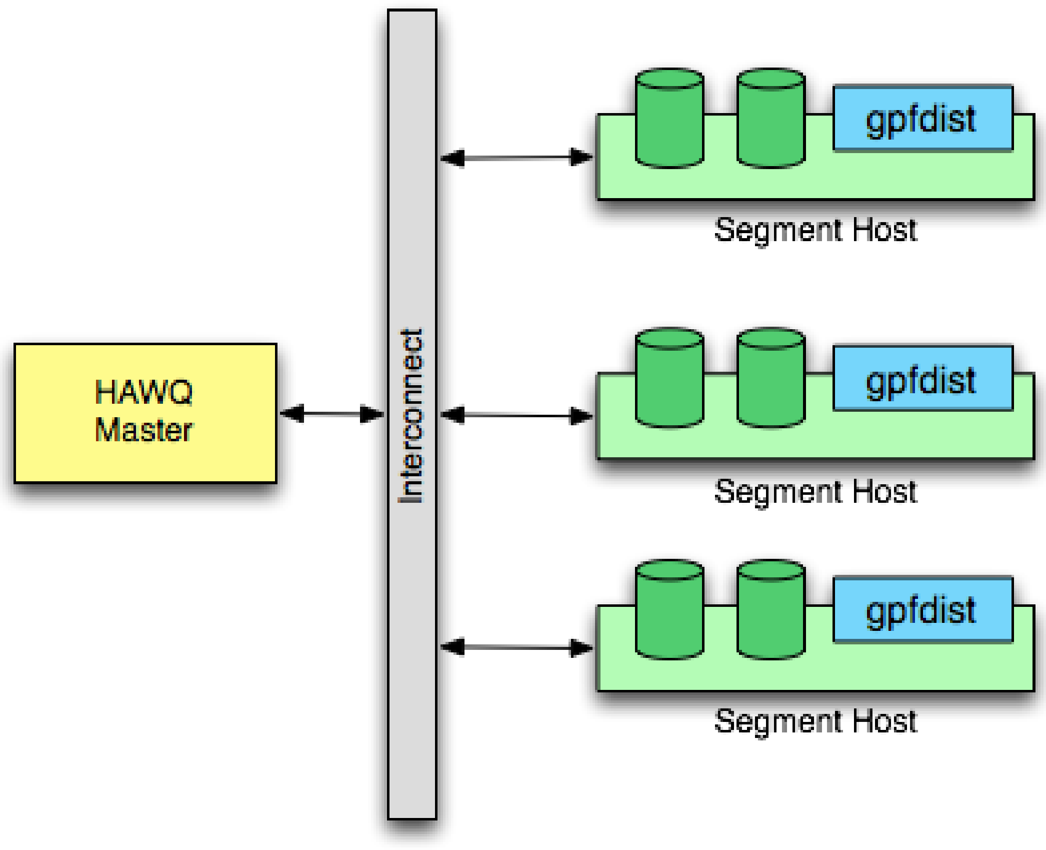


Figure 4: Deploying gpfdist instances on each segment host

Example

This example of using `gpfdist` backs up and restores a 1TB `tpch` database. To do so, start two `gpfdist` instances on the backup host `sdw1` with two 1TB disks (One disk mounts at `/data1`, another disk mounts at `/data2`).

Using gpfdist to Back Up the tpch Database

1. Create backup locations and start the `gpfdist` instances.

In this example, issuing the first command creates two folders on two different disks with the same postfix `backup/tpch_20140627`. These folders are labeled as backups of the `tpch` database on 2014-06-27. In the next two commands, the example shows two `gpfdist` instances, one using port 8080, and another using port 8081:

```
sdw1$ mkdir -p /data1/gpadmin/backup/tpch_20140627 /data2/gpadmin/backup/tpch_
20140627
sdw1$ gpfdist -d /data1/gpadmin/backup/tpch_20140627 -p 8080 &
sdw1$ gpfdist -d /data2/gpadmin/backup/tpch_20140627 -p 8081 &
```

2. Save the schema for the database:

```
master_host$ pg_dump --schema-only -f tpch.schema tpch
master_host$ scp tpch.schema sdw1:/data1/gpadmin/backup/tpch_20140627
```

On the HAWQ master host, use the `pg_dump` utility to save the schema of the tpch database to the file `tpch.schema`. Copy the schema file to the backup location to restore the database schema.

3. Create a writable external table for each table in the database:

```
master_host$ psql tpch
tpch=# create writable external table wext_orders (like orders)
 location ('gpfdist://sdw1:8080/orders1.csv', 'gpfdist://sdw1:8081/orders2.csv')
 format 'CSV';
tpch=# create writable external table wext_lineitem (like lineitem)
 location ('gpfdist://sdw1:8080/lineitem1.csv', 'gpfdist://sdw1:8081/lineitem2.
csv') format 'CSV';
```

   The sample shows two tables in the `tpch` database: `orders` and `line item`. The sample shows that two corresponding external tables are created. Specify a location or each `gpfdist` instance in the `LOCATION` clause. This sample uses the CSV text format here, but you can also choose other delimited text formats. For more information, see the `CREATE EXTERNAL TABLE` SQL command.

4. Unload data to the external tables:

```
tpch=# begin;
tpch=# insert into wext_orders select * from orders;
tpch=# insert into wext_lineitem select * from lineitem;
tpch=# commit;
```

5. **(Optional)** Stop `gpfdist` servers to free ports for other processes:

   Find the progress ID and kill the process:

```
sdw1$ ps -ef | grep gpfdist
sdw1$ kill 612368; kill 612369
```

Recovering Using gpfdist

1. Restart `gpfdist` instances if they aren't running:

```
sdw1$ gpfdist -d /data1/gpadmin/backup/tpch_20140627 -p 8080 &
sdw1$ gpfdist -d /data2/gpadmin/backup/tpch_20140627 -p 8081 &
```

2. Create a new database and restore the schema:

```
master_host$ createdb tpch2
master_host$ scp sdw1:/data1/gpadmin/backup/tpch_20140627/tpch.schema .
master_host$ psql -f tpch.schema -d tpch2
```

3. Create a readable external table for each table:

```
master_host$ psql tpch2

tpch2=# create external table rext_orders (like orders) location ('gpfdist://
sdw1:8080/orders1.csv', 'gpfdist://sdw1:8081/orders2.csv') format 'CSV';
tpch2=# create external table rext_lineitem (like lineitem) location ('gpfdist://
sdw1:8080/lineitem1.csv', 'gpfdist://sdw1:8081/lineitem2.csv') format 'CSV';
```

   Note: The location clause is the same as the writable external table above.

4. Load data back from external tables:

```
tpch2=# insert into orders select * from rext_orders;
tpch2=# insert into lineitem select * from rext_lineitem;
```

5. Run the `ANALYZE` command after data loading:

```
tpch2=# analyze;
```

Troubleshooting gpfdist

Keep in mind that `gpfdist` is accessed at runtime by the segment instances. Therefore, you must ensure that the HAWQ segment hosts have network access to gpfdist. Since the `gpfdist` program is a web server, to test connectivity you can run the following command from each host in your HAWQ array (segments and master):

```
$ wget http://gpfdist_hostname:port/filename
```

Also, make sure that your `CREATE EXTERNAL TABLE` definition has the correct host name, port, and file names for `gpfdist`. The file names and paths specified should be relative to the directory where gpfdist is serving files (the directory path used when you started the `gpfdist` program). See “Defining External Tables - Examples”.

Using PXF

Pivotal Extension Framework (PXF) is an extensible framework that allows HAWQ to query external system data. The details of how to install and use PXF can be found in *PXF Installation and Administration*.

Using PXF to Back Up the tpch Database

1. Create a folder on HDFS for this backup:

```
master_host$ hdfs dfs -mkdir -p /backup/tpch-2014-06-27
```

2. Dump the database schema using `pg_dump` and store the schema file in a backup folder:

```
master_host$ pg_dump --schema-only -f tpch.schema tpch
master_host$ hdfs dfs -copyFromLocal tpch.schema /backup/tpch-2014-06-27
```

3. Create a writable external table for each table in the database.:

```
master_host$ psql tpch

tpch=# create writable external table wext_orders (like orders) location ('pxf://
namenode_host:50070/backup/tpch-2014-06-27/orders?Profile=
HdfsTextSimple&COMPRESSION_CODEC=org.apache.hadoop.io.compress.SnappyCodec')
 format 'TEXT';

tpch=# create writable external table wext_lineitem (like lineitem)
 location ('pxf://namenode_host:50070/backup/tpch-2014-06-27/lineitem?Profile=
HdfsTextSimple&COMPRESSION_CODEC=org.apache.hadoop.io.compress.SnappyCodec')
 format 'TEXT';
```

   Here, all backup files for the `orders` table go in the `/backup/tpch-2014-06-27/orders` folder, all backup files for the `lineitem` table go in `/backup/tpch-2014-06-27/lineitem` folder. We use snappy compression to save disk space.

4. Unload the data to external tables:

```
tpch=# begin;
tpch=# insert into wext_orders select * from orders;
tpch=# insert into wext_lineitem select * from lineitem;
tpch=# commit;
```

5. **(Optional)** Change the HDFS file replication factor for the backup folder. HDFS replicates each block into three blocks by default for reliability. You can decrease this number for your backup files if you need to:

```
master_host$ hdfs dfs -setrep 2 /backup/tpch-2014-06-27
```

   Note: This only changes the replication factor for existing files; new files will still use the default replication factor.

Recovering a PXF Backup

1. Create a new database and restore the schema:

```
master_host$ createdb tpch2
master_host$ hdfs dfs -copyToLocal /backup/tpch-2014-06-27/tpch.schema .
master_host$ psql -f tpch.schema -d tpch2
```

2. Create a readable external table for each table to restore:

```
master_host$ psql tpch2
```

```
tpch2=# create external table rext_orders (like orders) location ('pxf://namenode_
host:50070/backup/tpch-2014-06-27/orders?Profile=HdfsTextSimple') format 'TEXT';
tpch2=# create external table rext_lineitem (like lineitem) location ('pxf://
namenode_host:50070/backup/tpch-2014-06-27/lineitem?Profile=HdfsTextSimple')
 format 'TEXT';
```

The location clause is almost the same as above, except you don't have to specify the `COMPRESSION_CODEC` because PXF will automatically detect it.

3. Load data back from external tables:

```
tpch2=# insert into orders select * from rext_orders;
tpch2=# insert into lineitem select * from rext_lineitem;
```

4. Run `ANALYZE` after data loading:

```
tpch2=# analyze;
```

Chapter 9

Expanding the HAWQ System

This chapter provides information on adding additional resources to an existing HAWQ system to scale performance.

Planning Your HAWQ Expansion

Careful planning is critical to the success of a system expansion operation. By thoroughly preparing all new hardware and carefully planning all the steps of the expansion procedure, you can minimize risk and down time for the HAWQ database. For performance-related considerations when expanding large-scale systems, see *Planning Table Redistribution* later in this section.

This section provides an overview and a checklist for the system expansion process.

System Expansion Overview

System expansion consists of three phases:

- Adding and testing new hardware platforms
- Initializing new segments
- Redistributing tables

Adding and Testing New Hardware

You can refer to the general considerations for deploying new hardware described in *Planning New Hardware Platforms*. For more information on hardware platforms, consult the Pivotal platform engineers. After the new hardware platforms are provisioned and networked, you must run performance tests using Pivotal HAWQ utilities.

Initializing New Segments

Once the HAWQ binaries are installed on new hardware, you must initialize the new segments using `gpexpand` (not `gpinitsystem`). In this process, the utility creates a data and metadata directory and copies all the metadata from the existing segments to the new segments, capturing metadata for each user data table in an expansion schema, for status tracking. After this process has completed successfully, the expansion operation is committed, and cannot be reversed.

These operations are performed with the system offline. The `gpexpand` utility will shut down the database during initialization, if not already shut down.

Redistributing Tables

As part of the initialization process, `gpexpand` nullifies hash distribution policies (except for the parent tables of a partitioned table) and sets the distribution policy for all tables to random distribution. This action is performed on all tables, in all existing databases in the HAWQ instance.

> **Note:** Nullifying original distribution policies marks the distribution policies of all the user tables to Random. It does NOT involve any data movement or distribution. Physical movement of the data happens only when the final redistribution, according to the original policy, is performed.

Users can continue to access HAWQ after initialization is complete and the system is back online, though they could experience some performance degradation on systems that rely heavily on hash distribution of tables. During this process, normal operations such as ETL jobs, user queries, and reporting can continue, although users might experience slower response times.

To complete system expansion, you must run gpexpand to redistribute data tables across the newly added segments. Depending on the size and scale of your system, this might be accomplished in a single session during low-use hours, or it might require you to divide the process into batches over an extended period. Each table or partition will be unavailable for read or write operations during the period in which it is being redistributed. As each table is successfully redistributed across the new segments, according to its distribution key (if any), the performance of the database should incrementally improve until it equals and then exceeds pre-expansion performance levels.

In a typical operation, you will run the `gpexpand` utility four times, using different options, during the complete expansion process.

- To interactively create an expansion input file:

  ```
  gpexpand -f hosts_file
  ```

- To initialize segments and create expansion schema:

  ```
  gpexpand -i input_file -D database_name
  ```

- To redistribute tables:

```
gpexpand -d duration
```

- To remove the expansion schema:

```
gpexpand -c
```

In systems whose large scale requires multiple redistribution sessions, you may need to run `gpexpand` several more times to complete the expansion. For more information, see *Planning Table Redistribution*.

System Expansion Checklist

This checklist provides a quick overview of the steps required for a system expansion.

Online Pre-Expansion Preparation (Perfrom these tasks when the system is up and available)
• Devise and execute a plan for ordering, building, and networking new hardware platforms.
• Devise a database expansion plan. Map the number of segments per host, schedule the offline period to test performance and create the expansion schema, schedule the intervals for table redistribution.
• Install HAWQ database binaries on new hosts.
• Copy SSH keys to the new hosts (`gpssh-exkeys`).
• Validate the operating system environment of the new hardware (`gpcheck`).
• Validate disk I/O and memory bandwidth of the new hardware (`gpcheckperf`)
• Validate that the master data directory has no huge files under `pg_log` or `gpperfmon/data`
• Prepare an expansion input file (`gpexpand`).

Offline Expansion Tasks
• Validate the operating system environment of the combined existing and new hardware (`gpcheck`)
• Validate disk I/O and memory bandwidth of the combined existing and new hardware (`gpcheckperf`)
• Reinstall HAWQ Components
• Initialize new segments into the array and create an expansion schema (`gpexpand -i input_file`)
• Stop any automated snapshot or other processes that consume disk space.

Online Expansion Tasks
• Redistribute the tables through the expanded system (`gpexpand`).
• Remove expansion schema (`gpexpand -c`)
• Run analyze to update distribution statistics during the expansion using `gpexpand a`, or post-expansion using the `ANALYZE` SQL command.

Important: If you encounter problems during new segment initialization, you cannot use `gp_dump` to restore the system. You can *only* roll back a failed expansion operation. Once a setup operation is complete and the expansion is committed, you cannot roll back.

Planning New Hardware Platforms

Careful preparation of new hardware for system expansion is extremely important. Deliberate and thorough deployment of compatible hardware can greatly minimize the risk of issues developing later in the system expansion process.

Pivotal recommends the following:

- All new segment hosts for the expanded HAWQ Database array should have hardware resources and configurations matching those of the existing hosts.
- You work with HAWQ Platform Engineering prior to making a hardware purchase decision to expand the HAWQ Database.

The steps to plan and set up new hardware platforms will vary greatly for each unique deployment. Some of the possible considerations include:

- Preparing the physical space for the new hardware. Consider cooling, power supply, and other physical factors.
- Determining the physical networking and cabling required to connect the new and existing hardware.
- Mapping the existing IP address spaces and developing a networking plan for the expanded system.
- Capturing the system configuration (users, profiles, NICs, etc.) from existing hardware, to list it in detail for ordering the new hardware.
- Creating a custom build plan for deploying hardware with the desired configuration for the particular site and environment.

After selecting and adding new hardware to your network environment, make sure you perform the burn-in tasks described in *Verifying OS Settings*.

Planning Initialization of New Segments

Expanding the HAWQ Database requires a limited period of system down time. During this time period, you must run gpexpand to initialize new segments into the array and create an expansion schema.

The time required will depend on the number of schema objects in the HAWQ system, as well as other factors related to hardware performance.

Note

Note: After you begin initializing new segments, you can no longer restore the system using `gp_dump` files created for the per-expansion system. When initialization is successfully completed, the expansion is committed and cannot be rolled back.

Increasing Segments Per Host

By default, new hosts are initialized with the same number of segments as existing hosts. Optionally, you can increase the number of segments per host, or add new segments only to existing hosts.

For example, if existing hosts currently have two segments per host, you can use gpexpand to initialize two additional segments on existing hosts (for a total of four), and four new segments on new hosts.

Note: When you are adding new segments on existing nodes or hosts, you do not need to perform pre-expansion tasks such as HAWQ binary deployment, copying ssh keys,and other tasks, because these hosts have already been configured. You must make sure that these hosts have enough resources such as OS resources, as well as memory to manage new segments.

The interactive process for creating an expansion input file prompts for this option, and the input file format allows you to specify new segment directories manually as well. For more information, see *Creating an Input File for System Expansion*.

About the Expansion Schema

At initialization time, gpexpand creates an expansion schema. If you do not specify a particular database at initialization time (`gpexpand -D`), the schema is created in the database indicated by the `PGDATABASE` environment variable.

The expansion schema stores metadata for each table in the system, so that its status can be tracked throughout the expansion process. The expansion schema consists of two tables and a view for tracking the progress of an expansion operation:

- `gpexpand.status`
- `gpexpand.status_detail`
- `gpexpand.expansion_progress`

Planning Table Redistribution

The redistribution of tables is performed with the system online. For many HAWQ systems, table redistribution can be completed in a single gpexpand session scheduled during a low-use period. Larger systems may require you to plan multiple sessions and set the order of table redistribution, so as to minimize the performance impact.

Pivotal recommends completing the table redistribution in one session, if your database size and design permit it.

> **Important:** HDFS manages the disk space on your HAWQ system. Therefore, to perform table redistribution, verify that you have enough disk space on your Hadoop cluster, taking into consideration that HDFS replication (whose default is 3x) temporarily holds a copy of your largest table. Each table is unavailable for read and write operations while gpexpand is redistributing it among the segments.

The performance impact of table redistribution depends on the size, storage type, and partitioning design of a table. Redistributing a table with `gpexpand` takes approximately as much time per table as a `CREATE TABLE AS SELECT` operation would take. When redistributing a terabyte-scale fact table, the expansion utility can use a significant portion of available system resources, with resulting impact on the performance of queries or other database workload items.

Redistributing Append-Only and Compressed Tables

Append-only and compressed append-only tables are redistributed by gpexpand at different rates. The CPU capacity required to compress and decompress data tends to increase the impact on system performance. For similar-sized tables with similar-sized data, you may find overall performance differences, such as zlib-compressed append-only tables expanding at a significantly slower rate than uncompressed append-only tables (which can be potentially up to 80% slower).

Redistributing Tables with User-Defined Data Types

Certain sequences of alter operations on tables could render such tables unalterable from a redistribution perspective. gpexpand does not support redistribution of unalterable tables. For example, if you have a table initially created with a column of user-defined types and the column is subsequently dropped, this table may qualify as unalterable. If gpexpand reports a table as unalterable, you need to redistribute the table manually. To do this, create a new table matching the schema of the unalterable table and execute the following statement:

```
INSERT INTO <new_table>
```

```
SELECT * FROM <unalterable table>;
```

Redistributing Partitioned Tables

Because the expansion utility can process a large table partition-by-partition, an efficient partition design reduces the performance impact of table redistribution. Only the child tables of a partitioned table are set to a random distribution policy, and only one child partition table is unavailable during redistribution.

Preparing and Adding Nodes

To prepare new system nodes for expansion, install the HAWQ software binaries, exchange the required SSH keys and run performance tests. Pivotal recommends running performance tests at least twice: first on the new nodes only, and then on both the new and existing nodes together. The second set of tests must be run with the system offline, to prevent user activity from distorting test results.

Beyond these general guidelines, Pivotal recommends running performance tests any time that the networking of nodes is modified, or for any special conditions in the system environment. For example, if you plan to run the expanded system on two network clusters, run the performance tests on each cluster.

This rest of this section describes how to run the HAWQ administrative utilities to verify that your new nodes are ready for integration into the existing HAWQ system.

Adding New Nodes to the Trusted Host Environment

New nodes must exchange SSH keys with the existing nodes to allow HAWQ administrative utilities to connect to all segments without a password prompt.

Pivotal recommends performing the key exchange process twice: once as root (for administration convenience) and once as the gpadmin user (required for the HAWQ management utilities). Perform the following tasks in this order:

1. Exchange SSH keys as root.
2. Create the gpadmin user.
3. Exchange SSH keys as the gpadmin user.

Exchange SSH keys as root

1. Create two separate host list files:
   - One that has all of the existing host names in your HAWQ array.
   - One that has all of the new expansion hosts.

   For existing hosts, you can use the same host file that you used for the initial setup of SSH keys in the system.The files should include all hosts (master, backup master and segment hosts) and list one host name per line. If using a multi-NIC configuration, make sure to exchange SSH keys using all of the configured host names for a given host. Make sure there are no blank lines or extra spaces. For example:

```
mdw   OR  masterhost
sdw1-1    seghost1
sdw1-2    seghost2
sdw1-3    seghost3
sdw1-4
sdw2-1
sdw2-2
sdw2-3
sdw2-4
sdw3-1
sdw3-2
sdw3-3
sdw3-4
```

2. Log in as `root` on the master host, and source the `greenplum_path.sh` file from your HAWQ installation.

```
$ su -
# source /usr/local/hawq/greenplum_path.sh
```

3. Run the `gpssh-exkeys` utility, referencing the host list files. For example:

```
# gpssh-exkeys -f /home/gpadmin/existing_hosts_file -x /home/gpadmin/new_hosts_
file
```

4. `gpssh-exkeys` will check the remote hosts and perform the key exchange between all hosts. Enter the `root` user password when prompted. For example:

```
***Enter password for root@hostname: <root_password>
```

Create the gpadmin user

1. Use `gpssh` to create the `gpadmin` user on all of the new segment hosts (if the `gpadmin` user does not exist already). Use the list of new hosts that you created for the key exchange. For example:

```
# gpssh -f new_hosts_file '/usr/sbin/useradd gpadmin -d /home/gpadmin -s /bin/
bash'
```

2. Set the new `gpadmin` user's password. On Linux, you can do this on all segment hosts at once by using `gpssh`. For example:

```
# gpssh -f new_hosts_file 'echo gpadmin_password | passwd gpadmin --stdin'
```

   You must log in to each segment host and set the `gpadmin` user's password on each host. For example:

```
# ssh <segment_hostname>
# passwd gpadmin
# New password: <gpadmin_password>
# Retype new password: <gpadmin_password>
```

3. Verify that the `gpadmin` user has been created by searching for its home directory:

```
# gpssh -f new_hosts_file ls -l /home
```

Exchange SSH keys as the gpadmin user

Log in as `gpadmin`, and run the `gpssh-exkeys` utility, referencing the host list files. For example:

```
# gpssh-exkeys -e /home/gpadmin/existing_hosts_file -x /home/gpadmin/new_hosts_file
```

`gpssh-exkeys` will check the remote hosts and perform the key exchange between all hosts. Enter the `gpadmin` user password when prompted. For example:

```
***Enter password for gpadmin@hostname: <gpadmin_password>
```

Verifying OS Settings

Use the `gpcheck` utility to verify that all the new hosts in your array have the correct OS settings for running the HAWQ software.

To run gpcheck:

1. Log in on the master host as the user who will be running your HAWQ system (e.g., `gpadmin`):

```
$ su - gpadmin
```

2. Run the `gpcheck` utility using your host file for new hosts. For example:

```
$ gpcheck -f new_hosts_file
```

Validating Disk I/O and Memory Bandwidth

Use the `gpcheckperf` utility to test disk I/O and memory bandwidth.

To run gpcheckperf:

1. Run the `gpcheckperf` utility using the host file for new hosts. Use the `-d` option to specify the file systems you want to test on each host (you must have write access to these directories). For example:

   ```
   $ gpcheckperf -f new_hosts_file -d /data1 -d /data2 -v
   ```

2. The utility may take a while to perform the tests, as it is copying very large files between the hosts. When it is finished, you will see the summary results for the Disk Write, Disk Read, and Stream tests.

If your network is divided into subnets, repeat this procedure with a separate host file for each subnet.

Integrating New Hardware into the System

Before initializing the system with all new segments, repeat the performance tests on all nodes in the system, new and existing. Shut down the system and run these same tests using host files that include *all* nodes, existing and new:

- Verifying OS Settings
- Validating Disk I/O and Memory Bandwidth

Because user activity may skew the results of these test, you must shut down HAWQ (`gpstop`) before running them.

Installing HAWQ Components on the New Segments

This topic describes how to install the HAWQ components on the new segments created after running `gpexpand`.

Installing PL/R and pgcrypto after Expansion

If you have already installed the PL/R and pgcrypto packages on existing segments, use the following instructions to install these packages on the expanded segments:

1. Ensure that you have installed HAWQ binaries on all new segments.
2. Run `gpssh-exkeys` to set up password-less `ssh` on the cluster.
3. Untar the package if you are using the same versions of the packages installed on the existing cluster:
   - For PL/R:

     ```
     mkdir plr
     mv plr*.tgz plr
     cd plr
     tar -xzf plr*.tgz
     ```
   - For pgcrypto:

     ```
     mkdir pgcrypto
     mv pgcrypto.tgz pgcrypto
     cd pgcrypto
     tar -xzf pgcrypto.tgz
     ```
4. Ensure that the hostfile only lists new hostnames, each on a new line.
5. Run the install in expand mode:
   - For PL/R:

     ```
     ./plr_install.sh -f ~/hostfile -x
     ```
   - For pgcrypto:

     ```
     ./pgcrypto_install.sh -f ~/hostfile -x
     ```

Installing PL/Java after Expansion

If you have already installed PL/Java on existing segments, use the following instructions to install these packages on the expanded segments:

These instructions assume that you have a precompiled build of PL/Java from Pivotal.

Note: Before you install PL/Java:

- PL/Java is bundled with the HAWQ package at the following location: `/usr/local/hawq/share/postgresql/pljava/`.
- Ensure that the `$JAVA_HOME` variable is set to the same path on the master and all the segments.
- Ensure that the `LD_LIBRARY_PATH` variable also holds the path for `libjvm.so` file. For example, if the `libjvm.sois` at the location `LD_LIBRARY_PATH=$GPHOME/lib:$GPHOME/ext/python/`

lib:$LD_LIBRARY_PATH:/usr/java/jdk1.7.0_45/jre/lib/amd64/server/, change the `LD_LIBRARY_PATH` variable in `greenplum_path.sh` as follows:

```
export LD_LIBRARY_PATH=$GPHOME/lib:$GPHOME/ext/python/lib:$LD_LIBRARY_PATH:/
usr/java/jdk1.7.0_45/jre/lib/amd64/server/
```

- PL/Java is compatible with JDK 1.6 and 1.7.

1. Run the installer:

```
./pljava_install.sh -f ~/hosts.txt
```

Where `~/hosts.txt` is a text file containing hostnames of segment hosts in HAWQ deployment that are currently active. The file must contain one hostname per line.

2. Restart HAWQ.

```
source $GPHOME/greenplum_path.sh
gpstop -ar
```

3. Add the PL/Java class of configuration variables:

```
gpconfig -c custom_variable_classes -v \'pljava\'
```

If you have existing `custom_variable_classes` defined, prefix them with `pljava` in a comma-separated list.

4. Run the `CREATE LANGUAGE` command:

```
psql -d <dbname> -c "CREATE LANGUAGE pljava"
```

Run this command for every database where you want to install PL/Java.

Installing Custom JARS

1. Copy the jar file on the master host at `$GPHOME/lib/postgresql/java`.
2. Use `gpscp` to copy the jar file from the master to segments in the same location.

```
cd $GPHOME/lib/postgresql/java
gpscp -f ~/hosts.txt myfunc.jar =:$GPHOME/lib/postgresql/java/
```

3. Set `pljava_classpath` to include the newly-copied jar file.
4. From the `psql` session, execute the following:

```
set pljava_classpath='myfunc.jar';
```

This setting will be in effect only for the `psql` session. If you want it to affect all sessions, use:

```
gpconfig -c pljava_classpath -v \'myfunc.jar\'
```

Installing MADlib on Newly-Added Nodes

The following steps assume that MADlib was installed and running before adding new nodes. If MADlib was not installed, you can install it using the instructions provided in *HAWQ Installation and Upgrade*.

1. Download the MADlib RPM.
2. Make sure that you have HAWQ binaries installed properly on all master and segment nodes in your cluster.
3. Make sure the `hostfile` lists all the new segment nodes.
4. Run the following command:

   ```
   hawq_install.sh -r <RPM_FILEPATH> -f <HOSTFILE>
   ```
5. Complete the process by initializing the new segments. For more information, see *Initializing New Segments*.

Initializing New Segments

Use the `gpexpand` utility to initialize the new segments, create the expansion schema, and set a system-wide random distribution policy for the database. The utility performs these tasks by default the first time you run it with a valid input file on the HAWQ master. Subsequently, it will detect that an expansion schema has been created, and perform table redistribution.

Creating an Input File for System Expansion

To begin expansion, the gpexpand utility requires an input file containing information about the new segments and hosts. If you run `gpexpand` without specifying an input file, the utility displays an interactive interview that collects the required information and automatically creates an input file for you.

If you choose to create the input file by using the interactive interview, you can optionally specify a file containing a list of expansion hosts. If your platform or command shell limits the length of the list of hostnames you are allowed enter when prompted in the interview, specifying the hosts with `gpexpand -f` (as shown below) could be mandatory.

Creating an input file in Interactive Mode

Before running `gpexpand` to create an input file in interactive mode, make sure you have the following required information:

- Number of new hosts
- New hostnames (or a hosts file)
- Number of segments to add per host, if any

The utility automatically generates an input file based on this information and on the `dbid`, `content` ID, and data directory values stored in `gp_segment_configuration` and `pg_filespace`, then saves the file in the current directory.

To create an input file in interactive mode:

1. Log in to the master host as the user who will be running your HAWQ system (for example, `gpadmin`).
2. Run `gpexpand`. The utility displays messages about preparing for an expansion operation and prompts you to quit or continue. Optionally, you can specify a hosts file using `-f`. For example:

```
$ gpexpand -f /home/gpadmin/new_hosts_file
```

3. At the prompt, select `Y` to continue.
4. Enter a comma-separated list of the hostnames of the new expansion hosts. If you specified a hosts file using `-f`, go to the next step. Your list should appear as follows:

```
> sdw5, sdw6, sdw7, sdw8
```

   Note: To add segments to existing hosts only, enter a blank line at this prompt. Do not specify localhost or any existing host name.

5. Enter the number of new segments to add, if any. By default, the number of new hosts initialized corresponds to the number of existing segments. Optionally, you can increase the number of segments per host. For example, if existing hosts currently have two segments each, entering a value of 2 will initialize two additional segments on the existing hosts, and four new segments on new hosts.
6. If you are adding new segments, enter the metadata path for each new segment.

After you have entered all required information, the utility generates an input file and saves it in the current directory. For example:

```
gpexpand_inputfile_yyyymmdd_145134
```

If your system has shared filesystem filespaces, `gpexpand` expects a filespace configuration file (*`input_file_name`*`.fs`) to exist in the same directory as your expansion configuration file. See *User-defined Filespaces and gpexpand* for more information.

User-defined Filespaces and gpexpand

This topic describes two scenarios:

- HAWQ with no User-defined Filespaces
- HAWQ with a User-defined Filespace

HAWQ with no User-defined Filespaces

When you initialize a new HAWQ cluster, it has 2 filespaces by default: `pg_system` and `dfs_system` (Lookup system tables `pg_filespace` & `pg_filespace_entry`).

- `pg_system` stores all the metadata used by the Master and the segments. This is a local filesystem path that corresponds to that segment.
- `dfs_system` stores all the user data. Unlike `pg_system`, this is a shared filespace and is a path under HDFS.

Since HAWQ has these two default filespaces, the expansion utility expects corresponding filespaces for the new segments. `gpexpand` requests local filesystem paths for `pg_system` filespace, but auto-generates paths for shared filespace paths, to maintain consistency of paths between all the segment data directories

HAWQ with a User-defined Filespace

This means that one or more filespaces, other than the default, have been defined in the existing HAWQ system.

You can use the `gpfilespace` utility to add filespaces to your HAWQ system. User-defined filespaces always have a shared path.

Therefore, if you have one or more user-defined filespaces in your HAWQ system, `gpexpand` requests local filesystem paths for the `pg_system` filespace, but auto-generates paths for shared filespace paths so that it can maintain consistency of paths between all the segment data directories.

Expansion Input File Format

You can create your own input file in the required format. Unless you have special needs for your expansion scenario, Pivotal recommends creating the input file using the interactive interview process.

The format for the expansion `input.fs` file is:

```
filespaceOrder=filespace1_name :filespace2_name : ...
dbid:/path/for/filespace1 :/path/for/filespace2 : ...
dbid:/path/for/filespace1 :/path/for/filespace2 : ...
...
```

An expansion input file in this format requires the following information for each new segment:

Table 5: Input file format

Parameter	Values	Description
hostname	hostname	Hostname for the segment host
port	An available port number	Database listener port for the segment, incremented on the existing segment **port** base number.
fselocation	Directory name	The data directory (filespace) location for a segment as per the `pg_filespace_entry` system catalog.
dbid	Integer. Must not conflict with existing *dbid* values.	Database ID for the segment. The values you enter should be incremented sequentially from existing*dbid* values shown in the system catalog `gp_segment_configuration`. For example, to add four nodes to an existing ten-segment array with *dbid* values of 1-10, list new *dbid* values of 11, 12, 13 and 14.
content	Integer. Must not conflict with existing *content* .	The content ID of the segment. A primary segment and its mirror should have the same content ID, incremented sequentially from existing values. For more information, see **content** in the reference for `gp_segment_configuration`.
preferred_role	p	"p" (primary) is the only option.

Running gpexpand to Initialize New Segments

After you have created an input file, run `gpexpand` to initialize new segments. The utility will automatically stop HAWQ for the time required to initialize the segments, then restarts the system when finished.

To run gpexpand with an input file:

1. Log in to the master host as the user running your HAWQ system (for example, `gpadmin`).
2. Run the `gpexpand` utility, specifying the input file with `-i`.

   ```
   $ gpexpand -i input_file -D database1
   ```

   The utility detects if there is an existing expansion schema for the HAWQ system. If there is an existing schema, you must remove it with `gpexpand -c` before beginning a new expansion operation. See *Removing the Expansion Schema*. When the new segments are initialized and the expansion schema is successfully created, the utility prints a success message and exits.

When the initialization process is complete, you can connect to HAWQ and view the expansion schema. The schema resides in the database you specified with `-D`, or in the database specified by the `PGDATABASE` environment variable. For more information, see *About the Expansion Schema*.

Rolling Back a Failed Expansion Setup

You can roll back a failed expansion setup operation by using the command `gpexpand -r | --rollback`. However, this command is only allowed in a failure scenario. Once a setup operation has completed successfully, the expansion is committed, and you cannot roll back.

To roll back a failed expansion setup, use the following command, specifying the database that contains the expansion schema:

```
gpexpand --rollback -D database_name
```

Redistributing Tables

After successfully creating an expansion schema, you can bring HAWQ back online and redistribute tables across the entire array. You can redistribute tables with `gpexpand` at specified intervals, targeting low-use hours when the utility's CPU usage and table locks will have the least impact on database operations. Also, you can rank tables to ensure that the largest or most critical tables are redistributed in your preferred order.

While the redistribution of tables is underway:

- Any new tables or partitions created will be distributed across all segments exactly as they would be under normal operating conditions.
- Queries will use all segments, even though the relevant data may not yet have been redistributed to the tables on the new segments.

The table or partition currently being redistributed will be locked and unavailable for read or write operations. When its redistribution is completed, normal operations resume.

> **Note:** `gpexpand` does not support redistribution of unalterable tables. Some sequences of alter operations on tables could render those tables unalterable for redistribution. For example, if you createa table with a column of user-defined types, then subsequently drop the column, this table may become unalterable. As a workaround, if gpexpand reports a table as unalterable, you need to redistribute the table manually. To do this, create a new table matching the schema of the unalterable table and execute the following command:
>
> ```
> insert into <new_table> select * from <unalterable table>;
> ```

To redistribute tables with gpexpand:

1. Log in to the master host as the user who will be running your HAWQ system (for example, gpadmin).
2. Run the `gpexpand` utility. Optionally, you can use either the `-d` or `-e` option to define the time period for the expansion session. For example, to run the utility for a maximum of 60 consecutive hours:

   ```
   $ gpexpand -d 60:00:00
   ```

   The utility redistributes tables until the last table in the schema is successfully marked completed, or until the specified duration or end time is reached. Each time a session is started or finished, the utility updates the status and updated time in `gpexpand.status`.

Monitoring Table Redistribution

At any time during the process of redistributing tables, you can query the expansion schema. The view `gpexpand.expansion_progress` provides a summary of the current progress, including calculations of the estimated rate of table redistribution and estimated time to completion. The table `gpexpand.status_detail` can be queried for per-table status information.

Viewing Expansion Status

Because the estimates in `gpexpand.expansion_progress` are based on the rates achieved for each table, the view cannot calculate an accurate estimate until the first table has completed. Calculations are restarted each time you re-run gpexpand to start a new table redistribution session.

To monitor progress by querying `gpexpand.expansion_progress`, connect to HAWQ using psql or another supported client. Query `gpexpand.expansion_progress` with a command like the following:

```
=# select * from gpexpand.expansion_progress;
name      |   value
------------------------------+-----------------------
```

```
Bytes Left    |   5534842880
Bytes Done    |   142475264
Estimated Expansion Rate  |   680.75667095996092 MB/s
Estimated Time to Completion    |   00:01:01.008047
Tables Expanded     |   4
Tables Left         |   4
(6 rows)
```

Viewing Table Status

The table `gpexpand.status_detail` stores status, last updated time, and other useful information about each table in the schema. To monitor the status of a particular table by querying `gpexpand.status_detail`, connect to HAWQ using psql or another supported client. Query `gpexpand.status_detail` with a command similar to the following:

```
=> SELECT status, expansion_started, source_bytes FROM gpexpand.status_detail WHERE
 fq_name = ‘public.sales’;
status     |  expansion_started     |  source_bytes
-----------+----------------------------+-----------------------------------
COMPLETED  |  2009-02-20 10:54:10.043869  |  4929748992
(1 row)
```

Removing the Expansion Schema

The expansion schema can safely be removed after the expansion operation is completed and verified. To run another expansion operation on a HAWQ system, you must first remove the existing expansion schema.

To remove the expansion schema:

1. Log in to the master host as the user who will be running your HAWQ system (for example, `gpadmin`).
2. Run the `gpexpand` utility with the `-c` option. For example:

```
$ gpexpand -c
```

Chapter 10

HAWQ InputFormat for MapReduce

MapReduce is a programming model developed by Google for processing and generating large data sets on an array of commodity servers. You can use the HAWQ InputFormat option to enable MapReduce jobs to access HAWQ data stored in HDFS.

To use HAWQ InputFormat, you only need to provide the URL of the databasen to connect to, along with the table name you want to access. HAWQ InputFormat only fetches the metadata of the database and table of interest, which is much less data than the table data itself. After getting the metadata, the HAWQ InputFormat determines where and how the table data is stored in HDFS. It reads and parses those HDFS files and processes the parsed table tuples directly inside a Map task.

This chapter describes the document format and schema for defining HAWQ MapReduce jobs.

