Greenplum® Database 4.2

Developers Guide

Rev: DRAFT

Greenplum Database Developers Guide - DRAFT - 4.2 - Contents

Preface

This guide provides information for Developers creating applications that work with Greenplum Database systems.

About This Guide

This guide explains how clients connect to a Greenplum Database system, how to configure access control and workload management, perform basic administration tasks such as defining database objects, loading and unloading data, writing queries, and managing data, and provides guidance on identifying and troubleshooting common performance issues.

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

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

About the Greenplum Database Documentation Set

As of Release 4.2.3, the Greenplum Database documentation set consists of the following guides.

Table 1 Greenplum Database documentation set

Guide Name	Description
Greenplum Database Database Administrator Guide	Every day DBA tasks such as configuring access control and workload management, writing queries, managing data, defining database objects, and performance troubleshooting.
Greenplum Database System Administrator Guide	Describes the Greenplum Database architecture and concepts such as parallel processing, and system administration tasks for Greenplum Database such as configuring the server, monitoring system activity, enabling high-availability, backing up and restoring databases, and expanding the system.
Greenplum Database Reference Guide	Reference information for Greenplum Database systems: SQL commands, system catalogs, environment variables, character set support, datatypes, the Greenplum MapReduce specification, postGIS extension, server parameters, the gp_toolkit administrative schema, and SQL 2008 support.
Greenplum Database Utility Guide	Reference information for command-line utilities, client programs, and Oracle compatibility functions.
Greenplum Database Installation Guide	Information and instructions for installing and initializing a Greenplum Database system.

Document Conventions

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

Text Conventions

Table 2 Text Conventions

Text Convention	Usage	Examples
bold	Button, menu, tab, page, and field names in GUI applications	Click **Cancel** to exit the page without saving your changes.
italics	New terms where they are defined Database objects, such as schema, table, or columns names	The *master instance* is the `postgres` process that accepts client connections. Catalog information for Greenplum Database resides in the *pg_catalog* schema.
`monospace`	File names and path names Programs and executables Command names and syntax Parameter names	Edit the `postgresql.conf` file. Use `gpstart` to start Greenplum Database.
`monospace italics`	Variable information within file paths and file names Variable information within command syntax	`/home/gpadmin/`*`config_file`* `COPY` *`tablename`* `FROM '`*`filename`*`'`
`monospace bold`	Used to call attention to a particular part of a command, parameter, or code snippet.	Change the host name, port, and database name in the JDBC connection URL: `jdbc:postgresql://`**`host:5432/mydb`**
`UPPERCASE`	Environment variables SQL commands Keyboard keys	Make sure that the Java `/bin` directory is in your `$PATH`. `SELECT * FROM` *`my_table`*`;` Press `CTRL+C` to escape.

Command Syntax Conventions

Table 0.1 Command Syntax Conventions

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

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

Getting Support

EMC support, product, and licensing information can be obtained as follows.

Product information

For documentation, release notes, software updates, or for information about EMC products, licensing, and service, go to the EMC Powerlink website (registration required) at:

http://Powerlink.EMC.com

Technical support

For technical support, go to Powerlink and choose **Support**. On the Support page, you will see several options, including one for making a service request. Note that to open a service request, you must have a valid support agreement. Please contact your EMC sales representative for details about obtaining a valid support agreement or with questions about your account.

1. Introduction to Greenplum

Greenplum Database is a massively parallel processing (MPP) database server based on PostgreSQL open-source technology. MPP (also known as a *shared nothing* architecture) refers to systems with two or more processors that cooperate to carry out an operation - each processor with its own memory, operating system and disks. Greenplum uses this high-performance system architecture to distribute the load of multi-terabyte data warehouses, and can use all of a system's resources in parallel to process a query.

Greenplum Database is essentially several PostgreSQL database instances acting together as one cohesive database management system (DBMS). It is based on PostgreSQL 8.2.15, and in most cases is very similar to PostgreSQL with regard to SQL support, features, configuration options, and end-user functionality. Database users interact with Greenplum Database as they would a regular PostgreSQL DBMS.

The internals of PostgreSQL have been modified or supplemented to support the parallel structure of Greenplum Database. For example, the system catalog, query planner, optimizer, query executor, and transaction manager components have been modified and enhanced to be able to execute queries simultaneously across all of the parallel PostgreSQL database instances. The Greenplum *interconnect* (the networking layer) enables communication between the distinct PostgreSQL instances and allows the system to behave as one logical database.