Supported Data Types

HAWQ InputFormat supports the following data types:

SQL/HAWQ	JDBC/JAVA	setXXX	getXXX
DECIMAL/NUMERIC	java.math.BigDecimal	setBigDecimal	getBigDecimal
FLOAT8/DOUBLE PRECISION	double	setDouble	getDouble
INT8/BIGINT	long	setLong	getLong
INTEGER/INT4/INT	int	setInt	getInt
FLOAT4/REAL	float	setFloat	getFloat
SMALLINT/INT2	short	setShort	getShort
BOOL/BOOLEAN	boolean	setBoolean	getBoolean
VARCHAR/CHAR/TEXT	String	setString	getString
DATE	java.sql.Date	setDate	getDate
TIME/TIMETZ	java.sql.Time	setTime	getTime
TIMESTAMP/ TIMSTAMPTZ	java.sql.Timestamp	setTimestamp	getTimestamp
ARRAY	java.sq.Array	setArray	getArray
BIT/VARBIT	com.pivotal.hawq. mapreduce.datatype.	setVarbit	getVarbit
BYTEA	byte[]	setByte	getByte
INTERVAL	com.pivotal.hawq. mapreduce.datatype. HAWQInterval	setInterval	getInterval
POINT	com.pivotal.hawq. mapreduce.datatype. HAWQPoint	setPoint	getPoint
LSEG	com.pivotal.hawq. mapreduce.datatype. HAWQLseg	setLseg	getLseg
BOX	com.pivotal.hawq. mapreduce.datatype. HAWQBox	setBox	getBox
CIRCLE	com.pivotal.hawq. mapreduce.datatype. HAWQCircle	setVircle	getCircle
PATH	com.pivotal.hawq. mapreduce.datatype. HAWQPath	setPath	getPath

SQL/HAWQ	JDBC/JAVA	setXXX	getXXX
POLYGON	com.pivotal.hawq.mapreduce.datatype.HAWQPolygon	setPolygon	getPolygon
MACADDR	com.pivotal.hawq.mapreduce.datatype.HAWQMacaddr	setMacaddr	getMacaddr
INET	com.pivotal.hawq.mapreduce.datatype.HAWQInet	setInet	getInet
CIDR	com.pivotal.hawq.mapreduce.datatype.HAWQCIDR	setCIDR	getCIDR

HAWQ InputFormat Example

The following example shows how you can use the HAWQ InputFormat to access HAWQ table data from MapReduce jobs.

```
package com.mycompany.app;
import com.pivotal.hawq.mapreduce.HAWQException;
import com.pivotal.hawq.mapreduce.HAWQInputFormat;
import com.pivotal.hawq.mapreduce.HAWQRecord;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.io.IntWritable;

import java.io.IOException;
public class HAWQInputFormatDemoDriver extends Configured
implements Tool {

 // CREATE TABLE employees (
 // id INTEGER NOT NULL, name VARCHAR(32) NOT NULL);
 public static class DemoMapper extends
  Mapper<Void, HAWQRecord, IntWritable, Text> {
    int id = 0;
    String name = null;
    public void map(Void key, HAWQRecord value, Context context)
  throws IOException, InterruptedException {
     try {
  id = value.getInt(1);
  name = value.getString(2);
     } catch (HAWQException hawqE) {
  throw new IOException(hawqE.getMessage());
     }
     context.write(new IntWritable(id), new Text(name));
    }
 }
 private static int printUsage() {
    System.out.println("HAWQInputFormatDemoDriver
    <database_url> <table_name> <output_path> [username]
    [password]");
    ToolRunner.printGenericCommandUsage(System.out);
    return 2;
 }

 public int run(String[] args) throws Exception {
    if (args.length < 3) {
     return printUsage();
    }
    Job job = new Job(getConf());
    job.setJobName("hawq-inputformat-demo");
    job.setJarByClass(HAWQInputFormatDemoDriver.class);
    job.setMapperClass(DemoMapper.class);
    job.setMapOutputValueClass(Text.class);
    job.setOutputValueClass(Text.class);
    String db_url = args[0];
    String table_name = args[1];
    String output_path = args[2];
    String user_name = null;
    if (args.length > 3) {
      user_name = args[3];
    }
    String password = null;
    if (args.length > 4) {
      password = args[4];
    }
    job.setInputFormatClass(HAWQInputFormat.class);
    HAWQInputFormat.setInput(job.getConfiguration(), db_url,
    user_name, password, table_name);
    FileOutputFormat.setOutputPath(job, new
    Path(output_path));
    job.setNumReduceTasks(0);
    int res = job.waitForCompletion(true) ? 0 : 1;
    return res;
 }

 public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(),
      new HAWQInputFormatDemoDriver(), args);
    System.exit(res);
 }
}
```

To compile and run the example:

1. Add the following dependencies into the project for compilation:
   a. HAWQInputFormat jars (located in the `$GPHOME/lib/postgresql/hawq-mr-io` directory):
      - `hawq-mapreduce-common.jar`
      - `hawq-mapreduce-ao.jar`
      - `hawq_mapreduce-parquet.jar`
      - `hawq-mapreduce-tool.jar`
   b. Required 3rd party jars (located in the `$GPHOME/lib/postgresql/hawq-mr-io/lib` directory):
      - `parquet-column-1.1.0.jar`
      - `parquet-common-1.1.0.jar`
      - `parquet-encoding-1.1.0.jar`
      - `parquet-format-1.1.0.jar`
      - `parquet-hadoop-1.1.0.jar`
      - `postgresql-jdbc.jar`
      - `snakeyaml.jar`
   c. Hadoop Mapreduce related jars (located in the install directory of your Hadoop distribution).
2. Check that you have installed HAWQ, HDFS and Yarn.
3. Create sample table:
   a. Log in to HAWQ:

      ```
       psql -d postgres
      ```
   b. Create the sample table:

      ```
      CREATE TABLE employees (
      id INTEGER NOT NULL PRIMARY KEY,
      name TEXT NOT NULL);
      ```

      Or a Parquet table:

      ```
      CREATE TABLE employees ( id INTEGER NOT NULL PRIMARY KEY, name TEXT NOT NULL)
       with (appendonly=true, orientation=parquet);
      ```
   c. Insert one tuple:

      ```
       INSERT INTO employees VALUES (1, 'Paul');
      ```

d. Use the following shell script snippet showing how to run the Mapreduce job:

```
# Suppose all five needed jars are under ./lib
export
LIBJARS=lib/hawq-mapreduce-common.jar,lib/hawq-mapreduce-ao.
jar,lib/hawq-mapreduce-tool.jar,lib/postgresql-9.2-1003-jdbc
4.jar,lib/snakeyaml-1.12.jar
export
HADOOP_CLASSPATH=lib/hawq-mapreduce-common.jar:lib/hawq-mapr
educe-ao.jar:lib/hawq-mapreduce-tool.jar:lib/postgresql-9.2-
1003-jdbc4.jar:lib/snakeyaml-1.12.jar
# Suppose the built application jar is my-app.jar
hadoop jar my-app.jar
com.mycompany.app.HAWQInputFormatDemoDriver -libjars
${LIBJARS} localhost:5432/postgres employees /tmp/employees
```

e. Use the following command to check the result of the Mapreduce job:

```
hadoop fs -cat /tmp/employees/*
```

The output will appear as follows:

```
1 Paul
```

Accessing HAWQ Data

You can access HAWQ data using the following interfaces:

- HAWQInputFormat.setInput API: Use this when HAWQ is running.
- Metadata Export Tool: Use this when HAWQ is not running.

HAWQInputFormat.setInput

```
/**
 * Initializes the map-part of the job with the appropriate input settings
 * through connecting to Database.
 *
 * @param conf
 * The map-reduce job configuration
 * @param db_url
 * The database URL to connect to
 * @param username
 * The username for setting up a connection to the database
 * @param password
 * The password for setting up a connection to the database
 * @param tableName
 * The name of the table to access to
 * @throws Exception
 */
public static void setInput(Configuration conf, String db_url,
 String username, String password, String tableName)
throws Exception;
```

Metadata Export Tool

Use the metadata export tool, gpextract, to export the metadata of the target table into a local YAML file:

```
gpextract [-h hostname] [-p port] [-U username] [-ddatabase] [-o output_file] [-W]
 <tablename>
```

Using the extracted metadata, access HAWQ data through the following interface:

```
/**
   * Initializes the map-part of the job with the appropriate input settings through
 reading metadata file stored in local filesystem.
   *
   * To get metadata file, please use gpextract first
   *
   * @param conf
   * The map-reduce job configuration
   * @param pathStr
   * The metadata file path in local filesystem. e.g.
   * /home/gpadmin/metadata/postgres_test
   * @throws Exception
   */
public static void setInput(Configuration conf, String pathStr)
   throws Exception;
```

Chapter 11

HAWQ Filespaces and High Availability Enabled HDFS

In previous versions of HAWQ, you may have initialized HAWQ on HDFS without the High Availability (HA) feature. Using the current version, you can now use the HDFS NameNode HA feature.

Enabling the HDFS NameNode HA feature

To enable the HDFS NameNode HA feature, you need to perform the following tasks:

- Collect information about the target filespace
- Stop the HAWQ Cluster and backup catalog
- Move the filespace location using the command line tool
- Configure ${GPHOME}/etc/hdfs-client.xml
- Reinitialize the standby master after moving the filespace

Collecting Information about the Target Filespace

A default filespace named dfs_system exists in the pg_filespace catalog and the parameter, pg_filespace_entry, contains detailed information for each filespace.

1. Use the following SQL query to gather information about the filespace located on HDFS:

```
SELECT
    fsname, fsedbid, fselocation
FROM
    pg_filespace as sp, pg_filespace_entry as entry, pg_filesystem as fs
WHERE
    sp.fsfsys = fs.oid and fs.fsysname = 'hdfs' and sp.oid = entry.fsefsoid
ORDER BY
    entry.fsedbid;
```

The sample output is as follows:

```
fsname    | fsedbid |              fselocation
------------+---------+--------------------------------------
 dfs_system |       1 | /data/hawq-kerberos/dfs/gpseg-1
 dfs_system |       2 | hdfs://mdw:9000/hawq-security/gpseg0
 dfs_system |       3 | hdfs://mdw:9000/hawq-security/gpseg1
 dfs_system |       4 | hdfs://mdw:9000/hawq-security/gpseg2
 dfs_system |       5 | hdfs://mdw:9000/hawq-security/gpseg3
 dfs_system |       6 | hdfs://mdw:9000/hawq-security/gpseg4
 dfs_system |       7 | hdfs://mdw:9000/hawq-security/gpseg5
(7 rows)
```

The output can contain the following:

- Master instance path information.
- Standby master instance path information, if the standby master is configured (not in this example).
- HDFS paths that share the same prefix for segment instances.

2. To enable HA HDFS, you need the segment location comprising the filespace name and the common prefix of segment HDFS paths. The segment location is formatted like a URL. The sample output displays the segment location, hdfs://mdw:9000/hawq-security. mdw:9000 is the NameNode host and RPC port, you must replace it with your HA HDFS cluster service ID to get the new segment location. For example hdfs://phdcluster/hawq-security.

```
Filespace Name: dfs_system
New segment location: hdfs://phdcluster/hawq-security
```

Note: To move the filespace location to a segment location that is different from the old segment location, you must move the data to new path on HDFS.

For example, you can do this by moving the filespace from hdfs://phdcluster/hawq-security to hdfs://phdcluster/hawq/another/path.

Stopping the HAWQ Cluster and Backing Up the Catalog

To enable HA HDFS, you are changing the HAWQ catalog and persistent tables. You cannot preform transactions while persistent tables are being updated. Therefore, before you move the filespace location, Pivotal recommends that you back up the catalog. This is to ensure that you do not lose data due to a hardware failure or during an operation (such as killing the HAWQ process).

1. Disconnect all workload connections. Check the active connection with:

   ```
   psql -c "select * from pg_catalog.pg_stat_activity"
   ```

2. Issue a checkpoint:

   ```
   psql -c "checkpoint"
   ```

3. Shut down the HAWQ cluster:

   ```
   gpstop -a -M fast
   ```

4. Copy the master data directory:

   ```
   $cp -r MASTER_DATA_DIRECTORY /catalog/backup/location
   ```

Moving the Filespace Location

HAWQ provides the command line tool, `gpfilespace`, to move the location of the filespace.

Define `$MASTER_DATA_DIRECTORY` to point to the *MASTER_DATA_DIRECTORY* path. For example:

```
export MASTER_DATA_DIRECTORY=/data1/master/gpseg-1/
```

Note: The path in this example may not reflect your actual *MASTER_DATA_DIRECTORY* path.

Run the following command to move a file space location:

```
gpfilespace --movefilespace default --location=hdfs://phdcluster/hawq-security
```

Note:

1. If the target filespace is not default filespace, replace default in command line with the actual filespace name.
2. Replace `hdfs://phdcluster/hawq-security` with new segment location.

Important: Errors while moving the location of the filespace:

Non-fatal error can occur if you provide invalid input or if you have not stopped HAWQ before attempting a filespace location change. Check that you have followed the instructions from the beginning, or correct the input error before you re-run gpfilespace.

Fatal errors can occur due to hardware failure or if you fail to kill a HAWQ process before attempting a filespace location change. When a fatal error occurs, you will see the message, "PLEASE RESTORE MASTER DATA DIRECTORY" in the output. If this occurs, shut down the database and restore the `$MASTER_DATA_DIRECTORY`.

Configure ${GPHOME}/etc/hdfs-client.xml

Configure the `hdfs-client.xml` file. See *HAWQ Installation and Upgrade* for more information.

Reinitialize the Standby Master

The standby master catalog is rendered invalid during the move, and needs to be reinitialized. If you did not have a standby master configured, you can skip this task.

```
gpstart -a                              #start HAWQ cluster
gpinitstandby -r                        #remove standby master
gpinitstandby -s <standby host name>  #initialize a standby master
```

Chapter 12

SQL Command Reference

This section contains a description and the syntax of the SQL commands supported by HAWQ.

DROP FUNCTION
DROP GROUP
DROP OPERATOR
DROP OPERATOR CLASS
DROP OWNED
DROP RESOURCE QUEUE
DROP ROLE
DROP SCHEMA
DROP SEQUENCE
DROP TABLE
DROP TABLESPACE
DROP TYPE
DROP USER
DROP VIEW
END
EXECUTE
EXPLAIN
FETCH
GRANT
INSERT
PREPARE
REASSIGN OWNED
RELEASE SAVEPOINT
RESET
REVOKE
ROLLBACK
ROLLBACK TO SAVEPOINT
SAVEPOINT
SELECT
SELECT INTO
SET
SET ROLE
SET SESSION AUTHORIZATION
SHOW
TRUNCATE
VACUUM

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 HAWQ extension present for historical reasons. ROLLBACK is the equivalent standard SQL command.

See Also

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

Related Links

SQL Command Reference

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`*

Related Links

SQL Command Reference

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

HAWQ 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 HAWQ.

See Also

CREATE AGGREGATE , *DROP AGGREGATE*

Related Links

SQL Command Reference

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.

Example

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

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

Compatibility

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

See Also

CREATE OPERATOR , *DROP OPERATOR*

Related Links

SQL Command Reference

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` statement in the SQL standard.

See Also

CREATE OPERATOR , *DROP OPERATOR CLASS*

Related Links

SQL Command Reference

ALTER ROLE

Changes a database role (user or group).

Synopsis

```
ALTER ROLE name RENAME TO newname

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
    | CREATE-ROLE | NOCREATE-ROLE
    | 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 HAWQ role. There are several variants of this command:

- **RENAME** — Changes the name of the role. Database superusers can rename any role. Roles having `CREATE-ROLE` 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 `CREATE-ROLE` privilege can change defaults for non-superuser roles. See "Server Configuration Parameters" for more information on 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 `CREATE-ROLE` 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 *HAWQ 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
CREATE-ROLE | NOCREATE-ROLE
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
```

`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.

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.

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 CREATE-ROLE 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 HAWQ extension.

See Also

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

Related Links

SQL Command Reference

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
      [ ENCODING ( storage_directive [,…] ) ]
      [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]
  SET WITHOUT OIDS
  INHERIT parent_table
  NO INHERIT parent_table
  OWNER TO new_owner
```

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 name
          partition_element
      [ ( subpartition_spec ) ]
  EXCHANGE DEFAULT PARTITION WITH TABLE table_name
        [ WITH | WITHOUT VALIDATION ]
  EXCHANGE PARTITION { partition_name | FOR (RANK(number)) |
       FOR (value) } 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}
    BLOCKSIZE={8192-2097152}
    ORIENTATION={COLUMN|ROW}
    COMPRESSTYPE={ZLIB|QUICKLZ|RLE_TYPE|NONE}
    COMPRESSLEVEL={0-9}
    FILLFACTOR={10-100}
    OIDS[=TRUE|FALSE]
```

where *storage_directive* is:

```
    COMPRESSTYPE={ZLIB | QUICKLZ | RLE_TYPE | NONE}
  | COMPRESSLEVEL={0-9}
  | BLOCKSIZE={8192-2097152}
```

where *column_reference_storage_directive* is:

```
    COLUMN column_name ENCODING ( storage_directive [, … ] ), …
  | DEFAULT COLUMN ENCODING ( storage_directive [, … ] )
```

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 HAWQ 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 HAWQ 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 HAWQ. Also a table is only allowed one unique constraint and the uniqueness must be within the HAWQ 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.

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.

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-only 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).

WITH TABLE *table_name* - The name of the table you are swapping in to the partition design.

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.

`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

Take special care when altering or dropping columns that are part of the HAWQ distribution key as this can change the distribution policy for the table. HAWQ does not currently support foreign key constraints. For a unique constraint to be enforced in HAWQ, 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.

> **Note:** Note: The table name specified in the `ALTER TABLE` command cannot be the name of a partition within a table.

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.

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

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 HAWQ 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`*

Related Links

SQL Command Reference

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_* (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*

Related Links

SQL Command Reference

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*

Related Links

SQL Command Reference

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
    | CREATE-ROLE | NOCREATE-ROLE
    | 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 HAWQ extension. The SQL standard leaves the definition of users to the implementation.

See Also

`ALTER ROLE`

Related Links

SQL Command Reference

ANALYZE

Collects statistics about a database.

Synopsis

```
ANALYZE [VERBOSE] [ROOTPARTITION] 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, the query planner uses these statistics to help determine the most efficient execution plans for queries.

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

Parameters

`VERBOSE`

Enables display of progress messages. When specified, `ANALYZE` emits progress messages to indicate which table is currently being processed. Various statistics about the tables are printed as well.

`ROOTPARTITION`

For partitioned tables, `ANALYZE` on the parent (i.e. root in multi-level partitioning) table without this option will collect statistics on each individual leaf partition as well as the global partition table, both of which are needed for query planning. In scenarios when all the individual child partitions have up-to-date statistics (for example, after loading and analyzing a daily partition), the `ROOTPARTITION` option can be used to collect only the global stats on the partition table. This could save the time of re-analyzing each individual leaf partition.

If you use `ROOTPARTITION` on a non-root or non-partitioned table, `ANALYZE` will skip the option and issue a warning. You can also analyze all root partition tables in the database by using `ROOTPARTITION ALL`

Note: Use `ROOTPARTITION ALL` to analyze all root partition tables in the database.

`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 will help the query planner to choose the most appropriate query plan, and thereby improve the speed of query processing. A common strategy is to run `VACUUM` and `ANALYZE` once a day during a low-usage time of day.

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

There may be situations where the remote Analyzer may not be able to perform a task on a PXF table. For example, if a PXF Java component is down, the remote analyzer may not perform the task, so that the database transaction can succeed. In these cases the statistics remain with the default external table values.

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`*

Related Links

SQL Command Reference

BEGIN

Starts a transaction block.

Synopsis

```
BEGIN [WORK | TRANSACTION] [SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
 UNCOMMITTED]
      [READ WRITE | READ ONLY]
```

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`), HAWQ 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.

Parameters

`WORK`
`TRANSACTION`

Optional key words. They have no effect.

`SERIALIZABLE`
`REPEATABLE READ`
`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 HAWQ, `READ UNCOMMITTED` is treated the same as `READ COMMITTED`. `SERIALIZABLE` is supported the same as `REPEATABLE READ` wherein all statements of the current transaction can only see rows committed before the first statement was executed in the transaction. `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

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;
```

Compatibility

`BEGIN` is a HAWQ language extension. It is equivalent to the SQL-standard command `START TRANSACTION lang="EN"`.

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`*, *`SAVEPOINT`*

Related Links

SQL Command Reference

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 HAWQ 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 HAWQ language extension.

Related Links

SQL Command Reference

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

HAWQ 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*

Related Links

SQL Command Reference

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`* , *`ROLLBACK`*

Related Links

SQL Command Reference

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 [, ...]] ]
```

Description

`COPY` moves data between HAWQ 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 HAWQ 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 HAWQ 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. You can then examine this error table to see error rows that were not loaded (if any). If the *error_table* specified already exists, it will be used. If it does not exist, it will be automatically generated. If the command auto-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 auto-generated 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`. An error 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;
```

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 HAWQ 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.

Notes

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

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 HAWQ 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 HAWQ master host machine, not the client. They must be accessible to and readable or writable by the HAWQ 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 HAWQ 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.

COPY supports creating readable foreign tables with error tables. The default for concurrently inserting into the error table is 127. You can use error tables with foreign tables under the following circumstances:

- Multiple foreign tables can use different error tables
- Multiple foreign tables cannot use the same error table

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 HAWQ 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. HAWQ `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 HAWQ.

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`

Related Links

SQL Command Reference

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 HAWQ. 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.

HAWQ 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 HAWQ, 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.

PREFUNC is optional. If defined, it is executed on master. Input to PREFUNC is partial results from segments, and not the tuples. If PREFUNC is not defined, the aggregate cannot be executed in parallel. PREFUNC and gp_enable_multiphase_agg are used as follows:

- gp_enable_multiphase_agg = off: SFUNC is executed sequentially on master. PREFUNC, even if defined, is unused.
- gp_enable_multiphase_agg = on and PREFUNC is defined: SFUNC is executed in parallel, on segments. PREFUNC is invoked on master to aggregate partial results from segments.

```
CREATE OR REPLACE FUNCTION my_avg_accum(bytea,bigint) returns bytea
 as 'int8_avg_accum' language internal strict immutable;
CREATE OR REPLACE FUNCTION my_avg_merge(bytea,bytea) returns bytea as
 'int8_avg_amalg' language internal strict immutable;
CREATE OR REPLACE FUNCTION my_avg_final(bytea) returns numeric as
 'int8_avg' language internal strict immutable;
CREATE AGGREGATE my_avg(bigint) (   stype = bytea,sfunc = my_avg_
accum,prefunc = my_avg_merge,finalfunc = my_avg_final,initcond = ''
  );
```

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 HAWQ, it is required that the *sfunc*, *ffunc*, and *prefunc* functions used to create the aggregate are defined as `IMMUTABLE`.

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

Examples

Create a sum of cubes aggregate:

```
CREATE FUNCTION scube_accum(numeric, numeric) RETURNS numeric
 AS 'select $1 + $2 * $2 * $2'
 LANGUAGE SQL
 IMMUTABLE
 RETURNS NULL ON NULL INPUT;
CREATE AGGREGATE scube(numeric) (
 SFUNC = scube_accum,
 STYPE = numeric,
 INITCOND = 0 );
```

To test this aggregate:

```
CREATE TABLE x(a INT);
INSERT INTO x VALUES (1),(2),(3);
SELECT scube(a) FROM x;
```

Correct answer for reference:

```
SELECT sum(a*a*a) FROM x;
```

Compatibility

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

See Also

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

Related Links

SQL Command Reference

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 HAWQ. 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.

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

DROP DATABASE

Related Links

SQL Command Reference

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')
      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 [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'] )]
            | '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

`CREATE EXTERNAL TABLE` or `CREATE EXTERNAL WEB TABLE` creates a new readable external table definition in HAWQ. 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.

`CREATE WRITABLE EXTERNAL TABLE` or `CREATE WRITABLE EXTERNAL WEB TABLE` creates a new writable external table definition in HAWQ. 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. 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`) for all protocols and comma separated values (`CSV`) format for `gpfdist` and file protocols. 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 HAWQ.

Parameters

READABLE | WRITABLE

Specifiies the type of external table, readable being the default. Readable external tables are used for loading data into HAWQ. Writable external tables are used for unloading data.

WEB

Creates a readable or wrtiable web external table definition in HAWQ. 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 HAWQ 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'
```

For writable external tables, specifies the URI location of the `gpfdist` process that will collect data output from the HAWQ 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.

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 HAWQ 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 HAWQ superuser (`gpadmin`).
- `ON MASTER` runs the command on the master host only.
- `ON` *`number`* means the command will be executed by the specified number of segments. The particular segments are chosen randomly at runtime by the HAWQ 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 HAWQ 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. The *content* ID of the HAWQ 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' ***(options)***

Specifies the format of the external or web table data - either plain text (`TEXT`) or comma separated values (`CSV`) format.

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.

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 HAWQ 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.

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. It specifies an error table where rows with formatting errors will be logged when running in single row error isolation mode. You can then examine this error table to see error rows that were not loaded (if any). If the error_table specified already exists, it will be used. If it does not exist, it will be automatically generated.

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 HAWQ 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.

DISTRIBUTED BY (*column*, [...])
DISTRIBUTED RANDOMLY

Used to declare the HAWQ 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;
```

Compatibility

`CREATE EXTERNAL TABLE` is a HAWQ extension. The SQL standard makes no provisions for external tables.

See Also

`CREATE TABLE` , *`CREATE TABLE AS`* , *`COPY`* , *`INSERT`* , *`SELECT INTO`*

Related Links

SQL Command Reference

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 "User Defined Functions" in *Using HAWQ to Query Data*.

Limited Use of VOLATILE and STABLE Functions

To prevent data from becoming out-of-sync across the segments in HAWQ, 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 HAWQ 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 HAWQ.

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 HAWQ. 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 HAWQ 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 HAWQ 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 HAWQ 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.

HAWQ 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$SomeTag$
```

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 HAWQ 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*

Related Links

SQL Command Reference

CREATE GROUP

Defines a new database role.

Synopsis

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

where *option* can be:

```
       SUPERUSER | NOSUPERUSER
     | CREATEDB | NOCREATEDB
     | CREATE-ROLE | NOCREATE-ROLE
     | 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

`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

`CREATE ROLE`

Related Links

SQL Command Reference

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 HAWQ 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 HAWQ 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. A language handler has also been added for PL/R, but the PL/R language package is not pre-installed with HAWQ. For more information, see *Using Procedural Languages*.

The PL/R libraries require the correct versions of R to be installed, respectively. Download the appropriate extensions from *Pivotal Network*, then install the extensions. See the Pivotal HAWQ *Installation Guide* for details. This ensures that all underlying dependencies are installed along with the extensions.

To use the PL/R procedural language you must make sure that the systems that run HAWQ (master and all segments) have the R language installed and the PL/R package library (`plr.so`) added to their Pivotal HAWQ installation on all hosts. Pivotal provides compiled packages for R and PL/R that you can install.

There are two forms of the `CREATE LANGUAGE` command. In the first form, the user specifies the name of the desired language and the HAWQ 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

Indicates that this is a procedural language.

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`, `plpython`, `plpythonu`, and `plr`. `plpgsql` is already installed by default in HAWQ.

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 HAWQ 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 language is installed by default in HAWQ.

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. However, a language can be installed into the `template1` database, which will cause it to be available automatically in all subsequently-created 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 HAWQ 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 HAWQ extension.

See Also

`CREATE FUNCTION`

Related Links

SQL Command Reference

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 HAWQ 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 HAWQ language extension. The SQL standard does not provide for user-defined operators.

See Also

CREATE FUNCTION , *CREATE TYPE* , *ALTER OPERATOR* , *DROP OPERATOR*

Related Links

SQL Command Reference

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 6: 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 7: 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

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 8: 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 9: 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 HAWQ extension. There is no `CREATE OPERATOR CLASS` statement in the SQL standard.

See Also

ALTER OPERATOR CLASS , *DROP OPERATOR CLASS* , *CREATE FUNCTION*

Related Links

SQL Command Reference

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 HAWQ 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 HAWQ 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. Pivotal 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;
```

As a general guideline, 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.

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 HAWQ 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).

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 HAWQ 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

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 *HAWQ 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);
```

View the status of all resource queues:

```
SELECT q.oid as queueid, q.rsqname as rsqname,
 t1.value::int as rsqcountlimit,
 t2.value::int as rsqcountvalue,
 t3.value::real as rsqcostlimit,
 t4.value::real as rsqcostvalue,
 t5.value::real as rsqmemorylimit,
 t6.value::real as rsqmemoryvalue,
 t7.value::int as rsqwaiters,
 t8.value::int as rsqholders
FROM pg_resqueue q,
 pg_resqueue_status_kv() t1 (queueid oid, key text, value text),
 pg_resqueue_status_kv() t2 (queueid oid, key text, value text),
 pg_resqueue_status_kv() t3 (queueid oid, key text, value text),
 pg_resqueue_status_kv() t4 (queueid oid, key text, value text),
 pg_resqueue_status_kv() t5 (queueid oid, key text, value text),
 pg_resqueue_status_kv() t6 (queueid oid, key text, value text),
 pg_resqueue_status_kv() t7 (queueid oid, key text, value text),
 pg_resqueue_status_kv() t8 (queueid oid, key text, value text)
WHERE
 q.oid = t1.queueid
 AND t1.queueid = t2.queueid
 AND t2.queueid = t3.queueid
 AND t3.queueid = t4.queueid
 AND t4.queueid = t5.queueid
 AND t5.queueid = t6.queueid
 AND t6.queueid = t7.queueid
 AND t7.queueid = t8.queueid
 AND t1.key = 'rsqcountlimit'
 AND t2.key = 'rsqcountvalue'
 AND t3.key = 'rsqcostlimit'
 AND t4.key = 'rsqcostvalue'
 AND t5.key = 'rsqmemorylimit'
 AND t6.key = 'rsqmemoryvalue'
 AND t7.key = 'rsqwaiters'
 AND t8.key = 'rsqholders'
;
```

View the status of a specific status queue:

```
SELECT q.oid as queueid, q.rsqname as rsqname,
 t1.value::int as rsqcountlimit,
 t2.value::int as rsqcountvalue,
 t3.value::real as rsqcostlimit,
 t4.value::real as rsqcostvalue,
 t5.value::real as rsqmemorylimit,
 t6.value::real as rsqmemoryvalue,
 t7.value::int as rsqwaiters,
 t8.value::int as rsqholders
FROM pg_resqueue q,
 pg_resqueue_status_kv() t1 (queueid oid, key text, value text),
 pg_resqueue_status_kv() t2 (queueid oid, key text, value text),
 pg_resqueue_status_kv() t3 (queueid oid, key text, value text),
 pg_resqueue_status_kv() t4 (queueid oid, key text, value text),
 pg_resqueue_status_kv() t5 (queueid oid, key text, value text),
 pg_resqueue_status_kv() t6 (queueid oid, key text, value text),
 pg_resqueue_status_kv() t7 (queueid oid, key text, value text),
 pg_resqueue_status_kv() t8 (queueid oid, key text, value text)
WHERE q.oid = t1.queueid
 AND t1.queueid = t2.queueid
 AND t2.queueid = t3.queueid
 AND t3.queueid = t4.queueid
 AND t4.queueid = t5.queueid
 AND t5.queueid = t6.queueid
 AND t6.queueid = t7.queueid
 AND t7.queueid = t8.queueid
 AND t1.key = 'rsqcountlimit'
 AND t2.key = 'rsqcountvalue'
 AND t3.key = 'rsqcostlimit'
 AND t4.key = 'rsqcostvalue'
 AND t5.key = 'rsqmemorylimit'
 AND t6.key = 'rsqmemoryvalue'
 AND t7.key = 'rsqwaiters'
 AND t8.key = 'rsqholders'
 AND q.rsqname = 'queue_name'
;
```

Compatibility

`CREATE RESOURCE QUEUE` is a HAWQ extension. There is no provision for resource queues or workload management in the SQL standard.

See Also

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

Related Links

SQL Command Reference

CREATE ROLE

Defines a new database role (user or group).

Synopsis

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

where *option* can be:

```
       SUPERUSER | NOSUPERUSER
     | CREATEDB | NOCREATEDB
     | CREATE-ROLE | NOCREATE-ROLE
     | 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 HAWQ 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 `CREATE-ROLE` 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 HAWQ 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.

CREATE-ROLE
NOCREATE-ROLE

If `CREATEDB` is specified, the role being defined will be allowed to create new roles, alter other roles, and drop other roles. `NOCREATE-ROLE` (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
```

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.

The `INHERIT` attribute is the default for reasons of backwards compatibility. In prior releases of HAWQ, 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 `CREATE-ROLE` privilege. There is no concept of inheritance for the privileges of a `CREATE-ROLE`-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 `CREATE-ROLE` privilege but not the `CREATEDB` privilege, it can create a new role with the `CREATEDB` privilege. Therefore, regard roles that have the `CREATE-ROLE` 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 CREATE-ROLE;
```

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 HAWQ, 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 HAWQ 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*

Related Links

SQL Command Reference

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`, and `GRANT` are accepted as clauses within `CREATE SCHEMA`. Other kinds of objects may be created in separate commands after the schema is created.

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 HAWQ.

The SQL standard specifies that the subcommands in `CREATE SCHEMA` may appear in any order. The present HAWQ 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. HAWQ 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

`DROP SCHEMA`

Related Links

SQL Command Reference

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, 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 HAWQ 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 HAWQ, 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 HAWQ 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 HAWQ:

- `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 HAWQ.

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 HAWQ (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 HAWQ extension.

See Also

`DROP SEQUENCE`

Related Links

SQL Command Reference

CREATE TABLE

Defines a new table.

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}] ...}
   [, ... ] ]
   [column_reference_storage_directive [, …] ]
   )
   [ 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 *storage_parameter* is:

```
   APPENDONLY={TRUE}
   BLOCKSIZE={8192-2097152}
   ORIENTATION={COLUMN | ROW | PARQUET}
   COMPRESSTYPE={ZLIB | QUICKLZ | SNAPPY | GZIP | NONE}
   COMPRESSLEVEL={0-9}
   FILLFACTOR={10-100}
   OIDS=[TRUE | FALSE]
   PAGESIZE={1024-1073741823}
   ROWGROUPSIZE={1024-1073741823}
```

where *column_constraint* is:

```
   [CONSTRAINT constraint_name]
   NOT NULL | NULL
   | CHECK ( expression )
```

and *table_constraint* is:

```
   [CONSTRAINT constraint_name]
   | CHECK ( expression )
```

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 [, ... ] ) ]
[column_reference_storage_directive [, …] ]
[ 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 [, ... ] ) ]
[column_reference_storage_directive [, …] ]
[ TABLESPACE tablespace ]
```

where *storage_parameter* is:

```
   APPENDONLY={TRUE}
   BLOCKSIZE={8192-2097152}
   ORIENTATION={COLUMN | ROW | PARQUET}
   COMPRESSTYPE={ZLIB | QUICKLZ | NONE}
   COMPRESSLEVEL={1-9}
   FILLFACTOR={10-100}
   OIDS=[TRUE|FALSE]
   PAGESIZE={1024-1073741823}
   ROWGROUPSIZE={1024-1073741823}
```

where *storage_directive* is:

```
   COMPRESSTYPE={ZLIB | QUICKLZ | NONE}
 | COMPRESSLEVEL={0-9}
 | BLOCKSIZE={8192-2097152}
```

where *column_reference_storage_directive* is:

```
   COLUMN column_name ENCODING (storage_directive [, ... ] ), ...
 |
   DEFAULT COLUMN ENCODING (storage_directive [, ... ] )
```

Description

`CREATE TABLE` creates a new, initially empty table in the current database. The table is owned by the user issuing the command. If a schema name is given then the table is created in the specified schema. Otherwise it is created in the current schema. Temporary tables exist in a special schema, so a schema name may not be given when creating a temporary table. The name of the table must be distinct from the name of any other table, external table, sequence, 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.

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 HAWQ distribution policy. If a `DISTRIBUTED BY` or `DISTRIBUTED RANDOMLY` clause is not supplied, then HAWQ assigns a hash distribution policy to the table using the first column of the table as the distribution key. Columns of geometric or user-defined data types are not eligible as HAWQ 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 HAWQ 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, HAWQ 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 planner to determine which table partitions to scan in order to satisfy a given query predicate. These partition constraints are managed automatically by HAWQ.

Parameters

GLOBAL | LOCAL

These keywords are present for SQL standard compatibility, but have no effect in HAWQ.

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.

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.

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 HAWQ, 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-only 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.

NULL | NOT NULL

Specifies if the column is or is not allowed to contain null values. `NULL` is the default.

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.

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.

Note: You cannot create a table with both column encodings and compression parameters in a WITH clause.

The following storage options are available:

APPENDONLY — Set to `TRUE` to create the table as an append-only table. If `FALSE` is specified, an error message displays stating that heap tables are not supported.

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, or parquet. 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-only 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. This option is valid only if `APPENDONLY=TRUE`.

OIDS — Set to `OIDS=FALSE` (the default) so that rows do not have object identifiers assigned to them. Pivotal strongly recommends that you do not enable OIDS when creating a table. On large tables, such as those in a typical HAWQ 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 HAWQ 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-only 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 dfs_default is used. Creating table on tablespace `pg_default` is not allowed.

`DISTRIBUTED BY (`*`column`*`, [ ... ] )`
`DISTRIBUTED RANDOMLY`

Used to declare the HAWQ 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. If not supplied, then hash distribution is chosen using the first eligible column of the table as the distribution key.

PARTITION BY

Declares one or more columns by which to partition the 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-only 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. This subpartition specification would then apply to all parent partitions.

Notes

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.

HAWQ has some special conditions for primary key and unique constraints with regards to columns that are the *distribution key* in a HAWQ table. For a unique constraint to be enforced in HAWQ, 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.

Primary key and foreign key constraints are not supported in HAWQ. For inherited tables, table privileges *are not* inherited in the current implementation.

HAWQ also supports the parquet storage format.

Setting Parameters for Parquet Tables:

You can set three kinds of parameters for a parquet table.

1. Set the parquet orientation parameter:

```
with (appendonly=true, orientation=parquet);
```

2. Set the compression type parameter: Parquet tables can compressed either as `SNAPPY` or `GZIP`. When setting to `SNAPPY`, setting a compression level causes it to fail. You can set the `GZIP` compression level between 1 - 9. If you specify a compression level but not a compression type when creating parquet table, the compression type defaults to `GZIP`.

   Note: The compress types `ZLIB` and `QUICKLZ` are valid for AO/CO tables, while `SNAPPY` and `GZIP` are valid for parquet tables.

3. Set the data storage parameter: By default, the two parameters, `PAGESIZE` and `ROWGROUPSIZE` are set to 1MB/8MB for common and partitioned tables.

   Note: The page size should be less than the rowgroup size. This is because rowgroup includes the metadata information of a single page even for a single column table. The parameters `PAGESIZE` and `ROWGROUPSIZE` are valid for parquet tables, while `BLOCKSIZE` is valid for AO/CO tables

About Parquet Storage

DDL and DML: Most DDL and DML operations are valid for a parquet table. The usage for DDL and DML operations is similar to AO tables. Valid operations on parquet tables include:

- parquet table creation (with/without partition, with/without compress type)
- insert and select

Compress type and level: You can only set the compression type at the table level. HAWQ does not support setting column level. The specified compression type is propagated to the columns. All the columns must have same compress type/level.

Data type: HAWQ supports all data types except arrays and User defined types.

Alter table: HAWQ does not support adding and dropping a new column to an existing parquet table. You can use `ALTER TABLE` for a partition operation.

FillFactor/OIDS/Checksum: HAWQ does not support these operations when creating parquet tables. The default value for checksum for a parquet table is false. You cannot set this value or specify fillfactor and oids.

Memory occupation: When inserting or loading data to a parquet table, the whole rowgroup is stored in physical memory. It is stored in physical memory until the size exceeds the threshold or the end of the `INSERT` operation. Once either occurs, the entire rowgroup is flushed to the disk.Also, at the beginning of the `INSERT` operation, each column is pre-allocated a page buffer. The column pre-allocated page buffer size should be `min(pageSizeLimit, rowgroupSizeLimit/estimatedColumnWidth/estimatedRecordWidth)` for the first rowgroup. For the following rowgroup, it should be `min(pageSizeLimit, actualColumnChunkSize in last rowgroup * 1.05)`, of which 1.05 is the estimated scaling factor. When reading data from a parquet table, the requested columns of the row group are loaded into memory. Memory is allocated 8 MB by default. Ensure that memory occupation not exceed physical memory when setting `ROWGROUPSIZE` or `PAGESIZE`, otherwise you may encounter OOM.

Parquet Examples

Parquet Example 1

```
CREATE TABLE parquet_wt_subpartgzip1(id SERIAL, int,char(5),numeric, boolean DEFAULT
 false, char DEFAULT 'd', text, timestamp, character varying(705),bigint, date,a11
 varchar(600),text, decimal, real, bigint, int4, bytea, timestamp with time zone,
 timetz, path, box, macaddr, interval, character varying(800), lseg, point, double
 precision, circle, int4, numeric(8), polygon, date, real, money, cidr, inet, time,
 text, bit, bit varying(5), smallint, int);
WITH (appendonly=true, orientation=parquet);
distributed randomly;
Partition by range(a1)
Subpartition by list(a2)
subpartition template (
default subpartition df_sp, subpartition sp1 values('M'),
subpartition sp2 values('F')
WITH (appendonly=true, orientation=parquet,compresstype=gzip,compresslevel=1))
(start(1) end(5000) every(1000) );
```

Parquet Example 2

```
CREATE TABLE parquet_wt_subpartgzip1_exch(
id SERIAL,int, char(5), numeric, boolean DEFAULT false, char DEFAULT 'd', text,
 timestamp, character varying(705), bigint, date,a11 varchar(600), text, decimal,
 real, bigint, int4 , bytea, timestamp with time zone, timetz, path, box, macaddr,
 interval, character varying(800), lseg, point, double precision, circle, int4,
 numeric(8), polygon, date, real, money, cidr, inet, time, text, bit, bit varying(5),
 smallint, int)
WITH (appendonly=true, orientation=parquet, compresstype=gzip)
distributed randomly;
```

Parquet Example 3

```
Alter table parquet_wt_subpartgzip1 alter partition FOR (
RANK(1))
exchange partition sp1 with table parquet_wt_subpartgzip1_exch;\d+ parquet_wt_
subpartgzip1_1_prt_1_2_prt_sp1
```

AO Examples

AO Example 1: 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);
```

AO Example 2: Create table films and table distributors (the first column will be used as the HAWQ distribution key by default):

```
CREATE TABLE films (
code char(5),
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,
name varchar(40) NOT NULL CHECK (name <> '')
);
```

AO Example 3: Create a gzip-compressed, append-only table:

```
CREATE TABLE sales (txn_id int, qty int, date date)
WITH (appendonly=true, compresslevel=5)
DISTRIBUTED BY (txn_id);
```

AO Example 4: 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. HAWQ 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 HAWQ. HAWQ 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 HAWQ 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. HAWQ does not enforce this restriction; it treats column and table check constraints alike.
- **NULL Constraint** — The `NULL` constraint is a HAWQ 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 HAWQ 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 HAWQ.
- **Partitioning** — Table partitioning via the `PARTITION BY` clause is a HAWQ language extension.
- **Zero-column tables** — HAWQ 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 Pivotal decided to ignore this spec restriction.

- **WITH clause** — The `WITH` clause is a Pivotal extension; neither storage parameters nor OIDs are in the standard.
- **Tablespaces** — The HAWQ concept of tablespaces is not part of the SQL standard. The clauses `TABLESPACE` and `USING INDEX TABLESPACE` are extensions.
- **Data Distribution** — The HAWQ 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`*

Related Links

SQL Command Reference

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}
   BLOCKSIZE={8192-2097152}
   ORIENTATION={COLUMN | ROW | PARQUET}
   COMPRESSTYPE={ZLIB | QUICKLZ RLE_TYPE | SNAPPY | GZIP | NONE}
   COMPRESSLEVEL={1-9 | 1}
   FILLFACTOR={10-100}
   OIDS=[TRUE | FALSE]
   PAGESIZE={1024-1073741823}
   ROWGROUPSIZE={1024-1073741823}
```

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 HAWQ.

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-only table. If `FALSE`, an error message displays stating that heap tables are not supported.

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, or parquet. 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-only 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`.

OIDS — Set to `OIDS=FALSE` (the default) so that rows do not have object identifiers assigned to them. Pivotal strongly recommends that you do not enable OIDS when creating a table. On large tables, such as those in a typical HAWQ 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 HAWQ 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`* command, or an *`EXECUTE`* command that runs a prepared `SELECT` query.

`DISTRIBUTED BY (`*`column`*`, [ ... ] )`
`DISTRIBUTED RANDOMLY`

Used to declare the HAWQ distribution policy for the table. One or more columns can be used as the distribution key, meaning those columns are used by the hashing algorithm to divide the data evenly across all of the segments. 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 to distribute randomly, which will send the data round-robin to the segment instances. If not supplied, then either the `PRIMARY KEY` (if the table has one) or the first eligible column of the table will be used.

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 HAWQ, these parentheses are optional.
- The standard defines a `WITH [NO] DATA` clause; this is not currently implemented by HAWQ. The behavior provided by HAWQ is equivalent to the standard's `WITH DATA` case. `WITH NO DATA` can be simulated by appending `LIMIT 0` to the query.
- HAWQ handles temporary tables differently from the standard; see `CREATE TABLE` for details.
- The `WITH` clause is a HAWQ extension; neither storage parameters nor `OIDs` are in the standard.
- The HAWQ concept of tablespaces is not part of the standard. The `TABLESPACE` clause is an extension.

See Also

CREATE EXTERNAL TABLE , *EXECUTE* , *SELECT* , *SELECT INTO*

Related Links

SQL Command Reference

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 HAWQ 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`* or *`CREATE TABLE`* to have the data files for these objects stored within the specified tablespace.

In HAWQ, 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 HAWQ 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` for more information.

Parameters

tablespacename

The name of a tablespace to be created. The name cannot begin with `pg_`, 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 *filespace_name*

The name of a HAWQ filespace that was defined using the `gpfilespace` management utility.

Notes

You must first create a filespace to be used by the tablespace. See `gpfilespace` 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 HAWQ extension.

See Also

`CREATE DATABASE` , *`CREATE TABLE`* , *`DROP TABLESPACE`*

Related Links

SQL Command Reference

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. This is the only form currently supported by HAWQ. 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 HAWQ, 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 HAWQ. 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, HAWQ 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 HAWQ 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, HAWQ 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

`CREATE TYPE` command is a HAWQ 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`*

Related Links

SQL Command Reference

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
     | CREATE-ROLE | NOCREATE-ROLE
     | 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

HAWQ does not support `CREATE USER`. This command has been replaced by *`CREATE ROLE`* .

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

`CREATE ROLE`

Related Links

SQL Command Reference

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`* command which will provide the columns and rows of the view.

Notes

Views in HAWQ 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, if you run the following command:

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

The result is poor: 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 the following command:

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

Check that you have permission to access the tables referenced in the view. View ownership determines permissions, not your status as current user. This is true, even if you are a superuser. This concept is unusual, since superusers typically have access to all objects. In the case of views, even superusers must be explicitly granted access to tables referenced if they do not own 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 HAWQ. 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 HAWQ language extension. So is the concept of a temporary view.

See Also

`SELECT` , *`DROP VIEW`*

Related Links

SQL Command Reference

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*

Related Links

SQL Command Reference

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.

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 HAWQ, 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 HAWQ, 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.

query

A *SELECT* command which will provide the rows to be returned by the cursor.

FOR READ ONLY

`FOR READ ONLY` indicates that the cursor is used in a read-only mode. Cursors can only be used in a read-only mode in HAWQ. HAWQ does not support updatable cursors (FOR UPDATE), so this is the default behavior.

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 HAWQ 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.

Scrollable cursors are not currently supported in HAWQ. You can only use `FETCH` to move the cursor position forward, not backwards.

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. HAWQ permits cursors to be used interactively.

HAWQ 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 HAWQ cursors are forward moving only (not scrollable).

Binary cursors are a HAWQ extension.

See Also

CLOSE , *FETCH* , *SELECT*

Related Links

SQL Command Reference

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`*

Related Links

SQL Command Reference

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

`CREATE DATABASE`

Related Links

SQL Command Reference

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`

Related Links

SQL Command Reference

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

`DROP TABLESPACE` , `gpfilespace`

Related Links

SQL Command Reference

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

ALTER FUNCTION , *CREATE FUNCTION*

Related Links

SQL Command Reference

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`

Related Links

SQL Command Reference

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*

Related Links

SQL Command Reference

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.

Notes

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.

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` , *`CREATE OPERATOR`* *`CREATE OPERATOR CLASS`*

Related Links

SQL Command Reference

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 HAWQ extension.

See Also

REASSIGN OWNED , *DROP ROLE*

Related Links

SQL Command Reference

DROP RESOURCE QUEUE

Removes a resource queue.

Synopsis

```
DROP RESOURCE QUEUE queue_name
```

Description

This command removes a workload management resource queue from HAWQ. 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 HAWQ extension.

See Also

CREATE RESOURCE QUEUE , *ALTER ROLE*

Related Links

SQL Command Reference

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 `CREATE-ROLE` 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 HAWQ uses.

See Also

ALTER ROLE , *CREATE ROLE* , *DROP OWNED* , *REASSIGN OWNED* , *SET ROLE*

Related Links

SQL Command Reference

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 HAWQ extension.

See Also

CREATE SCHEMA

Related Links

SQL Command Reference

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 HAWQ extension.

See Also

`CREATE SEQUENCE`

Related Links

SQL Command Reference

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 HAWQ extension.

See Also

`CREATE TABLE` , *`ALTER TABLE`* , *`TRUNCATE`*

Related Links

SQL Command Reference

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 HAWQ extension.

See Also

`CREATE TABLESPACE` , *`ALTER TABLESPACE`*

Related Links

SQL Command Reference

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 HAWQ extension. But note that the `CREATE TYPE` command and the data type extension mechanisms in HAWQ differ from the SQL standard.

See Also

`ALTER TYPE` , *`CREATE TYPE`*

Related Links

SQL Command Reference

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`

Related Links

SQL Command Reference

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 HAWQ extension.

See Also

`CREATE VIEW`

Related Links

SQL Command Reference

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 HAWQ 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 HAWQ extension that provides functionality equivalent to *`COMMIT`* , which is specified in the SQL standard.

See Also

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

Related Links

SQL Command Reference

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*

Related Links

SQL Command Reference

EXPLAIN

Shows the query plan of a statement.

Synopsis

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

Description

`EXPLAIN` displays the query plan that the HAWQ 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. 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 HAWQ 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 planner 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 planner 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.

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 planner 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 HAWQ, 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 HAWQ, 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`

Related Links

SQL Command Reference

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 HAWQ, 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 HAWQ. 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

HAWQ 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 HAWQ.

`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. HAWQ 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 `FETCH ALL`, in which `FORWARD` is implicit, are HAWQ 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*

Related Links

SQL Command Reference

GRANT

Defines access privileges.

Synopsis

```
GRANT { {SELECT | INSERT | UPDATE | DELETE | REFERENCES |
TRIGGER } [,...] | 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 `CREATE-ROLE` 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 HAWQ. 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: HAWQ does not support triggers.

`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 HAWQ, 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.

HAWQ 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 HAWQ. The SQL standard does not support setting the privileges on more than one object per command.

HAWQ 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`, and `DELETE` privileges. This is not possible according to the SQL standard. HAWQ 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 HAWQ extensions.

See Also

REVOKE

Related Links

SQL Command Reference

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.

Note: HAWQ supports 127 concurrent inserts currently.

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.

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*

Related Links

SQL Command Reference

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. HAWQ 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. HAWQ 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 HAWQ has chosen for a prepared statement, use `EXPLAIN`.

For more information on query planning and the statistics collected by HAWQ 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`*

Related Links

SQL Command Reference

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 HAWQ extension.

See Also

`DROP OWNED` , *`DROP ROLE`*

Related Links

SQL Command Reference

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 HAWQ allows it to be omitted.

See Also

BEGIN , *SAVEPOINT* , *ROLLBACK TO SAVEPOINT* , *COMMIT*

Related Links

SQL Command Reference

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 *HAWQ Configuration Parameter Reference* for more information.

Parameters

configuration_parameter

The name of a system configuration parameter. See *HAWQ Configuration Parameter Reference* 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 HAWQ extension.

See Also

SET

Related Links

SQL Command Reference

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 HAWQ assumes `RESTRICT` by default.

See Also

GRANT

Related Links

SQL Command Reference

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`*

Related Links

SQL Command Reference

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 HAWQ (and Oracle) allow it to be omitted. SQL allows only `WORK`, not `TRANSACTION`, as a stopword after `ROLLBACK`. Also, SQL has an optional clause `AND [NO] CHAIN` which is not currently supported by HAWQ. Otherwise, this command conforms to the SQL standard.

See Also

BEGIN , *COMMIT* , *SAVEPOINT* , *RELEASE SAVEPOINT* , *ROLLBACK*

Related Links

SQL Command Reference

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 HAWQ, 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`*

Related Links

SQL Command Reference

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]
```

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.

You must have `SELECT` privilege on a table to read its values.

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. There are a number of constructs that can be classified as an expression but do not follow any general syntax rules.

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 HAWQ.

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.

HAWQ 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 currentrow (`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 HAWQ 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.

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 planner 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.

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

HAWQ allows you 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 HAWQ 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. HAWQ extends each of these clauses to allow the other choice as well (but it uses the standard's interpretation if there is ambiguity). HAWQ 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, HAWQ 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 HAWQ, 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 Also

`EXPLAIN`

Related Links

SQL Command Reference

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 HAWQ 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`*

Related Links

SQL Command Reference

SET

Changes the value of a HAWQ 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`. See *HAWQ Server Configuration Parameters*. `SET` only affects the value used by the current session.

If `SET` or `SET SESSION` is issued within a transaction that is later aborted, the effects of the `SET` command disappear when the transaction is rolled back. 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` only last 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.

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 HAWQ configuration parameter. Only parameters classified as *session* can be changed with `SET`. See *HAWQ Server Configuration Parameters*.

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).

Examples

Set the schema search path:

```
SET search_path TO my_schema, public;
```

Increase work memory to 200 MB:

```
SET work_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 HAWQ allows more flexible time-zone specifications. All other `SET` features are HAWQ extensions.

See Also

`RESET` , *`SHOW`*

Related Links

SQL Command Reference

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

HAWQ 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; HAWQ does not make this restriction. The `SESSION` and `LOCAL` modifiers are a HAWQ extension, as is the `RESET` syntax.

See Also

`SET SESSION AUTHORIZATION`

Related Links

SQL Command Reference

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. HAWQ allows identifier syntax (*rolename*), which SQL does not. SQL does not allow this command during a transaction; HAWQ does not make this restriction. The `SESSION` and `LOCAL` modifiers are a HAWQ extension, as is the `RESET` syntax.

See Also

`SET ROLE`

Related Links

SQL Command Reference

SHOW

Shows the value of a system configuration parameter.

Synopsis

```
SHOW configuration_parameter

SHOW ALL
```

Description

`SHOW` displays the current settings of HAWQ system configuration parameters. These parameters can be set using the `SET` statement, or by editing the postgresql.conf configuration file of the HAWQ master. Note that some parameters viewable by `SHOW` are read-only — their values can be viewed but not set. See *HAWQ Server Configuration Parameters*.

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 HAWQ extension.

See Also

SET , *RESET*

Related Links

SQL Command Reference

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.This is most useful on large tables.

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 HAWQ) it has no effect.

RESTRICT

Since this key word applies to foreign key references (which are not supported in HAWQ) it has no effect.

Notes

Only the owner of a table may `TRUNCATE` it. `TRUNCATE` will not perform the following:

- Run any user-defined `ON DELETE` triggers that might exist for the tables.
- Truncate any tables that inherit from the named table. Only the named table is truncated, not its child tables.

Examples

Empty the table `films`:

```
TRUNCATE films;
```

Compatibility

There is no `TRUNCATE` command in the SQL standard.

See Also

DROP TABLE

Related Links

SQL Command Reference

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 HAWQ 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 catalog tables. `VACUUM` has no effect on a normal HAWQ table, since the delete or update operations are not supported on normal HAWQ table.

With no parameter, `VACUUM` processes every table in the current database. With a parameter, `VACUUM` processes only that table. `VACUUM ANALYZE` performs a `VACUUM` and then an `ANALYZE` for each selected table. This is a handy combination form for routine maintenance scripts. See *`ANALYZE`* for more details about its processing.

Plain `VACUUM` (without `FULL`) simply reclaims space and makes it available for re-use. This form of the command can operate in parallel with normal reading and writing of the table, as an exclusive lock is not obtained. `VACUUM FULL` does more extensive processing, including moving of tuples across blocks to try to compact the table to the minimum number of disk blocks. This form is much slower and requires an exclusive lock on each table while it is being processed.

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.

Note: A VACUUM FULL is not recommended in HAWQ. See *Notes*.

FREEZE

Specifying `FREEZE` is equivalent to performing `VACUUM` with the `vacuum_freeze_min_age` server configuration parameter set to zero. The `FREEZE` option is deprecated and will be removed in a future release. Set the parameter in the master `postgresql.conf` file instead.

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, it may be a good idea to issue a `VACUUM ANALYZE` command for the affected table. This will update the system catalogs with the results of all recent changes, and allow the HAWQ query planner to make better choices in planning queries.

`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. The `auto vacuum` daemon feature, that automates the execution of `VACUUM` and `ANALYZE` commands is currently disabled in HAWQ.

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 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 FULL` will reclaim all expired row space, but is a very expensive operation and may take an unacceptably long time to finish on large, distributed HAWQ tables. If you do get into a situation where the free space map has overflowed, it may be more timely to recreate the table with a `CREATE TABLE AS` statement and drop the old table.

`VACUUM FULL` is not recommended in HAWQ. It is best to size the free space map appropriately. The free space map is configured with the following server configuration parameters:

- `max_fsm_pages`
- `max_fsm_relations`

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 planner:

```
VACUUM ANALYZE;
```

Compatibility

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

See Also

ANALYZE

Related Links

SQL Command Reference

Chapter 13

Management Utility Reference

This section provides references for the command-line management utilities provided with HAWQ. HAWQ provides the standard client and server programs, and additional management utilities to administer a distributed HAWQ system.

The HAWQ management utilities are located in `$GPHOME/bin`.

Backend Server Programs

The following server programs handle the parallelism and distribution of a HAWQ system. Users and administrators do not access these programs directly, but do so through the HAWQ management tools and utilities.

Program Name	Description	Alternative
initdb	This program is called by gpinitsystem when initializing a HAWQ array. It is used internally to create the individual segment instances and the master instance.	gpinitsystem
ipclean	Cleans up shared memory and semaphores from aborted backend servers.	N/A
gypsyncmaster	This is the HAWQ program that starts the gpsyncagent process on the standby master host. Administrators do not call this program directly, but do so through the management scripts that initialize and/or activate a standby master for a HAWQ system. This process is responsible for keeping the standby master up to date with the primary master via a transactionlog replication process.	gpinitstandby gpactivatestandby
pg_controldata	Displays control information of the database cluster.	gpstate
pg_ctl	This program is called by gpstart and gpstop when starting or stopping a HAWQ array. It is used internally to stop and start the individual segment instances and the master instance in parallel and with the correct options.	gpstartgpstop

Program Name	Description	Alternative
pg_resetxlog	Resets the transaction log.	N/A
postgres	The postgres executable is the actual PostgreSQL server process that processes queries.	The main postgres process (postmaster) creates other postgres subprocesses and postgres sessions, as needed to handle client connections.
postmaster	postmaster starts the postgres database server listener process that accepts client connections. In HAWQ, a postgres database listener process runs on the HAWQ Master Instance and on each Segment Instance.	In HAWQ, you use gpstart and gpstop to start all postmasters (postgres processes) in the system at once, in the correct order and with the correct options.

gpactivatestandby
gpcheckperf
gpconfig
gpexpand
gpextract
gpfdist
gpfilespace
gpinitstandby
gpinitsystem
gpload
gplogfilter
gppkg
gpmigrator
gprecoverseg
gpscp
gpstart
gpssh
gpssh-exkeys
gpstate
gpstop

gpactivatestandby

Activates a standby master host and makes it the active master for the HAWQ system.

Synopsis

```
gpactivatestandby -d standby_master_datadir [-c new_standby_master] [-f] [-a] [-q]
    [-l logfile_directory]

gpactivatestandby -v

gpactivatestandby -? | -h | --help
```

Description

The `gpactivatestandby` utility activates a backup, standby master host and brings it into operation as the active master instance for a HAWQ system. The activated standby master effectively becomes the HAWQ master, accepting client connections on the master port.

The port number must be set to the same number on the master host and the backup master host. You must run this utility from the master host you want to activate and not from the host you need to disable. Running this utility assumes you have a backup master host configured for the system *gpinitstandby*.

The utility performs the following steps:

- Stops the synchronization process (`gpsyncagent`) on the backup master
- Updates the system catalog tables of the backup master using the logs.
- Activates the backup master to be the new active master for the system.
- (optional) Makes the host specified with the -c option the new standby master host.
- Restarts the HAWQ system with the new master host.

A backup HAWQ master host serves as a ‘warm standby’ in the event of the primary HAWQ master host becoming inoperable. The backup master is kept up to date by a transaction log replication process (gpsyncagent), which runs on the backup master host and keeps the data between the primary and backup master hosts synchronized.

If the primary master fails, the log replication process is shutdown, and the backup master can be activated in its place by using the gpactivatestandby utility. Upon activation of the backup master, the replicated logs are used to reconstruct the state of the HAWQ master host at the time of the last successfully committed transaction. To specify a new standby master host after making your current standby master active, use the -c option..

In order to use `gpactivatestandby` to activate a new primary master host, the master host that was previously serving as the primary master cannot be running. The utility checks for a `postmaster.pid` file in the data directory of the disabled master host, and if it finds it there, it will assume the old master host is still active. In some cases, you may need to remove the `postmaster.pid` file from the disabled master host data directory before running `gpactivatestandby` (for example, if the disabled master host process was terminated unexpectedly).

After activating a standby master, run `ANALYZE` to update the database query statistics. For example:

```
psql dbname -c 'ANALYZE;'
```

Options

`-a (do not prompt)`

Do not prompt the user for confirmation.

-c *new_standby_master_hostname*

Optional. After you activate your standby master, you may want to specify another host to be the new standby, otherwise your HAWQ system will no longer have a standby master configured. Use this option to specify the hostname of the new standby master host. You can also use gpinitstandby at a later time to configure a new standby master host.

-d *standby_master_datadir*

Required. The absolute path of the data directory for the master host you are activating.

-f (force activation)

Use this option to force activation of the backup master host when the synchronization process (gpsyncagent) is not running. Only use this option if you are sure that the backup and primary master hosts are consistent, and you know the gpsyncagent process is not running on the backup master host. This option may be useful if you have just initialized a new backup master using gpinitstandby, and want to activate it immediately.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-q (no screen output)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-v (show utility version)

Displays the version, status, last updated date, and check sum of this utility.

-? | -h | --help (help)

Displays the online help.

Example

Activate the backup master host and make it the active master instance for a HAWQ system (run from backup master host you are activating):

```
gpactivatestandby -d /gpdata
```

Activate the backup master host and at the same time configure another host to be your new standby master:

```
gpactivatestandby -d /gpdata -c new_standby_hostname
```

Related Links

Management Utility Reference

gpcheckperf

Verifies the baseline hardware performance of the specified hosts.

Synopsis

```
gpcheckperf -d test_directory [-d test_directory ...]
    {-f hostfile_gpcheckperf | - h hostname [-h hostname ...]}
    [-r ds] [-B block_size] [-S file_size] [-D] [-v|-V]

gpcheckperf -d temp_directory
    {-f hostfile_gpchecknet | - h hostname [-h hostname ...]}
    [ -r n|N|M [--duration time] [--netperf] ] [-D] [-v | -V]

gpcheckperf -?

gpcheckperf --version
```

Description

The `gpcheckperf` utility starts a session on the specified hosts and runs the following performance tests:

- **Disk I/O Test (dd test)** — To test the sequential throughput performance of a logical disk or file system, the utility uses the **dd** command, which is a standard UNIX utility. It times how long it takes to write and read a large file to and from disk and calculates your disk I/O performance in megabytes (MB) per second. By default, the file size that is used for the test is calculated at two times the total random access memory (RAM) on the host. This ensures that the test is truly testing disk I/O and not using the memory cache.
- **Memory Bandwidth Test (stream)** — To test memory bandwidth, the utility uses the STREAM benchmark program to measure sustainable memory bandwidth (in MB/s). This tests that your system is not limited in performance by the memory bandwidth of the system in relation to the computational performance of the CPU. In applications where the data set is large (as in HAWQ), low memory bandwidth is a major performance issue. If memory bandwidth is significantly lower than the theoretical bandwidth of the CPU, then it can cause the CPU to spend significant amounts of time waiting for data to arrive from system memory.
- **Network Performance Test (gpnetbench*)** — To test network performance including the HAWQ interconnect, the utility runs a network benchmark program that transfers a 5 second stream of data from the current host to each remote host included in the test. The data is transferred in parallel to each remote host and the minimum, maximum, average and median network transfer rates are reported in megabytes (MB) per second. If the summary transfer rate is slower than expected (less than 100 MB/s), you can run the network test serially using the `-r n` option to obtain per-host results. To run a full-matrix bandwidth test, you can specify `-r M` which will cause every host to send and receive data from every other host specified. This test is best used to validate if the switch fabric can tolerate a full-matrix workload.

To specify the hosts to test, use the `-f` option to specify a file containing a list of host names, or use the `-h` option to name single host names on the command-line. If running the network performance test, all entries in the host file must be for network interfaces within the same subnet. If your segment hosts have multiple network interfaces configured on different subnets, run the network test once for each subnet.

You must also specify at least one test directory (with `-d`). The user who runs `gpcheckperf` must have write access to the specified test directories on all remote hosts. For the disk I/O test, the test directories should correspond to your segment data directories (primary and/or mirrors). For the memory bandwidth and network tests, a temporary directory is required for the test program files.

Before using `gpcheckperf`, you must have a trusted host setup between the hosts involved in the performance test. You can use the utility `gpssh-exkeys` to update the known host files and exchange public keys between hosts if you have not done so already. Note that `gpcheckperf` calls to `gpssh` and `gpscp`, so these HAWQ utilities must also be in your `$PATH`.

Options

`-B` ***`block_size`***

Specifies the block size (in KB or MB) to use for disk I/O test. The default is 32KB, which is the same as the HAWQ page size. The maximum block size is 1 MB.

`-d` ***`test_directory`***

For the disk I/O test, specifies the file system directory locations to test. You must have write access to the test directory on all hosts involved in the performance test. You can use the `-d` option multiple times to specify multiple test directories (for example, to test disk I/O of your primary and mirror data directories).

`-d` ***`temp_directory`***

For the network and stream tests, specifies a single directory where the test program files will be copied for the duration of the test. You must have write access to this directory on all hosts involved in the test.

`-D (display per-host results)`

Reports performance results for each host for the disk I/O tests. The default is to report results for just the hosts with the minimum and maximum performance, as well as the total and average performance of all hosts.

`--duration` ***`time`***

Specifies the duration of the network test in seconds (s), minutes (m), hours (h), or days (d). The default is 15 seconds.

`-f` ***`hostfile_gpcheckperf`***

For the disk I/O and stream tests, specifies the name of a file that contains one host name per host that will participate in the performance test. The host name is required, and you can optionally specify an alternate user name and/or SSH port number per host. The syntax of the host file is one host per line as follows:

```
[username@]hostname[:ssh_port]
```

`-f` ***`hostfile_gpchecknet`***

For the network performance test, all entries in the host file must be for host adresses within the same subnet. If your segment hosts have multiple network interfaces configured on different subnets, run the network test once for each subnet. For example (a host file containing segment host address names for interconnect subnet 1):

```
sdw1-1
sdw2-1
sdw3-1
```

`-h` ***`hostname`***

Specifies a single host name (or host address) that will participate in the performance test. You can use the `-h` option multiple times to specify multiple host names.

`--netperf`

Specifies that the netperf binary should be used to perform the network test instead of performing the HAWQ network test. To use this option, you must download `netperf` from *http://www.netperf.org* and install it into `$GPHOME/bin/lib` on all HAWQ hosts (master and segments).

`-r ds{n|N|M}`

Specifies which performance tests to run. The default is `dsn`:

- Disk I/O test (`d`)
- Stream test (`s`)
- Network performance test in sequential (`n`), parallel (`N`), or full-matrix (`M`) mode. The optional `--duration` option specifies how long (in seconds) to run the network test. To use the parallel (`N`) mode, you must run the test on an *even* number of hosts.

  If you would rather use `netperf` (*http://www.netperf.org*) instead of the HAWQ network test, you can download it and install it into `$GPHOME/bin/lib` on all HAWQ hosts (master and segments). You would then specify the optional `--netperf` option to use the `netperf` binary instead of the default `gpnetbench*` utilities.

-S *file_size*

Specifies the total file size to be used for the disk I/O test for all directories specified with `-d`. *file_size* should equal two times total RAM on the host. If not specified, the default is calculated at two times the total RAM on the host where `gpcheckperf` is executed. This ensures that the test is truly testing disk I/O and not using the memory cache. You can specify sizing in KB, MB, or GB.

-v (verbose) | -V (very verbose)

Verbose mode shows progress and status messages of the performance tests as they are run. Very verbose mode shows all output messages generated by this utility.

--version

Displays the version of this utility.

-? (help)

Displays the online help.

Examples

Run the disk I/O and memory bandwidth tests on all the hosts in the file *host_file* using the test directory of */data1* and */data2*:

```
$ gpcheckperf -f hostfile_gpcheckperf -d /data1 -d /data2 -r ds
```

Run only the disk I/O test on the hosts named *sdw1* and *sdw2* using the test directory of */data1*. Show individual host results and run in verbose mode:

```
$ gpcheckperf -h sdw1 -h sdw2 -d /data1 -r d -D -v
```

Run the parallel network test using the test directory of */tmp,* where *hostfile_gpcheck_ic** specifies all network interface host address names within the same interconnect subnet:

```
$ gpcheckperf -f hostfile_gpchecknet_ic1 -r N -d /tmp
$ gpcheckperf -f hostfile_gpchecknet_ic2 -r N -d /tmp
```

Run the same test as above, but use `netperf` instead of the HAWQ network test (note that `netperf` must be installed in `$GPHOME/bin/lib` on all HAWQ hosts):

```
$ gpcheckperf -f hostfile_gpchecknet_ic1 -r N --netperf -d /tmp
$ gpcheckperf -f hostfile_gpchecknet_ic2 -r N --netperf -d /tmp
```

See Also

gpssh, *gpscp*

Related Links

Management Utility Reference

gpconfig

Sets server configuration parameters on all segments within a HAWQ system.

Synopsis

```
gpconfig -c param_name -v value [-m master_value | --masteronly]
       | -r param_name [--masteronly | -l
       [--skipvalidation] [--verbose] [--debug]

gpconfig -s param_name [--verbose] [--debug]

gpconfig --help
```

Description

The `gpconfig` utility allows you to set, unset, or view configuration parameters from the `postgresql.conf` files of all instances (master, segments, and mirrors) in your HAWQ system. When setting a parameter, you can also specify a different value for the master, if necessary. For example, parameters such as `max_connections` require a different setting on the master from what is used for the segments. If you want to set or unset a global or master only parameter, use the `--masteronly` option.

`gpconfig` cannot change the configuration parameters if there are failed segments in the system. gpconfig can only be used to manage certain parameters. For example, you cannot use it to set parameters such as `port`, which is required to be distinct for every segment instance.

Use the `-l (list)` option to see a complete list of configuration parameters supported by `gpconfig`. When `gpconfig` sets a configuration parameter in a segment postgresql.conf file, the new parameter setting always displays at the bottom of the file. When you use `gpconfig` to remove a configuration parameter setting, gpconfig comments out the parameter in all segment `postgresql.conf` files, thereby restoring the system default setting. For example, if you use `gpconfig` to remove (comment out) a parameter and later add it back (set a new value), there will be two instances of the parameter; one that is commented out, and one that is enabled and inserted at the bottom of the `postgresql.conf` file.

After setting a parameter, you must restart your HAWQ system or reload the postgresql.conf files for the change to take effect. Whether you require a restart or a reload depends on the parameter. See the *Server Configuration Parameters* reference for more information about the server configuration parameters.

To show the currently set values for a parameter across the system, use the `-s` option.

`gpconfig` uses the following environment variables to connect to the HAWQ master instance and obtain system configuration information:

- `PGHOST`
- `PGPORT`
- `PGUSER`
- `PGPASSWORD`
- `PGDATABASE`

Options

`-c | --change` ***`param_name`***

Changes a configuration parameter setting by adding the new setting to the bottom of the `postgresql.conf` files.

`-v | --value` ***`value`***

The value to use for the configuration parameter you specified with the `-c` option. By default, this value is applied to all segments, their mirrors, the master, and the standby master.

`-m | --mastervalue` ***`master_value`***

The master value to use for the configuration parameter you specified with the `-c` option. If specified, this value only applies to the master and standby master. This option can only be used with `-v`.

`--masteronly`

When specified, `gpconfig` will only edit the master `postgresql.conf` file.

`-r | --remove` ***`param_name`***

Removes a configuration parameter setting by commenting out the entry in the `postgresql.conf` files.

`-l | --list`

Lists all configuration parameters supported by the `gpconfig` utility.

`-s | --show` ***`param_name`***

Shows the value for a configuration parameter used on all instances (master and segments) in the HAWQ system. If there is a discrepancy in a parameter value between segment instances, the `gpconfig` utility displays an error message. Note that the `gpconfig` utility reads parameter values directly from the database, and not the `postgresql.conf` file. If you are using `gpconfig` to set configuration parameters across all segments, then running `gpconfig -s` to verify the changes, you might still see the previous (old) values. You must reload the configuration files (`gpstop -u`) or restart the system (`gpstop -r`) for changes to take effect.

`--skipvalidation`

Overrides the system validation checks of `gpconfig` and allows you to operate on any server configuration parameter, including hidden parameters and restricted parameters that cannot be changed by `gpconfig`. When used with the `-l` option (list), it shows the list of restricted parameters. This option should only be used to set parameters when directed by Pivotal Support.

`--verbose`

Displays additional log information during `gpconfig` command execution.

`--debug`

Sets logging output to debug level.

`-? | -h | --help`

Displays the online help.

Examples

Set the `max_connections` setting to 100 on all segments and 10 on the master:

```
gpconfig -c max_connections -v 100 -m 10
```

Comment out all instances of the `default_statistics_target` configuration parameter, and restore the system default:

```
gpconfig -r default_statistics_target
```

List all configuration parameters supported by `gpconfig`:

```
gpconfig -l
```

Show the values of a particular configuration parameter across the system:

```
gpconfig -s max_connections
```

See Also

gpstop

Related Links

Management Utility Reference

gpexpand

Expands an existing HAWQ database across new hosts in the array.

Synopsis

```
gpexpand [-f hosts_file]
        | -i input_file [-B batch_size] [-V]
        | {-d hh:mm:ss | -e 'YYYY-MM-DD hh:mm:ss'} [-analyze]
          [-n  parallel_processes]
        | --rollback
        | --clean
        [-D database_name] [--verbose] [--silent]

gpexpand -? | -h | --help

gpexpand --version
```

Prerequisites

- You are logged in as the HAWQ superuser (`gpadmin`).
- The new segment hosts have been installed and configured as per the existing segment hosts. This involves:
  - Configuring the hardware and OS
  - Installing the HAWQ software
  - Creating the `gpadmin` user account
  - Exchanging SSH keys.
- Enough disk space on your segment hosts to temporarily hold a copy of your largest table.

Description

The `gpexpand` utility performs system expansion in two phases: segment initialization and then table redistribution.

In the initialization phase, `gpexpand` runs with an input file that specifies data directories, *dbid* values, and other characteristics of the new segments. You can create the input file manually, or by following the prompts in an interactive interview.

If you choose to create the input file using the interactive interview, you can optionally specify a file containing a list of expansion hosts. If your platform or command shell limits the length of the list of hostnames that you can type when prompted in the interview, specifying the hosts with `-f` may be mandatory.

In addition to initializing the segments, the initialization phase performs these actions:

- Creates an expansion schema to store the status of the expansion operation, including detailed status for tables.
- Changes the distribution policy for all tables to `DISTRIBUTED RANDOMLY`. The original distribution policies are later restored in the redistribution phase.

To begin the redistribution phase, you must run `gpexpand` with either the `-d` (duration) or `-e` (end time) options. Until the specified end time or duration is reached, the utility will redistribute tables in the expansion schema. Each table is reorganized using `ALTER TABLE` commands to rebalance the tables across new segments, and to set tables to their original distribution policy. If `gpexpand` completes the reorganization of all tables before the specified duration, it displays a success message and ends.

Options

`-a | --analyze`

Run `ANALYZE` to update the table statistics after expansion. The default is to not run `ANALYZE`.

`-B` ***`batch_size`***

Batch size of remote commands to send to a given host before making a one-second pause. Default is `16`. Valid values are 1-128.

The `gpexpand` utility issues a number of setup commands that may exceed the host's maximum threshold for authenticated connections as defined by `MaxStartups` in the SSH daemon configuration. The one-second pause allows authentications to be completed before `gpexpand` issues any more commands.

The default value does not normally need to be changed. However, it may be necessary to reduce the maximum number of commands if `gpexpand` fails with connection errors such as `'ssh_exchange_identification: Connection closed by remote host.'`

`-c | --clean`

Remove the expansion schema.

`-d | --duration` ***`hh:mm:ss`***

Duration of the expansion session from beginning to end.

`-D` ***`database_name`***

Specifies the database in which to create the expansion schema and tables. If this option is not given, the setting for the environment variable `PGDATABASE` is used. The database templates *template1* and *template0* cannot be used.

`-e | --end '`*`YYYY-MM-DD hh:mm:ss`*`'`

Ending date and time for the expansion session.

`-f | --hosts-file` ***`filename`***

Specifies the name of a file that contains a list of new hosts for system expansion. Each line of the file must contain a single host name.

This file can contain hostnames with or without network interfaces specified. The `gpexpand` utility handles either case, adding interface numbers to end of the hostname if the original nodes are configured with multiple network interfaces.

`-i | --input` ***`input_file`***

Specifies the name of the expansion configuration file, which contains one line for each segment to be added in the format of:

hostname:address:port:fselocation:dbid:content:preferred_role:replication_port

If your system has filespaces, `gpexpand` will expect a filespace configuration file (*input_file_name*`.fs`) to exist in the same directory as your expansion configuration file. The filespace configuration file is in the format of:

```
filespaceOrder=filespace1_name:filespace2_name: ...
dbid:/path/for/filespace1:/path/for/filespace2: ...
dbid:/path/for/filespace1:/path/for/filespace2: ...
...
```

`-n` ***`parallel_processes`***

The number of tables to redistribute simultaneously. Valid values are 1 - 16.

Each table redistribution process requires two database connections: one to alter the table, and another to update the table's status in the expansion schema. Before increasing `-n`, check the current value of the server configuration parameter `max_connections` and make sure the maximum connection limit is not exceeded.

-r | --rollback

Roll back a failed expansion setup operation. If the rollback command fails, attempt again using the `-D` option to specify the database that contains the expansion schema for the operation that you want to roll back.

-s | --silent

Runs in silent mode. Does not prompt for confirmation to proceed on warnings.

-v | --verbose

Verbose debugging output. With this option, the utility will output all DDL and DML used to expand the database.

--version

Display the utility's version number and exit.

-V | --novacuum

Do not vacuum catalog tables before creating schema copy.

-? | -h | --help

Displays the online help.

Examples

Run `gpexpand` with an input file to initialize new segments and create the expansion schema in the default database:

```
$ gpexpand -i input_file
```

Run `gpexpand` for sixty hours maximum duration to redistribute tables to new segments:

```
$ gpexpand -d 60:00:00
```

See Also

gpssh-exkeys

Related Links

Management Utility Reference

gpextract

Extracts the metadata of a specified table into a YAML file.

Synopsis

```
gpextract [-h hostname] [-p port] [-U username] [-d database]
     [-o output_file] [-W] <tablename>

gpextract -?

gpextract --version
```

Description

`gpextract` is a utility that extracts a table's metadata into a YAML formatted file. HAWQ's InputFormat uses this YAML-formatted file to read a HAWQ file stored on HDFS directly into the MapReduce program.

Note: `gpextract` is bound by the following rules:

- You must start up HAWQ to use `gpextract`.
- `gpextract` only supports AO and Parquet tables.
- `gpextract` supports partitioned tables, but does not support sub-partitions.

Arguments

`<tablename>`

Name of the table that you need to extract metadata. You can use the format *namespace_name.table_name*.

Options

`-o` ***`output_file`***

Is the name of a file that `gpextract` uses to write the metadata. If you do not specify a name, `gpextract` writes to `stdout`.

`-v (verbose mode)`

Optional. Displays the verbose output of the extraction process.

`-? (help)`

Displays the online help.

`--version`

Displays the version of this utility.

Connection Options

`-d` ***`database`***

The database to connect to. If not specified, it reads from the environment variable `$PGDATABASE` or defaults to `template1`.

`-h` ***`hostname`***

Specifies the host name of the machine on which the HAWQ master database server is running. If not specified, it reads from the environment variable `$PGHOST` or defaults to `localhost`.

`-p` ***`port`***

Specifies the TCP port on which the HAWQ master database server is listening for connections. If not specified, reads from the environment variable `$PGPORT` or defaults to 5432.

`-U` *`username`*

The database role name to connect as. If not specified, reads from the environment variable `$PGUSER` or defaults to the current system user name.

`-W (force password prompt)`

Force a password prompt. If not specified, reads the password from the environment variable `$PGPASSWORD` or from a password file specified by `$PGPASSFILE` or in `~/.pgpass`.

Metadata File Format

`gpextract` exports the table metadata into a file using YAML 1.1 document format. The file contains various key information about the table, such as table schema, data file locations and sizes, partition constraints and so on.

The basic structure of the metadata file is as follows:

```
Version: string (1.0.0)
DBVersion: string
FileFormat: string (AO/Parquet)
TableName: string (schemaname.tablename)
DFS_URL: string (hdfs://127.0.0.1:9000)
Encoding: UTF8
AO_Schema:
 - name: string
    type: string

AO_FileLocations:
   Blocksize: int
    Checksum: boolean
   CompressionType: string
   CompressionLevel: int
   PartitionBy: string ('PARTITION BY ...')
   Files:
   - path: string (/gpseg0/16385/35469/35470.1)
   size: long

   Partitions:
   - Blocksize: int
     Checksum: Boolean
     CompressionType: string
     CompressionLevel: int
     Name: string
     Constraint: string (PARTITION Jan08 START (date '2008-01-01') INCLUSIVE)
     Files:
     - path: string
       size: long


Parquet_FileLocations:
  RowGroupSize: long
  PageSize: long
  CompressionType: string
  CompressionLevel: int
  Checksum: boolean
  EnableDictionary: boolean
  PartitionBy: string
  Files:
  - path: string
    size: long
  Partitions:
  - Name: string
    RowGroupSize: long
    PageSize: long
    CompressionType: string
    CompressionLevel: int
    Checksum: boolean
    EnableDictionary: boolean
    Constraint: string
    Files:
    - path: string
      size: long
```

Example: Extracting an AO table

Extract the `rank` table's metadata into a file named `rank_table.yaml`:

```
$ gpextract -o rank_table.yaml rank
```

Output content in rank_table.yaml

```
AO_FileLocations:
 Blocksize: 32768
 Checksum: false
 CompressionLevel: 0
 CompressionType: null
 Files:
 - path: /gpseg0/16385/35469/35692.1
   size: 0
 - path: /gpseg1/16385/35469/35692.1
   size: 0
 PartitionBy: PARTITION BY list (gender)
 Partitions:
 - Blocksize: 32768
   Checksum: false
   CompressionLevel: 0
   CompressionType: null
   Constraint: PARTITION girls VALUES('F') WITH (appendonly=true)
 Files:
 - path: /gpseg0/16385/35469/35697.1
   size: 0
 - path: /gpseg1/16385/35469/35697.1
   size: 0
   Name: girls
 - Blocksize: 32768
   Checksum: false
   CompressionLevel: 0
   CompressionType: null
   Constraint: PARTITION boys VALUES('M') WITH (appendonly=true)
   Files:
   - path: /gpseg0/16385/35469/35703.1
     size: 0
   - path: /gpseg1/16385/35469/35703.1
     size: 0
   Name: boys
 - Blocksize: 32768
   Checksum: false
   CompressionLevel: 0
   CompressionType: null
   Constraint: DEFAULT PARTITION other WITH appendonly=true)
   Files:
   - path: /gpseg0/16385/35469/35709.1
     size: 90071728
   - path: /gpseg1/16385/35469/35709.1
     size: 90071512
   Name: other
 AO_Schema:
 - name: id
   type: int4
 - name: rank
   type: int4
 - name: year
```

```
    type: int4
  - name: gender
    type: bpchar
  - name: count
     type: int4
  DFS_URL: hdfs://127.0.0.1:9000
  Encoding: UTF8
  FileFormat: AO
  TableName: public.rank
  Version: 1.0.0
```

Example: Extracting a Parquet table

Extract the `orders` table's metadata into a file named `orders.yaml`:

```
$ gpextract -o orders.yaml orders
```

Output content in orders.yaml

```
DFS_URL: hdfs://127.0.0.1:9000
Encoding: UTF8
FileFormat: Parquet
TableName: public.orders
Version: 1.0.0
Parquet_FileLocations:
  Checksum: false
  CompressionLevel: 0
  CompressionType: none
  EnableDictionary: false
  Files:
  - path: /hawq-data/gpseg0/16385/16626/16657.1
    size: 0
  - path: /hawq-data/gpseg1/16385/16626/16657.1
    size: 0
  PageSize: 1048576
  PartitionBy: PARTITION BY range (o_orderdate)
  Partitions:
  - Checksum: false
    CompressionLevel: 0
    CompressionType: none
    Constraint: PARTITION p1_1 START ('1992-01-01'::date) END ('1994-12-31'::date)
      EVERY ('3 years'::interval) WITH (appendonly=true, orientation=parquet,
 pagesize=1048576,
      rowgroupsize=8388608, compresstype=none, compresslevel=0)
    EnableDictionary: false
    Files:
    - path: /hawq-data/gpseg0/16385/16626/16662.1
      size: 8140599
    - path: /hawq-data/gpseg1/16385/16626/16662.1
      size: 8099760
    Name: orders_1_prt_p1_1
    PageSize: 1048576
    RowGroupSize: 8388608
  - Checksum: false
    CompressionLevel: 0
    CompressionType: none
    Constraint: PARTITION p1_11 START ('1995-01-01'::date) END ('1997-12-31'::date)
      EVERY ('e years'::interval) WITH (appendonly=true, orientation=parquet,
 pagesize=1048576,
      rowgroupsize=8388608, compresstype=none, compresslevel=0)
    EnableDictionary: false
    Files:
    - path: /hawq-data/gpseg0/16385/16626/16668.1
      size: 8088559
    - path: /hawq-data/gpseg1/16385/16626/16668.1
      size: 8075056
    Name: orders_1_prt_p1_11
    PageSize: 1048576
    RowGroupSize: 8388608
  - Checksum: false
    CompressionLevel: 0
    CompressionType: none
    Constraint: PARTITION p1_21 START ('1998-01-01'::date) END ('2000-12-31'::date)
      EVERY ('3 years'::interval) WITH (appendonly=true, orientation=parquet,
 pagesize=1048576,
      rowgroupsize=8388608, compresstype=none, compresslevel=0)
    EnableDictionary: false
    Files:
    - path: /hawq-data/gpseg0/16385/16626/16674.1
      size: 8065770
    - path: /hawq-data/gpseg1/16385/16626/16674.1
      size: 8126669
    Name: orders_1_prt_p1_21
    PageSize: 1048576
    RowGroupSize: 8388608
  RowGroupSize: 8388608
```

See Also

gpload

Related Links

Management Utility Reference

gpfdist

Serves data files to or writes data files out from HAWQ segments.

Synopsis