Greenplum Database also includes features designed to optimize PostgreSQL for business intelligence (BI) workloads. For example, Greenplum has added parallel data loading (external tables), resource management, query optimizations, and storage enhancements, which are not found in standard PostgreSQL. Many features and optimizations developed by Greenplum make their way into the PostgreSQL community. For example, table partitioning is a feature first developed by Greenplum, and it is now in standard PostgreSQL.

Chapter 3, "About Greenplum Query Processing" describes Greenplum Database query processing.

2. About the Greenplum Architecture

Greenplum Database stores and processes large amounts of data by distributing the data and processing workload across several servers or *hosts*. Greenplum Database is an *array* of individual databases based upon PostgreSQL 8.2 working together to present a single database image. The *master* is the entry point to the Greenplum Database system. It is the database instance to which clients connect and submit SQL statements. The master coordinates its work with the other database instances in the system, called *segments*, which store and process the data.

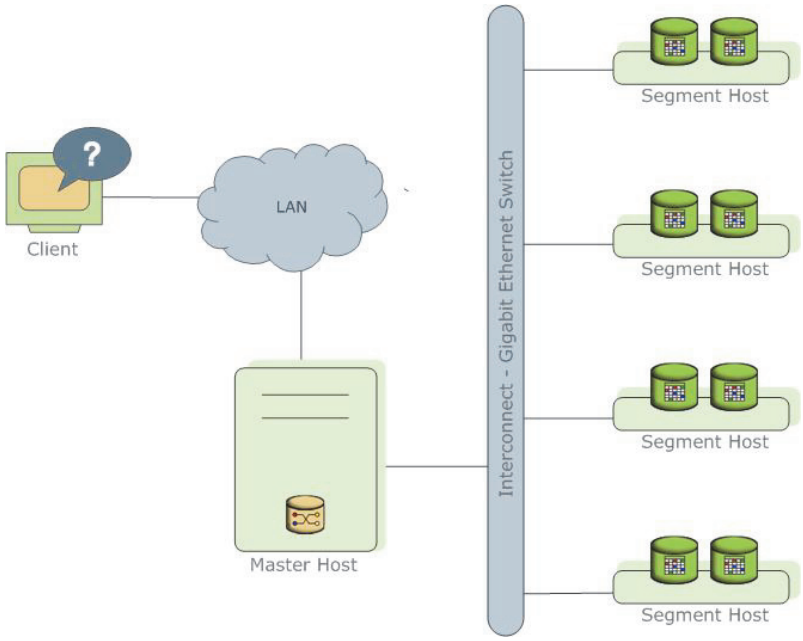


Figure 2.1 High-Level Greenplum Database Architecture

This section describes the components that make up a Greenplum Database system and how they work together:

- About the Greenplum Master
- About the Greenplum Segments
- About the Greenplum Interconnect
- About Redundancy and Failover in Greenplum Database
- About Parallel Data Loading
- About Management and Monitoring

About the Greenplum Master

The *master* is the entry point to the Greenplum Database system. It is the database process that accepts client connections and processes SQL commands that system users issue.

Greenplum Database end-users interact with Greenplum Database (through the master) as they would with a typical PostgreSQL database. They 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 Greenplum Database system itself. The master does not contain any user data; data resides only on the *segments*. 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.

About the Greenplum Segments

In Greenplum Database, the *segments* are where data is stored and the majority of query processing takes place. When a user connects to the database and issues a query, processes are created on each segment to handle the work of that query. For more information about query processes, see the *Greenplum Database Database Administrator Guide*.

User-defined tables and their indexes are distributed across the available segments in a Greenplum Database system; each segment contains a distinct portion of data. The database server processes that serve segment data run under the corresponding segment instances. Users interact with segments in a Greenplum Database system through the master.

In the recommended Greenplum Database hardware configuration, there is one active segment per effective CPU or CPU core. For example, if your segment hosts have two dual-core processors, you would have four primary segments per host.

About the Greenplum Interconnect

The *interconnect* is the networking layer of Greenplum Database. The interconnect refers to the inter-process communication between segments and the network infrastructure on which this communication relies. The Greenplum interconnect uses a standard Gigabit Ethernet switching fabric.