```
gpfdist [-d directory] [-p http_port] [-l log_file] [-t timeout]
   [-S] [-v | -V] [-m max_length] [--ssl certificate_path]

gpfdist -?

gpfdist --version
```

Description

`gpfdist` is HAWQ's parallel file distribution program. It is used by readable external tables and `gpload` to serve external table files to all HAWQ segments in parallel. It is used by writable external tables to accept output streams from HAWQ segments in parallel and write them out to a file.

In order for `gpfdist` to be used by an external table, the `LOCATION` clause of the external table definition must specify the external table data using the `gpfdist://` protocol (see *CREATE EXTERNAL TABLE*).

> **Note:** If the `--ssl` option is specified to enable SSL security, create the external table with the `gpfdists://` protocol.

The benefit of using `gpfdist` is that you are guaranteed maximum parallelism while reading from or writing to external tables, thereby offering the best performance as well as easier administration of external tables.

For readable external tables, `gpfdist` parses and serves data files evenly to all the segment instances in the HAWQ system when users `SELECT` from the external table.

For writable external tables, `gpfdist` accepts parallel output streams from the segments when users `INSERT` into the external table, and writes to an output file.

For readable external tables, if load files are compressed using `gzip` or `bzip2` (have a `.gz` or `.bz2` file extension), `gpfdist` uncompresses the files automatically before loading provided that `gunzip` or `bunzip2` is in your path.

> **Note:** Currently, readable external tables do not support compression on Windows platforms, and writable external tables do not support compression on any platforms.

Most likely, you will want to run `gpfdist` on your ETL machines rather than the hosts where HAWQ is installed. To install `gpfdist` on another host, simply copy the utility over to that host and add `gpfdist` to your `$PATH`. You can also run `gpfdist` as a Windows Service. See *Running gpfdist as a Windows Service* for more details.

Options

`-d` ***`directory`***

The directory from which `gpfdist` will serve files for readable external tables or create output files for writable external tables. If not specified, defaults to the current directory.

`-l` ***`log_file`***

The fully qualified path and log file name where standard output messages are to be logged.

`-p` ***`http_port`***

The HTTP port on which `gpfdist` will serve files. Defaults to 8080.

`-t` ***`timeout`***

Sets the time allowed for HAWQ to establish a connection to a `gpfdist` process. Default is 5 seconds. Allowed values are 2 to 30 seconds. May need to be increased on systems with large amounts of network traffic.

-m *max_length*

Sets the maximum allowed data row length in bytes. Default is 32768. Should be used when user data includes very wide rows (or when `line too long` error message occurs). Should not be used otherwise as it increases resource allocation. Valid range is 32K to 256MB. (The upper limit is 1MB on Windows systems.)

-S (use O_SYNC)

Opens the file for synchronous I/O with the `O_SYNC` flag. Any writes to the resulting file descriptor block `gpfdist` until the data is physically written to the underlying hardware.

--ssl *certificate_path*

Adds SSL encryption to data transferred with `gpfdist`. After executing `gpfdist` with the `--ssl` *`certificate_path`* option, the only way to load data from this file server is with the `gpfdist://` protocol.

The location specified in *certificate_path* must contain the following files:

- The server certificate file, `server.crt`
- The server private key file, `server.key`
- The trusted certificate authorities, `root.crt`

The root directory (`/`) cannot be specified as *certificate_path*.

-v (verbose)

Verbose mode shows progress and status messages.

-V (very verbose)

Verbose mode shows all output messages generated by this utility.

-? (help)

Displays the online help.

--version

Displays the version of this utility.

Running gpfdist as a Windows Service

HAWQ Loaders allow `gpfdist` to run as a Windows Service.

Follow the instructions below to download, register and activate `gpfdist` as a service:

1. Update your HAWQ Loader package to the latest version. This package is available from *Pivotal Network*.
2. Register `gpfdist` as a Windows service:
   a. Open a Windows command window
   b. Run the following command:

   ```
   sc create gpfdist binpath= "path_to_gpfdist.exe -p 8081 -d External\load\files
   \path -l Log\file\path"
   ```

   You can create multiple instances of `gpfdist` by running the same command again, with a unique name and port number for each instance:

   ```
   sc create gpfdistN binpath= "path_to_gpfdist.exe -p 8082 -d External\load\files
   \path -l Log\file\path"
   ```
3. Activate the `gpfdist` service:

a. Open the Windows Control Panel and select **Administrative Tools > Services**.
b. Highlight then right-click on the `gpfdist` service in the list of services.
c. Select **Properties** from the right-click menu, the Service Properties window opens.

   Note that you can also stop this service from the Service Properties window.
d. Optional: Change the **Startup Type** to **Automatic** (after a system restart, this service will be running), then under **Service** status, click **Start**.
e. Click **OK**.

Repeat the above steps for each instance of `gpfdist` that you created.

Examples

Serve files from a specified directory using port 8081 (and start `gpfdist` in the background):

```
gpfdist -d /var/load_files -p 8081 &
```

Start `gpfdist` in the background and redirect output and errors to a log file:

```
gpfdist -d /var/load_files -p 8081 -l /home/gpadmin/log &
```

To stop `gpfdist` when it is running in the background:

--First find its process id:

```
ps ax | grep gpfdist
```

OR on Solaris

```
ps -ef | grep gpfdist
```

--Then kill the process, for example:

```
kill 3456
```

See Also

`gpload`, *CREATE EXTERNAL TABLE*

Related Links

Management Utility Reference

gpfilespace

Creates a filespace using a configuration file that defines per-segment file system locations. Filespaces describe the physical file system resources to be used by a tablespace.

Synopsis

```
gpfilespace [connection_option ...] [-l logfile_directory]
             [-o output_file_name]

gpfilespace [connection_option ...] [-l logfile_directory]
             [-c fs_config_file]

gpfilespace -v | -?
```

Description

A tablespace requires a file system location to store its database files. In HAWQ, 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 HAWQ system is referred to as a *filespace*. Once a filespace is defined, it can be used by one or more tablespaces.

When used with the `-o` option, the `gpfilespace` utility looks up your system configuration information in the HAWQ catalog tables and prompts you for the appropriate file system locations needed to create the filespace. It then outputs a configuration file that can be used to create a filespace. If a file name is not specified, a `gpfilespace_config_#` file will be created in the current directory by default.

Once you have a configuration file, you can run `gpfilespace` with the `-c` option to create the filespace in HAWQ.

Options

`-c | --config` ***`fs_config_file`***

A configuration file containing:

- An initial line denoting the new filespace name. For example:

  filespace:*myfs*
- One line each for the master, the primary segments, and the mirror segments. A line describes a file system location that a particular segment database instance should use as its data directory location to store database files associated with a tablespace. Each line is in the format of:

```
hostname:dbid:/filesystem_dir/seg_datadir_name
```

`-l | --logdir` ***`logfile_directory`***

The directory to write the log file. Defaults to `~/gpAdminLogs`.

`-o | --output` ***`output_file_name`***

The directory location and file name to output the generated filespace configuration file. You will be prompted to enter a name for the filespace, a master file system location, the primary segment file system locations, and the mirror segment file system locations. For example, if your configuration has 2 primary and 2 mirror segments per host, you will be prompted for a total of 5 locations (including the master). The file system locations must exist on all hosts in your system prior to running the `gpfilespace` utility. The utility will designate segment-specific data directories within the location(s) you specify, so it is possible to use the same location for multiple segments. However, primaries and mirrors cannot use the same location. After the utility creates the configuration file, you can manually edit the file to make any required changes to the filespace layout before creating the filespace in HAWQ.

-v | --version (show utility version)

Displays the version of this utility.

-? | --help (help)

Displays the utility usage and syntax.

Connection Options

-h host | --host *host*

The host name of the machine on which the HAWQ master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to localhost.

-p port | --port *port*

The TCP port on which the HAWQ master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U username | --username *superuser_name*

The database superuser role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system user name. Only database superusers are allowed to create filespaces.

-W | --password

Force a password prompt.

Note: `gpfilespace`, `showfilespace`, `showtempfilespace`, `movetransfilespace`, `showtransfilespace`, and `movetempfilespace` are not supported.

Examples

Create a filespace configuration file. You will be prompted to enter a name for the filespace, choose a file system name, file replica number, and a DFS URL for store data.

```
$ gpfilespace -o .
Enter a name for this filespace
> example_hdfs
Available filesystem name:
filesystem: hdfs
Choose filesystem name for this filespace

> hdfs

Enter replica num for filespace. If 0, default replica num is used (default=3):
>3
Checking your configuration:
Your system has 1 hosts with 2 segments per host.
Configuring hosts: [sdw1]
Please specify the DFS location for the segments (for
example: localhost:9000/fs)
location> 127.0.0.1:9000/hdfs
```

Example filespace configuration file:

```
filespace:fastdisk
mdw:1:/gp_master_filespc/gp-1
sdw1:2:/gp_pri_filespc/gp0
sdw1:3:/gp_mir_filespc/gp1
sdw2:4:/gp_mir_filespc/gp0
sdw2:5:/gp_pri_filespc/gp1
```

Execute the configuration file to create the filespace in HAWQ:

```
$ gpfilespace -c gpfilespace_config_1
```

Related Links

Management Utility Reference

gpinitstandby

Adds and/or initializes a standby master host for a HAWQ system.

Synopsis

```
gpinitstandby
      { -s standby_hostname | -r | -n }
      [-M smart | -M fast] [-a] [-q] [-D] [-L]
      [-l logfile_directory]

gpinitstandby -? | -v
```

Description

The `gpinitstandby` utility adds a backup, standby master host to your HAWQ system. If your system has an existing standby master host configured, use the `-r` option to remove it before adding the new standby master host.

Before running this utility, make sure that the HAWQ software is installed on the standby master host and that you have exchanged SSH keys between the hosts. It is recommended that the master port is set to the same port number on the master host and the backup master host.

This utility should be run on the currently active *primary* master host. See the *HAWQ Installation and Upgrade* for instructions.

The utility performs the following steps:

- Shut down your HAWQ system.
- Update the HAWQ system catalog to remove the existing backup master host information (if the -r option is supplied).
- Update the HAWQ system catalog to add the new backup master host information (use the -n option to skip this step).
- Edit the pg_hba.conf files of the segment instances to allow access from the newly added standby master.
- Set up the backup master instance on the alternate master host.
- Start the synchronization process.
- Restart your HAWQ system.

A backup master host serves as a ‘warm standby’ in the event of the primary master host becoming inoperable. The backup master is kept up-to-date by a transaction log replication process (`gpsyncagent`), which runs on the backup master host and keeps the data between the primary and backup master hosts synchronized. If the primary master fails, the log replication process is shut down, and the backup master can be activated in its place by using the utility. Upon activation of the backup master, the replicated logs are used to reconstruct the state of the master host at the time of the last successfully committed transaction.

The activated standby master effectively becomes the HAWQ master, accepting client connections on the master port and performing normal master operations such as SQL command processing and workload management.

Options

`-a (do not prompt)`

Do not prompt the user for confirmation.

`-D (debug)`

Sets logging level to debug.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-L (leave database stopped)

Leave HAWQ in a stopped state after removing the warm standby master.

-M fast (fast shutdown - rollback)

Use fast shut down when stopping HAWQ at the beginning of the standby initialization process. Any transactions in progress are interrupted and rolled back.

-M smart (smart shutdown - warn)

Use smart shut down when stopping HAWQ at the beginning of the standby initialization process. If there are active connections, this command fails with a warning. This is the default shutdown mode.

-n (resynchronize)

Use this option if you already have a standby master configured, and just want to resynchronize the data between the primary and backup master host. The HAWQ system catalog tables will not be updated.

-q (no screen output)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-r (remove standby master)

Removes the currently configured standby master host from your HAWQ system.

-s standby_hostname

The host name of the standby master host.

-v (show utility version)

Displays the version, status, last updated date, and check sum of this utility.

-? (help)

Displays the online help.

Examples

Add a backup master host to your HAWQ system and start the synchronization process:

```
gpinitstandby -s host09
```

Remove the existing standby master from your HAWQ system configuration:

```
gpinitstandby -r
```

Start an existing backup master host and synchronize the data with the current primary master host. Do not add a new HAWQ backup master host to the system catalog.

```
gpinitstandby -n
```

Note: Do not specify the -n and -s options in the same command.

Related Links

Management Utility Reference

gpinitsystem

Initializes a HAWQ system using configuration parameters specified in the `gpinitsystem_config` file.

Synopsis

```
gpinitsystem -c gpinitsystem_config
      [-h hostfile_gpinitsystem]
      [-B parallel_processes]
      [-p postgresql_conf_param_file]
      [-s standby_master_host]
      [--max_connections=number] [--shared_buffers=size]
      [--locale=locale] [--lc-collate=locale]
      [--lc-ctype=locale] [--lc-messages=locale]
      [--lc-monetary=locale] [--lc-numeric=locale]
      [--lc-time=locale] [--su_password=password]
            [-a] [-q] [-l logfile_directory] [-D]

gpinitsystem -?

gpinitsystem -v
```

Description

The `gpinitsystem` utility will create a HAWQ instance using the values defined in a configuration file. See *gpinitsystem* for more information about this configuration file. Before running this utility, make sure that you have installed the HAWQ software on all the hosts in the array.

In a HAWQ system, each database instance (the master and all segments) must be initialized across all of the hosts in the system in such a way that they can all work together as a unified DBMS. The `gpinitsystem` utility takes care of initializing the HAWQ master and each segment instance, and configuring the system as a whole.

Before running `gpinitsystem`, you must set the `$GPHOME` environment variable to point to the location of your HAWQ installation on the master host and exchange SSH keys between all host addresses in the array using `gpssh-exkeys`.

This utility performs the following tasks:

- Verifies that the parameters in the configuration file are correct.
- Ensures that a connection can be established to each host address. If a host address cannot be reached, the utility will exit.
- Verifies the locale settings.
- Displays the configuration that will be used and prompts the user for confirmation.
- Initializes the master instance.
- Initializes the standby master instance (if specified).
- Initializes the primary segment instances.
- Configures the HAWQ system and checks for errors.
- Starts the HAWQ system.

Options

`-a (do not prompt)`

Do not prompt the user for confirmation.

`-B` ***`parallel_processes`***

The number of segments to create in parallel. If not specified, the utility will start up to 4 parallel processes at a time.

`-c` ***`gpinitsystem_config`***

Required. The full path and filename of the configuration file, which contains all of the defined parameters to configure and initialize a new HAWQ system. See ***gpinitsystem*** for a description of this file.

`-D (debug)`

Sets log output level to debug.

`-h` ***`hostfile_gpinitsystem`***

Optional. The full path and filename of a file that contains the host addresses of your segment hosts. If not specified on the command line, you can specify the host file using the `MACHINE_LIST_FILE` parameter in the `gpinitsystem_config` file.

`-l` ***`logfile_directory`***

The directory to write the log file. Defaults to `~/gpAdminLogs`.

`--locale=`*`locale`*` | -n `*`locale`*

Sets the default locale used by HAWQ. If not specified, the `LC_ALL`, `LC_COLLATE`, or `LANG` environment variable of the master host determines the locale. If these are not set, the default locale is C (POSIX). A locale identifier consists of a language identifier and a region identifier, and optionally a character set encoding. For example, `sv_SE` is Swedish as spoken in Sweden, `en_US` is U.S. English, and `fr_CA` is French Canadian. If more than one character set can be useful for a locale, then the specifications look like this: `en_US.UTF-8` (locale specification and character set encoding). On most systems, the command `locale` will show the locale environment settings and `locale -a` will show a list of all available locales.

`--lc-collate=`*`locale`*

Similar to `--locale`, but sets the locale used for collation (sorting data). The sort order cannot be changed after HAWQ is initialized, so it is important to choose a collation locale that is compatible with the character set encodings that you plan to use for your data. There is a special collation name of C or POSIX (byte-order sorting as opposed to dictionary-order sorting). The C collation can be used with any character encoding.

`--lc-ctype=`*`locale`*

Similar to `--locale`, but sets the locale used for character classification (what character sequences are valid and how they are interpreted). This cannot be changed after HAWQ is initialized, so it is important to choose a character classification locale that is compatible with the data you plan to store in HAWQ.

`--lc-messages=`*`locale`*

Similar to `--locale`, but sets the locale used for messages output by HAWQ. The current version of HAWQ does not support multiple locales for output messages (all messages are in English), so changing this setting will not have any effect.

`--lc-monetary=`*`locale`*

Similar to `--locale`, but sets the locale used for formatting currency amounts.

`--lc-numeric=`*`locale`*

Similar to `--locale`, but sets the locale used for formatting numbers.

`--lc-time=`*`locale`*

Similar to `--locale`, but sets the locale used for formatting dates and times.

`--max_connections=`*`number`*` | -m `*`number`*

Sets the maximum number of client connections allowed to the master. The default is 250.

`-p` ***`postgresql_conf_param_file`***

Optional. The name of a file that contains `postgresql.conf` parameter settings that you want to set for HAWQ. These settings will be used when the individual master and segment instances are initialized. You can also set parameters after initialization using the `gpconfig` utility.

`-q (no screen output)`

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

`--shared_buffers=`*`size`*` | -b `*`size`*

Sets the amount of memory a HAWQ server instance uses for shared memory buffers. You can specify sizing in kilobytes (kB), megabytes (MB) or gigabytes (GB). The default is 125MB.

`-s `*`standby_master_host`*

Optional. If you wish to configure a backup master host, specify the host name using this option. The HAWQ software must already be installed and configured on this host.

`--su_password=`*`superuser_password`*` | -e `*`superuser_password`*

Use this option to specify the password to set for the HAWQ superuser account (such as `gpadmin`). If this option is not specified, the default password `gparray` is assigned to the superuser account. You can use the `ALTER ROLE` command to change the password at a later time.

Recommended security best practices:

- Do not use the default password option for production environments.
- Change the password immediately after installation.

`-? (help)`

Displays the online help.

`-v (show utility version)`

Displays the version of this utility.

Initialization Configuration File Format

`gpinitsystem` requires a configuration file with the following parameters defined. An example initialization configuration file can be found in `$GPHOME/docs/cli_help/gpconfigs/gpinitsystem_config`.

`ARRAY_NAME`

Required. A name for the array you are configuring. You can use any name you like. Enclose the name in quotes if the name contains spaces.

`MACHINE_LIST_FILE`

Optional. Can be used in place of the `-h` option. This specifies the file that contains the list of segment host address names that comprise the HAWQ system. The master host is assumed to be the host from which you are running the utility and should not be included in this file. If your segment hosts have multiple network interfaces, then this file would include all addresses for the host. Give the absolute path to the file.

`SEG_PREFIX`

Required. This specifies a prefix that will be used to name the data directories on the master and segment instances. The naming convention for data directories in a HAWQ system is *SEG_PREFIXnumber* where *number* starts with 0 for segment instances (the master is always -1). So for example, if you choose the prefix `gpseg`, your master instance data directory would be named `gpseg-1`, and the segment instances would be named `gpseg0`, `gpseg1`, `gpseg2`, `gpseg3`, and so on.

`PORT_BASE`

Required. This specifies the base number by which primary segment port numbers are calculated. The first primary segment port on a host is set as `PORT_BASE`, and then incremented by one for each additional primary segment on that host. Valid values range from 1 through 65535.

`DATA_DIRECTORY`

Required. This specifies the data storage location(s) where the utility will create the primary segment data directories. The number of locations in the list dictate the number of primary segments that will get created per physical host (if multiple addresses for a host are listed in the host file, the number of segments will be spread evenly across the specified interface addresses). It is OK to list the same data storage area multiple times if you want your data directories created in the same location. The user who runs `gpinitsystem` (for example, the `gpadmin` user) must have permission to write to these directories. For example, this will create six primary segments per host:

```
declare -a DATA_DIRECTORY=(/data1/primary /data1/primary
/data1/primary /data2/primary /data2/primary /data2/primary)
```

`MASTER_HOSTNAME`

Required. The host name of the master instance. This host name must exactly match the configured host name of the machine (run the `hostname` command to determine the correct hostname).

`MASTER_DIRECTORY`

Required. This specifies the location where the data directory will be created on the master host. You must make sure that the user who runs `gpinitsystem` (for example, the `gpadmin` user) has permissions to write to this directory.

`MASTER_PORT`

Required. The port number for the master instance. This is the port number that users and client connections will use when accessing the HAWQ system.

`TRUSTED_SHELL`

Required. The shell the `gpinitsystem` utility uses to execute commands on remote hosts. Allowed values are `ssh`. You must set up your trusted host environment before running the `gpinitsystem` utility (you can use `gpssh-exkeys` to do this).

`CHECK_POINT_SEGMENTS`

Required. Maximum distance between automatic write ahead log (WAL) checkpoints, in log file segments (each segment is normally 16 megabytes). This will set the `checkpoint_segments` parameter in the `postgresql.conf` file for each segment instance in the HAWQ system.

`ENCODING`

Required. The character set encoding to use. This character set must be compatible with the `--locale` settings used, especially `--lc-collate` and `--lc-ctype`. HAWQ supports the same character sets as PostgreSQL.

`DATABASE_NAME`

Optional. The name of a HAWQ database to create after the system is initialized. You can always create a database later using the `CREATE DATABASE` command or the `createdb` utility.

`MIRROR_PORT_BASE`

Optional. This specifies the base number by which mirror segment port numbers are calculated. The first mirror segment port on a host is set as `MIRROR_PORT_BASE`, and then incremented by one for each additional mirror segment on that host. Valid values range from 1 through 65535 and cannot conflict with the ports calculated by `PORT_BASE`.

`REPLICATION_PORT_BASE`

Optional. This specifies the base number by which the port numbers for the primary file replication process are calculated. The first replication port on a host is set as `REPLICATION_PORT_BASE`, and then incremented by one for each additional primary segment on that host. Valid values range from 1 through 65535 and cannot conflict with the ports calculated by `PORT_BASE` or `MIRROR_PORT_BASE`.

`MIRROR_REPLICATION_PORT_BASE`

Optional. This specifies the base number by which the port numbers for the mirror file replication process are calculated. The first mirror replication port on a host is set as `MIRROR_REPLICATION_PORT_BASE`, and then incremented by one for each additional mirror segment on that host. Valid values range from 1 through 65535 and cannot conflict with the ports calculated by `PORT_BASE`, `MIRROR_PORT_BASE`, or `REPLICATION_PORT_BASE`.

`MIRROR_DATA_DIRECTORY`

Optional. This specifies the data storage location(s) where the utility will create the mirror segment data directories. There must be the same number of data directories declared for mirror segment instances as for primary segment instances (see the `DATA_DIRECTORY` parameter). The user who runs `gpinitsystem` (for example, the `gpadmin` user) must have permission to write to these directories. For example:

```
declare -a MIRROR_DATA_DIRECTORY=(/data1/mirror
/data1/mirror /data1/mirror /data2/mirror /data2/mirror
/data2/mirror)
```

Examples

Initialize a HAWQ array by supplying a configuration file and a segment host address file:

```
$ gpinitsystem -c gpinitsystem_config -h hostfile_gpinitsystem -S
```

Initialize a HAWQ array and set the superuser remote password:

```
$ gpinitsystem -c gpinitsystem_config -h hostfile_gpinitsystem --su-password=
mypassword
```

Initialize a HAWQ array with an optional standby master host:

```
$ gpinitsystem -c gpinitsystem_config -h hostfile_gpinitsystem -s host09
```

See Also

`gpssh-exkeys`

Related Links

Management Utility Reference

gpload

Runs a load job as defined in a YAML formatted control file.

Synopsis

```
gpload -f control_file [-l log_file] [-h hostname] [-p port]
   [-U username] [-d database] [-W] [--gpfdist_timeout seconds]
   [--no_auto_trans] [[-v | -V] [-q]] [-D]

gpload -?

gpload --version
```

Prerequisites

The client machine where `gpload` is executed must have the following:

- Python 2.6.2 or later, `pygresql` (the Python interface to PostgreSQL), and `pyyaml`. Note that Python and the required Python libraries are included with the HAWQ server installation, so if you have HAWQ installed on the machine where `gpload` is running, you do not need a separate Python installation.

  Note: HAWQ Loaders for Windows support only Python 2.5 (available from *www.python.org*).
- The *`gpfdist`* parallel file distribution program must be installed and in your `$PATH`. This program is located in `$GPHOME/bin` of your HAWQ server installation.
- Network access to and from all hosts in your HAWQ array (master and segments).
- Network access to and from the hosts where the data to be loaded resides (ETL servers).

Description

`gpload` is a data loading utility that acts as an interface to HAWQ's external table parallel loading feature. Using a load specification defined in a YAML formatted control file, gpload executes a load by invoking the HAWQ parallel file server (gpfdist), creating an external table definition based on the source data defined, and executing an `INSERT` operation to load the source data into the target table in the database.

Note: `UPDATE` or `MERGE` operations are not supported at this time.

Options

`-f` ***`control_file`***

Required. A YAML file that contains the load specification details. See *Control File Format*.

`--gpfdist_timeout` ***`seconds`***

Sets the timeout for the `gpfdist` parallel file distribution program to send a response. Enter a value from `0` to `30` seconds (entering "`0`" to disables timeouts). Note that you might need to increase this value when operating on high-traffic networks.

`-l` ***`log_file`***

Specifies where to write the log file. Defaults to `~/gpAdminLogs/gpload_`*`YYYYMMDD`*. For more information about the log file, see *Log File Format*.

`-q (no screen output)`

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

`-D (debug mode)`

Check for error conditions, but do not execute the load.

`-v (verbose mode)`

Show verbose output of the load steps as they are executed.

`-V (very verbose mode)`

Shows very verbose output.

`-? (show help)`

Show help, then exit.

`--version`

Show the version of this utility, then exit.

Connection Options

`-d` ***`database`***

The database to load into. If not specified, reads from the load control file, the environment variable `$PGDATABASE` or defaults to the current system user name.

`-h` ***`hostname`***

Specifies the host name of the machine on which the HAWQ master database server is running. If not specified, reads from the load control file, the environment variable `$PGHOST` or defaults to `localhost`.

`-p` ***`port`***

Specifies the TCP port on which the HAWQ master database server is listening for connections. If not specified, reads from the load control file, the environment variable `$PGPORT` or defaults to 5432.

`-U` ***`username`***

The database role name to connect as. If not specified, reads from the load control file, the environment variable `$PGUSER` or defaults to the current system user name.

`-W (force password prompt)`

Force a password prompt. If not specified, reads the password from the environment variable `$PGPASSWORD` or from a password file specified by `$PGPASSFILE` or in `~/.pgpass`. If these are not set, then `gpload` will prompt for a password even if `-W` is not supplied.

Control File Format

The `gpload` control file uses the *YAML 1.1* document format and then implements its own schema for defining the various steps of a HAWQ load operation. The control file must be a valid YAML document.

The `gpload` program processes the control file document in order and uses indentation (spaces) to determine the document hierarchy and the relationships of the sections 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.

The basic structure of a load control file is:

```
---
VERSION: 1.0.0.1
DATABASE: db_name
USER: db_username
HOST: master_hostname
PORT: master_port
GPLOAD:
   INPUT:
    - SOURCE:
         LOCAL_HOSTNAME:
           - hostname_or_ip
         PORT: http_port
       | PORT_RANGE: [start_port_range, end_port_range]
```

```
          FILE:
            - /path/to/input_file
          SSL: true | false
          CERTIFICATES_PATH: /path/to/certificates
     - COLUMNS:
            - field_name: data_type
     - TRANSFORM: 'transformation'
     - TRANSFORM_CONFIG: 'configuration-file-path'
     - MAX_LINE_LENGTH: integer
     - FORMAT: text | csv
     - DELIMITER: 'delimiter_character'
     - ESCAPE: 'escape_character' | 'OFF'
     - NULL_AS: 'null_string'
     - FORCE_NOT_NULL: true | false
     - QUOTE: 'csv_quote_character'
     - HEADER: true | false
     - ENCODING: database_encoding
    OUTPUT:
     - TABLE: schema.table_name
     - MODE: insert
     - MAPPING:
              target_column_name: source_column_name | 'expression'
    PRELOAD:
     - TRUNCATE: true | false
     - REUSE_TABLES: true | false
    SQL:
     - BEFORE: "sql_command"
     - AFTER: "sql_command"
```

VERSION

Optional. The version of the `gpload` control file schema. The current version is 1.0.0.1.

DATABASE

Optional. Specifies which database in HAWQ to connect to. If not specified, defaults to `$PGDATABASE` if set or the current system user name. You can also specify the database on the command line using the `-d` option.

USER

Optional. Specifies which database role to use to connect. If not specified, defaults to the current user or `$PGUSER` if set. You can also specify the database role on the command line using the `-U` option.

If the user running `gpload` is not a HAWQ superuser, then the server configuration parameter `gp_external_grant_privileges` must be set to `on` in order for the load to be processed.

HOST

Optional. Specifies HAWQ master host name. If not specified, defaults to localhost or `$PGHOST` if set. You can also specify the master host name on the command line using the `-h` option.

PORT

Optional. Specifies HAWQ master port. If not specified, defaults to 5432 or `$PGPORT` if set. You can also specify the master port on the command line using the `-p` option.

GPLOAD

Required. Begins the load specification section. A `GPLOAD` specification must have an `INPUT` and an `OUTPUT` section defined.

INPUT

Required. Defines the location and the format of the input data to be loaded. `gpload` will start one or more instances of the *`gpfdist`* file distribution program on the current host and create the required external table definition(s) in HAWQ that point to the source data. Note that the host from which you run `gpload` must be accessible over the network by all HAWQ hosts (master and segments).

SOURCE

Required. The `SOURCE` block of an `INPUT` specification defines the location of a source file. An `INPUT` section can have more than one `SOURCE` block defined. Each `SOURCE` block defined corresponds to one instance of the *`gpfdist`* file distribution program that will be started on the local machine. Each `SOURCE` block defined must have a `FILE` specification.

LOCAL_HOSTNAME

Optional. Specifies the host name or IP address of the local machine on which `gpload` is running. If this machine is configured with multiple network interface cards (NICs), you can specify the host name or IP of each individual NIC to allow network traffic to use all NICs simultaneously. The default is to use the local machine's primary host name or IP only.

PORT

Optional. Specifies the specific port number that the *`gpfdist`* file distribution program should use. You can also supply a `PORT_RANGE` to select an available port from the specified range. If both `PORT` and `PORT_RANGE` are defined, then `PORT` takes precedence. If neither `PORT` or `PORT_RANGE` are defined, the default is to select an available port between 8000 and 9000.

If multiple host names are declared in `LOCAL_HOSTNAME`, this port number is used for all hosts. This configuration is desired if you want to use all NICs to load the same file or set of files in a given directory location.

PORT_RANGE

Optional. Can be used instead of `PORT` to supply a range of port numbers from which `gpload` can choose an available port for this instance of the *`gpfdist`* file distribution program.

FILE

Required. Specifies the location of a file, named pipe, or directory location on the local file system that contains data to be loaded. You can declare more than one file so long as the data is of the same format in all files specified.

If the files are compressed using `gzip` or `bzip2` (have a `.gz` or `.bz2` file extension), the files will be uncompressed automatically (provided that `gunzip` or `bunzip2` is in your path).

When specifying which source files to load, you can use the wildcard character (`*`) or other C-style pattern matching to denote multiple files. The files specified are assumed to be relative to the current directory from which `gpload` is executed (or you can declare an absolute path).

SSL

Optional. Specifies usage of SSL encryption.

CERTIFICATES_PATH

Required when SSL is `true`; cannot be specified when SSL is `false` or unspecified. The location specified in `CERTIFICATES_PATH` must contain the following files:

- The server certificate file, `server.crt`
- The server private key file, `server.key`
- The trusted certificate authorities, `root.crt`

The root directory (`/`) cannot be specified as `CERTIFICATES_PATH`.

COLUMNS

Optional. Specifies the schema of the source data file(s) in the format of *`field_name`*`:`*`data_type`*. The `DELIMITER` character in the source file is what separates two data value fields (columns). A row is determined by a line feed character (`0x0a`).

If the input `COLUMNS` are not specified, then the schema of the output `TABLE` is implied, meaning that the source data must have the same column order, number of columns, and data format as the target table.

The default source-to-target mapping is based on a match of column names as defined in this section and the column names in the target `TABLE`. This default mapping can be overridden using the `MAPPING` section.

TRANSFORM

Optional. Specifies the name of the input XML transformation passed to `gpload`.

TRANSFORM_CONFIG

Optional. Specifies the location of the XML transformation configuration file that is specified in the `TRANSFORM` parameter, above.

MAX_LINE_LENGTH

Optional. An integer that specifies the maximum length of a line in the XML transformation data passed to `gpload`.

FORMAT

Optional. Specifies the format of the source data file(s) - either plain text (`TEXT`) or comma separated values (`CSV`) format. Defaults to `TEXT` if not specified.

DELIMITER

Optional. 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. You can also specify a non-printable ASCII character via an escape sequence using the Unicode representation of the ASCII character. For example, `"\014"` represents the shift out character.

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.

NULL_AS

Optional. 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 nulls from empty strings. Any source data item that matches this string will be considered a null value.

FORCE_NOT_NULL

Optional. 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.

QUOTE

Required when `FORMAT` is `CSV`. Specifies the quotation character for `CSV` mode. The default is double-quote (`"`).

HEADER

Optional. Specifies that the first line in the data file(s) is a header row (contains the names of the columns) and should not be included as data to be loaded. If using multiple data source files, all files must have a header row. The default is to assume that the input files do not have a header row.

ENCODING

Optional. Character set encoding of the source data. Specify a string constant (such as `'SQL_ASCII'`), an integer encoding number, or `'DEFAULT'` to use the default client encoding. If not specified, the default client encoding is used.

ERROR_LIMIT

Optional. Enables single row error isolation mode for this load operation. When enabled, input rows that have format errors will be discarded provided that the error limit count is not reached on any HAWQ segment instance during input processing. If the error limit is not reached, all good rows will be loaded and any error rows will either be discarded or logged to the table specified in `ERROR_TABLE`. The default is to abort the load operation on the first error encountered. Note that single row error isolation only applies to data rows with format errors; for example, extra or missing attributes, attributes of a wrong data type, or invalid client encoding sequences. Constraint errors, such as primary key violations, will still cause the load operation to abort if encountered.

ERROR_TABLE

Optional when `ERROR_LIMIT` is declared. Specifies an error table where rows with formatting errors will be logged when running in single row error isolation mode. You can then examine this error table to see error rows that were not loaded (if any). If the `ERROR_TABLE` specified already exists, it will be used. If it does not exist, it will be automatically generated.

OUTPUT

Required. Defines the target table and final data column values that are to be loaded into the database.

TABLE

Required. The name of the target table to load into.

MODE

Optional. Currently, the only mode is INSERT.

INSERT - Loads data into the target table using the following method:

```
INSERT INTO target_table SELECT * FROM input_data;
```

MAPPING

Optional. If a mapping is specified, it overrides the default source-to-target column mapping. The default source-to-target mapping is based on a match of column names as defined in the source `COLUMNS` section and the column names of the target `TABLE`. A mapping is specified as either:

`target_column_name``:` *`source_column_name`*

or

`target_column_name``: '`*`expression`*`'`

Where *expression* is any expression that you would specify in the `SELECT` list of a query, such as a constant value, a column reference, an operator invocation, a function call, and so on.

PRELOAD

Optional. Specifies operations to run prior to the load operation. Right now the only preload operation is `TRUNCATE`.

TRUNCATE

Optional. If set to true, `gpload` will remove all rows in the target table prior to loading it.

REUSE_TABLES

Optional. If set to true, `gpload` will not drop the external table objects and staging table objects it creates. These objects will be reused for future load operations that use the same load specifications. This improves performance of trickle loads (ongoing small loads to the same target table).

SQL

Optional. Defines SQL commands to run before and/or after the load operation. You can specify multiple `BEFORE` and/or `AFTER` commands. List commands in the order of desired execution.

BEFORE

Optional. An SQL command to run before the load operation starts. Enclose commands in quotes.

AFTER

Optional. An SQL command to run after the load operation completes. Enclose commands in quotes.

Log File Format

Log files output by `gpload` have the following format:

```
timestamp|level|message
```

Where *timestamp* takes the form: `YYYY-MM-DD HH:MM:SS`, *level* is one of `DEBUG`, `LOG`, `INFO`, `ERROR`, and *message* is a normal text message.

Some `INFO` messages that may be of interest in the log files are (where # corresponds to the actual number of seconds, units of data, or failed rows):

```
INFO|running time: #.## seconds
INFO|transferred #.# kB of #.# kB.
INFO|gpload succeeded
INFO|gpload succeeded with warnings
INFO|gpload failed
INFO|1 bad row
INFO|# bad rows
```

Notes

If your database object names were created using a double-quoted identifier (delimited identifier), you must specify the delimited name within single quotes in the `gpload` control file. For example, if you create a table as follows:

```
CREATE TABLE "MyTable" ("MyColumn" text);
```

Your YAML-formatted `gpload` control file would refer to the above table and column names as follows:

```
- COLUMNS:
   - '"MyColumn"': text
OUTPUT:
   - TABLE: public.'"MyTable"'
```

Examples

Run a load job as defined in `my_load.yml`:

```
gpload -f my_load.yml
```

Example load control file:

```
---
VERSION: 1.0.0.1
DATABASE: ops
USER: gpadmin
HOST: mdw-1
PORT: 5432
GPLOAD:
 INPUT:
   - SOURCE:
  LOCAL_HOSTNAME:
   - etl1-1
   - etl1-2
   - etl1-3
   - etl1-4
  PORT: 8081
  FILE:
    - /var/load/data/*
   - COLUMNS:
   - name: text
   - amount: float4
   - category: text
   - desc: text
   - date: date
   - FORMAT: text
   - DELIMITER: '|'
 OUTPUT:
   - TABLE: payables.expenses
   - MODE: INSERT
  SQL:
   - BEFORE: "INSERT INTO audit VALUES('start',current_timestamp)"
   - AFTER: "INSERT INTO audit VALUES('end',current_timestamp)"
```

See Also

gpfdist, *CREATE EXTERNAL TABLE*

Related Links

Management Utility Reference

gplogfilter

Searches through HAWQ log files for specified entries.

Synopsis

```
gplogfilter [timestamp_options] [pattern_options]
     [output_options] [input_options] [input_file]

gplogfilter --help

gplogfilter --version
```

Description

The `gplogfilter` utility can be used to search through a HAWQ log file for entries matching the specified criteria. If an input file is not supplied, then `gplogfilter` will use the `$MASTER_DATA_DIRECTORY` environment variable to locate the HAWQ master log file in the standard logging location. To read from standard input, use a dash (`-`) as the input file name. Input files may be compressed using `gzip`. In an input file, a log entry is identified by its timestamp in `YYYY-MM-DD [hh:mm[:ss]]` format.

You can also use `gplogfilter` to search through all segment log files at once by running it through the *`gpssh`* utility. For example, to display the last three lines of each segment log file:

```
gpssh -f seg_host_file
=> source /usr/local/greenplum-db/greenplum_path.sh
=> gplogfilter -n 3 /gpdata/*/pg_log/gpdb*.csv
```

By default, the output of `gplogfilter` is sent to standard output. Use the `-o` option to send the output to a file or a directory. If you supply an output file name ending in `.gz`, the output file will be compressed by default using maximum compression. If the output destination is a directory, the output file is given the same name as the input file.

Options

Timestamp Options

`-b` *`datetime`* `| --begin=`*`datetime`*

Specifies a starting date and time to begin searching for log entries in the format of `YYYY-MM-DD [hh:mm[:ss]]`.

`-e` *`datetime`* `| --end=`*`datetime`*

Specifies an ending date and time to stop searching for log entries in the format of `YYYY-MM-DD [hh:mm[:ss]]`.

`-d` *`time`* `| --duration=`*`time`*

Specifies a time duration to search for log entries in the format of `[hh][:mm[:ss]]`. If used without either the `-b` or `-e` option, will use the current time as a basis.

Pattern Matching Options

`-c i [gnore] | r [espect] | --case=i [gnore] | r [espect]`

Matching of alphabetic characters is case sensitive by default unless proceeded by the `--case=ignore` option.

`-C '`*`string`*`' | --columns='`*`string`*`'`

Selects specific columns from the log file. Specify the desired columns as a comma-delimited string of column numbers beginning with 1, where the second column from left is 2, the third is 3, and so on.

-f '*string*' | --find='*string*'

Finds the log entries containing the specified string.

-F '*string*' | --nofind='*string*'

Rejects the log entries containing the specified string.

-m *regex* | --match=*regex*

Finds log entries that match the specified Python regular expression. See *http://docs.python.org/library/re.html* for Python regular expression syntax.

-M *regex* | --nomatch=*regex*

Rejects log entries that match the specified Python regular expression. See *http://docs.python.org/library/re.html* for Python regular expression syntax.

-t | --trouble

Finds only the log entries that have `ERROR:`, `FATAL:`, or `PANIC:` in the first line.

Output Options

-n *integer* | --tail=*integer*

Limits the output to the last *integer* of qualifying log entries found.

-s *offset* [limit] | --slice=*offset* [limit]

From the list of qualifying log entries, returns the *limit* number of entries starting at the *offset* entry number, where an *offset* of zero (`0`) denotes the first entry in the result set and an *offset* of any number greater than zero counts back from the end of the result set.

-o *output_file* | --out=*output_file*

Writes the output to the specified file or directory location instead of `STDOUT`.

-z 0-9 | --zip=0-9

Compresses the output file to the specified compression level using `gzip`, where `0` is no compression and `9` is maximum compression. If you supply an output file name ending in `.gz`, the output file will be compressed by default using maximum compression.

-a | --append

If the output file already exists, appends to the file instead of overwriting it.

Input Options

input_file

The name of the input log file(s) to search through. If an input file is not supplied, `gplogfilter` will use the `$MASTER_DATA_DIRECTORY` environment variable to locate the HAWQ master log file. To read from standard input, use a dash (`-`) as the input file name.

-u | --unzip

Uncompress the input file using `gunzip`. If the input file name ends in `.gz`, it will be uncompressed by default.

--help

Displays the online help.

--version

Displays the version of this utility.

Examples

Display the last three error messages in the master log file:

```
gplogfilter -t -n 3
```

Display all log messages in the master log file timestamped in the last 10 minutes:

```
gplogfilter -d :10
```

Display log messages in the master log file containing the string `|con6 cmd11|`:

```
gplogfilter -f '|con6 cmd11|'
```

Using *`gpssh`*, run `gplogfilter` on the segment hosts and search for log messages in the segment log files containing the string `con6` and save output to a file.

```
gpssh -f seg_hosts_file -e 'source
/usr/local/greenplum-db/greenplum_path.sh ; gplogfilter -f
con6 /gpdata/*/pg_log/gpdb*.csv' > seglog.out
```

See Also

`gpssh`, *`gpscp`*

Related Links

Management Utility Reference

gppkg

Installs HAWQ extensions such as pgcrypto, PL/R, PL/Java, and MADlib, along with their dependencies, across an entire cluster.

Synopsis

```
gppkg [-i package | -u package | -r  name-version | -c]
        [-d master_data_directory] [-a] [-v]

gppkg --migrate GPHOME_1 GPHOME_2 [-a] [-v]

gppkg [-q | --query] query_option

gppkg -? | --help | -h

gppkg --version
```

Description

The Package Manager (`gppkg`) utility installs HAWQ extensions, along with any dependencies, on all hosts across a cluster. It will also automatically install extensions on new hosts in the case of system expansion and segment recovery.

First, download one or more of the available packages from *Pivotal Network* then copy it to the master host. Use the Package Manager to install each package using the options described below.

Note: After a major upgrade to HAWQ, you must download and install all extensions again.

Examples of database extensions and packages software that are delivered using the Package Manager are:

- PL/Java
- PL/R
- MADlib
- pgcrypto

Options

`-a (do not prompt)`

Do not prompt the user for confirmation.

`-c | --clean`

Reconciles the package state of the cluster to match the state of the master host. Running this option after a failed or partial install/uninstall ensures that the package installation state is consistent across the cluster.

`-d` ***`master_data_directory`***

The master data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

`-i` ***`package`*** **`| --install=`*****`package`***

Installs the given package. This includes any pre/post installation steps and installation of any dependencies.

`--migrate` ***`GPHOME_1 GPHOME_2`***

Migrates packages from a separate `$GPHOME`. Carries over packages from one version of HAWQ to another.

For example:

```
gppkg --migrate /usr/local/XXXXXXX /usr/local/XXXXXXX
```

This option is automatically invoked by the installer during minor upgrades. This option is given here for cases when the user wants to migrate packages manually.

Migration can only proceed if `gppkg` is executed from the installation directory to which packages are being migrated. That is, `GPHOME_2` must match the `$GPHOME` from which the currently executing `gppkg` is being run.

-q | --query *query_option*

Provides information specified by `query_option` about the installed packages. Only one `query_option` can be specified at a time. The following table lists the possible values for query_option. `<package_file>` is the name of a package.

Table 10: Query Options for gppkg

query_option	Returns
`<package_file>`	Whether the specified package is installed.
`--info <package_file>`	The name, version, and other information about the specified package.
`--list <package_file>`	The file contents of the specified package.
`--all`	List of all installed packages.

-r *name-version* | --remove=*name-version*

Removes the specified package.

-u *package* | --update=*package*

Updates the given package.

> **Warning:** The process of updating a package includes removing all previous versions of the system objects related to the package. For example, previous versions of shared libraries are removed. After the update process, a database function will fail when it is called if the function references a package file that has been removed.

--version (show utility version)

Displays the version of this utility.

-v | --verbose

Sets the logging level to verbose.

-? | -h | --help

Displays the online help.

Related Links

Management Utility Reference

gpmigrator

Upgrades an existing HAWQ 1.1.x system to 1.2.x.

Synopsis

```
gpmigrator old_GPHOME_path new_GPHOME_path
           [-d master_data_directory]
           [-l logfile_directory] [-q] [--debug]
           [--check-only] [-R]

gpmigrator --version | -v

gpmigrator --help | -h
```

Prerequisites

The following tasks should be performed prior to executing an upgrade:

- Make sure you are logged in to the master host as the HAWQ superuser (`gpadmin`).
- Install the HAWQ 1.2 binaries on all HAWQ hosts.
- Copy or preserve any additional folders or files (such as backup folders) that you have added in the HAWQ data directories or `$GPHOME` directory. Only files or folders strictly related to HAWQ operations are preserved by the migration utility.
- (**Optional**) Run `VACUUM` on all databases, and remove old server log files from `pg_log` in your master and segment data directories. This is not required, but will reduce the size of HAWQ files to be backed up and migrated.
- Check for and recover any failed segments in your current HAWQ system (`gpstate`, `gprecoverseg`).
- (**Optional, but highly recommended**) Back up your current databases. If you find any issues when testing your upgraded system, you can restore this backup.
- Remove the standby master from your system configuration (`gpinitstandby -r`).
- Do a clean shutdown of your current system (`gpstop`).
- Update your environment to source the 1.2 installation.
- Inform all database users of the upgrade and lockout time frame. Once the upgrade is in process, users will not be allowed on the system until the upgrade is complete.

Description

The `gpmigrator` utility upgrades an existing Greenplum Database 1.1.x system to 1.2.x. This utility updates the system catalog and internal version number, but not the actual software binaries. During the migration process, all client connections to HAWQ will be locked out.

Options

old_GPHOME_path

Required. The absolute path to the current version of HAWQ software you want to migrate away from.

new_GPHOME_path

Required. The absolute path to the new version of HAWQ software you want to migrate to.

-d *master_data_directory*

Optional. The current master host data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-q (quiet mode)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-R (revert)

In the event of an error during upgrade, reverts all changes made by `gpmigrator`.

--check-only

Runs pre-migrate checks to verify that your database is healthy.

Checks include:

- Check catalog health
- Check that the Greenplum Database binaries on each segment match those on the master
- Check for a minimum amount of free disk space

Performing a pre-migration check of your database should be done during a database maintenance period. If the utility detects catalog errors, the utility stops the database.

--help | -h

Displays the online help.

--debug

Sets logging level to debug.

--version | -v

Displays the version of this utility.

See Also

gpstop, *gpstate*, *gprecoverseg*

Related Links

Management Utility Reference

gprecoverseg

Recovers a primary or mirror segment instance that has been marked as down.

Synopsis

```
gprecoverseg [-p new_recover_host[,...]] [-d master_data_directory]
             [-B parallel_processes]

gprecoverseg -?

gprecoverseg --version
```

Description

The `gprecoverseg` utility reactivates failed segment instances.Once gprecoverseg completes this process, the system will be recovered.

A segment instance can fail for several reasons, such as a host failure, network failure, or disk failure. When a segment instance fails, its status is marked as *down* in the HAWQ database system catalog, and its mirror is activated in *change tracking* mode. In order to bring the failed segment instance back into operation again, you must first correct the problem that made it fail in the first place, and then recover the segment instance in HAWQ database using `gprecoverseg`.

Segment recovery using `gprecoverseg` requires that you have an active mirror to recover from. For systems that do not have mirroring enabled, or in the event of a double fault (a primary and mirror pair both down at the same time) — do a system restart to bring the segments back online (`gpstop -r`).

By default, a failed segment is recovered in place, meaning that the system brings the segment back online on the same host and data directory location on which it was originally configured.

If the data directory was removed or damaged, gprecoverseg can recover the data directory (using `-F`). This requires that you have at least one alive segment to recover from.

In some cases, this may not be possible (for example, if a host was physically damaged and cannot be recovered). In this situation, `gprecoverseg` allows you to recover failed segments to a completely new host (using `-p`). In this scenario, to prevent HAWQ having an unbalanced workload, all the the segments on the failed host should be moved to the new host. You must manually kill the other alive segments left on the failed host before you try to run `gprecoverseg`.

The new recovery segment host must be pre-installed with the HAWQ software and configured exactly the same as the existing segment hosts. A spare data directory location must exist on all currently configured segment hosts and have enough disk space to accommodate the failed segments.

If you do not have mirroring enabled or if you have both a primary and its mirror down, you must take manual steps to recover the failed segment instances and then restart the system, for example:

```
gpstop -r
```

Options

`-a (do not prompt)`

Do not prompt the user for confirmation. If `gprecovery` **-a** cannot recover successfully, HAWQ will raise an exception and tell user to use the **-F** or **-p** option.

`-B` ***`parallel_processes`***

The number of segments to recover in parallel. If not specified, the utility will start up to four parallel processes depending on how many segment instances it needs to recover.

-d *master_data_directory*

Optional. The master host data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

-F (full recovery)

Optional. Perform a full copy of the active segment instance in order to recover the failed segment. The default is to only restart the failed segment in-place.

Comments

Lines beginning with `#` are treated as comments and ignored.

Filespace Order

The first comment line that is not a comment specifies filespace ordering. This line starts with `filespaceOrder=` and is followed by list of filespace names delimited by a colon. For example:

```
filespaceOrder=raid1:raid2
```

The default `pg_system` filespace should not appear in this list. The list should be left empty on a system with no filespaces other than the default `pg_system` filespace. For example:

```
filespaceOrder=
```

Examples

Recovery of a single mirror to a new host on a system with an extra filespace

```
filespaceOrder=
sdw1-1:50001:/data1/mirror/gpseg16SPACEsdw4-1:
50001:51001:/data1/recover1/gpseg16
```

Obtaining a Sample File

You can use the `-o` option to output a sample recovery configuration file to use as a starting point.

-l *logfile_directory*

The directory to write the log file. Defaults to `~/gpAdminLogs`.

-p *new_recover_host*[,...]

Specifies a spare host outside of the currently configured HAWQ array on which to recover invalid segments. In the case of multiple failed segment hosts, you can specify a comma-separated list. The spare host must have the HAWQ Database software installed and configured, and have the same hardware and OS configuration as the current segment hosts (same OS version, locales, `gpadmin` user account, data directory locations created, ssh keys exchanged, number of network interfaces, network interface naming convention, and so on.).

Before using this option:

- Make sure that all the segments on the failed hosts are marked down.
- If you see any segments that are alive on the failed hosts, kill the segments or shutdown the failed hosts.
- Specify a new host for each failed host.

-q (no screen output)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-v (verbose)

Sets logging output to verbose.

`--version (version)`

Displays the version of this utility.

`-? (help)`

Displays the online help.

Examples

Recover any failed segment instances in place:

```
$ gprecoverseg
```

Recreate any failed segment instances in place

```
$ gprecoverseg -f
```

Replace any failed host to a set of new hosts:

```
$ gprecoverseg -p new1, new2
```

See Also

`gpstart`, *`gpstop`*

Related Links

Management Utility Reference

gpscp

Copies files between multiple hosts at once.

Synopsis

```
gpscp { -f hostfile_gpssh | -h hostname [-h hostname ...] }
      [-J character] [-v] [[user@]hostname:]file_to_copy [...]
      [[user@]hostname:]copy_to_path

gpscp -?

gpscp --version
```

Description

The `gpscp` utility allows you to copy one or more files from the specified hosts to other specified hosts in one command using SCP (secure copy). For example, you can copy a file from the HAWQ master host to all of the segment hosts at the same time.

To specify the hosts involved in the SCP session, use the `-f` option to specify a file containing a list of host names, or use the `-h` option to name single host names on the command-line. At least one host name (`-h`) or a host file (`-f`) is required. The `-J` option allows you to specify a single character to substitute for the *hostname* in the `copy from` and `copy to` destination strings. If `-J` is not specified, the default substitution character is an equal sign (`=`). For example, the following command will copy `.bashrc` from the local host to `/home/gpadmin` on all hosts named in `hostfile_gpssh`:

```
gpscp -f hostfile_gpssh .bashrc =:/home/gpadmin
```

If a user name is not specified in the host list or with *user*`@` in the file path, `gpscp` will copy files as the currently logged in user. To determine the currently logged in user, do a `whoami` command. By default, `gpscp` goes to `$HOME` of the session user on the remote hosts after login. To ensure the file is copied to the correct location on the remote hosts, it is recommended that you use absolute paths.

Before using `gpscp`, you must have a trusted host setup between the hosts involved in the SCP session. You can use the utility `gpssh-exkeys` to update the known host files and exchange public keys between hosts if you have not done so already.

Options

-f *hostfile_gpssh*

Specifies the name of a file that contains a list of hosts that will participate in this SCP session. The syntax of the host file is one host per line as follows:

```
<hostname>
```

-h *hostname*

Specifies a single host name that will participate in this SCP session. You can use the `-h` option multiple times to specify multiple host names.

-J *character*

The `-J` option allows you to specify a single character to substitute for the *hostname* in the `copy from` and `copy to` destination strings. If `-J` is not specified, the default substitution character is an equal sign (`=`).

-v (verbose mode)

Optional. Reports additional messages in addition to the SCP command output.

file_to_copy

Required. The file name (or absolute path) of a file that you want to copy to other hosts (or file locations). This can be either a file on the local host or on another named host.

copy_to_path

Required. The path where you want the file(s) to be copied on the named hosts. If an absolute path is not used, the file will be copied relative to `$HOME` of the session user. You can also use the equal sign '`=`' (or another character that you specify with the `-J` option) in place of a *hostname*. This will then substitute in each host name as specified in the supplied host file (`-f`) or with the `-h` option.

`-? (help)`

Displays the online help.

`--version`

Displays the version of this utility.

Examples

Copy the file named `installer.tar` to `/` on all the hosts in the file `hostfile_gpssh`:

```
gpscp -f hostfile_gpssh installer.tar =:/
```

Copy the file named *myfuncs.so* to the specified location on the hosts named `sdw1` and `sdw2`:

```
gpscp -h sdw1 -h sdw2 myfuncs.so =:/usr/local/HAWQ-db/lib
```

Related Links

Management Utility Reference

gpstart

Starts a HAWQ system.

Synopsis

```
gpstart [-d master_data_directory] [-B parallel_processes] [-R]
        [-m] [-y] [-a] [-t timeout_seconds] [-l logfile_directory]
        [-v | -q]

gpstart -? | -h | --help

gpstart --version
```

Description

The `gpstart` utility is used to start the HAWQ server processes. When you start a HAWQ system, you are actually starting several `postgres` database server listener processes at once (the master and all of the segment instances). The `gpstart` utility handles the startup of the individual instances. Each instance is started in parallel.

The first time an administrator runs `gpstart`, the utility creates a hosts cache file named `.gphostcache` in the user's home directory. Subsequently, the utility uses this list of hosts to start the system more efficiently. If new hosts are added to the system, you must manually remove this file from the `gpadmin` user's home directory. The utility will create a new hosts cache file at the next startup.

Before you can start a HAWQ system, you must have initialized the system using `gpinitsystem` first.

Options

`-a (do not prompt)`

Do not prompt the user for confirmation.

`-B` ***`parallel_processes`***

The number of segments to start in parallel. If not specified, the utility will start up to 60 parallel processes, depending on how many segment instances are needed.

`-d` ***`master_data_directory`***

Optional. The master host data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

`-l` ***`logfile_directory`***

The directory to write the log file. Defaults to `~/gpAdminLogs`.

`-m (master only)`

Optional. Starts the master instance only, which may be useful for maintenance tasks. This mode only allows connections to the master in utility mode. For example:

```
PGOPTIONS='-c gp_session_role=utility' psql
```

`-q (no screen output)`

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

`-R (restricted mode)`

Starts HAWQ in restricted mode (only database superusers are allowed to connect).

`-t` ***`timeout_seconds`***

Specifies a timeout in seconds to wait for a segment instance to start up. If a segment instance was shutdown abnormally (due to power failure or killing its `postgres` database listener process, for example), it may take longer to start up due to the database recovery and validation process. If not specified, the default timeout is 60 seconds.

`-v (verbose output)`

Displays detailed status, progress and error messages output by the utility.

`-y (do not start standby master)`

Optional. Do not start the standby master host. The default is to start the standby master host and synchronization process.

`-? | -h | --help (help)`

Displays the online help.

`--version (show utility version)`

Displays the version of this utility.

Examples

Start a HAWQ system:

```
gpstart
```

Start a HAWQ system in restricted mode (only allow superuser connections):

```
gpstart -R
```

Start the HAWQ master instance only and connect in utility mode:

```
gpstart -m PGOPTIONS='-c gp_session_role=utility' psql
```

Display the online help for the gpstart utility:

```
gpstart -?
```

See Also

`gpstop`

Related Links

Management Utility Reference

gpssh

Provides SSH access to multiple hosts at once.

Synopsis

```
gpssh { -f hostfile_gpssh | - h hostname [-h hostname ...] } [-u userid ...]
[-v] [-e] [bash_command]

gpssh -?

gpssh --version
```

Description

The `gpssh` utility allows you to run bash shell commands on multiple hosts at once using SSH (secure shell). You can execute a single command by specifying it on the command-line, or omit the command to enter into an interactive command-line session.

To specify the hosts involved in the SSH session, use the `-f` option to specify a file containing a list of host names, or use the `-h` option to name single host names on the command-line. At least one host name (`-h`) or a host file (`-f`) is required. Note that the current host is ***not*** included in the session by default — to include the local host, you must explicitly declare it in the list of hosts involved in the session.

Before using `gpssh`, you must have a trusted host setup between the hosts involved in the SSH session. You can use the utility `gpssh-exkeys` to update the known host files and exchange public keys between hosts if you have not done so already.

If you do not specify a command on the command-line, `gpssh` will go into interactive mode. At the `gpssh` command prompt (`=>`), you can enter a command as you would in a regular bash terminal command-line, and the command will be executed on all hosts involved in the session. To end an interactive session, press `CTRL`+`D` on the keyboard or type `exit` or `quit`.

If a user name is not specified in the host file, `gpssh` will execute commands as the currently logged in user. To determine the currently logged in user, do a `whoami` command. By default, `gpssh` goes to `$HOME` of the session user on the remote hosts after login. To ensure commands are executed correctly on all remote hosts, you should always enter absolute paths.

Options

bash_command

A bash shell command to execute on all hosts involved in this session (optionally enclosed in quotes). If not specified, `gpssh` will start an interactive session.

-e (echo)

Optional. Echoes the commands passed to each host and their resulting output while running in non-interactive mode.

-f *hostfile_gpssh*

Specifies the name of a file that contains a list of hosts that will participate in this SSH session. The host name is required, and you can optionally specify an alternate user name and/or SSH port number per host. The syntax of the host file is one host per line as follows:

```
[username@]hostname[:ssh_port]
```

-h *hostname*

Specifies a single host name that will participate in this SSH session. You can use the `-h` option multiple times to specify multiple host names.

-u*userid*

Specifies the userid for this SSH session.

-v (verbose mode)

Optional. Reports additional messages in addition to the command output when running in non-interactive mode.

--version

Displays the version of this utility.

-? (help)

Displays the online help.

Examples

Start an interactive group SSH session with all hosts listed in the file `hostfile_gpssh`:

```
$ gpssh -f hostfile_gpssh
```

At the `gpssh` interactive command prompt, run a shell command on all the hosts involved in this session.

```
=> ls -a /data/primary/*
```

Exit an interactive session:

```
=> exit
```

Start a non-interactive group SSH session with the hosts named `sdw1` and `sdw2` and pass a file containing several commands named `command_file` to `gpssh`:

```
$ gpssh -h sdw1 -h sdw2 -v -e < command_file
```

Execute single commands in non-interactive mode on hosts `sdw2` and `localhost`:

```
$ gpssh -h sdw2 -h localhost -v -e 'ls -a /data/primary/*'
$ gpssh -h sdw2 -h localhost -v -e 'echo $GPHOME'
$ gpssh -h sdw2 -h localhost -v -e 'ls -1 | wc -l'
```

See Also

gpssh-exkeys, *gpscp*

Related Links

Management Utility Reference

gpssh-exkeys

Exchanges SSH public keys between hosts.

Synopsis

```
gpssh-exkeys -f hostfile_exkeys | - h hostname [-h hostname ...] [-p <password>]

gpssh-exkeys -e hostfile_exkeys -x hostfile_gpexpand

gpssh-exkeys -?

gpssh-exkeys --version
```

Description

The `gpssh-exkeys` utility exchanges SSH keys between the specified host names (or host addresses). This allows SSH connections between HAWQ hosts and network interfaces without a password prompt. The utility is used to initially prepare a HAWQ system for password-free SSH access, and also to add additional ssh keys when expanding a HAWQ system.

To specify the hosts involved in an initial SSH key exchange, use the `-f` option to specify a file containing a list of host names (recommended), or use the `-h` option to name single host names on the command-line. At least one host name (`-h`) or a host file is required. Note that the local host is included in the key exchange by default.

To specify new expansion hosts to be added to an existing HAWQ system, use the `-e` and `-x` options. The `-e` option specifies a file containing a list of existing hosts in the system that already have SSH keys. The `-x` option specifies a file containing a list of new hosts that need to participate in the SSH key exchange.

Keys are exchanged as the currently logged in user. Pivotal recommends performing the key exchange process twice: once as `root` and once as the `gpadmin` user (the user designated to own your HAWQ installation). The HAWQ management utilities require that the same non-root user be created on all hosts in the HAWQ system, and the utilities must be able to connect as that user to all hosts without a password prompt.

The `gpssh-exkeys` utility performs key exchange using the following steps:

- Creates an RSA identification key pair for the current user if one does not already exist. The public key of this pair is added to the `authorized_keys` file of the current user.
- Updates the `known_hosts` file of the current user with the host key of each host specified using the `-h`, `-f`, `-e`, and `-x` options.
- Connects to each host using `ssh` and obtains the `authorized_keys`, `known_hosts`, and `id_rsa.pub` files to set up password-free access.
- Adds keys from the `id_rsa.pub` files obtained from each host to the `authorized_keys` file of the current user.
- Updates the `authorized_keys`, `known_hosts`, and `id_rsa.pub` files on all hosts with new host information (if any).

Options

`-e` ***`hostfile_exkeys`***

When doing a system expansion, this is the name and location of a file containing all configured host names and host addresses (interface names) for each host in your *current* HAWQ system (master, standby master and segments), one name per line without blank lines or extra spaces. Hosts specified in this file cannot be specified in the host file used with `-x`.

-f *hostfile_exkeys*

Specifies the name and location of a file containing all configured host names and host addresses (interface names) for each host in your HAWQ system (master, standby master and segments), one name per line without blank lines or extra spaces.

-h *hostname*

Specifies a single host name (or host address) that will participate in the SSH key exchange. You can use the `-h` option multiple times to specify multiple host names and host addresses.

-p *password*

Specifies the password used to login to the hosts. The hosts should share the same password.

--version

Displays the version of this utility.

-x *hostfile_gpexpand*

When doing a system expansion, this is the name and location of a file containing all configured host names and host addresses (interface names) for each *new segment host* you are adding to your HAWQ system, one name per line without blank lines or extra spaces. Hosts specified in this file cannot be specified in the host file used with `-e`.

-? (help)

Displays the online help.

Examples

Exchange SSH keys between all host names and addresses listed in the file `hostfile_exkeys`:

```
$ gpssh-exkeys -f hostfile_exkeys
```

Exchange SSH keys between the hosts `sdw1`, `sdw2`, and `sdw3`:

```
$ gpssh-exkeys -h sdw1 -h sdw2 -h sdw3
```

Exchange SSH keys between existing hosts `sdw1`, `sdw2`, and `sdw3`, and new hosts `sdw4` and `sdw5` as part of a system expansion operation:

```
$ cat hostfile_exkeys
 mdw
 mdw-1
 mdw-2
 smdw
 smdw-1
 smdw-2
 sdw1
 sdw1-1
 sdw1-2
 sdw2
 sdw2-1
 sdw2-2
 sdw3
 sdw3-1
 sdw3-2
$ cat hostfile_gpexpand
 sdw4
 sdw4-1
 sdw4-2
 sdw5
 sdw5-1
 sdw5-2
$ gpssh-exkeys -e hostfile_exkeys -x hostfile_gpexpand
```

See Also

gpssh, *gpscp*

Related Links

Management Utility Reference

gpstate

Shows the status of a running HAWQ system.

Synopsis

```
gpstate [-d master_data_directory] [-B parallel_processes]
         [-s | -b | -Q] [-p] [-i] [-f] [-v | -q]
         [-l log_directory]

gpstate -? | -h | --help
```

Description

The `gpstate` utility displays information about a running HAWQ instance. There is additional information you may want to know about a HAWQ system, since it is comprised of multiple database instances (segments) spanning multiple machines. The `gpstate` utility provides additional status information for a HAWQ system, such as:

- Which segments are down.
- Master and segment configuration information (hosts, data directories, etc.).
- The ports used by the system.

Options

`-b (brief status)`

Optional. Display a brief summary of the state of the HAWQ system. This is the default option.

`-B` ***`parallel_processes`***

The number of segments to check in parallel. If not specified, the utility will start up to 60 parallel processes depending on how many segment instances it needs to check.

`-d` ***`master_data_directory`***

Optional. The master data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

`-f (show standby master details)`

Display details of the standby master host if configured.

`-i (show HAWQ version)`

Display the HAWQ software version information for each instance.

`-l` ***`logfile_directory`***

The directory to write the log file. Defaults to `~/gpAdminLogs`.

`-p (show ports)`

List the port numbers used throughout the HAWQ system.

`-q (no screen output)`

Optional. Run in quiet mode. Except for warning messages, command output is not displayed on the screen. However, this information is still written to the log file.

`-Q (quick status)`

Optional. Checks segment status in the system catalog on the master host. Does not poll the segments for status.

`-s (detailed status)`

Optional. Displays detailed status information for the HAWQ system.

`-v (verbose output)`

Optional. Displays error messages and outputs detailed status and progress information.

`-? | -h | --help (help)`

Displays the online help.

Output Field Definitions

The following output fields are reported by `gpstate -s` for the master:

Table 11: gpstate output data for the master

Output Data	Description
Master host	Host name of the master
Master process ID	PID of the master database listener process
Master data directory	File system location of the master data directory
Master port	Port of the master `postgres` database listener process

Master current role	dispatch = regular operating mode
	utility = maintenance mode

HAWQ array configuration type	Standard = one NIC per host
	Multi-Home = multiple NICs per host

HAWQ initsystem version	Version of HAWQ when system was first initialized
HAWQ current version	Current version of HAWQ
Postgres version	version of Postgres that HAWQ is based on
HAWQ mirroring status	Physical mirroring, SAN or none
Master standby	Host name of the standby master

The following output fields are reported by `gpstate -s` for each segment:

Table 12: gpstate output data for segments

Output Data	Description
Hostname	System-configured host name
Address	Network address host name (NIC name)
Datadir	File system location of segment data directory
Port	Port number of segment `postgres` database listener process
Current Role	Current role of a segment: *Primary*
Preferred Role	Role at system initialization time: *Primary*
File `postmaster.pid`	Status of `postmaster.pid` lock file: *Found* or *Missing*

Output Data	Description
PID from `postmaster.pid` file	PID found in the `postmaster.pid` file
Lock files in `/tmp`	A segment port lock file for its `postgres` process is created in `/tmp` (file is removed when a segment shuts down)
Active PID	Active process ID of a segment
Master reports status as	Segment status as reported in the system catalog: *Up* or *Down*
Database status	Status of HAWQ to incoming requests: *Up*, *Down*, or *Suspended*. A *Suspended* state means database activity is temporarily paused while a segment transitions from one state to another.

Examples

Show detailed status information of a HAWQ system:

```
gpstate -s
```

Do a quick check for down segments in the master host system catalog:

```
gpstate -Q
```

Show information about the standby master configuration:

```
gpstate -f
```

Display the HAWQ software version information:

```
gpstate -I
```

See Also

`gpstart`, *`gplogfilter`*

Related Links

Management Utility Reference

gpstop

Stops or restarts a HAWQ system.

Synopsis

```
gpstop [-d master_data_directory] [-B parallel_processes]
       [-M smart | fast | immediate] [-t timeout_seconds] [-r] [-y] [-a]
       [-l logfile_directory] [-v | -q]

gpstop -m [-d master_data_directory] [-y] [-l logfile_directory] [-v | -q]

gpstop -u [-d master_data_directory] [-l logfile_directory] [-v | -q]

gpstop --version

gpstop -? | -h | --help
```

Description

The `gpstop` utility is used to stop the database servers that comprise a HAWQ system. When you stop a HAWQ system, you are actually stopping several `postgres` database server processes at once (the master and all of the segment instances). The `gpstop` utility handles the shutdown of the individual instances. Each instance is shutdown in parallel.

By default, you are not allowed to shut down HAWQ if there are any client connections to the database. Use the `-M fast` option to roll back all in progress transactions and terminate any connections before shutting down. If there are any transactions in progress, the default behavior is to wait for them to commit before shutting down.

With the `-u` option, the utility uploads changes made to the master `pg_hba.conf` file or to *runtime* configuration parameters in the master `postgresql.conf` file without interruption of service. Note that any active sessions will not pickup the changes until they reconnect to the database.

Options

`-a (do not prompt)`

Do not prompt the user for confirmation.

`-B` ***`parallel_processes`***

The number of segments to stop in parallel. If not specified, the utility will start up to 60 parallel processes depending on how many segment instances it needs to stop.

`-d` ***`master_data_directory`***

Optional. The master host data directory. If not specified, the value set for `$MASTER_DATA_DIRECTORY` will be used.

`-l` ***`logfile_directory`***

The directory to write the log file. Defaults to `~/gpAdminLogs`.

`-m (master only)`

Optional. Shuts down a HAWQ master instance that was started in maintenance mode.

`-M fast (fast shutdown - rollback)`

Fast shut down. Any transactions in progress are interrupted and rolled back.

`-M immediate (immediate shutdown - abort)`

Immediate shut down. Any transactions in progress are aborted. This shutdown mode is not recommended. This mode kills all database processes without allowing the database server to complete transaction processing or clean up any temporary or in-process work files.

-M smart (smart shutdown - warn)

Smart shut down. If there are active connections, this command fails with a warning. This is the default shutdown mode.

-q (no screen output)

Run in quiet mode. Command output is not displayed on the screen, but is still written to the log file.

-r (restart)

Restart after shutdown is complete.

-t *timeout_seconds*

Specifies a timeout threshold (in seconds) to wait for a segment instance to shutdown. If a segment instance does not shutdown in the specified number of seconds, `gpstop` displays a message indicating that one or more segments are still in the process of shutting down and that you cannot restart HAWQ until the segment instance(s) are stopped. This option is useful in situations where `gpstop` is executed and there are very large transactions that need to rollback. These large transactions can take over a minute to rollback and surpass the default timeout period of 600 seconds.

-u (reload pg_hba.conf and postgresql.conf files only)

This option reloads the `pg_hba.conf` files of the master and segments and the runtime parameters of the `postgresql.conf` files but does not shutdown the HAWQ array. Use this option to make new configuration settings active after editing `postgresql.conf` or `pg_hba.conf`. Note that this only applies to configuration parameters that are designated as *runtime* parameters. In HAWQ if there are some failed segments, this option can not be executed.

-v (verbose output)

Displays detailed status, progress and error messages output by the utility.

--version (show utility version)

Displays the version of this utility.

-y (do not stop standby master)

Do not stop the standby master process. The default is to stop the standby master.

-? | -h | --help (help)

Displays the online help.

Examples

Stop a HAWQ system in smart mode:

```
gpstop
```

Stop a HAWQ system in fast mode:

```
gpstop -M fast
```

Stop all segment instances and then restart the system:

```
gpstop -r
```

Stop a master instance that was started in maintenance mode:

```
gpstop -m
```

Reload the `postgresql.conf` and `pg_hba.conf` files after making configuration changes but do not shutdown the HAWQ array:

```
gpstop -u
```

See Also

`gpstart`

Related Links

Management Utility Reference

Chapter 14

Client Utility Reference

This section provides references for the command-line client utilities provided with HAWQ. HAWQ provides the standard client and server programs, and additional client utilities to administer a distributed HAWQ DBMS.

The HAWQ client programs are located in `$GPHOME/bin`.

createdb

Creates a new database.

Synopsis

```
createdb [connection_option ...] [-D tablespace] [-E encoding]
        [-O owner] [-T template] [-e] [dbname ['description']]

createdb --help

createdb --version
```

Description

Creates a new database in a HAWQ system. It is a wrapper around the SQL command `CREATE DATABASE`.

When you create a database with this command, you will own the new database. You can specify a different owner using the `-O` option, if you have the appropriate privileges.

Options

dbname

Select a unique name for the new database in the HAWQ system. If you do not specify a name, the utility reads the environment variables `PGDATABASE`, `PGUSER`, or defaults to the current system user.

description

Describe the newly created database. Enclose white space within quotes.

-D *tablespace* | --tablespace *tablespace*

The default tablespace for the database.

-e echo

Echo the commands that createdb generates and sends to the server.

-E *encoding* | --encoding *encoding*

Character set encoding used in the new database. Specify a string constant (such as '`UTF8`'), an integer encoding number, or `DEFAULT` to use the default encoding.

-O *owner* | --owner *owner*

The name of the database user who will own the new database. Defaults to the user executing this command.

-T *template* | --template *template*

The name of the template used to create the new database. Defaults to `template1`.

Connection Options

-h *host* | --host *host*

The host name of the HAWQ master database server. If not specified, reads the name from the environment variable `PGHOST` or defaults to localhost.

-p *port* | --port *port*

The TCP port where the HAWQ master database server listens for connections. If not specified, reads the environment variable `PGPORT`, or defaults to `5432`.

-U *username* | --username *username*

Specifies the database role name used to connect. If not specified, reads the environment variable `PGUSER` or defaults to the current system role name.

`-w | --no-password`

Use this to run automated batch jobs and scripts. In general, if the server requires password authentication, ensure that it can be accessed through a `.pgpass` file. Otherwise, the connection attempt will fail.

`-W | --password`

Force a password prompt.

Examples

To create the database `test` using the default options:

```
createdb test
```

To create the database `demo` using the HAWQ master on host `gpmaster`, port `54321`, using the `LATIN1` encoding scheme:

```
createdb -p 54321 -h gpmaster -E LATIN1 demo
```

See Also

`CREATE DATABASE`

Related Links

Client Utility Reference

createlang

Defines a new procedural language for a database.

Synopsis

```
createlang [connection_option ...] [-e] langname [[-d] dbname]

createlang [connection-option ...] -l dbname

createlang --help

createlang --version
```

Description

The `createlang` utility adds a new programming language to a database. `createlang` is a wrapper around the `SQL` command `CREATE LANGUAGE`.

The procedural language packages currently included in the standard HAWQ Database distribution are: `PL/pgSQL`, `PL/R`, and `PL/Java`. The `PL/pgSQL` language is already registered in all databases by default.

A language handler has also been added for `PL/R`, but the `PL/R` language package is not pre-installed with the HAWQ Database. See *Using Procedural Languages*.

Options

`langname`

Specifies the name of the procedural programming language to be defined.

`[-d] *dbname* | [--dbname] *dbname*`

Specifies to which database the language should be added. The default is to use the `PGDATABASE` environment variable setting, or the same name as the current system user.

`-e | --echo`

Echo the commands that `createlang` generates and sends to the server.

`-l *dbname* | --list *dbname*`

Show a list of already installed languages in the target database.

Connection Options

`-h *host* | --host *host*`

The host name of the machine on which the HAWQ master database server is running. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

`-p *port* | --port *port*`

The TCP port on which the HAWQ master database server is listening for connections. If not specified, reads from the environment variable `PGPORT` or defaults to `5432`.

`-U *username* | --username *username*`

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

`-w | --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

-W | --password

Force a password prompt.

Examples

To install the language `plperl` into the database `template1`:

```
createlang plperl template1
```

See Also

`CREATE LANGUAGE`

Related Links

Client Utility Reference

createuser

Creates a new database role.

Synopsis

```
createuser [connection_option ...] [role_attribute ...] [-e] role_name

createuser --help | --version
```

Description

`createuser` creates a new HAWQ role. You must be a superuser or user with `CREATE-ROLE` privileges. To create a new superuser, you must be a superuser.

> **Note:** Making someone a superuser grants privileges such as bypassing access permission checks within the database.

`createuser` is a wrapper around the SQL command `CREATE ROLE`.

Options

role_name

Select a unique name for the role to be created. This name must be different from all existing roles in this HAWQ installation.

-c number | --connection-limit *number*

Set a maximum number of connections for the new role. The default is no limit.

-D | --no-createdb

By default, the new role cannot create databases.

-d | --createdb

The new role can create databases.

-e | --echo

Echo the commands that createuser generates and sends to the server.

-E | --encrypted

Encrypts and stores the password for the role. If not specified, uses the default password.

-i | --inherit

By default, the new role inherits the privileges of the groups to which it belongs.

-I | --no-inherit

The new role will not inherit the privileges of the groups to which it belongs.

-l | --login

By default, the new role can log in to HAWQ.

-L | --no-login

The new role cannot log in to HAWQ (a group-level role).

-N | --unencrypted

Does not encrypt the stored password for the role. If not specified, the default password behavior is used.

-P | --pwprompt

If given, createuser prompts you for the password for the new role. Use this only if you want to enforce password authentication.

-r | --createrole

The new role can create new roles (`CREATE-ROLE` privilege).

-R | --no-createrole

The new role cannot create new roles. This is the default.

-s | --superuser

Create the new role as superuser.

-S | --no-superuser

Do not create the new role to be a superuser. This is the default.

Connection Options

-h *host* | --host *host*

The host name of the HAWQ master database server. If not specified, reads the name from the environment variable `PGHOST` or defaults to `localhost`.

-p *port* | --port *port*

The TCP port where the HAWQ master database server listens for connections. If not specified, reads the environment variable `PGPORT`, or defaults to `5432`.

- U *username* | --username *username*

Specifies the database role name used to connect. If not specified, reads the environment variable `PGUSER` or defaults to the current system role name.

-w | --no-password

Use this to run automated batch jobs and scripts. In general, if the server requires password authentication, ensure that it can be accessed through a `.pgpass` file. Otherwise the connection attempt will fail.

-W | --password

Force a password prompt.

Examples

Create the role, `joe`, with default options:

```
createuser joe
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
CREATE ROLE
```

Create the role, `joe`, with default connection options:

```
createuser -h masterhost -p 54321 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATE-ROLE INHERIT
LOGIN;
CREATE ROLE
```

Create the role, `joe` as a superuser, and provide password prompts:

```
createuser -P -s -e joe
Enter password for new role: admin123
Enter it again: admin123
CREATE ROLE joe PASSWORD 'admin123' SUPERUSER CREATEDB
CREATE-ROLE INHERIT LOGIN;
CREATE ROLE
```

Note: The example shows how the new password is echoed if the `-e` option is used.

See Also

`CREATE ROLE`

Related Links

Client Utility Reference

dropdb

Removes a database.

Synopsis

```
dropdb [connection_option ...] [-e] [-i] dbname

dropdb --help

dropdb --version
```

Description

`dropdb` removes an existing database. The user who executes this command must be a superuser or own the database being dropped.

`dropdb` is a wrapper around the SQL command `DROP DATABASE`.

Options

dbname

The unique name of the database being removed.

-e | --echo

Echo the commands that `dropdb` generates and sends to the server.

-i | --interactive

Presents verification prompts to ensure that you want to perform this process.

Connection Options

-h *host* | --host *host*

The host name of the HAWQ master database server. If not specified, reads the name of the environment variable `PGHOST` or defaults to localhost.

-p *port* | --port *port*

The TCP port where the HAWQ master database server listens for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U *username* | --username *username*

Specifies the database role name used to connect. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

-w | --no-password

Use this to run automated batch jobs and scripts. In general, if the server requires password authentication, ensure that it can be accessed through a `.pgpass` file. Otherwise the connection attempt will fail.

-W | --password

Force a password prompt.

Examples

Destroy the database named `demo` using default connection parameters:

```
dropdb demo
```

Destroy the database named `demo` using connection options, with verification, and a peek at the underlying command:

```
dropdb -p 54321 -h masterhost -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE "demo"
DROP DATABASE
```

See Also

DROP DATABASE

Related Links

Client Utility Reference

dropuser

Removes a database role.

Synopsis

```
dropuser [connection_option ...] [-e] [-i] role_name

dropuser --help

dropuser --version
```

Description

`dropuser` removes an existing role from HAWQ.. Only superusers and users with the `CREATE-ROLE` privilege can remove roles. To remove a superuser role, you must yourself be a superuser.

`dropuser` is a wrapper around the SQL command `DROP ROLE`.

Options

role_name

The name of the role to be removed. If you do not specify the name, you will be prompted on the command line.

-e | --echo

Echo the commands that `dropuser` generates and sends to the server.

-i | --interactive

Prompt for confirmation before actually removing the role.

Connection Options

-h *host* | --host *host*

The host name of the HAWQ master database server. If not specified, reads from the environment variable `PGHOST` or defaults to localhost.

-p *port* | --port *port*

The TCP port where the HAWQ master database server listens for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

-U *username* | --username *username*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

-w | --no-password

Use this to run automated batch jobs and scripts. In general, if the server requires password authentication ensure that it can be accessed through a `.pgpass` file. Otherwise the connection attempt will fail.

-W | --password

Force a password prompt.

Examples

Remove the role `joe` using default connection options:

```
dropuser joe
DROP ROLE
```

Remove the role *joe* using connection options, with verification, and a peek at the underlying command:

```
dropuser -p 54321 -h masterhost -i -e joe
Role "joe" will be permanently removed.
Are you sure? (y/n) y
DROP ROLE "joe"
DROP ROLE
```

See Also

DROP ROLE

Related Links

Client Utility Reference

pg_dump

Extracts a database into a single script file or other archive file.

Synopsis