By default, the interconnect uses User Datagram Protocol (UDP) to send messages over the network. The Greenplum software performs 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 TCP. If the interconnect used TCP, Greenplum Database would have a scalability limit of 1000 segment instances. With UDP as the current default protocol for the interconnect, this limit is not applicable.

About Redundancy and Failover in Greenplum Database

You can deploy Greenplum Database without a single point of failure. This section explains the redundancy components of Greenplum Database.

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

About Segment Mirroring

When you deploy your Greenplum Database system, you can optionally configure *mirror* segments. Mirror segments allow database queries to fail over to a backup segment if the primary segment becomes unavailable. To configure mirroring, you must have enough hosts in your Greenplum Database system so the secondary (mirror) segment always resides on a different host than its primary segment. Figure 2.2 shows how table data is distributed across segments when mirroring is configured..

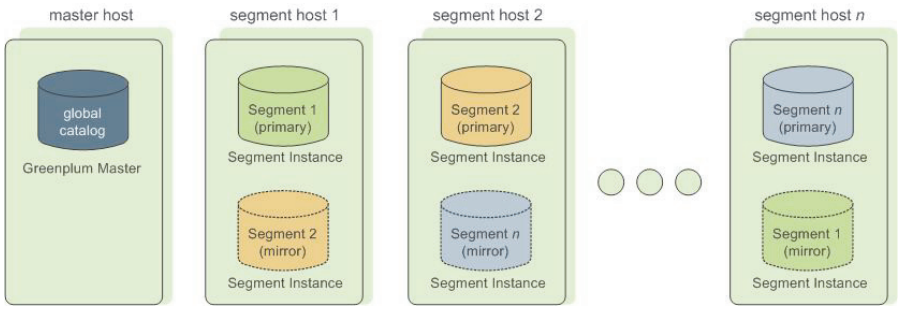


Figure 2.2 Data Mirroring in Greenplum Database

Segment Failover and Recovery

When mirroring is enabled in a Greenplum Database system, the system will automatically fail over to the mirror copy if a primary copy becomes unavailable. A Greenplum Database system can remain operational if a segment instance or host goes down as long as all the data is available on the remaining active segments.

If the master cannot connect to a segment instance, it marks that segment instance as *down* in the Greenplum Database system catalog and brings up the mirror segment in its place. A failed segment instance will remain out of operation until an administrator takes steps to bring that segment back online. An administrator can recover a failed segment while the system is up and running. The recovery process copies over only the changes that were missed while the segment was out of operation.

If you do not have mirroring enabled, the system will automatically shut down if a segment instance becomes invalid. You must recover all failed segments before operations can continue.

About Master Mirroring

You can also 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 unoperational. The standby master is kept up to date by a transaction log replication process, which runs on the standby master host and synchronizes the data between the primary and standby master hosts.

If the primary master fails, the log replication process stops, and the standby master can be activated in its place. Upon activation of the standby 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 Greenplum Database master, accepting client connections on the master port (which must be set to the same port number on the master host and the backup master host).

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.

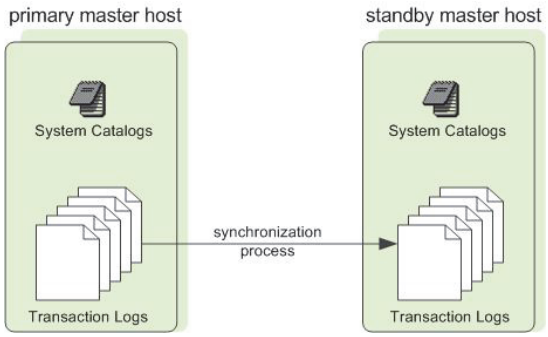


Figure 2.3 Master Mirroring in Greenplum Database

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 redundant Gigabit connections to the Greenplum Database host (master and segment) servers.

About Parallel Data Loading

In a large scale, multi-terabyte data warehouse, large amounts of data must be loaded within a relatively small maintenance window. Greenplum supports fast, parallel data loading with its external tables feature. Administrators can also load external tables in *single row error isolation* mode to filter bad rows into a separate error table while continuing to load properly formatted rows. Administrators can specify an error threshold for a load operation to control how many improperly formatted rows cause Greenplum to abort the load operation.

By using external tables in conjunction with Greenplum Database's parallel file server (`gpfdist`), administrators can achieve maximum parallelism and load bandwidth from their Greenplum Database system.