```
pg_dump [connection_option ...] [dump_option ...] dbname
```

Description

`pg_dump` is a standard client utility for backing up a database, and is also supported in HAWQ. It creates a single (non-parallel) dump file.

Use `pg_dump` if you are migrating your data to a different database vendor, or to a HAWQ system with a different segment configuration. For example, a different HAWQ system configuration may have more or fewer segment instances.

To restore dump files:

- From archive format you must use the pg_restore utility.
- From plain text format you can use a client program such as psql.

About using `pg_dump` utility with HAWQ:

- The dump operation can take a several hours for very large databases. Make sure you have sufficient disk space to create the dump file.
- To migrate data from one HAWQ system to another, use the `--gp-syntax` command-line option to include the `DISTRIBUTED BY` clause in `CREATE TABLE` statements. This ensures that HAWQ table data is distributed with the correct distribution key columns upon restore.
- `pg_dump` makes consistent backups even if the database is being used concurrently.
- `pg_dump` does not block other users accessing the database (readers or writers).

When used with one of the archive file formats and combined with `pg_restore`, `pg_dump` provides a flexible archival and transfer mechanism. `pg_dump` can be used to backup an entire database, then `pg_restore` can be used to examine the archive and/or select the parts of the database you want to restore.

The *custom* format (`-Fc`) is flexible. You can select and reorder all the archived items, and compress the archive by default.

The tar format (`-Ft`) is not compressed and does not reorder data when loading. It can be manipulated with standard UNIX tools such as `tar`.

Options

dbname

Specifies the name of the database to be dumped. If the database is not specified, the environment variable `PGDATABASE` is used. If `PGDATABASE` is not set, the user name specified for the connection is used.

Dump Options

-a | --data-only

Dumps the data, not the schema (data definitions). This option is only meaningful for plain-text format. For archive formats, specify the option when you call `pg_restore`.

-b | --blobs

Include large objects in the dump. This is the default behavior except when `--schema`, `--table`, or `--schema-only` is specified.

-c | --clean

Adds commands to the text output file to clean (drop) database objects prior to creating them. The `DROP` commands are added to the DDL dump output files so that when you restore, the `DROP` commands run before the `CREATE` commands. This option is only works with plain-text format. For archive formats, you may specify the option when you call `pg_restore`.

-C | --create

Begin the output with a command to create the database itself and reconnect to the created database. With a script of this form, it doesn't matter which database you connect to before running the script. This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call `pg_restore`.

-d | --inserts

Dump data as `INSERT` commands (rather than `COPY`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL-based databases. Also, since this option generates a separate command for each row, an error in reloading a row causes only that row to be lost rather than the entire table contents. Note that the restore may fail altogether if you have rearranged column order. The `-D` option is safe against column order changes, though even slower.

-D | --column-inserts | --attribute-inserts

Dump data as `INSERT` commands with explicit column names `(INSERT INTO`*table*`(`*column*`, ...) VALUES ...)`. This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL-based databases. Also, since this option generates a separate command for each row, an error in reloading a row causes only that row to be lost rather than the entire table contents.

-E *encoding* | --encoding=*encoding*

Create the dump in the specified character set encoding. By default, the dump is created in the database encoding. (Another way to get the same result is to set the `PGCLIENTENCODING` environment variable to the desired dump encoding.)

-f *file* | --file=*file*

Send output to the specified file. If this is omitted, the standard output is used.

-F p|c|t | --format=plain|custom|tar

Selects the format of the output. format can be one of the following:

`p` | `plain` — Output a plain-text SQL script file. This is the default.

`c` | `custom` — Output a custom archive suitable for input into `pg_restore`. This is the most flexible format in that it allows reordering of loading data as well as object definitions. This format is also compressed by default.

`t` | `tar` — Output a tar archive suitable for input into `pg_restore`. Using this archive format allows reordering and/or exclusion of database objects at the time the database is restored. It is also possible to limit which data is reloaded at restore time.

-i | --ignore-version

Ignore version mismatch between `pg_dump` and the database server. `pg_dump` can dump from servers running previous releases of HAWQ, but very old versions may not be supported anymore. Use this option if you need to override the version check.

-n *schema* | --schema=*schema*

Dump only schemas matching the schema pattern; this selects both the schema itself, and all its contained objects. When this option is not specified, all non-system schemas in the target database will be dumped. Multiple schemas can be selected by writing multiple `-n` switches. Also, the schema parameter is interpreted as a pattern according to the same rules used by `psql`'s `\d` commands, so multiple schemas can also be selected by writing wildcard characters in the pattern. Add wildcards within quotes to prevent the shell from expanding the wildcards.

> **Note:** When `-n` is specified, `pg_dump` makes no attempt to dump any other database objects that the selected schema(s) may depend upon. Therefore, there is no guarantee that the results of a specific-schema dump can be successfully restored by themselves into a clean database.

> **Note:** Non-schema objects such as blobs are not dumped when `-n` is specified. You can add blobs back to the dump with the `--blobs` switch.

-N *schema* | --exclude-schema=*schema*

Do not dump any schemas matching the schema pattern. The pattern is interpreted according to the same rules as for `-n`. You can use `-N` more than once to exclude schemas matching several patterns. When both `-n` and `-N` are given, the utility dumps the schemas that match at least one `-n` switch but not the `-N` switches. If `-N` appears without `-n`, then schemas matching `-N` are excluded.

-o | --oids

Dump object identifiers (OIDs) as part of the data for every table. Use of this option is not recommended for files that are intended to be restored into HAWQ.

-O | --no-owner

Do not output commands to set ownership of objects to match the original database.

By default, `pg_dump` issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created database objects. To successfully run this script, you must be a superuser or own the objects in the script. Specify `-O` to make a script that can be restored by any user, and grant them ownership of all the objects. This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call `pg_restore`.

-s | --schema-only

Dump only the object definitions (schema), not data.

-S *username* | --superuser=*username*

Specify the superuser user name to use when disabling triggers. This is only relevant if `--disable-triggers` is used. It is better to leave this out, and instead start the resulting script as a superuser.

-t *table* | --table=*table*

Dump only tables (or views or sequences) matching the table pattern. Specify the table in the format `schema.table`.

Multiple tables can be selected by writing multiple `-t` switches. Also, the table parameter is interpreted as a pattern according to the same rules used by `psql`'s `\d` commands, so multiple tables can also be selected by writing wildcard characters in the pattern. When using wildcards, be careful to quote the pattern if needed to prevent the shell from expanding the wildcards. The `-n` and `-N` switches have no effect when `-t` is used, because tables selected by `-t` will be dumped regardless of those switches, and non-table objects will not be dumped.

> **Note:** When `-t` is specified, `pg_dump` makes no attempt to dump any other database objects that the selected table(s) may depend upon. Therefore, there is no guarantee that the results of a specific-table dump can be successfully restored by themselves into a clean database.
>
> Also, `-t` cannot be used to specify a child table partition. To dump a partitioned table, you must specify the parent table name.

-T *table* | --exclude-table=*table*

Do not dump any tables matching the table pattern. The pattern is interpreted according to the same rules as for `-t`. `-T` can be given more than once to exclude tables matching any of several patterns. When both `-t` and `-T` are given, the behavior is to dump just the tables that match at least one `-t` switch but no `-T` switches. If `-T` appears without `-t`, then tables matching `-T` are excluded from what is otherwise a normal dump.

`-v | --verbose`

Specifies verbose mode. This will cause `pg_dump` to output detailed object comments and start/stop times to the dump file, and progress messages to standard error.

`-x | --no-privileges | --no-acl`

Prevent dumping of access privileges (`GRANT/REVOKE` commands).

`--disable-dollar-quoting`

This option disables the use of dollar quoting for function bodies, and forces them to be quoted using SQL standard string syntax.

`--disable-triggers`

This option is only relevant when creating a data-only dump. It instructs `pg_dump` to include commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have triggers on the tables that you do not want to invoke during data reload. The commands emitted for `--disable-triggers` must be done as superuser. Specify a superuser name with `-S`, and be careful when starting the script as a superuser. This option is only meaningful for the plain-text format.

For the archive formats, you may specify the option when you call `pg_restore. --use-set-session-authorization`.

Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to determine object ownership. This makes the dump more standards compatible, but depending on the history of the objects in the dump, may not restore properly. A dump using `SET SESSION AUTHORIZATION` will require superuser privileges to restore correctly, whereas `ALTER OWNER` requires lesser privileges.

`--gp-syntax | --no-gp-syntax`

Use `--gp-syntax` to dump HAWQ syntax in the `CREATE TABLE` statements. This preserves the distribution policy (`DISTRIBUTED BY` or `DISTRIBUTED RANDOMLY` clauses) to dump a HAWQ table and to restore into other HAWQ systems. The default is to include HAWQ syntax when connected to a HAWQ system, and to exclude it when connected to a regular PostgreSQL system.

`-Z 0..9 | --compress=0..9`

Specify the compression level to use in archive formats that support compression. Currently only the *custom* archive format supports compression.

Connection Options

`-h` *`host`*`| --host` *`host`*

The host name of the HAWQ master database server. If not specified, reads the name from the environment variable `PGHOST` or defaults to localhost.

`-p` *`port`*`| --port` *`port`*

The TCP port where the HAWQ master database server listens for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

`-U` *`username`*`| --username` *`username`*

The database role name to connect to. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

`-W | --password`

Force a password prompt.

Notes

When a data-only dump is chosen and the option `--disable-triggers` is used, `pg_dump` disables triggers on user tables before inserting the data and re-enables them after the data has been inserted. Stopping restore before it is complete could leave the system catalogs in the wrong state.

Members of `tar` archives are limited to a size less than 8 GB. (This is an inherent limitation of the `tar` file format.) Therefore, this format cannot be used if the textual representation of any one table exceeds that size. The total size of a tar archive and any of the other output formats is not limited, except possibly by the operating system.

The dump file produced by `pg_dump` does not contain the statistics used by the optimizer to make query planning decisions. Therefore, it is wise to run `ANALYZE` after restoring from a dump file to ensure good performance.

Examples

Dump a database called `mydb` into a SQL-script file:

```
pg_dump mydb > db.sql
```

Reload a SQL-script into a new database named `newdb`:

```
psql -d newdb -f db.sql
```

Dump a HAWQ database in tar file format and include distribution policy information:

```
pg_dump -Ft --gp-syntax mydb > db.tar
```

Dump a database into a custom-format archive file:

```
pg_dump -Fc mydb > db.dump
```

Reload an archive file into a new database named `newdb`:

```
pg_restore -d newdb db.dump
```

Dump a single table named `mytab`:

```
pg_dump -t mytab mydb > db.sql
```

To specify an upper-case or mixed-case name in `-t` and related switches, use double-quotes around the name. Otherwise, it will be folded to lower case. Since double quotes are special to the shell, use the following syntax:

```
pg_dump -t '"MixedCaseName"' mydb > mytab.sql
```

See Also

`pg_dumpall`, *`pg_restore`*, *`psql`*

Related Links

Client Utility Reference

pg_dumpall

Extracts all databases in a HAWQ system to a single script file or other archive file.

Synopsis

```
pg_dumpall [connection_option ...] [dump_option ...]
```

Description

`pg_dumpall` is a standard client utility for backing up all databases or HAWQ instances. It creates a single (non-parallel) dump file.

`pg_dumpall` creates a single script file that contains SQL commands. These commands can be used as input to *`psql`* to restore the databases. It does this by calling *`pg_dump`* for each database. It also dumps all the common global database objects.

This script also includes information about database users and groups, and access permissions that apply to databases as a whole.

You must connect as superuser to run `pg_dumpall` because the script reads tables from all databases to produce a complete dump. Also you need superuser privileges to add users and groups, and to create databases. The SQL script is written to the standard output. Shell operators can be used to redirect it into a file.

`pg_dumpall` needs to connect several times to the HAWQ master server once for each database. If you use password authentication you can store the password in a `~/.pgpass` file.

Options

Dump Options

`-a | --data-only`

Dump only the data, not the schema (data definitions). This option only works for the plain-text format. For the archive formats, you can specify the option when you call `pg_restore`.

`-c | --clean`

Output commands to drop database objects before creating them. This option only works for the plain-text format. For the archive formats, you can specify the option when you call `pg_restore`.

`-d | --inserts`

Dump data as `INSERT` commands (rather than `COPY`). This restores the data slowly, but is useful for making dumps to load into non-PostgreSQL-based databases. Also, since this option generates a separate command for each row, an error in reloading a row is limited to a single row. Note that the restore may fail altogether if you have rearranged the column order. Using the `-D` option is safe from column order changes, though it is slower that `-d`.

`-D | --column-inserts | --attribute-inserts`

Dump data as `INSERT` commands with explicit column names `(INSERT INTO` *`table`* `(`*`column`*`, ...) VALUES ...)`. This will make restoration very slow; it is used to load dumps into non-PostgreSQL-based databases. Also, since this option generates a separate command for each row, an error in reloading a row is limited to a single row.

`-f | --filespaces`

Dump filespace definitions.

`-g | --globals-only`

Dump only global objects (roles and tablespaces), no databases.

-i | --ignore-version

Ignore version mismatch between *pg_dump* and the database server. `pg_dump`. `pg_dump` can dump from servers running previous releases of HAWQ, but very old versions may not be supported. Use this option if you need to override the version check.

-o | --oids

Dump object identifiers (OIDs) as part of the data for every table. Pivotal recommends you use this option to restore files into HAWQ.

-O | --no-owner

Do not output commands to set ownership of objects to match the original database.

By default, `pg_dump` issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created database objects. To successfully run this script, you must be a superuser or own the objects in the script. Specify `-O` to make a script that can be restored by any user, and grant them ownership of all the objects. This option is only meaningful for the plain-text format. For the archive formats, you may specify the option when you call `pg_restore`.

-r | --resource-queues

Dump resource queue definitions.

-s | --schema-only

Dump only the object definitions (schema), not data.

-S *username* | --superuser=*username*

Specify the superuser user name to use when disabling triggers. This is only relevant if `--disable-triggers` is used. It is better to omit this option, and instead start the resulting script as a superuser.

-v | --verbose

Specifies verbose mode. This will cause `pg_dump` to output detailed object comments and start/stop times to the dump file, and send progress messages to standard error.

-x | --no-privileges | --no-acl

Prevent dumping of access privileges (`GRANT/REVOKE` commands).

--disable-dollar-quoting

This option disables the use of dollar quoting for function bodies, and forces them to be quoted using SQL standard string syntax.

--disable-triggers

This option is only relevant when creating a data-only dump. It instructs `pg_dumpall` to include commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have triggers on the tables that you do not want to invoke during data reload. The commands emitted for `--disable-triggers` must be performed as superuser. Specify a superuser name with `-S`, and be careful when starting the script as a superuser.

--use-set-session-authorization

Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to determine object ownership. This makes the dump more standards compatible, but depending on the history of the objects in the dump, may not restore properly. A dump using `SET SESSION AUTHORIZATION` will require superuser privileges to restore correctly, whereas `ALTER OWNER` requires lesser privileges.

--gp-syntax

Output HAWQ syntax in the `CREATE TABLE` statements. This allows the distribution policy (`DISTRIBUTED BY` or `DISTRIBUTED RANDOMLY` clauses) of a HAWQ table to be dumped, which is useful for restoring into other HAWQ systems.

Connection Options

`-h` *`host`* `|` `--host` *`host`*

The host name of the HAWQ master database server. If not specified, reads from the environment variable `PGHOST` or defaults to `localhost`.

`-p` *`port`* `|` `--port` *`port`*

The TCP port where the HAWQ master database server listens for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

`-U` *`username`* `|` `--username` *`username`*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

`-W | --password`

Force a password prompt.

Notes

Since `pg_dumpall` calls *`pg_dump`* internally, some diagnostic messages will refer to `pg_dump`.

Once a database is restored, it is wise to run `ANALYZE` on each database so the query planner has useful statistics. You can also run `vacuumdb -a -z` to analyze all databases.

To restore or create databases in non-default locations while using `pg_dumpall`, check that you have all the necessary tablespace, filespace, and directories.

Examples

Dump all databases:

```
pg_dumpall > db.out
```

Reload the specified file:

```
psql template1 -f db.out
```

Dump only global objects (including filespaces and resource queues):

```
pg_dumpall -g -f -r
```

See Also

`pg_dump`

Related Links

Client Utility Reference

pg_restore

Restores a database from an archive file created by `pg_dump`.

Synopsis

```
pg_restore [connection_option ...] [restore_option ...] filename
```

Description

`pg_restore` is a utility for restoring a database from an archive created by *`pg_dump`* in one of the non-plain-text formats. `pg_restore` reconstructs the database to the state it was in at the time it was saved. The archive files also allow `pg_restore` to be selective about what is restored, or even to reorder the items prior to being restored.

`pg_restore` can operate in two modes. If a database name is specified, the archive is restored directly into the database. Otherwise, a script containing the SQL commands necessary to rebuild the database is created and written to a file or standard output. The script output is equivalent to the plain text output format of `pg_dump`. Some of the options controlling the output are therefore analogous to `pg_dump` options.

`pg_restore` cannot restore information that is not present in the archive file. For instance, if the archive was made using the "dump data as `INSERT` commands" option, `pg_restore` will not be able to load the data using `COPY` statements.

Options

filename

Specifies the location of the archive file to be restored. If not specified, the standard input is used.

Restore Options

-a | --data-only

Restore only the data, not the schema (data definitions).

-c | --clean

Clean (drop) database objects before recreating them.

-C | --create

Create the database before restoring into it. (When this option is used, the database named with `-d` is used only to issue the initial `CREATE DATABASE` command. All data is restored into the database name that appears in the archive.)

-d *dbname* | --dbname=*dbname*

Connect to this database and restore directly into this database. The default is to use the `PGDATABASE` environment variable setting, or the same name as the current system user.

-e | --exit-on-error

Exit if an error is encountered while sending SQL commands to the database. The default is to continue and to display a count of errors at the end of the restoration.

-f *outfilename* | --file=*outfilename*

Specify output file for generated script, or for the listing when used with `-l`. Default is the standard output.

-F t |c | --format=tar | custom

The format of the archive produced by *`pg_dump`*. It is not necessary to specify the format, since `pg_restore` will determine the format automatically. Format can be either `tar` or `custom`.

`-i | --ignore-version`

Ignore database version checks.

`-I` ***`index`*** **`| --index=`*****`index`***

Restore definition of named index only.

`-l | --list`

List the contents of the archive. The output of this operation can be used with the `-L` option to restrict and reorder the items that are restored.

`-L` ***`list-file`*** **`| --use-list=`*****`list-file`***

Restore elements in the *list-file* only. Lines can be moved and may also be commented out by placing a `;` at the start of the line.

`-n` ***`schema`*** **`| --schema=`*****`schema`***

Restore only objects that are in the named schema. This can be combined with the `-t` option to restore a specific table.

`-O | --no-owner`

Do not output commands to set ownership of objects to match the original database.

By default, `pg_restore` issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created schema elements. These statements will fail unless you connect to the database as a superuser, or own the objects in the script. With `-O`, any user name can be used for the initial connection, and this user will own all the created objects.

`-P '`***`function-name`*****`(`*****`argtype`*****` [, ...])' | --function='`*****`function-name`*****`(`*****`argtype`*****` [, ...])'`**

Restore the named function only. The function name must be enclosed in quotes. Be careful to spell the function name and arguments exactly as they appear in the dump file's table of contents (as shown by the `--list` option).

`-s | --schema-only`

Restore only the schema (data definitions), not the data (table contents). Sequence current values will not be restored, either. (Do not confuse this with the `--schema` option, which uses the word schema in a different meaning.)

`-S` ***`username`*** **`| --superuser=`*****`username`***

Specify the superuser user name to use when disabling triggers. This is only relevant if `--disable-triggers` is used.

`-t` ***`table`*** **`| --table=`*****`table`***

Restore definition and/or data of named table only.

`-T` ***`trigger`*** **`| --trigger=`*****`trigger`***

Restore named trigger only.

`-v | --verbose`

Specifies verbose mode.

`-x | --no-privileges | --no-acl`

Prevent restoration of access privileges (`GRANT/REVOKE` commands).

`--disable-triggers`

This option is only relevant when performing a data-only restore. It instructs `pg_restore` to execute commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have triggers on the tables that you do not want to invoke during data reload. You must be superuser to issue the `--disable-triggers` command. So, you should also specify a superuser name with `-S`, or preferably run `pg_restore` as a superuser.

`--no-data-for-failed-tables`

By default, table data is restored even if the creation command for the table failed (e.g., because it already exists). With this option, data for such a table is skipped.

This behavior is useful when the target database may already contain the desired table contents. Specifying this option prevents duplicate or obsolete data from being loaded. This option is effective only when restoring directly into a database, not when producing SQL script output.

Connection Options

`-h` ***`host`*** **`| --host`** ***`host`***

The host name of the HAWQ master database server.If not specified, reads from the environment variable `PGHOST` or defaults to localhost.

`-p` ***`port`*** **`| --port`** ***`port`***

The TCP port where the HAWQ master database server listens for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

`-U` ***`username`*** **`| --username`** ***`username`***

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

`-W | --password`

Force a password prompt.

`-1 | --single-transaction`

Execute the restore as a single transaction. This ensures that either all the commands complete successfully, or no changes are applied.

Notes

If your installation has any local additions to the `template1` database, load the output of `pg_restore` into an empty database; otherwise you will see errors for duplicate definitions of the added objects. To make an empty database without any local additions, copy from `template0` not `template1`. For example:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

When restoring data to a pre-existing table and the option `--disable-triggers` is used, `pg_restore` emits commands to disable triggers on user tables before inserting the data then emits commands to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs could be left in the wrong state.

`pg_restore` will not restore large objects for a single table. If an archive contains large objects, then all large objects will be restored.

See also the `pg_dump` documentation for details on limitations of `pg_dump`.

Once restored, it is wise to run `ANALYZE` on each restored table so the query planner has useful statistics.

Examples

Assume we have dumped a database called `mydb` into a custom-format dump file:

```
pg_dump -Fc mydb > db.dump
```

Drop the database and recreate it from the dump:

```
dropdb mydb
pg_restore -C -d template1 db.dump
```

Reload the dump into a new database called `newdb`. Notice there is no `-C`, we instead connect directly to the database to be restored into. Also note that we clone the new database from `template0` not `template1`, to ensure it is initially empty:

```
createdb -T template0 newdb
pg_restore -d newdb db.dump
```

Reorder database items, it is first necessary to dump the table of contents of the archive:

```
pg_restore -l db.dump > db.list
```

The listing file consists of a header and one line for each item. For example,

```
; Archive created at Fri Jul 28 22:28:36 2006
;     dbname: mydb
;     TOC Entries: 74
;     Compression: 0
;     Dump Version: 1.4-0
;     Format: CUSTOM
;
; Selected TOC Entries:
;
2; 145344 TABLE species postgres
3; 145344 ACL species
4; 145359 TABLE nt_header postgres
5; 145359 ACL nt_header
6; 145402 TABLE species_records postgres
7; 145402 ACL species_records
8; 145416 TABLE ss_old postgres
9; 145416 ACL ss_old
10; 145433 TABLE map_resolutions postgres
11; 145433 ACL map_resolutions
12; 145443 TABLE hs_old postgres
13; 145443 ACL hs_old
```

Semicolons start a comment, and the numbers at the start of lines refer to the internal archive ID assigned to each item. Lines in the file can be commented out, deleted, and reordered. For example:

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

Could be used as input to `pg_restore` and would only restore items 10 and 6, in that order:

```
pg_restore -L db.list db.dump
```

See Also

pg_dump

Related Links

Client Utility Reference

psql

Interactive command-line interface for HAWQ

Synopsis

```
psql [option ...] [dbname [username]]
```

Description

`psql` is a terminal-based front-end to HAWQ. It enables you to type in queries interactively, issue them to HAWQ, and see the query results. Alternatively, input can be from a file. In addition, it provides a number of meta-commands and various shell-like features to facilitate writing scripts and automating a wide variety of tasks.

Options

-a | --echo-all

Print all input lines to standard output as they are read. This is more useful for script processing rather than interactive mode.

-A | --no-align

Switches to unaligned output mode. (The default output mode is aligned.)

-c '*command*' | --command '*command*'

Specifies that `psql` is to execute the specified command string, and then exit. This is useful in shell scripts. *command* must be either a command string that is completely parseable by the server, or a single backslash command. Thus you cannot mix SQL and `psql` meta-commands with this option. To achieve that, you could pipe the string into `psql`, for example:

```
echo '\x \\ SELECT * FROM foo;' | psql
```

(`\\` is the separator meta-command.)

If the command string contains multiple SQL commands, they are processed in a single transaction, unless there are explicit `BEGIN/COMMIT` commands included in the string to divide it into multiple transactions. This is different from the behavior when the same string is fed to `psql`'s standard input.

-d *dbname* | --dbname *dbname*

Specifies the name of the database to connect to. This is equivalent to specifying dbname as the first non-option argument on the command line.

If this parameter contains an equals sign, it is treated as a `conninfo` string; for example you can pass `'dbname=hawq user=username password=mypass'` as `dbname`.

-e | --echo-queries

Copy all SQL commands sent to the server to standard output as well.

-E | --echo-hidden

Echo the actual queries generated by `\d` and other backslash commands. You can use this to study `psql`'s internal operations.

-f *filename* | --file *filename*

Use a file as the source of commands instead of reading commands interactively. After the file is processed, `psql` terminates. If *filename* is `-` (hyphen), then standard input is read.

Using this option is subtly different from writing `psql <`*filename*. In general, both will do what you expect, but using `-f` enables some useful features such as error messages with line numbers.

`-F` ***`separator`*** **`| --field-separator`** ***`separator`***

Use the specified separator as the field separator for unaligned output.

`-H | --html`

Turn on HTML tabular output.

`-l | --list`

List all available databases, then exit. Other non-connection options are ignored.

`-L` ***`filename`*** **`| --log-file`** ***`filename`***

Write all query output into the specified log file, in addition to the normal output destination.

`-o` ***`filename`*** **`| --output`** ***`filename`***

Put all query output into the specified file.

`-P` ***`assignment`*** **`| --pset`** ***`assignment`***

Allows you to specify printing options in the style of `\pset` on the command line.

Note that here you have to separate name and value with an equal sign instead of a space. Thus to set the output format to LaTeX, you could write `-P format=latex`.

`-q | --quiet`

Specifies that `psql` should do its work quietly. By default, it prints welcome messages and various informational output. If this option is used, none of this happens. This is useful with the `-c` option.

`-R` ***`separator`*** **`| --record-separator`** ***`separator`***

Use *separator* as the record separator for unaligned output.

`-s | --single-step`

Run in single-step mode. That means the user is prompted before each command is sent to the server, with the option to cancel execution as well. Use this to debug scripts.

`-S | --single-line`

Runs in single-line mode where a new line terminates an SQL command, as a semicolon does.

`-t | --tuples-only`

Turn off printing of column names and result row count footers, etc. This command is equivalent to `\pset tuples_only` and is provided for convenience.

`-T` ***`table_options`*** **`| --table-attr`** ***`table_options`***

Allows you to specify options to be placed within the HTML table tag. See `\pset` for details.

`-v` ***`assignment`*** **`| --set`** ***`assignment`*** **`| --variable`** ***`assignment`***

Perform a variable assignment, like the `\set` internal command. Note that you must separate name and value, if any, by an equal sign on the command line. To unset a variable, leave off the equal sign. To just set a variable without a value, use the equal sign but leave off the value. These assignments are done during a very early stage of start-up, so variables reserved for internal purposes could get overwritten later.

`-V | --version`

Print the `psql` version and exit.

`-x | --expanded`

Turn on the expanded table formatting mode.

`-X | --no-psqlrc`

Do not read the start-up file (neither the system-wide `psqlrc` file nor the user's `~/.psqlrc` file).

`-1 | --single-transaction`

When `psql` executes a script with the `-f` option, adding this option wraps `BEGIN/COMMIT` around the script to execute it as a single transaction. This ensures that either all the commands complete successfully, or no changes are applied.

If the script itself uses `BEGIN`, `COMMIT`, or `ROLLBACK`, this option will not have the desired effects. Also, if the script contains any command that cannot be executed inside a transaction block, specifying this option will cause that command (and hence the whole transaction) to fail.

`-? | --help`

Show help about `psql` command line arguments, and exits.

Connection Options

`-h` ***`host`*** **`| --host`** ***`host`***

The host name of the HAWQ master database server. If not specified, reads from the environment variable `PGHOST` or defaults to localhost.

`-p` ***`port`*** **`| --port`** ***`port`***

The TCP port where the HAWQ master database server listens for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

`-U` ***`username`*** **`| --username`** ***`username`***

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system role name.

`-W | --password`

Pivotal recommends using this option to force a password prompt. If no password prompt is issued and the server requires password authentication, the connection attempt will fail.

`-w --no-password`

Does not issue a password prompt. If the server requires password authentication, storing the password a .pgpass file ensures a successful connection when running batch jobs and scripts.

Note: This option remains set for the entire session, and so it affects uses of the meta-command `\connect` as well as the initial connection attempt.

Exit Status

`psql` returns the following values:

- 0 to the shell if it finished normally.
- 1 if a psql fatal error (out of memory, file not found) occurs.
- 2 if the connection to the server went bad and the session was not interactive.
- 3 if an error occurred in a script and the variable `ON_ERROR_STOP` was set.

Usage

Connecting to a Database

`psql` is a client application for HAWQ. To connect to a database you need to know the following information:

- Name of your target database
- Host name and port number of the HAWQ master server
- Database user name

Those parameters can be communicated to `psql` via command line options, namely `-d`, `-h`, `-p`, and `-U`, respectively. If an argument is found that does not belong to any option, it is interpreted as the database name (or the user name, if the database name is already given).

Not all these options are required; there are useful defaults. If you omit the host name, `psql` will connect via a UNIX-domain socket to a master server on the local host, or via TCP/IP to `localhost` on machines that do not have UNIX-domain sockets. The default master port number is 5432. If you use a different port for the master, you must specify the port. The default database user name is your UNIX user name, as is the default database name. Note that you cannot just connect to any database under any user name. Your database administrator should have informed you about your access rights.

When the defaults are not right, you can save yourself some typing by setting any or all of the environment variables `PGAPPNAME`, `PGDATABASE`, `PGHOST`, `PGPORT`, and `PGUSER` to appropriate values.

It is also convenient to have a `~/.pgpass` file to avoid regularly having to type in passwords. This file should reside in your home directory and contain lines of the following format:

```
hostname:port:database:username:password
```

The permissions on `.pgpass` must disallow any access to world or group (for example: `chmod 0600 ~/.pgpass`). If the permissions are less strict than this, the file will be ignored. (The file permissions are not currently checked on Microsoft Windows clients, however.)

If the connection could not be made for any reason (insufficient privileges, server is not running, etc.), `psql` will return an error and terminate.

Entering SQL Commands

In normal operation, `psql` provides a prompt with the name of the database to which `psql` is currently connected, followed by the string **=>** for a regular user or **=#** for a superuser. For example:

```
testdb=>
testdb=#
```

At the prompt, the user may type in SQL commands. Ordinarily, input lines are sent to the server when a command-terminating semicolon is reached. An end of line does not terminate a command. Thus commands can be spread over several lines for clarity. If the command was sent and executed without error, the results of the command are displayed on the screen.

Meta-Commands

Anything you enter in `psql` that begins with an unquoted backslash is a `psql` meta-command that is processed by `psql` itself. These commands help make `psql` more useful for administration or scripting. Meta-commands are more commonly called slash or backslash commands.

The format of a `psql` command is the backslash, followed immediately by a command verb, then any arguments. The arguments are separated from the command verb and each other by any number of whitespace characters.

To include whitespace into an argument you may quote it with a single quote. To include a single quote into such an argument, use two single quotes. Anything contained in single quotes is furthermore subject to C-like substitutions for `\n` (new line), `\t` (tab), `\digits` (octal), and `\xdigits` (hexadecimal).

If an unquoted argument begins with a colon (`:`), it is taken as a `psql` variable and the value of the variable is used as the argument instead.

Arguments that are enclosed in backquotes (`` ` ``) are taken as a command line that is passed to the shell. The output of the command (with any trailing newline removed) is taken as the argument value. The above escape sequences also apply in backquotes.

Some commands take an SQL identifier (such as a table name) as argument. These arguments follow the syntax rules of SQL: Unquoted letters are forced to lowercase, while double quotes (`"`) protect letters from case conversion and allow incorporation of whitespace into the identifier. Within double quotes, paired double quotes reduce to a single double quote in the resulting name. For example, `FOO"BAR"BAZ` is interpreted as `fooBARbaz`, and `"A weird"" name"` becomes `A weird" name`.

Parsing for arguments stops when another unquoted backslash occurs. This is taken as the beginning of a new meta-command. The special sequence `\\` (two backslashes) marks the end of arguments and continues parsing SQL commands, if any. That way SQL and `psql` commands can be freely mixed on a line. But in any case, the arguments of a meta-command cannot continue beyond the end of the line.

The following meta-commands are defined:

`\a`

If the current table output format is unaligned, it is switched to aligned. If it is not unaligned, it is set to unaligned. This command is kept for backwards compatibility. See `\pset` for a more general solution.

`\cd [`*`directory`*`]`

Changes the current working directory. Without argument, changes to the current user's home directory. To print your current working directory, use `\!pwd`.

`\C [`*`title`*`]`

Sets the title of any tables being printed as the result of a query or unset any such title. This command is equivalent to `\pset title`.

`\c | \connect [`*`dbname`*` [`*`username`*`] [`*`host`*`] [`*`port`*`]]`

Establishes a new connection. If the new connection is successfully made, the previous connection is closed. If any of dbname, username, host or port are omitted, the value of that parameter from the previous connection is used. If the connection attempt failed, the previous connection will only be kept if `psql` is in interactive mode. When executing a non-interactive script, processing will immediately stop with an error. This distinction was chosen as a user convenience against typos, and a safety mechanism that scripts are not accidentally acting on the wrong database.

`\conninfo`

Displays information about the current connection including the database name, the user name, the type of connection (UNIX domain socket, `TCP/IP`, etc.), the host, and the port.

`\copy {`*`table`*` [(`*`column_list`*`)] | (`*`query`*`)} {from | to} {`*`filename`*` | stdin | stdout | pstdin | pstdout} [with] [binary] [oids] [delimiter [as] '`*`character`*`'] [null [as] '`*`string`*`'] [csv [header] [quote [as] 'character'] [escape [as] '`*`character`*`'] [force quote column_list] [force not null column_list]]`

Performs a frontend (client) copy. This is an operation that runs an SQL `COPY` command, but instead of the server reading or writing the specified file, `psql` reads or writes the file and routes the data between the server and the local file system. This means that file accessibility and privileges are those of the local user, not the server, and no SQL superuser privileges are required.

The syntax of the command is similar to that of the SQL `COPY` command. Note that, because of this, special parsing rules apply to the `\copy` command. In particular, the variable substitution rules and backslash escapes do not apply.

`\copy ... from stdin | to stdout` reads/writes based on the command input and output respectively. All rows are read from the same source that issued the command, continuing until `\.` is read or the stream reaches `EOF`. Output is sent to the same place as command output. To read/write from `psql`'s standard input or output, use `pstdin` or `pstdout`. This option is useful for populating tables in-line within a SQL script file.

This operation is not as efficient as the SQL `COPY` command because all data must pass through the client/server connection.

\copyright

Shows the copyright and distribution terms of PostgreSQL on which HAWQ is based.

\d [*relation_pattern*] | \d+ [*relation_pattern*] | \dS [*relation_pattern*]

For each relation (table, external table, view, index, or sequence) matching the relation pattern, show all columns, their types, the tablespace (if not the default) and any special attributes such as `NOT NULL` or defaults, if any. Associated indexes, constraints, rules, and triggers are also shown, as is the view definition if the relation is a view.

- The command form `\d+` is identical, except that more information is displayed: any comments associated with the columns of the table are shown, as is the presence of OIDs in the table.
- The command form `\dS` is identical, except that system information is displayed as well as user information. For example, `\dt` displays user tables, but not system tables; `\dtS` displays both user and system tables. Both these commands can take the `+` parameter to display additional information, as in `\dt+` and `\dtS+`.

  If `\d` is used without a pattern argument, it is equivalent to `\dtvs` which will show a list of all tables, views, and sequences.

\da [*aggregate_pattern*]

Lists all available aggregate functions, together with the data types they operate on. If a pattern is specified, only aggregates whose names match the pattern are shown.

\db [*tablespace_pattern*] | \db+ [*tablespace_pattern*]

Lists all available tablespaces and their corresponding filespace locations. If pattern is specified, only tablespaces whose names match the pattern are shown. If + is appended to the command name, each object is listed with its associated permissions.

\dc [*conversion_pattern*]

Lists all available conversions between character-set encodings. If pattern is specified, only conversions whose names match the pattern are listed.

\dC

Lists all available type casts.

\dd [*object_pattern*]

Lists all available objects. If pattern is specified, only matching objects are shown.

\dD [*domain_pattern*]

Lists all available domains. If pattern is specified, only matching domains are shown.

\df [*function_pattern*] | \df+ [*function_pattern*]

Lists available functions, together with their argument and return types. If pattern is specified, only functions whose names match the pattern are shown. If the form `\df+` is used, additional information about each function, including language and description, is shown. To reduce clutter, `\df` does not show data type I/O functions. This is implemented by ignoring functions that accept or return type `cstring`.

\dg [*role_pattern*]

Lists all database roles. If pattern is specified, only those roles whose names match the pattern are listed.

\distPvxS [*index* | *sequence* | *table* | *parent table* | *view* | *external_table* | *system_object*]

This is not the actual command name: the letters `i`, `s`, `t`, `P`, `v`, `x`, `S` stand for index, sequence, table, parent table, view, external table, and system table, respectively. You can specify any or all of these letters, in any order, to obtain a listing of all the matching objects. The letter `S` restricts the listing to system objects; without `S`, only non-system objects are shown. If + is appended to the command name, each object is listed with its associated description, if any. If a pattern is specified, only objects whose names match the pattern are listed.

\dl

This is an alias for `\lo_list`, which shows a list of large objects.

\do [*operator_pattern*]

Lists available operators with their operand and return types. If pattern is specified, only operators whose names match the pattern are listed.

\dp [*relation_pattern_to_show_privileges*]

Produces a list of all available tables, views and sequences with their associated access privileges. If pattern is specified, only tables, views and sequences whose names match the pattern are listed. The `GRANT` and `REVOKE` commands are used to set access privileges.

\dT [*datatype_pattern*] | \dT+ [*datatype_pattern*]

Lists all data types or only those that match pattern. The command form `\dT+` shows extra information.

\du [*role_pattern*]

Lists all database roles, or only those that match pattern.

\e | \edit [*filename*]

If a file name is specified, the file is edited; after the editor exits, its content is copied back to the query buffer. If no argument is given, the current query buffer is copied to a temporary file which is then edited in the same fashion. The new query buffer is then re-parsed according to the normal rules of `psql`, where the whole buffer is treated as a single line. (Thus you cannot make scripts this way. Use `\i` for that.) This means also that if the query ends with (or rather contains) a semicolon, it is immediately executed. In other cases it will merely wait in the query buffer.

`psql` searches the environment variables `PSQL_EDITOR`, `EDITOR`, and `VISUAL` (in that order) for an editor to use. If all of them are unset, `vi` is used on UNIX systems, `notepad.exe` on Windows systems.

\echotext [...]

Prints the arguments to the standard output, separated by one space and followed by a newline. This can be useful to intersperse information in the output of scripts.

If you use the `\o` command to redirect your query output you may wish to use `'echo` instead of this command.

\encoding [*encoding*]

Sets the client character set encoding. Without an argument, this command shows the current encoding.

\f [*field_separator_string*]

Sets the field separator for unaligned query output. The default is the vertical bar (`|`). See also `\pset` for a generic way of setting output options.

\g [{*filename* | |*command* }]

Sends the current query input buffer to the server and optionally stores the query's output in a file or pipes the output into a separate UNIX shell executing command. A bare `\g` is virtually equivalent to a semicolon. A `\g` with argument is a one-shot alternative to the `\o` command.

\h | \help [*sql_command*]

Gives syntax help on the specified SQL command. If a command is not specified, then `psql` will list all the commands for which syntax help is available. Use an asterisk (*) to show syntax help on all SQL commands. To simplify typing, commands that consists of several words do not have to be quoted.

`\H`

Turns on HTML query output format. If the HTML format is already on, it is switched back to the default aligned text format. This command is for compatibility and convenience, but see `\pset` about setting other output options.

`\i` ***`input_filename`***

Reads input from a file and executes it as though it had been typed on the keyboard. If you want to see the lines on the screen as they are read you must set the variable `ECHO` to all.

`\l | \list | \l+ | \list+`

List the names, owners, and character set encodings of all the databases in the server. If `+` is appended to the command name, database descriptions are also displayed.

`\lo_export` ***`loid filename`***

Reads the large object with OID *loid* from the database and writes it to *filename*. Note that this is subtly different from the server function `lo_export`, which acts with the permissions of the user that the database server runs as and on the server's file system. Use `\lo_list` to find out the large object's OID.

`\lo_import` ***`large_object_filename`*** **`[`*`comment`*`]`**

Stores the file into a large object. Optionally, it associates the given comment with the object. Example:

```
mydb=> \lo_import '/home/gpadmin/pictures/photo.xcf' 'a
picture of me'
lo_import 152801
```

The response indicates that the large object received object ID 152801 which one ought to remember if one wants to access the object ever again. For that reason it is recommended to always associate a human-readable comment with every object. Those can then be seen with the `\lo_list` command. Note that this command is subtly different from the server-side `lo_import` because it acts as the local user on the local file system, rather than the server's user and file system.

`\lo_list`

Shows a list of all large objects currently stored in the database, along with any comments provided for them.

`\lo_unlink` ***`largeobject_oid`***

Deletes the large object of the specified OID from the database. Use `\lo_list` to find out the large object's OID.

`\o [ {`*`query_result_filename`*` | |`*`command`*`} ]`

Saves future query results to a file or pipes them into a UNIX shell command. If no arguments are specified, the query output will be reset to the standard output. Query results include all tables, command responses, and notices obtained from the database server, as well as output of various backslash commands that query the database (such as `\d`), but not error messages. To intersperse text output in between query results, use `'echo`.

`\p`

Print the current query buffer to the standard output.

`\password [`*`username`*`]`

Changes the password of the specified user (by default, the current user). This command prompts for the new password, encrypts it, and sends it to the server as an `ALTER ROLE` command. This makes sure that the new password does not appear in cleartext in the command history, the server log, or elsewhere.

`\prompt [` ***`text`*** **`]`** ***`name`***

Prompts the user to set a variable *name*. Optionally, you can specify a prompt. Enclose prompts longer than one word in single quotes.

By default, `\prompt` uses the terminal for input and output. However, use the `-f` command line switch to specify standard input and standard output.

`\pset` *`print_option`* `[`*`value`*`]`

This command sets options affecting the output of query result tables. *print_option* describes which option is to be set. Adjustable printing options are:

- **`format`** – Sets the output format to one of **u**`naligned`, **a**`ligned`, **h**`tml`, **l**`atex`, **t**`roff-ms`, or **w**`rapped`. First letter abbreviations are allowed. Unaligned writes all columns of a row on a line, separated by the currently active field separator. This is intended to create output that might be intended to be read in by other programs. Aligned mode is the standard, human-readable, nicely formatted text output that is default. The HTML and LaTeX modes put out tables that are intended to be included in documents using the respective mark-up language. They are not complete documents! (This might not be so dramatic in HTML, but in LaTeX you must have a complete document wrapper.)

  The wrapped option sets the output format like the `aligned` parameter , but wraps wide data values across lines to make the output fit in the target column width. The target width is set with the `columns` option. To specify the column width and select the wrapped format, use two \pset commands; for example, to set the with to 72 columns and specify wrapped format, use the commands `\pset columns 72` and then `\pset format wrapped`.

  > **Note:** Since `psql` does not attempt to wrap column header titles, the wrapped format behaves the same as aligned if the total width needed for column headers exceeds the target.

- **`border`** – The second argument must be a number. In general, the higher the number the more borders and lines the tables will have, but this depends on the particular format. In HTML mode, this will translate directly into the `border=...` attribute, in the others only values `0` (no border), `1` (internal dividing lines), and `2` (table frame) make sense.
- **`columns`** – Sets the target width for the `wrapped` format, and also the width limit for determining whether output is wide enough to require the pager. The default is *zero*. Zero causes the target width to be controlled by the environment variable `COLUMNS`, or the detected screen width if `COLUMNS` is not set. In addition, if `columns` is zero then the wrapped format affects screen output only. If columns is nonzero then file and pipe output is wrapped to that width as well.

  After setting the target width, use the command `\pset format wrapped` to enable the wrapped format.

- **`expanded`** | **`x)`** – Toggles between regular and expanded format. When expanded format is enabled, query results are displayed in two columns, with the column name on the left and the data on the right. This mode is useful if the data would not fit on the screen in the normal horizontal mode. Expanded mode is supported by all four output formats.
- **`linestyle`** [**`unicode`** | **`ascii`** | **`old-ascii`**] – Sets the border line drawing style to one of unicode, ascii, or old-ascii. Unique abbreviations, including one letter, are allowed for the three styles. The default setting is `ascii`. This option only affects the `aligned` and `wrapped` output formats.

  `ascii` – uses plain ASCII characters. Newlines in data are shown using a + symbol in the right-hand margin. When the wrapped format wraps data from one line to the next without a newline character, a dot (.) is shown in the right-hand margin of the first line, and again in the left-hand margin of the following line.

`old-ascii` – style uses plain ASCII characters. Newlines in data are shown using a **`:`** symbol in place of the left-hand column separator.

When the data is wrapped from one line to the next without a newline character, a **`;`** symbol is used in place of the left-hand column separator.

`unicode` – style uses Unicode box-drawing characters. Newlines in data are shown using a carriage return symbol in the right-hand margin. When the data is wrapped from one line to the next without a newline character, an ellipsis symbol is shown in the right-hand margin of the first line, and again in the left-hand margin of the following line.

When the `border` setting is greater than zero, this option also determines the characters with which the border lines are drawn. Plain ASCII characters work everywhere, but Unicode characters look nicer on displays that recognize them.

- **`null 'string'`** – The second argument is a string to print whenever a column is null. The default is not to print anything, which can easily be mistaken for an empty string. For example, the command `\psetnull '(empty)'` displays *(empty)* in null columns.
- **`fieldsep`** – Specifies the field separator to be used in unaligned output mode. That way one can create, for example, tab- or comma-separated output, which other programs might prefer. To set a tab as field separator, type `\pset fieldsep '\t'`. The default field separator is `'|'` (a vertical bar).
- **`footer`** – Toggles the display of the default footer (*x* rows).
- **`numericlocale`** – Toggles the display of a locale-aware character to separate groups of digits to the left of the decimal marker. It also enables a locale-aware decimal marker.
- **`recordsep`** – Specifies the record (line) separator to use in unaligned output mode. The default is a newline character.
- **`title`** [*text*] – Sets the table title for any subsequently printed tables. This can be used to give your output descriptive tags. If no argument is given, the title is unset.
- **`tableattr`** | **`T`** [*text*] – Allows you to specify any attributes to be placed inside the HTML table tag. This could for example be `cellpadding` or `bgcolor`. Note that you probably don't want to specify border here, as that is already taken care of by `\pset border`.
- **`tuples_only`** | **`t`** [*novalue* | *on* | *off*] – The `\pset tuples_only` command by itselt toggles between tuples only and full display. The values *on* and *off* set the tuples display, regardless of the current setting. Full display may show extra information such as column headers, titles, and various footers. In tuples only mode, only actual table data is shown The `\t` command is equivalent to `\psettuples_only` and is provided for convenience.
- **`pager`** – Controls the use of a pager for query and `psql` help output. When `on`, if the environment variable `PAGER` is set, the output is piped to the specified program. Otherwise a platform-dependent default (such as `more`) is used. When `off`, the pager is not used. When `on`, the pager is used only when appropriate. Pager can also be set to `always`, which causes the pager to be always used.

`\q`

Quits the `psql` program.

`\qechotext [ ... ]`

This command is identical to `\echo` except that the output will be written to the query output channel, as set by `\o`.

`\r`

Resets (clears) the query buffer.

`\s [`*`history_filename`*`]`

Print or save the command line history to *filename*. If *filename* is omitted, the history is written to the standard output.

`\set [`*`name`*` [`*`value`*` [ ... ]]]`

Sets the internal variable *name* to *value* or, if more than one value is given, to the concatenation of all of them. If no second argument is given, the variable is just set with no value. To unset a variable, use the `\unset` command.

Valid variable names can contain characters, digits, and underscores. Variable names are case-sensitive.

Although you are welcome to set any variable to anything you want, `psql` treats several variables as special. They are documented in the topic about variables.

This command is totally separate from the SQL command `SET`.

`\t [novalue | on | off]`

The `\t` command by itself toggles a display of output column name headings and row count footer. The values `on` and `off` set the tuples display, regardless of the current setting.This command is equivalent to `\pset tuples_only` and is provided for convenience.

`\T `*`table_options`*

Allows you to specify attributes to be placed within the table tag in HTML tabular output mode.

`\timing [novalue | on | off]`

The `\timing` command by itself toggles a display of how long each SQL statement takes, in milliseconds. The values `on` and `off` set the time display, regardless of the current setting.

`\w {`*`filename`*` | |`*`command`*`}`

Outputs the current query buffer to a file or pipes it to a UNIX command.

`\x`

Toggles expanded table formatting mode.

`\z [`*`relation_to_show_privileges`*`]`

Produces a list of all available tables, views and sequences with their associated access privileges. If a pattern is specified, only tables, views and sequences whose names match the pattern are listed. This is an alias for `\dp`.

`\! [`*`command`*`]`

Escapes to a separate UNIX shell or executes the UNIX command. The arguments are not further interpreted, the shell will see them as is.

`\?`

Shows help information about the `psql` backslash commands.

Patterns

The various `\d` commands accept a pattern parameter to specify the object name(s) to be displayed. In the simplest case, a pattern is just the exact name of the object. The characters within a pattern are normally folded to lower case, just as in SQL names; for example, `\dt FOO` will display the table named `foo`. As in SQL names, placing double quotes around a pattern stops folding to lower case. Should you need to include an actual double quote character in a pattern, write it as a pair of double quotes within a double-quote sequence; again this is in accord with the rules for SQL quoted identifiers. For example, `\dt "FOO""BAR"` will display the table named `FOO"BAR` (not `foo"bar`). Unlike the normal rules for SQL names, you can put double quotes around just part of a pattern, for instance `\dt FOO"FOO"BAR` will display the table named `fooFOObar`.

Within a pattern, `*` matches any sequence of characters (including no characters) and `?` matches any single character. (This notation is comparable to UNIX shell file name patterns.) For example, `\dt int*` displays all tables whose names begin with `int`. But within double quotes, `*` and `?` lose these special meanings and are just matched literally.

A pattern that contains a dot (`.`) is interpreted as a schema name pattern followed by an object name pattern. For example, `\dt foo*.bar*` displays all tables whose table name starts with `bar` that are in schemas whose schema name starts with `foo`. When no dot appears, then the pattern matches only objects that are visible in the current schema search path. Again, a dot within double quotes loses its special meaning and is matched literally.

Advanced users can use regular-expression notations. The `.` character is taken as a separator. The `*` is translated to the regular-expression notation `.*`. The `?` is translated to `.`. You can emulate these pattern characters at need by writing `?` for `.`, `(R+|)` for `R*`, or `(R|)` for `R?`. Remember that the pattern must match the whole name, unlike the usual interpretation of regular expressions; write `*` at the beginning and/or end if you don't wish the pattern to be anchored. Note that within double quotes, all regular expression special characters lose their special meanings and are matched literally. Also, the regular expression special characters are matched literally in operator name patterns (such as the argument of `\do`).

Whenever the pattern parameter is omitted completely, the `\d` commands display all objects that are visible in the current schema search path – this is equivalent to using the pattern `*`. To see all objects in the database, use the pattern `*.*`.

Advanced Features

Variables

`psql` provides variable substitution features similar to common UNIX command shells. Variables are simply name/value pairs, where the value can be any string of any length. To set variables, use the `psql` meta-command `\set`:

```
testdb=> \set foo bar
```

This sets the variable `foo` to the value `bar`. To retrieve the content of the variable, precede the name with a colon and use it as the argument of any slash command:

```
testdb=> \echo :foo
bar
```

Note: The arguments of `\set` are subject to the same substitution rules as with other commands. Thus you can construct interesting references such as `\set :foo 'something'` and get 'soft links' or 'variable variables' of Perl or PHP fame, respectively. Unfortunately, there is no way to do anything useful with these constructs. On the other hand, `\set bar :foo` is a perfectly valid way to copy a variable.

If you call `\set` without a second argument, the variable is set, with an empty string as *value*. To unset (or delete) a variable, use the command `\unset`.

`psql`'s internal variable names can consist of letters, numbers, and underscores in any order and any number of them. A number of these variables are treated specially by `psql`. They indicate certain option settings that can be changed at run time by altering the value of the variable or represent some state of the application. Although you can use these variables for any other purpose, this is not recommended, as the program behavior might behave unexpectedly. By convention, all specially treated variables consist of all upper-case letters (and possibly numbers and underscores). To ensure maximum compatibility in the future, avoid using such variable names for your own purposes. A list of all specially treated variables are as follows:

AUTOCOMMIT

When on (the default), each SQL command is automatically committed upon successful completion. To postpone commit in this mode, you must enter a `BEGIN` or `START TRANSACTION` SQL command. When off or unset, SQL commands are not committed until you explicitly issue `COMMIT` or `END`. The autocommit-on mode works by issuing an implicit `BEGIN` for you, just before any command that is not already in a transaction block and is not itself a `BEGIN` or other transaction-control command, nor a command that cannot be executed inside a transaction block (such as `VACUUM`).

In autocommit-off mode, you must explicitly abandon any failed transaction by entering `ABORT` or `ROLLBACK`. Also keep in mind that if you exit the session without committing, your work will be lost.

Leavng autocommit off is closer to the SQL spec. If you prefer autocommit-off, you may wish to set it in your `~/.psqlrc` file.

DBNAME

The name of the database you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

ECHO

If set to all, all lines entered from the keyboard or from a script are written to the standard output before they are parsed or executed. To select this behavior on program start-up, use the switch `-a`. If set to queries, `psql` merely prints all queries as they are sent to the server. The switch for this is `-e`.

ECHO_HIDDEN

When this variable is set and a backslash command queries the database, the query is first shown. This way you can study the HAWQ internals and provide similar functionality in your own programs. (To select this behavior on program start-up, use the switch `-E`.) If you set the variable to the value `noexec`, the queries are just shown but are not actually sent to the server and executed.

ENCODING

The current client character set encoding.

FETCH_COUNT

If this variable is set to an integer value > 0, the results of `SELECT` queries are fetched and displayed in groups of that many rows, rather than the default behavior of collecting the entire result set before display. Therefore only a limited amount of memory is used, regardless of the size of the result set. Settings of 100 to 1000 are commonly used when enabling this feature. Keep in mind that when using this feature, a query may fail after having already displayed some rows.

Although you can use any output format with this feature, the default aligned format tends to look bad because each group of `FETCH_COUNT` rows will be formatted separately, leading to varying column widths across the row groups. The other output formats work better.

HISTCONTROL

If this variable is set to `ignorespace`, lines which begin with a space are not entered into the history list. If set to a value of `ignoredups`, lines matching the previous history line are not entered. A value of `ignoreboth` combines the two options. If unset, or if set to any other value than those above, all lines read in interactive mode are saved on the history list.

HISTFILE

The file name that will be used to store the history list. The default value is `~/.psql_history`. For example, putting

```
\set HISTFILE ~/.psql_history- :DBNAME
```

in `~/.psqlrc` will cause `psql` to maintain a separate history for each database.

HISTSIZE

The number of commands to store in the command history. The default value is 500.

HOST

The database server host you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

IGNOREEOF

If unset, sending an `EOF` character (usually `CTRL+D`) to an interactive session of `psql` will terminate the application. If set to a numeric value, that many `EOF` characters are ignored before the application terminates. If the variable is set but has no numeric value, the default is `10`.

LASTOID

The value of the last affected OID, as returned from an `INSERT` or `lo_insert` command. This variable is only guaranteed to be valid until after the result of the next SQL command has been displayed.

ON_ERROR_ROLLBACK

When on, if a statement in a transaction block generates an error, the error is ignored and the transaction continues. When interactive, such errors are only ignored in interactive sessions, and not when reading script files. When off (the default), a statement in a transaction block that generates an error aborts the entire transaction. The on_error_rollback-on mode works by issuing an implicit `SAVEPOINT` for you, just before each command that is in a transaction block, and rolls back to the savepoint on error.

ON_ERROR_STOP

By default, if non-interactive scripts encounter an error, such as a malformed SQL command or internal meta-command, processing continues. This has been the traditional behavior of `psql` but it is sometimes not desirable. If this variable is set, script processing will immediately terminate. If the script was called from another script it will terminate in the same fashion. If the outermost script was not called from an interactive `psql` session but rather using the `-f` option, `psql` will return error code 3, to distinguish this case from fatal error conditions (error code 1).

PORT

The database server port to which you are currently connected. This is set every time you connect to a database (including program start-up), but can be unset.

PROMPT1
PROMPT2
PROMPT3

These specify what the prompts `psql` issues should look like. See "Prompting".

QUIET

This variable is equivalent to the command line option `-q`. It is not very useful in interactive mode.

SINGLELINE

This variable is equivalent to the command line option `-S`.

SINGLESTEP

This variable is equivalent to the command line option `-s`.

USER

The database user you are currently connected as. This is set every time you connect to a database (including program start-up), but can be unset.

VERBOSITY

This variable can be set to the values `default`, `verbose`, or `terse` to control the verbosity of error reports.

SQL Interpolation

An additional useful feature of `psql` variables is that you can substitute (interpolate) them into regular SQL statements. The syntax for this is again to prepend the variable name with a colon (`:`).

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

Variables can even contain unbalanced quotes or backslash commands. You must make sure that it makes sense where you put it. Variable interpolation will not be performed into quoted SQL entities.

A popular application of this facility is to refer to the last inserted OID in subsequent statements to build a foreign key scenario. Another possible use of this mechanism is to copy the contents of a file into a table column. First load the file into a variable and then proceed as above.

```
testdb=> \set content '''' `cat my_file.txt` ''''
testdb=> INSERT INTO my_table VALUES (:content);
```

One problem with this approach is that `my_file.txt` might contain single quotes. These need to be escaped so that they don't cause a syntax error when the second line is processed. This could be done with the program `sed`:

```
testdb=> \set content '''' `sed -e "s/'/''/g" < my_file.txt`
''''
```

Note the use of different shell quoting conventions so that neither the single quote marks nor the backslashes are special to the shell. Backslashes are still special to `sed`, however, so we need to double them.

Since colons may legally appear in SQL commands, the following rule applies: the character sequence "`:name`" is not changed unless "`name`" is the name of a variable that is currently set. In any case you can escape a colon with a backslash to protect it from substitution. (The colon syntax for variables is standard SQL for embedded query languages, such as ECPG. The colon syntax for array slices and type casts are HAWQ extensions, hence the conflict.)

Prompting

The prompts `psql` issues can be customized to your preference. The three variables `PROMPT1`, `PROMPT2`, and `PROMPT3` contain strings and special escape sequences that describe the appearance of the prompt. Prompt 1 is the normal prompt that is issued when `psql` requests a new command. Prompt 2 is issued when more input is expected during command input because the command was not terminated with a semicolon or a quote was not closed. Prompt 3 is issued when you run an SQL `COPY` command and you are expected to type in the row values on the terminal.

The value of the selected prompt variable is printed literally, except where a percent sign (`%`) is encountered. Depending on the next character, certain other text is substituted instead. Defined substitutions are:

`%M`

The full host name (with domain name) of the database server, or `[local]` if the connection is over a UNIX domain socket, or `[local:/dir/name]`, if the UNIX domain socket is not at the compiled in default location.

`%m`

The host name of the database server, truncated at the first dot, or `[local]` if the connection is over a UNIX domain socket.

`%>`

The port number at which the database server is listening.

`%n`

The database session user name. (The expansion of this value might change during a database session as the result of the command `SET SESSION AUTHORIZATION`.)

%/

The name of the current database.

%~

Like `%/`, but the output is `~` (tilde) if the database is your default database.

%#

If the session user is a database superuser, then a **#**, otherwise a **>**. (The expansion of this value might change during a database session as the result of the command `SET SESSION AUTHORIZATION`.)

%R

In prompt 1 normally **=**, but **^** if in single-line mode, and **!** if the session is disconnected from the database (which can happen if `\connect` fails). In prompt 2 the sequence is replaced by **-**, *****, a single quote, a double quote, or a dollar sign, depending on whether `psql` expects more input because the command wasn't terminated yet, because you are inside a `/* ... */` comment, or because you are inside a quoted or dollar-escaped string. In prompt 3 the sequence doesn't produce anything.

%x

Transaction status: an empty string when not in a transaction block, or ***** when in a transaction block, or **!** when in a failed transaction block, or **?** when the transaction state is indeterminate (for example, because there is no connection).

%digits

The character with the indicated octal code is substituted.

%:name:

The value of the `psql` variable name.

%`command`

The output of command, similar to ordinary back-tick substitution.

%[... %]

Prompts may contain terminal control characters which, for example, change the color, background, or style of the prompt text, or change the title of the terminal window. In order for line editing to work properly, these non-printing control characters must be designated as invisible by surrounding them with `%[` and `%]`. Multiple pairs of these may occur within the prompt. For example,

```
testdb=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%[%033[0m%]%#'
```

results in a boldfaced (`1;`) yellow-on-black (`33;40`) prompt on VT100-compatible, color-capable terminals. To insert a percent sign into your prompt, write `%%`. The default prompts are `'%/%R%# '` for prompts 1 and 2, and `'>> '` for prompt 3.

Command-Line Editing

`psql` supports the NetBSD libedit library for convenient line editing and retrieval. The command history is automatically saved when `psql` exits and is reloaded when `psql` starts up. Tab-completion is also supported, although the completion logic makes no claim to be an SQL parser. If for some reason you do not like the tab completion, you can turn it off by putting this in a file named `.inputrc` in your home directory:

```
$if psql
set disable-completion on
$endif
```

Environment

PAGER

If the query results do not fit on the screen, they are piped through this command. Typical values are `more` or `less`. The default is platform-dependent. The use of the pager can be disabled by using the `\pset` command.

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters.

`PSQL_EDITOR`
`EDITOR`
`VISUAL`

Editor used by the `\e` command. The variables are examined in the order listed; the first that is set is used.

`SHELL`

Command executed by the `\!` command.

`TMPDIR`

Directory for storing temporary files. The default is `/tmp`.

Files

Before starting up, `psql` attempts to read and execute commands from the user's `~/.psqlrc` file.

The command-line history is stored in the file `~/.psql_history`.

Notes

`psql` only works smoothly with servers of the same version. That does not mean other combinations will fail outright, but subtle and not-so-subtle problems might come up. Backslash commands are particularly likely to fail if the server is of a different version.

Notes for Windows Users

`psql` is built as a console application. Since the Windows console windows use a different encoding than the rest of the system, you must take special care when using 8-bit characters within `psql`. If `psql` detects a problematic console code page, it will warn you at startup. To change the console code page, two things are necessary:

Set the code page by entering:

```
cmd.exe /c chcp 1252
```

`1252` is a character encoding of the Latin alphabet, used by Microsoft Windows for English and some other Western languages. If you are using Cygwin, you can put this command in `/etc/profile`.

Set the console font to Lucida Console, because the raster font does not work with the ANSI code page.

Examples

Start `psql` in interactive mode:

```
psql -p 54321 -U sally mydatabase
```

In `psql` interactive mode, spread a command over several lines of input. Notice the changing prompt:

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text)
```

```
testdb-> ;
CREATE TABLE
```

Look at the table definition:

```
testdb=> \d my_table
             Table "my_table"
 Attribute |  Type   |      Modifier
-----------+---------+--------------------
 first     | integer | not null default 0
 second    | text    |
```

Run `psql` in non-interactive mode by passing in a file containing SQL commands:

```
psql -f /home/gpadmin/test/myscript.sql
```

Related Links

Client Utility Reference

vacuumdb

Garbage-collects and analyzes a database.

Synopsis

```
vacuumdb [connection-option...] [--full | -f] [-F] [--verbose | -v]
    [--analyze | -z] [--table | -t table [( column [,...] )] ] [dbname]

vacuumdb [connection-options...] [--all | -a] [--full | -f] [-F]
    [--verbose | -v] [--analyze | -z]

vacuumdb --help

vacuumdb --version
```

Description

`vacuumdb` is a utility for cleaning a HAWQ database. `vacuumdb` will also generate internal statistics used by the legacy query optimizer.

`vacuumdb` is a wrapper around the SQL command `VACUUM`. There is no effective difference between vacuuming databases via this utility and via other methods for accessing the server.

Options

`-a | --all`

Vacuums all databases.

`[-d]` *`dbname`* `| [--dbname]` *`dbname`*

The name of the database to vacuum. If this is not specified and `-all` is not used, the database name is read from the environment variable `PGDATABASE`. If that is not set, the user name specified for the connection is used.

`-e | --echo`

Echo the commands that `reindexdb` generates and sends to the server.

`-f | --full`

Selects a full vacuum, which may reclaim more space, but takes much longer and exclusively locks the table.

Warning: A `VACUUM FULL` is not recommended in HAWQ.

`-F | --freeze`

Freeze row transaction information.

`-q | --quiet`

Do not display a response.

`-t` *`table`* `[(`*`column`*`)] | --table` *`table`* `[(`*`column`*`)]`

Clean or analyze this table only. Column names may be specified only in conjunction with the `--analyze` option. If you specify columns, you have to escape the parentheses from the shell.

`-v | --verbose`

Print detailed information during processing.

`-z | --analyze`

Collect statistics for use by the query planner.

Connection Options

`-h` *`host`* `|` `--host` *`host`*

The host name of the HAWQ master database server. If not specified, reads from the environment variable `PGHOST` or defaults to localhost.

`-p` *`port`* `|` `--port` *`port`*

The TCP port where the HAWQ master database server listens for connections. If not specified, reads from the environment variable `PGPORT` or defaults to 5432.

`-U` *`username`* `|` `--username` *`username`*

The database role name to connect as. If not specified, reads from the environment variable `PGUSER` or defaults to the current system user name.

`-w | --no-password`

Use this to run automated batch jobs and scripts. In general, if the server requires password authentication ensure that it can be accessed through a `.pgpass` file. Otherwise the connection attempt will fail.

`-W | --password`

Force a password prompt.

Notes

`vacuumdb` might need to connect several times to the master server, asking for a password each time. It is convenient to have a `~/.pgpass` file in such cases.

Examples

Clean a database named `test`:

```
vacuumdb test
```

Clean and analyze a database named `bigdb`:

```
vacuumdb --analyze bigdb
```

Clean a single table `foo` in a database named `mydb`, and analyze a single column `bar` of the table. Note the quotes around the table and column names to escape the parentheses from the shell:

```
vacuumdb --analyze --verbose --table 'foo(bar)' mydb
```

See Also

`VACUUM`, *`ANALYZE`*

Related Links

Client Utility Reference

Chapter 15

HAWQ Server Configuration Parameters

There are many configuration parameters that affect the behavior of the HAWQ system.

- *gp_enable_direct_dispatch*
- *gp_enable_fallback_plan*
- *gp_enable_fast_sri*
- *gp_enable_gpperfmon*
- *gp_enable_groupext_distinct_gather*
- *gp_enable_groupext_distinct_pruning*
- *gp_enable_multiphase_agg*
- *gp_enable_predicate_propagation*
- *gp_enable_preunique*
- *gp_enable_sequential_window_plans*
- *gp_enable_sort_distinct*
- *gp_enable_sort_limit*
- *gp_external_enable_exec*
- *gp_external_grant_privileges*
- *gp_external_max_segs*
- *gp_filerep_tcp_keepalives_count*
- *gp_filerep_tcp_keepalives_idle*
- *gp_filerep_tcp_keepalives_interval*
- *gp_fts_probe_interval*
- *gp_fts_probe_threadcount*
- *gp_fts_probe_timeout*
- *gp_gpperfmon_send_interval*
- *gp_hashjoin_tuples_per_bucket*
- *gp_interconnect_default_rtt*
- *gp_interconnect_fc_method*
- *gp_interconnect_hash_multiplier*
- *gp_interconnect_min_rto*
- *gp_interconnect_min_retries_before_timeout*
- *gp_interconnect_queue_depth*
- *gp_interconnect_setup_timeout*
- *gp_interconnect_snd_queue_depth*
- *gp_interconnect_timer_period*
- *gp_interconnect_timer_checking_period*
- *gp_interconnect_transmit_timeout*
- *gp_interconnect_type*
- *gp_log_format*
- *gp_max_csv_line_length*
- *gp_max_databases*
- *gp_max_filespaces*
- *gp_max_local_distributed_cache*
- *gp_max_packet_size*
- *gp_max_tablespaces*
- *gp_motion_cost_per_row*
- *gp_num_contents_in_cluster*
- *gp_reject_percent_threshold*
- *gp_reraise_signal*

- *gp_resqueue_memory_policy*
- *gp_resqueue_priority*
- *gp_resequeue_priority_cpucores_per_segment*
- *gp_resqueue_priority_sweeper_interval*
- *gp_role*
- *gp_safefswritesize*
- *gp_segment_connect_timeout*
- *gp_segments_for_planner*
- *gp_session_id*
- *gp_set_proc_affinity*
- *gp_set_read_only*
- *gp_snmp_community*
- *gp_snmp_monitor_address*
- *gp_snmp_use_inform_or_trap*
- *gp_statistics_pullup_from_child_partition*
- *gp_statistics_use_fkeys*
- *gp_vmem_idle_resource_timeout*
- *gp_vmem_protect_limit*
- *gp_vmem_protect_segworker_cache_limit*
- *gp_workfile_checksumming*
- *gp_workfile_compress_algorithm*
- *gp_workfile_limit_per_query*
- *gpperfmon_port*
- *integer_datetimes*
- *IntervalStyle*
- *join_collapse_limit*
- *krb_caseins_users*
- *krb_server_keyfile*
- *krb_srvname*
- *krb5_ccname*
- *lc_collate*
- *lc_ctype*
- *lc_messages*
- *lc_monetary*
- *lc_numeric*
- *lc_time*
- *listen_addresses*
- *local_preload_libraries*
- *log_autostats*
- *log_connections*
- *log_disconnections*
- *log_dispatch_stats*
- *log_duration*
- *log_error_verbosity*
- *log_executor_stats*
- *log_hostname*

- *log_min_duration_statement*
- *log_min_error_statement*
- *log_min_messages*
- *log_parser_stats*
- *log_planner_stats*
- *log_rotation_age*
- *log_rotation_size*
- *log_statement*
- *log_statement_stats*
- *log_timezone*
- *log_truncate_on_rotation*
- *max_appendonly_tables*
- *max_connections*
- *max_files_per_process*
- *max_fsm_pages*
- *max_fsm_relations*
- *max_function_args*
- *max_identifier_length*
- *max_locks_per_transaction*
- *max_prepared_transactions*
- *max_resource_portals_per_transaction*
- *max_resource_queues*
- *max_stack_depth*
- *max_statement_mem*
- *optimizer_log*
- *optimizer_minidump*
- *password_encryption*
- *pljava_vmoptions*
- *port*
- *pxf_enable_stat_collection*
- *random_page_cost*
- *regex_flavor*
- *resource_cleanup_gangs_on_wait*
- *resource_select_only*
- *search_path*
- *seq_page_cost*
- *server_encoding*
- *server_ticket_renew_interval*
- *server_version*
- *server_version_num*
- *shared_buffers*
- *shared_preload_libraries*
- *ssl*
- *ssl_ciphers*
- *standard_conforming_strings*
- *statement_mem*

- *statement_timeout*
- *stats_queue_level*
- *superuser_reserved_connections*
- *tcp_keepalives_count*
- *tcp_keepalives_idle*
- *tcp_keepalives_interval*
- *temp_buffers*
- *TimeZone*
- *timezone_abbreviations*
- *track_activities*
- *track_counts*
- *transaction_isolation*
- *transaction_read_only*
- *transform_null_equals*
- *unix_socket_directory*
- *unix_socket_group*
- *unix_socket_permissions*
- *update_process_title*
- *vacuum_cost_delay*
- *vacuum_cost_limit*
- *vacuum_cost_page_dirty*
- *vacuum_cost_page_hit*
- *vacuum_cost_page_miss*
- *vacuum_freeze_min_age*

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 work_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 HAWQ 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 `CREATE 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 HAWQ, 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.

Table 13: Settable Classifications

Set classification	Description
master or local	A **master** parameter only needs to be set in the postgresql.conf file of the HAWQ 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`), 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 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 HAWQ 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.
superuser	These session parameters can only be set by a database superuser. Regular database users cannot set this parameter.

Set classification	Description
read only	These parameters are not settable by database users or superusers. The current value of the parameter can be shown but not altered.

Parameter Categories

Configuration parameters affect categories of server behaviors, such as resource confumption, query tuning, and authentication. The following sections list the HAWQ configuration parameters by category:

- *Connection and Authentication Parameters*
- *System Resource Consumption Parameters*
- *Query Tuning Parameters*
- *Error Reporting and Logging Parameters*
- *System Monitoring Parameters*
- *Runtime Statistics Collection Parameters*
- *Automatic Statistics Collection Parameters*
- *Client Connection Default Parameters*
- *Lock Management Parameters*
- *Workload Management Parameters*
- *External Table Parameters*
- *Past PostgreSQL Version Compatibility Parameters*
- *HAWQ Array Configuration Parameters*

Connection and Authentication Parameters

These parameters control how clients connect and authenticate to HAWQ.

Connection Parameters

gp_vmem_idle_resource_timeout	*tcp_keepalives_count*
listen_addresses	*tcp_keepalives_idle*
max_connections	*tcp_keepalives_interval*
max_prepared_transactions	*unix_socket_directory*
superuser_reserved_connections	*unix_socket_group*
	unix_socket_permissions

Security and Authentication Parameters

authentication_timeout	*krb_srvname*
db_user_namespace	*password_encryption*
krb_caseins_users	*ssl*
krb_server_keyfile	*ssl_ciphers*

System Resource Consumption Parameters

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

max_appendonly_tables

max_prepared_transactions

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 anong the segment instances. The vacuum cost limit and delay is invoked at the segment level without taking into account the state of the entire HAWQ 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

Query Tuning Parameters

Query Plan Operator Control Parameters

The following parameters control the types of plan operations the query planner can use. Enable or disable plan operations to force the planner to choose a different plan. This is useful for testing and comparing query performance using different plan types.

gp_enable_adaptive_nestloop

gp_enable_agg_distinct

gp_enable_agg_distinct_pruning

gp_enable_direct_dispatch

gp_enable_fallback_plan

gp_enable_fast_sri

gp_enable_groupext_distinct_gather

gp_enable_groupext_distinct_pruning

gp_enable_multiphase_agg

gp_enable_predicate_propagation

gp_enable_preunique

gp_enable_sequential_window_plans

gp_enable_sort_distinct

gp_enable_sort_limit

Query Planner Costing Parameters

Warning: Pivotal recommends that you do not adjust these query costing parameters. They are tuned to reflect HAWQ hardware configurations and typical workloads. All of these parameters are related. Changing one without changing the others can have adverse affects on performance.

cpu_operator_cost

gp_motion_cost_per_row

gp_segments_for_planner

random_page_cost

seq_page_cost

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_hashjoin_tuples_per_bucket

gp_statistics_use_fkeys

gp_workfile_compress_algorithm

Other Query Planner Configuration Parameters

gp_enable_predicate_propagation

gp_statistics_pullup_from_child_partition

Error Reporting and Logging Parameters

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

What to Log

debug_pretty_print

debug_print_parse

debug_print_plan

debug_print_prelim_plan

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

SNMP Alerts

The following parameters send SNMP notifications when events occur.

gp_snmp_community

gp_snmp_monitor_address

gp_snmp_use_inform_or_trap

Performance Monitor Agents

The following parameters configure the data collection agents for the Performance Monitor.

gp_enable_gpperfmon

gpperfmon_port

gp_gpperfmon_send_interval

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 HAWQ master `postgresql.conf` file and restart HAWQ:

gp_autostats_mode

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

Statement Behavior Parameters

search_path

statement_timeout

vacuum_freeze_min_age

Locale and Formatting Parameters

client_encoding

lc_collate

lc_ctype

lc_messages

lc_monetary

lc_numeric

lc_time

Other Client Default Parameters

local_preload_libraries

Lock Management Parameters

deadlock_timeout

max_locks_per_transaction

Workload Management Parameters

The following configuration parameters configure the HAWQ workload management feature (resource queues), query prioritization, memory utilization and concurrency control.

gp_resqueue_priority

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

stats_queue_level

External Table Parameters

The following parameters configure the external tables feature of HAWQ.

gp_external_enable_exec

gp_external_grant_privileges

gp_external_max_segs

gp_reject_percent_threshold

Append-Only Table Parameters

The following parameters configure the append-only tables feature of HAWQ.

max_appendonly_tables

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 HAWQ.

add_missing_from

backslash_quote

regex_flavor

standard_conforming_strings

transform_null_equals

HAWQ Array Configuration Parameters

The parameters in this topic control the configuration of the HAWQ 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_setup_timeout
gp_interconnect_type
gp_max_packet_size

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_interval
gp_fts_probe_threadcount

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

Server Configuration Parameters

add_missing_from

Automatically adds missing table references to FROM clauses.

Value Range	Default	Set Classifications
Boolean	off	master session reload

application

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	on	master session reload

array_null

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.

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)	1 min	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 lower 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_operator_cost

Sets the planner's 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

Sets the planner's 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 query planner how many rows are expected to be fetched in a cursor query, thereby allowing the planner 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 module. 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

DataStyle

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 HAWQ 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

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
```

Value Range	Default	Set Classifications
none on_change on_no_stats	on_no_ stats	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_cached_segworkers_threshold

When a user starts a session with HAWQ 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 HAWQ Performance Monitor 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 EMC Support (does not log events to the `gpperfmon` database)

Value Range	Default	Set Classifications
on, off, local, remote	on	master system restart superuser

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_dbid

The local content DB ID if a segment.

Value Range	Default	Set Classifications
integer		read only

gp_debug_linger

Number of seconds for a HAWQ 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_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_enable_adaptive_nestloop

Enables the query planner to use a new type of join node called "Adaptive Nestloop" at query execution time. This causes the planner 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.

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 system restart

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_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 query 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 HAWQ Performance Monitor.

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 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 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 query planner's use of two or three-stage parallel aggregation plans. 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 query 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 Performance Monitor or MapReduce features.

Value Range	Default	Set Classifications
Boolean	on	master session reload

gp_external_grant_privileges

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 session 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 restart

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	master session 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	1 min	master session 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 session 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	10 secs	master session restart

gp_gpperfmon_send_interval

Sets the frequency that the HAWQ server processes send query execution updates to the Performance Monitor agent processes. 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)	10 min	master system restart

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_interconnect_default_rtt

Sets the default rtt (in ms) for UDP interconnect.

Value Range	Default	Set Classifications
1-1000ms	20ms	master session reload

gp_interconnect_fc_method

Specifies the flow control method used for UDP interconnect.

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" or "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_min_rto

Sets the minimum rto (in ms) for UDP interconnect.

Value Range	Default	Set Classifications
1-1000ms	20ms	master session reload

gp_interconnect_min_retries_before_timeout

Sets the minimum number of retries before reporting a transmit timeout in the interconnect.

Value Range	Default	Set Classifications
1-4096	100	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

Time to wait for the Interconnect to complete setup before it times out.

Value Range	Default	Set Classifications
Any valid time expression (number and unit)	5 min	master session reload

gp_interconnect_snd_queue_depth

Used to specify the average size of a send queue. The buffer pool size for each send process can be calculated by using `gp_interconnect_snd_queue_depth` * number of processes in the downstream gang.

Value Range	Default	Set Classifications
1 - 4096	2	master session reload

gp_interconnect_timer_period

Sets the timer period (in ms) for UDP interconnect.

Value Range	Default	Set Classifications
1-100ms	5ms	master session reload

gp_interconnect_timer_checking_period

Sets the timer checking period (in ms) for UDP interconnect.

Value Range	Default	Set Classifications
1-100ms	20ms	master session reload

gp_interconnect_transmit_timeout

Timeout (in seconds) on interconnect to transmit a packet.

Value Range	Default	Set Classifications
1-7200s	3600s	master session reload

gp_interconnect_type

Sets the networking protocol used for Interconnect traffic. With the TCP protocol, HAWQ 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 HAWQ software does the additional packet verification and checking not performed by UDP, so reliability and performance is equivalent to TCP.

Value Range	Default	Set Classifications
TCP UDP	UDP	

gp_log_format

Specifies the format of the server log files. If using the `hawq_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 HAWQ 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 HAWQ system.

Value Range	Default	Set Classifications
1-64	16	master system restart

gp_max_filespaces

The maximum number of filespaces allowed in a HAWQ system.

Value Range	Default	Set Classifications
1-32	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	master 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_tablespaces

The maximum number of tablespaces allowed in a HAWQ system.

Value Range	Default	Set Classifications
1-16	16	master system restart

gp_motion_cost_per_row

Sets the query 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 HAWQ system.

Value Range	Default	Set Classifications
-	-	read only

gp_reject_percent_threshold

For single row error handling on `COPY` and external table `SELECT`, 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 HAWQ memory management features.

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_resequeue_priority_cpucores_per_segment

Specifies the number of CPU units per segment. In a configuration where one segment is configured per CPU core on a host, this unit is 1.0 (default). If an 8-core host is configured with four segments, the value would be 2.0. A master host typically only has one segment running on it (the master instance), so the value for the master should reflect the usage of all available CPU cores.

Incorrect settings can result in CPU under-utilization.

The default values are appropriate for the EMC Greenplum Data Computing Appliance.

Value Range	Default	Set Classifications

0.1 - 25.0	segments = 4
	master = 24
		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-only tables in a non-mature file system. When a number of bytes greater than zero is specified, the append-only 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 HAWQ 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 HAWQ 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 restart

gp_segments_for_planner

Sets the number of primary segment instances for the 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 planner'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 HAWQ 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 HAWQ 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 query planner to utilize statistics from child tables when planning queries on the parent table.

Value Range	Default	Set Classifications
Boolean	on	master system reload

gp_statistics_use_fkeys

When enabled, allows the optimizer 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 system 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 HAWQ processes of an active segment instance can consume. To prevent over allocation of memory, set to:

```
( X * physical_memory ) / primary_segments
```

Where *X* is a value between 1.0 and 1.5. X=1 offers the best system performance. X=1.5 may cause more swapping on the system, but less queries will be cancelled.

For example, on a segment host with 16GB physical memory and 4 primary segment instances, the calculation would be:

```
(1 * 16) / 4 = 4GB
```

```
4 * 1024 = 4096MB
```

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).

Value Range	Default	Set Classifications
integer	8192 MB	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	master 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 system 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.

Value Range	Default	Set Classifications
none zlib	none	master system 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

gpperfmon_port

Sets the port on which all performance monitor agents communicate with the master.

Value Range	Default	Set Classifications
integer	8888	master system restart

integer_datetimes

Reports whether HAWQ 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 `DataStyle` parameter was set to ISO. 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 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

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 session reload

krb_server_keyfile

Sets the location of the Kerberos server key file.

Value Range	Default	Set Classifications
path and file name	unset	master session restart

krb_srvname

Sets the Kerberos service name.

Value Range	Default	Set Classifications
service name	postgres	master session restart

krb5_ccname

Sets the location of the Kerberos ticket cache.

Value Range	Default	Set Classifications
path and file name	`/tmp/postgrres.ccname`	master session restart

lc_collate

Reports the locale in which sorting of textual data is done. The value is determined when the HAWQ 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 HAWQ 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 host name as well. Note that depending on your host name resolution setup, this might impose a non-negligible performance penalty.

Value Range	Default	Set Classifications
Boolean	off	local 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 250 ms 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 lower the level, the fewer messages are sent to the log.

Value Range	Default	Set Classifications
DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1 INFO NOTICE WARNING ERROR LOG 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 query 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	master session 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.

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-only relations that can be written to or loaded concurrently. Append-only 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
2048	2048	master session restart

max_connections

The maximum number of concurrent connections to the database server. In a HAWQ system, user client connections go through the HAWQ 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.

Increasing this parameter may cause HAWQ to request more shared memory.

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.

Note: Increasing this value can improve performance of HAWQ, but bring heavier workload to HDFS.

Value Range	Default	Set Classifications
integer	150	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*	20000	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_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	64	local system restart

max_prepared_transactions

Sets the maximum number of transactions that can be in the prepared state simultaneously. HAWQ 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 HAWQ 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	8	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	200MB	master session reload superuser

optimizer_log

Indicates whether the Pivotal Query Optimizer or the legacy query optimizer produced the query execution plan. It also records the reason for using a plan generated by the existing planner. For more information about Pivotal Query Optimizer and the legacy query optimizer, see *HAWQ Query Processing*.

Value Range	Default	Set Classifications
Boolean	true	

optimizer_minidump

The Pivotal Query Optimizer generates minidumps to describe the optimization context for a given query. You can use the information in these files to reproduce failures or performance regressions during optimization in any environment. The minidump file is located under the master data directory and uses the following naming format:

```
Minidump_<date>_<time>.mpd
```

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 costs.

Value Range	Default	Set Classifications
ONERROR ALWAYS	ONERROR	

password_encryption

When a password is specified in `CREATE USER` or `ALTER USER` without stating either `ENCRYPTED` or `UNENCRYPTED`, this option determines whether the password is encrypted.

Value Range	Default	Set Classifications
Boolean		

pljava_vmoptions

Java VM options for pljava user-defined functions.

Value Range	Default	Set Classifications
integer		session or system

port

The database listener port for a HAWQ instance. The master and each segment has its own port. You must shut down your HAWQ system before changing port numbers.

Value Range	Default	Set Classifications
any valid port number	5432	local system restart

pxf_enable_stat_collection

Collects statistical information about PXF.

Value Range	Default	Set Classifications
Boolean	on	

random_page_cost

Sets the planner's estimate of the cost of a nonsequentially fetched disk page. This is measured as a multiple of the cost of a sequential page fetch.

Value Range	Default	Set Classifications
floating point	100	master session reload

regex_flavor

The `extended` value 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 session 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`, then `INSERT`, `UPDATE`, and `DELETE` commands will be evaluated as well.

Value Range	Default	Set Classifications
Boolean	off	master session reload

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

Sets the planner's 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 HAWQ 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_ticket_renew_interval

Sets the Kerberos ticket renew interval in milliseconds.

Value Range	Default	Set Classifications
integer	43200000	master system restart

server_version

Reports the version of PostgreSQL that this release of HAWQ 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 HAWQ is based on as an integer.

Value Range	Default	Set Classifications
integer	80215	read only

shared_buffers

Sets the amount of memory a HAWQ server instance uses for shared memory buffers. This setting must be at least 128 kilobytes and at least 16 kilobytes * `max_connections`.

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/pl`*`XXX`*' 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
		master 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	read only

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	off	

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.

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	local system restart

superuser_reserved_connections

Determines the number of connection slots that are reserved for HAWQ 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

The number of 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 `/share/postgresql/timezonesets/` in the installation directory.

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 system 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, look up 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, look up 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. `VACUUM` will limit the effective value to half the value of `autovacuum_freeze_max_age`, so that there is not an unreasonably short time between forced autovacuums.

Value Range	Default	Set Classifications
integer 0-100000000000	100000000	local system restart

Chapter 16

Character Set Support Reference

The character set support in HAWQ 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 HAWQ using gpinitsystem. It can be overridden when you create a database, so you can have multiple databases each with a different character set.

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	
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

Name	Description	Language	Server	Bytes/Char	Aliases
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	unspecified2	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	
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	

Name	Description	Language	Server	Bytes/Char	Aliases
WIN1256	Windows CP1256	Arabic	Yes	1	
WIN1257	Windows CP1257	Baltic	Yes	1	
WIN1258	Windows CP1258	Vietnamese	Yes	1	ABC, TCVN, TCVN5712, VSCII

Note:

- Not all the APIs support all the listed character sets. For example, the JDBC driver does not support MULE_INTERNAL, LATIN6, LATIN8, and LATIN10.
- The SQLASCII setting behaves considerable 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 nonASCII data, it is unwise to use the SQL_ASCII setting as a client encoding. SQL_ASCII is not supported as a server encoding.

Setting the Character Set

gpinitsystem defines the default character set for a HAWQ 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';
```

Note: 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

HAWQ 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. HAWQ comes with some predefined conversions or you can create a new conversion using the SQL command CREATE CONVERSION.