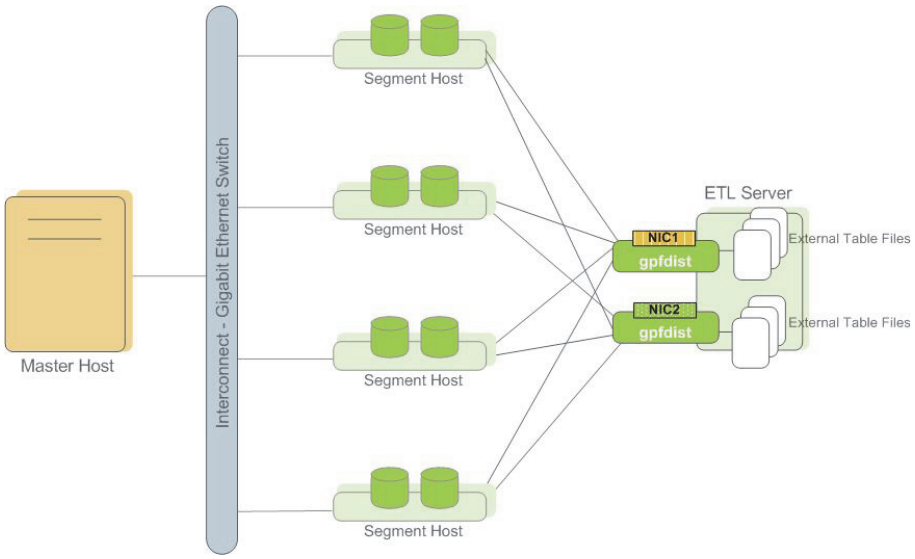


Figure 2.4 External Tables Using Greenplum Parallel File Server (gpfdist)

About Management and Monitoring

Administrators manage a Greenplum Database system using command-line utilities located in `$GPHOME/bin`. Greenplum provides utilities for the following administration tasks:

- Installing Greenplum Database on an Array
- Initializing a Greenplum Database System
- Starting and Stopping Greenplum Database
- Adding or Removing a Host
- Expanding the Array and Redistributing Tables among New Segments
- Managing Recovery for Failed Segment Instances
- Managing Failover and Recovery for a Failed Master Instance

- Backing Up and Restoring a Database (in Parallel)
- Loading Data in Parallel
- System State Reporting

Greenplum provides an optional system monitoring and management tool that administrators can install and enable with Greenplum Database. Greenplum Command Center uses data collection agents on each segment host to collect and store Greenplum system metrics in a dedicated database. Segment data collection agents send their data to the Greenplum master at regular intervals (typically every 15 seconds). Users can query the Command Center database to see query and system metrics. Greenplum Command Center has a graphical web-based user interface for viewing system metrics, which administrators can install separately from Greenplum Database. For more information, see the Greenplum Command Center documentation.

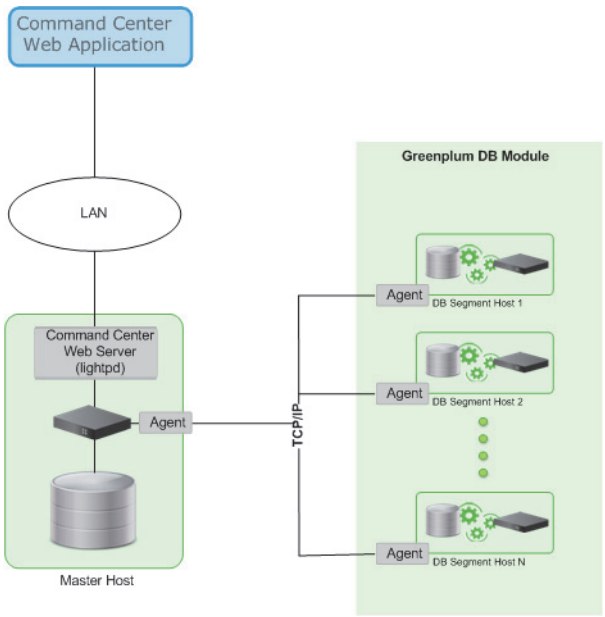


Figure 2.5 Greenplum Command Center Architecture

3. About Greenplum Query Processing

Users issue queries to Greenplum Database as they would to any database management system (DBMS). They connect to the database instance on the Greenplum master host using a client application such as `psql` and submit SQL statements.