Server Character Set	Available Client 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
SQL_ASCII	not supported as a server encoding

Server Character Set	Available Client Sets
UHC	not supported as a server encoding
UTF8	all supported encodings
WIN866	WIN866
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 HAWQ 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 SET client_encoding TO. Setting the client encoding can be done with this SQL command:

  ```
  SET CLIENT_ENCODING TO 'value';
  ```

  To query the current client encoding:

  ```
  SHOW client_encoding;
  ```

  To return 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 HAWQ 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 17

HAWQ Environment Variables

This is a reference for the environment variables to set for HAWQ. 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

Note: `GPHOME`, `PATH` and `LD_LIBRARY_PATH` can be set by sourcing the `greenplum_path.sh` file from your HAWQ installation directory.

GPHOME

This is the installed location of your HAWQ software. For example:

```
GPHOME=/usr/local/greenplum-db-4.1.x.x
export GPHOME
```

PATH

Your `PATH` environment variable should point to the location of the HAWQ bin directory. Solaris users must also add `/usr/sfw/bin` and `/opt/sfw/bin` to their `PATH`. For example:

```
PATH=$GPHOME/bin:$PATH
PATH=$GPHOME/bin:/usr/local/bin:/usr/sbin:/usr/sfw/bin:/opt/sfw/bin:$PATH
export PATH
```

LD_LIBRARY_PATH

The `LD_LIBRARY_PATH` environment variable should point to the location of the `HAWQ/PostgreSQL` library files. For Solaris, this also points to the GNU compiler and readline library files as well (readline libraries may be required for Python support on Solaris). For example:

```
LD_LIBRARY_PATH=$GPHOME/lib
LD_LIBRARY_PATH=$GPHOME/lib:/usr/sfw/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 HAWQ environment variables. You may want to add the connection-related environment variables to your profile, for convenience. That way, 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 HAWQ master host only.

PGAPPNAME

This is 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 HAWQ 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 Password File under *Configuring Client Authentication*.

PGOPTIONS

Sets additional configuration parameters for the HAWQ master server.

PGPORT

The port number of the HAWQ server on the master host. The default port is 5432.

PGUSER

The HAWQ 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 18

HAWQ Data Types

HAWQ 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
- char
- character
- date
- decimal
- double precision
- integer
- interval
- numeric
- real
- smallint
- time (with or without time zone)
- timestamp (with or without time zone)
- varchar

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 unique to HAWQ, 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 14: HAWQ 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)]	varbit	actual number of bits	bit string constant	variable-length bit string

Name	Alias	Size	Range	Description
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		1 byte + binarystring	sequence of octets	variable-length binary string
character [(n)]	char [(n)]	1 byte + n	strings up to n characters in length	fixed-length, blank padded
character varying [(n)]	varchar [(n)]	1 byte + binarystring	strings up to n characters in length	variable-length with limit
cidr		12 or 24 bytes		IPv4 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)]	numeric [(p,s)]	variable	no limit	user-specified, inexact
double precision	float8 float	8 bytes	15 decimal digits precision	variable-precision, inexact
inet		12 or 24 bytes		IPv4 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		16+16n bytes	[(x1,y1),...]	geometric path in the plane - not allowed in distribution key columns.

Name	Alias	Size	Range	Description
point		16 bytes	(x, y)	geometric path in the plane - not allowed in distribution key columns.
polygon		40+16n bytes	[(x1,y1),...]	closed geometric path in the plane - not allowed in the distribution key columns.
real	float4	4 bytes	6 decimal digits precision	variable-precision, inexact
serial	serial4	4 bytes	1 to 2147483647	autoincrementing integer
smallint	int2	2 bytes	-32768 to +32767	small range integer
text		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		1 byte + xml size	xml of any length	variable unlimited length

For variable length data types (such as char, varchar, text, xml, etc.) if the data is greater than or equal to 127 bytes, the storage overhead is 4 bytes instead of 1.

Time Zones

Time zones, and time-zone conventions, are influenced by political decisions, not just earth geometry. Time zones around the world became somewhat standardized during the 1900's, but continue to be prone to arbitrary changes, particularly with respect to daylight-savings rules. HAWQ uses the widely-used zoneinfo time zone database for information about historical time zone rules. For times in the future, the assumption is that the latest known rules for a given time zone will continue to be observed indefinitely far into the future.

HAWQ is compatible with the SQL standard definitions for typical usage. However, the SQL standard has an odd mix of date and time types and capabilities. Two obvious problems are:

- Although the date type cannot have an associated time zone, the time type can. Time zones in the real world have little meaning unless associated with a date as well as a time, since the offset can vary through the year with daylight-saving time boundaries.
- The default time zone is specified as a constant numeric offset from UTC. It is therefore impossible to adapt to daylight-saving time when doing date/time arithmetic across DST boundaries.

To address these difficulties, Pivotal recommends using date/time types that contain both date and time when using time zones. Pivotal recommends that you do not use the type time with time zone (although HAWQ supports this for legacy applications and for compliance with the SQL standard). HAWQ assumes your local time zone for any type containing only date or time.

All timezone-aware dates and times are stored internally in UTC. They are converted to local time in the zone specified by the timezone configuration parameter before being displayed to the client.

HAWQ allows you to specify time zones in three different forms:

- A full time zone name, for example America/New_York. HAWQ uses the widely-used zoneinfo time zone data for this purpose, so the same names are also recognized by much other software.
- A time zone abbreviation, for example PST. Such a specification merely defines a particular offset from UTC, in contrast to full time zone names which can imply a set of daylight savings transition-date rules as well. You cannot set the configuration parameters timezone or log_timezone to a time zone abbreviation, but you can use abbreviations in date/time input values and with the AT TIME ZONE operator.
- In addition to the timezone names and abbreviations, HAWQ accepts POSIX-style time zone specifications of the form STDoffset or STDoffsetDST, whereSTD is a zone abbreviation, offset is a numeric offset in hours west from UTC, and DST is an optional daylight-savings zone abbreviation, assumed to stand for one hour ahead of the given offset. For example, if EST5EDT were not already a recognized zone name, it would be accepted and would be functionally equivalent to United States East Coast time. When a daylight-savings zone name is present, it is assumed to be used according to the same daylight-savings transition rules used in the zoneinfo time zone database's posixrules entry. In a standard HAWQ installation, posixrules is the same as US/Eastern, so that POSIX-style time zone specifications follow USA daylight-savings rules. If needed, you can adjust this behavior by replacing the posixrules file.

In short, this is the difference between abbreviations and full names: abbreviations always represent a fixed offset from UTC, whereas most of the full names imply a local daylight-savings time rule, and so have two possible UTC offsets.

One should be wary that the POSIX-style time zone feature can lead to silently accepting bogus input, since there is no check on the reasonableness of the zone abbreviations. For example, SET TIMEZONE TO FOOBAR0 will work, leaving the system effectively using a rather peculiar abbreviation for UTC. Another issue to keep in mind is that in POSIX time zone names, positive offsets are used for locations west of Greenwich. Everywhere else, PostgreSQL follows the ISO-8601 convention that positive timezone offsets are east of Greenwich.

In all cases, timezone names are recognized case-insensitively.

Neither full names nor abbreviations are hard-wired into the server, see *Date and Time Configuration Files*.

The timezone configuration parameter can be set in the file postgresql.conf. There are also several special ways to set it:

- If timezone is not specified in postgresql.conf or as a server command-line option, the server attempts to use the value of the TZ environment variable as the default time zone. If TZ is not defined or is not any of the time zone names known to PostgreSQL, the server attempts to determine the operating system's default time zone by checking the behavior of the C library function localtime(). The default time zone is selected as the closest match from HAWQ's known time zones.
- The SQL command SET TIME ZONE sets the time zone for the session. This is an alternative spelling of SET TIMEZONE TO with a more SQL-spec-compatible syntax.
- The PGTZ environment variable is used by libpq clients to send a SET TIME ZONE command to the server upon connection.

Date and Time Configuration Files

Since timezone abbreviations are not well standardized, HAWQ provides a means to customize the set of abbreviations accepted by the server. The timezone_abbreviations run-time parameter determines the active set of abbreviations. While this parameter can be altered by any database user, the possible values for it are under the control of the database administrator — they are in fact names of configuration files stored in .../share/timezonesets/ of the installation directory. By adding or altering files in that directory, the administrator can set local policy for timezone abbreviations.

timezone_abbreviations can be set to any file name found in .../share/timezonesets/, if the file's name is entirely alphabetic. (The prohibition against non-alphabetic characters in timezone_abbreviations prevents reading files outside the intended directory, as well as reading editor backup files and other extraneous files.)

A timezone abbreviation file can contain blank lines and comments beginning with #. Non-comment lines must have one of these formats:

```
time_zone_nameoffsettime_zone_nameoffset D
@INCLUDE file_name
@OVERRIDE
```

A time_zone_name is just the abbreviation being defined. The offset is the zone's offset in seconds from UTC, positive being east from Greenwich and negative being west. For example, -18000 would be five hours west of Greenwich, or North American east coast standard time. D indicates that the zone name represents local daylight-savings time rather than standard time. Since all known time zone offsets are on 15 minute boundaries, the number of seconds has to be a multiple of 900.

The @INCLUDE syntax allows inclusion of another file in the .../share/timezonesets/ directory. Inclusion can be nested, to a limited depth.

The @OVERRIDE syntax indicates that subsequent entries in the file can override previous entries (i.e., entries obtained from included files). Without this, conflicting definitions of the same timezone abbreviation are considered an error.

In an unmodified installation, the file Default contains all the non-conflicting time zone abbreviations for most of the world. Additional files Australia and India are provided for those regions: these files first include the Default file and then add or modify timezones as needed.

For reference purposes, a standard installation also contains files Africa.txt, America.txt, etc, containing information about every time zone abbreviation known to be in use according to the zoneinfo timezone database. The zone name definitions found in these files can be copied and pasted into a custom configuration file as needed.

Note: These files cannot be directly referenced as timezone_abbreviations settings, because of the dot embedded in their names.

Chapter 19

System Catalog Reference

This is a reference of the system catalog tables and views of the HAWQ Database. All system tables related to the parallel features of HAWQ are prefixed with **gp_** . Tables prefixed with **pg_** are standard system catalog tables, or features HAWQ that enhance data warehousing workloads. Note that the global system catalog for HAWQ resides on the master instance.

- *gp_configuration_history*
- *gp_distributed_log*
- *gp_distributed_xacts*
- *gp_distribution_policy*
- *gp_fastsequence*
- *gpexpand.expansion_progress*
- *gpexpand.status*
- *gpexpand.status_detail*
- *gp_fault_strategy*
- *gp_global_sequence*
- *gp_id*
- *gp_interfaces*
- *gp_master_mirroring*
- *gp_persistent_database_node*
- *gp_persistent_filespace_node*
- *gp_persistent_relation_node*
- *gp_persistent_tablespace_node*
- *gp_pgdatabase*
- *gp_relation_node*
- *gp_san_configuration*
- *gp_segment_configuration*
- *gp_transaction_log*
- *gp_version_at_initdb*
- *pg_aggregate*
- *pg_am*
- *pg_amop*
- *pg_amproc*
- *pg_appendonly*
- *pg_attrdef*
- *pg_attribute*
- *pg_attribute_encoding*
- *pg_auth_members*
- *pg_authid*
- *pg_cast*
- *pg_class*
- *pg_compression*

- *pg_constraint*
- *pg_conversion*
- *pg_database*
- *pg_depend*
- *pg_description*
- *pg_exttable*
- *pg_filespace*
- *pg_filespace_entry*
- *pg_index*
- *pg_inherits*
- *pg_language*
- *pg_largeobject*
- *pg_locks*
- *pg_opclass*
- *pg_namespace*
- *pg_operator*
- *pg_partition*
- *pg_partition_columns*
- *pg_partition_encoding*
- *pg_partition_rule*
- *pg_partition_templates*
- *pg_partitions*
- *pg_pltemplate*
- *pg_proc*
- *pg_resourcetype*
- *pg_resqueue*
- *pg_resqueue_attributes*
- *pg_resqueuecapability*
- *pg_rewrite*
- *pg_roles*
- *pg_shdepend*
- *pg_shdescription*
- *pg_stat_activity*
- *pg_stat_last_operation*
- *pg_stat_last_shoperation*
- *pg_stat_operations*
- *pg_stat_partition_operations*
- *pg_statistic*
- *pg_stat_resqueues*
- *pg_tablespace*
- *pg_trigger*
- *pg_type*
- *pg_type_encoding*
- *pg_user_mapping*
- *pg_window*

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 Pivotal 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.

Column	Type	References	Description
time	timestamp with time zone		Timestamp for the event recorded.
dbid	amllint	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 gprecoverseg, see *Management Utility Reference*.

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. HAWQ's distributed transaction manager ensures that the segments stay in synch. This view allows you to see the status of distributed transactions.

Column	Type	References	Description
segment_id	smallint	gp_segment_ configuration.content	The content id of 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 gloabl transaction id.
distributed_id	text		A system assigned ID for a distributed transaction (Committed or Aborted)
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 HAWQ distributed transactions. A distributed transaction is a transaction that involves modifying data on the segment instances. HAWQ'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.

Column	Typ	References	Description
distributed_xid	xid		The transaction ID used by the distributed transaction across the HAWQ 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.
gp_session_id	int		The ID number of the HAWQ database session associated with the 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 HAWQ 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.

Column	Type	References	Description
localoid	oid	pg_class>oid	The table object identifier (OID).
attrmums	smallint[]	pg_atrribute.attnum	The column number(s) of the distribution column(s).

gp_fastsequence

The gp_fastsequence table contains information about indexes on append-only column-oriented tables. It is used to track the maximum row number used by a file segment of an append-only column-oriented table.

Column	Type	References	Description
objid	oid	pg_class.oid	Object id of the pg_aoseg.pg_aocsseg_* table used to track append-only file segments.
objmod	bigint		Object modifier.
last_sequence	bigint		The last sequence number used by the object.
contentid	integer		The content identifier.

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`.

Column	Type	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: gpexpand.status

Column	Type	Description

status	text	Tracks the status of an expansion operation. Valid values are:
		`SETUP`
		`SETUP DONE`
		`EXPANSION STARTED`
		`EXPANSION STOPPED`
		`COMPLETED`

updated	timestamp with timezone	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: gpexpand.status_detail

Column	Type	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`
last updated	timestamp with time zone	Timestamp of the last change in status for this table.
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.

Column	Type	Description
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_fault_strategy

The `gp_fault_strategy` table specifies the fault action.

Column	Type	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.

Column	Type	Description
sequence_num	bigint	log sequence number position in the transaction log.

gp_id

The `gp_id` system catalog table identifies the HAWQ 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.

Column	Type	Description
gpname	name	The name of the HAWQ database system.
numsegments	smallint	The number of segments in the HAWQ Database system.
dbid	smallint	The unique identifier of this segment (or master) instance.
content	smallint	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 the HAWQ 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.

Column	Type	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_master_mirroring

The `gp_master_mirroring` table contains state information about the standby master host and its associated write-ahead log (WAL) replication process. If this synchronization process (gpsyncagent) fails on the standby master, it may not always be noticeable to users of the system. This catalog is a place where HAWQ administrators can check to see if the standby master is current and fully synchronized.

Column	Type	Description
summary_state	text	The current state of the log replication process between the master and standby master - logs are either ‘Synchronized’ or ‘Not Synchronized’
detail_state	text	If not synchronized, this column will have information about the cause of the error.
log_time	timestamptz	This contains the timestamp of the last time the master sent its logs to the standby master.
error_message	text	If not synchronized, this column will have the error message from the failed synchronization attempt.

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.

Column	Type	References	Description
contentid	integer	gp_segment_ configuration.content	The content identifier for a segment instance, in Hawq it means a partition of data. Will be removed in the future release.
tablespace_oid	oid	pg_tablespace.oid	Table space object id.
database_oid	oid	pg_database.oid	Database 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
create_mirror_data_ loss_tracking_session_ num	bigint		Unused.
mirror_existence_state	smallint		Unused.
reserved	integer		Unused.
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 HAWQ to internally manage persistent representations of file system objects.
shared_storage	boolean		Indicate is this database on HDFS storage. Will be removed in the future release.

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.

Column	Type	References	Description
filespace_oid	oid	pg_filespace.oid	Object id of the filespace.
contentid	integer	gp_segment_ configuration.content	The content identifier for a segment instance, in Hawq it means a partition of data. Will be removed in the future release.
db_id_1	smallint		Primary segment id.
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
create_mirror_data_ loss_tracking_session_ num	bigint		Unused.
mirror_existence_state	smallint		Unused.
reserved	integer		Unused.
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 HAWQ Database to internally manage persistent representations of file system objects.
shared_storage	boolean		Indicate is this database on HDFS storage. Will be removed in the future release.

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.

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.refilenode	The object id of the relation file node.
segment_file_num	integer		For append-only tables, the append-only segment file number.
relation_storage_ manager	smallint		Whether the relation is heap storage or append-only 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
create_mirror_data_ loss_tracking_session_ num	bigint		Unused.
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

Column	Type	References	Description
mirror_data_ synchronization_state	smallint		Unused.
mirror_bufpool_marked_ for_scan_incremental_ resync	boolean		Unused.
mirror_bufpool_resync_ changed_page_count	bigint		Unused.
mirror_bufpool_resync_ ckpt_loc	gpxlogloc		Unused.
mirror_bufpool_resync_ ckpt_block_num	integer		Unused.
mirror_append_only_ loss_eof	bigint		Unused.
mirror_append_only_ new_eo	bigint		Unused.
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 HAWQ to internally manage persistent representations of file system objects.
shared_storage	boolean		Unused.
contentid	integer		Content identifier.

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.

Column	Type	References	Description
contentid	integer	gp_segment_ configuration.content	The content identifier for a segment instance.
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
create_mirror_data_ loss_tracking_session_ num	bigint		Unused.
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
reserved	integer		Unused.
parent_xid	integer		Global transaction ID.
persistent_serial_ number	bigint		Log sequence number position in the transaction log for a file block.

Column	Type	References	Description
previous_free_tid	tid		Used by HAWQ Database to internally manage persistent representations of file system objects.
shared_storage	boolean		Indicate is this database on HDFS storage. Will be removed in the future release.

gp_pgdatabase

The `gp_pgdatabase` view shows status information about HAWQ segment instances and whether they are acting as the mirror or the primary. This view is used internally by the HAWQ fault detection and recovery utilities to determine failed segments.

Column	Type	Reference	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 HAWQ Database. For the master, the value is -1.
valid	boolean		Whether or not the instance is valid.
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).

Column	Type	References	Description
relfilenode_oid	oid	pg_class.relfilenode	The object id of the relation file node.
segment_file_num	integer		For append-only tables, the append-only segment file number.
create_mirror_data_loss_tracking_session_num	bigint		Unused.
persistent_tid	tid		Used by HAWQ 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.
contentid	integer	gp_segment_configuration.content	The content identifier for a segment instance, in Hawq it means a partition of data. Will be removed in the future release.

gp_san_configuration

The `gp_san_configuration` table contains mount-point information for SAN failover.

Column	Type	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 indicates 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.
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.

Column	Type	Description
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.

Column	Type	References	Description
dbid	smallint		The unique identifier of a segment (or master) instance.
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.

Column	Type	References	Description
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.
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.

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 HAWQ system. It identifies the version of HAWQ 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.

Column	Type	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.

Column	Type	References	Description
aggfnoid	reproc	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_procof the inverse function of aggtransfn.
agginvprelimfn	regproc	pg_proc.oid	The OID in pg_procof the inverse function of aggtransfn.
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.

Column	Type	References	Description
amname	name		Name of the access method.
amstrategies	smallint		Number of operator strategies for this access method
amsupport	smallint		Number of support routines for this access method
amorderstrategy	smallint		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?
amcanshrink	boolean		
aminsert	regproc		"Insert this tuple" function.
ambeginscan	regproc		"Start new scan" function.
amgettuple	regproc		"Next valid tuple" function.

Column	Type	References	Description
amgetmulti	regproc		"Fetch multiple tuples" function.
amrescan	regproc		"Restart this scan" function.
amendscan	regproc		"End this scan" function.
ammarkpos	regproc		"Mark current scan position" function.
amrestrpos	regproc		"Restore marked scan position" function.
ambuild	regproc		"Build new index" function.
ambulkdelete	regproc		Bulk-delete function.
amvacuumcleanup	regproc		Post-VACUUM cleanup function.
amcostestimate	regproc		Function to estimate cost of an index scan.
amoptions	regproc		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.

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 the class.i operator.

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-only tables. This table is populated only on the master.

Column	Type	Description
relid	oid	The table object identifier (OID) of the compressed table.
blocksize	integer	For AO/CO tables,blocksize indicates the block size used for compression of append-only tables. Valid values are [8KB, 2MB], with a default of 32KB; For Parquet tables, blocksize indicates the row group size for the Parquet table. Valid values are [1KB, 1GB). with a default of 8MB.
safefswritesize	integer	Minimum size for safe write operations to append-only 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.
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. This data is stored only if gp_appendonly_verify_block_checksums is enabled (this parameter is disabled by default to optimize performance).
compresstype	text	Type of compression used to compress append-only tables. Valid values are zlib (gzip compression) and quicklz.

Column	Type	Description
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.
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.
version	integer	The version of the AppendOnlyEntry for table pg_ appendonly. Currently, the version number is 2.
pagesize	integer	The variable for Parquet tables: the page size for each column inside a row group. Valid value are [1KB, 1GB). Should be less than rowgroup size. Default size is 1MB.

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.

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.

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).

Column	Type	References	Description
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.
atttypmod	int4		Records type-specific data supplied at table creation time (for example, the maximum length of a carchar 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. typealign 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.

Column	Type	References	Description
attisdropped	boolean		This column has been dropped and no longer valid. A dropped column is still physically present in the table, but is ignored by the parser. It 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.
aatinhcount	int4		The number of direct ancestors this column has. A column with a nonzero number of ancestors cannot be dropped not renamed.

pg_attribute_encoding

The `pg_attribute_encoding` system catalog table contains column storage information.

Column	Type	Modifiers	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 HAWQ Database system.

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 HAWQ Database system: there is only one copy of **pg_authid** per system, not one per database.

Column	Type	Description
rolname	name	Role name
rolsuper	boolean	Role has superuser privileges.
rolinherit	boolean	Role automatically inherits privileges of roles it is a member of.
rolcreatearole	boolean	Role may create more roles.
rolecreatedb	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 n limit.
rolpassword	text	Password (possible 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.
rolresqueue	oid	
rolcreaterextgpfd	boolean	Role may create readable external tables that use the gpfdist protocol
rolcreaterexthttp	boolean	Role may create readable external tables that use the http protocol

Column	Type	Description
rolcreatewextgpfd	boolean	Role may create writable external tables that use the gpfdist protocol
rolcreaterexthdfs	boolean	Unused
rolcreatewexthdfs	boolean	Unused

pg_cast

The catalog `pg_cast` 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.

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).
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.

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_authod.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		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.)
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.

Column	Type	References	Description
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		For an aoseg table, OID of segno index.
relaosegrelid	oid		OID of an aoseg table; 0 if none.
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.
relkind	char		The type of object r = heap or append-only table, i = index, S = sequence, v = view, c = composite type, t = TOAST value, o = internal append-only segment files and EOFs, c = composite type, u = uncataloged temporary heap table
relstorage	char		The storage mode of a table a = append-only, h = heap, v = virtual, x= external table.

Column	Type	References	Description
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.
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	acliterm[]		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.

Column	Type	Modifiers	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 compressor.
compdecompressor	regproc	not null	plain	Name of decompressor.
compvalidator	regproc	not null	plain	Name of 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. Check constraints on domains are stored here, too.

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 relation
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 update 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.

Column	Type	Reference	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.

Column	Type	References	Description
datname	name		Database name.
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; useful particularly to pg_dump/gp_dump.

Column	Type	References	Description
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.
dattablespace	oid	pg_tablesapce.oid	The default tablespace for the database catalog. 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.
dat2tablespace	oid	pg_tablesapce.oid	The default tablespace for the table created in this database. dattablespace is used to store the catalog of this database.
datconfig	text[]		Session defaults for user-settable server configuration parameters.
datacl	aclite[]		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 HAWQ.

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. Example: a trigger that's created to enforce a foreign-key constraint is made internally dependent on the constraint's **pg_constraint** entry.
- 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.

Column	Type	Reference	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.
objsubid	int4		For a table column, this is the column number. For all the 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.

Column	Type	Reference	Description
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 HAWQ.

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.

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 HAWQ 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 HAWQ, 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 HAWQ system is referred to as a 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.
fsfsys	oid		The filesystem ID.
fsrep	smallint		fs replica number.

pg_filespace_entry

A tablespace requires a file system location to store its database files. In HAWQ, 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 HAWQ system is referred to as a **filespace** . The `pg_filespace_entry` table contains information about the collection of file system locations across a HAWQ Database system that comprise a HAWQ Database filespace.

Column	Type	References	Description
fsefoid	oid	pg_filespace.oid	The object ID of the filespace
fsedbid	smallint	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`.

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.
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.

Column	Type	References	Description
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
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.
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 HAWQ, inheritance relationships are created by both the INHERITS clause (standalone inheritance) and the PARTITION BY clause (partitioned child table inheritance) of CREATE TABLE.

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.

Column	Type	References	Description
lanname	name		Name of the language.
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.

Column	Type	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_locks

The view `pg_locks` provides access to information about the locks held by open transactions within HAWQ 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.

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_opclass.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_opclass.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 (them **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		Is the lock held by a writer process?
gp_segment_id	integer		The HAWQ segment id (dbid) where the lock is held.

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.

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_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.

Column	Type	References	Description
nspname	name		Namespace name.
nspowner	oid	pg_authis.oid	Owner of the namespace.
nspacl	aclitem[]		Access privileges as given by GRANT and REVOKE.

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.

Column	Type	References	Description
opname	name		Operator name.
opnamesapce	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	bool		This operator supports hash joins.
oprleft	oid	pg_type.oid	Type of the left operand.
oprright	oid	pg_type.oid	Type of the left operand.
oprresult	oid	pg_type.oid	Type of the result.
oprcom	oid	pg_operator.oid	Commutator of this operator, if any.
opnegate	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_resqueue.oid	If this operator supports merge joins, the less-than operator that compares the left and right operand types (L<R).
oprgtcmpop	oid	pg_authid.oid	If this operator supports merge joins, the greater-than operator that compares the left and right operand types (L>R).

Column	Type	References	Description
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.

Column	Type	References	Description
parrelid	oid	pg_class.oid	The object identifier of the table.
pakind	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	pg_attribute.oid	The number of attributes that define this level.
paratts	int2vector		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.

Column	Type	Description
schemaname	name	The name of the schema where the partitioned table is located.
tablename	name	The 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.
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.

Column	Type	Modifiers	Storage
parecoid	oid	not null	plain
parecattnum	smallint	not null	plain
parecattoptions	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).

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 parchildrelid) 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.
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.

Column	Type	References	Description
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.
partemplatespace	oid		

pg_partition_templates

The `pg_partition_templates` system view is used to show the subpartitions that were created using a subpartition template.

Column	Type	Description
schemaname	name	The name of the schema where the partitioned table is located.
tablename	name	The name of the top-level parent table.
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.

Column	Type	Description
schemaname	name	The name of the schema where the partitioned table is located.
tablename	name	The name of the top-level parent table.
partitionschemaname	name	
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.
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.

Column	Type	Description
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	
partitionboundary	text	
parenttablespace	name	
partitiontablespace	name	

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 a HAWQ 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.

Column	Type	Description
tmplname	name	Name of the language for this template.
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.

Column	Type	References	Description
proname	name		Name of the function.
pronamesapce	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 .
prodataaccess	char		

pg_resourcetype

T he `pg_resourcetype` system catalog table contains information about the extended attributes that can be assigned to HAWQ 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.

Column	Type	Description
restypid	smallint	The resource type ID.
resname	name	The name of theresource 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 HAWQ 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.

Column	Type	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.
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.

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 HAWQ 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.

Column	Type	References	Description
resqueueid	oid	pg_resqueue.oid	The object ID of the associated resource queue.
restypid	smallint		The resource type, derived from the pg_ resourcetype system table.
ressetting	opague 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.

Column	Type	References	Description
rulename	name		Rule name.
ev_class	oid	pg_class.oid	Rule specific to table.
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.

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		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.

Column	Type	References	Description
rolcreaterextgpfd	boolean		Role may create readable external tables that use the gpfdist protocol.
rolcreaterexthttp	boolean		Role may create readable external tables that use the gpfdist protocol.
rolcreatewextgpfd	boolean		Role may create writable external tables that use the gpfdist protocol.
rolcreaterexthdfs	boolean		
rolcreatewexthdfs	boolean		

pg_shdepend

The `pg_shdepend` system catalog table records the dependency relationships between database objects and shared objects, such as roles. This information allows the HAWQ 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 in a HAWQ 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

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.
refclassid	oid	any OID column	The OID of the system catalog the referenced object is in (must be a shared catalog).
refobjid	oid		The OID of the specific referenced object.
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 in a HAWQ system: there is only one copy of `pg_shdescription` per system, not one per database.

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 catalog where this object appears..
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`.

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.
usesysid	oid	pg_type.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	timestamptz		Time query began execution.
backend_start	timestamptz		Time backend process was started.
client_addr	inet		Client address.
client_port	integer		Client port.
application_name	text		Client application name.
xact_start	timestamptz		Transaction start time.

pg_stat_last_operation

The `pg_stat_last_operation` table contains metadata tracking information about database objects (tables, views, etc.).

Column	Type	References	Description
classid	oid	pg_class.oid	The OID of the system catalog where this object appears.
objid	oid	any OID column	OID of the object in the system catalog.
staactionname	name		Action 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.
statime	timestamp with timezone		The timestamp of the operation. This is the same timestamp that is written to the HAWQ 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.)

Column	Type	References	Description
classid	oid	pg_class.oid	The OID of the system catalog where this object appears.
objid	oid	any OID column	OID of the object in the system catalog.
staactionname	name		Action 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 HAWQ 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).

Column	Type	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	timestamp with timezone	The timestamp of the operation. This is the same timestamp that is written to the HAWQ 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 view `pg_stat_partition_operations` shows details about the last operation performed on a partitioned table.

Column	Type	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	timestamp with timezone	The timestamp of the operation. This is the same timestamp that is written to the HAWQ 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.

Column	Type	Description
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_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 planner. 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.

Column	Type	References	Description
starelid	oid	pg_class.oid	The table or index that the described column belongs to.
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	pg_authid.oid	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.

Column	Type	References	Description
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	float[]		Numerical statistics of the appropriate kind for the N th slot, or NULL if the slot kind does not involve numerical values.
stavalues N	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 HAWQ 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.

Column	Type	Description
queueid	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 HAWQ system: there is only one copy of `pg_tablespace` per system, not one per database.

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.

Column	Type	References	Description
tgrelid	oid	pg_class.oid Note that HAWQ does not enforce referential integrity.	The table name where this trigger is set.
tgname	name		Unique trigger name (trigger names must be unique for the same table).
tgfoid	oid	pg_proc.oid Note that HAWQ does not enforce referential integrity.	The function to be called.
tgtype	int2	pg_authid.oid	Bit mask identifying trigger conditions.
tgenabled	boolean		True if trigger is enabled.
tgisconstraint	boolean		True if a trigger implements a referential integrity constraint.
tgconstrname	name		Referential integrity constraint name.
tgconstrrelid	oid	pg_class.oid Note that HAWQ does not enforce referential integrity.	The table referenced by a 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 unused.
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.

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.
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.

Column	Type	References	Description
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.
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.

Column	Type	References	Description
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 (typlen > 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 HAWQ. 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	integer		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.
typndims	integer		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.

Column	Type	References	Description
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.

Column	Type	Modifiers	Storage	Description
typeid	oid	not null	plain	Foreign key to pg_ attribute.
tyoptions	text[]		extended	The actual options.

pg_user_mapping

The `pg_user_mapping` catalog stores the mappings from local users to remote users. You must have administrator privileges to view this catalog.

Column	Type	Modifiers	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.

Column	Type	References	Description
winfnoid	regproc	pg_proc.oid	The OID in the 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.
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.

Column	Type	References	Description
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 20

hawq_toolkit Reference

This chapter describes the `hawq_toolkit` administrative schema views and their usage.

What is hawq_toolkit?

Pivotal provides an administrative schema called `hawq_toolkit` that you can use to query the system catalogs, log files, and operating environment for system status information. The `hawq_toolkit` schema contains a number of views that you can access using SQL commands. The `hawq_toolkit` schema is accessible to all database users, although some objects may require superuser permissions. For convenience, you may want to add the `hawq_toolkit` schema to your schema search path. For example:

```
=> ALTER ROLE myrole SET search_path TO my schema,hawq_toolkit;
```

Viewing HAWQ Database Server Log Files

Each component of a HAWQ Database system (master, standby master and primary segments) keeps its own server log files, The `hawq_log_*` family of views allows you to issue SQL queries against the server log files to find particular entries of interest. The user of these views require superuser permissions.

hawq_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 user of this view requires superuser permissions.

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.

hawq_log_database

This view uses an external table to read log files of the entire HAWQ system (master and segments) 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.

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").

Column	Description
logsegment	The segment content identifier (prefixed with "seg". 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	Detailed message test associated with an error message.
loghint	Hint message test 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.

hawq_log_master_concise

This view uses an external table to read a subset of the log fields from the master log files. The use of this view requires superuser permissions.

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").

Column	Description
logseverity	LOG, ERROR, FATAL, PANIC, DEBUG1, DEBUG2.
logmessage	Log or error message text.

hawq_log_system

This view uses an external table to read the server log files of the entire HAWQ system (master and segments) and lists all log entries. Associated log entries can be identified by session id (`logsession`) and command id (`logcmdcount`). The use of this view requires superuser permissions.

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"). 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	Detailed message test associated with an error message.
loghint	Hint message test associated with an error message.
logquery	The internally-generated query text.

Column	Description
logquerypos	The cursor index into the internally-generated query text.
logcontext	The context in which this message gets generated.
logdebug	Query string with full details 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 for Tables that Require Routine Maintenance

hawq_stats_missing

This view shows tables that do not have statistics and therefore may require an ANALYZE to be run on the table.

Column	Description
smischema	The schema name.
smitable	The table name.
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, parent tables of partitioned tables are always empty and will always return a false result.
smicols	The number of columns in the table.
smirecs	The number of rows in the table.

Checking Query Disk Spill Space Usage

The `hawq_workfile_*` views show information about all the queries that are currently using disk spill space. HAWQ 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 HAWQ Database configuration parameters `hawq_workfile_limit_per_query` and `hawq_workfile_limit_per_segment`.

hawq_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.

Column	Type	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	HAWQ 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.
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.

hawq_workfile_usage_per_query

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.

Column	Type	Description
command_cnt	integer	Command ID of the query.

Column	Type	Description
content	smallint	The content identifier for a segment instance.
current_query	text	Current query that the process is running.
datname	name	HAWQ 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.
numflies	bigint	The number of files created.
state	text	The state of the query that created the work file
usename	name	Role name.

hawq_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.

Column	Type	Description
content	smallint	The content identifier for a segment instance.
size	bigint	The total size of the work files on a segment.
numflies	bigint	The number of files created.

Checking Database Object Sizes and Disk Space

hawq_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 permissions to access.

Column	Description
sotdoid	The object ID of the table.
sotdsize	The total size of the table in bytes (main relation, plus oversized (toast) attributes, plus 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-only (AO) tables.
sotdschemaname	The schema name.
sotdtablename	The table name.

hawq_size_of_table_uncompressed

This view shows the uncompressed table size for append-only (AO) tables. Otherwise, the table size on disk is shown. The use of this view requires superuser permissions.

Column	Description
sotuoid	The object ID of the table.
sotusize	The uncompressed 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.

hawq_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 permissions to access.

Column	Description
sotaidoid	The object ID of the parent table.
sotaidtablesize	The disk size of the table.
sotaididsize	The total size of all indexes on the table.
sotaidschemaname	The name of the schema.
sotaidtablename	The name of the table.

hawq_size_of_schema_disk

This view shows schema sizes for the schemas in the current database. This view is accessible to all users, however non-superusers will only be able to see schemas that they have permissions to access.

Column	Description
sosdnsp	The name of the schema.
sosdschematablesize	The total size of tables in the schema in bytes.
sosdschemaidxsize	The total size of indexes in the schema in bytes.

hawq_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.

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-only 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.

hawq_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 permissions to access.

Note: In HAWQ, the data of a database consists of data stored in the local storage of the master, and data stored in HDFS. When the HDFS is online, the sodddatsize includes the sizes of both two kinds of data; while the HDFS is offline, the sodddatsize only includes the size of data stored in the local storage of the master.

Column	Description
sodddatname	The name of the database.
sodddatsize	The size of the database in bytes.

Chapter 21

Managing HAWQ Log Files

HAWQ server log file output tends to be voluminous, especially at higher debug levels, and can file up quickly. These files do not need to be saved indefinitely. Administrators should rotate the log files periodically so new log files are started and old ones are removed. HAWQ log file rotation should be enabled on the master and all segment instances.

For more information about the HAWQ log files, see *HAWQ Server Configuration Parameters*.

Daily Log Files

Daily log files are created in `pg_log` of the master and each segment data directory using the naming convention of: `gpdb-YYYY-MM-DD.log`. Although log files are rolled over daily, they are not automatically truncated or deleted. Administrators need to implement scripts or programs to periodically clean up old log files in the `pg_log` directory of the master and each segment instance.

Management Utility Log Files

Log files for the HAWQ management utilities are written to `~/gpAdminLogs` by default (this location can be changed, if desired). The naming convention for management log files is:

```
<script_name>_<date>.log
```

The log entry format is:

<*timestamp*>:<*utility*>:<*host*>:<*user*>:[INFO|WARN|FATAL]:<*message*>

The log file for a particular utility execution is appended to its daily log file each time that utility is run. More information on HAWQ log file entries as they relate to its command-line management utilities can be found in the section *Management Utility Reference*.