Understanding Query 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 3.1. The master dispatches targeted query plans to a single segment, as shown in Figure 3.2. Each segment is responsible for executing local database operations on its own set of data.

Most database operations—such as table scans, joins, aggregations, and sorts—execute across all segments in parallel. Each operation is performed on a segment database independent of the data stored in the other segment databases.

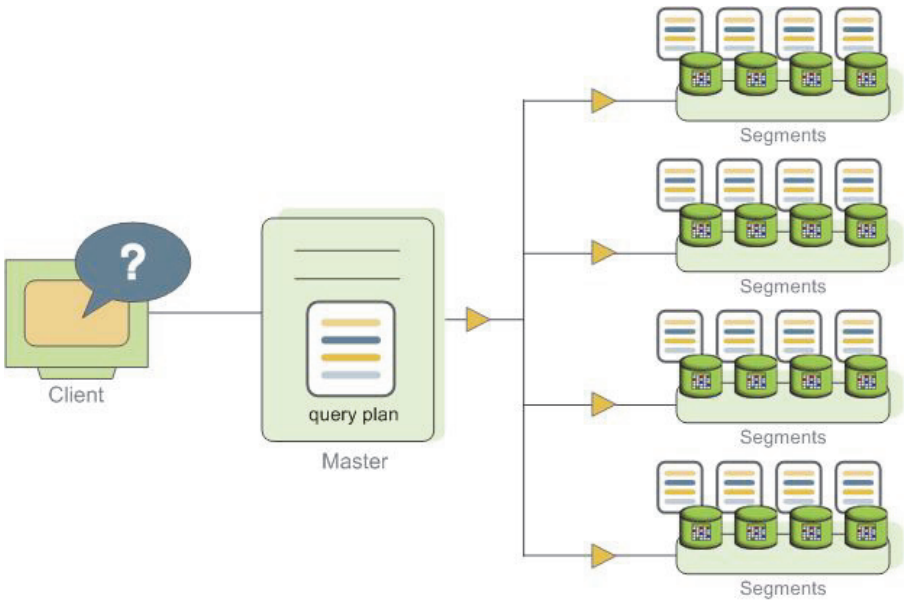


Figure 3.1 Dispatching the Parallel Query Plan

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).

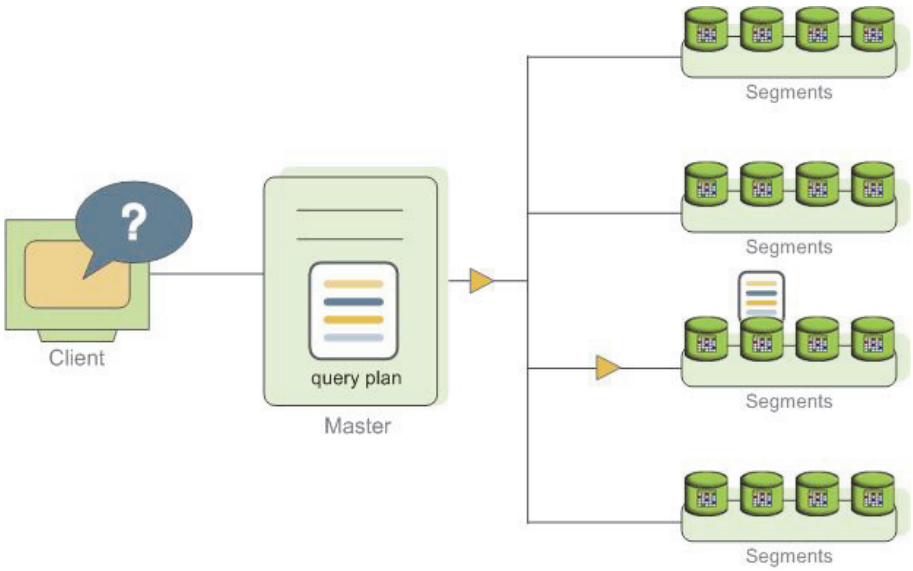


Figure 3.2 Dispatching a Targeted Query Plan

Understanding Greenplum Query Plans

A *query plan* is the set of operations Greenplum Database 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, Greenplum Database 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, Greenplum 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';
```

Figure 3.3 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.

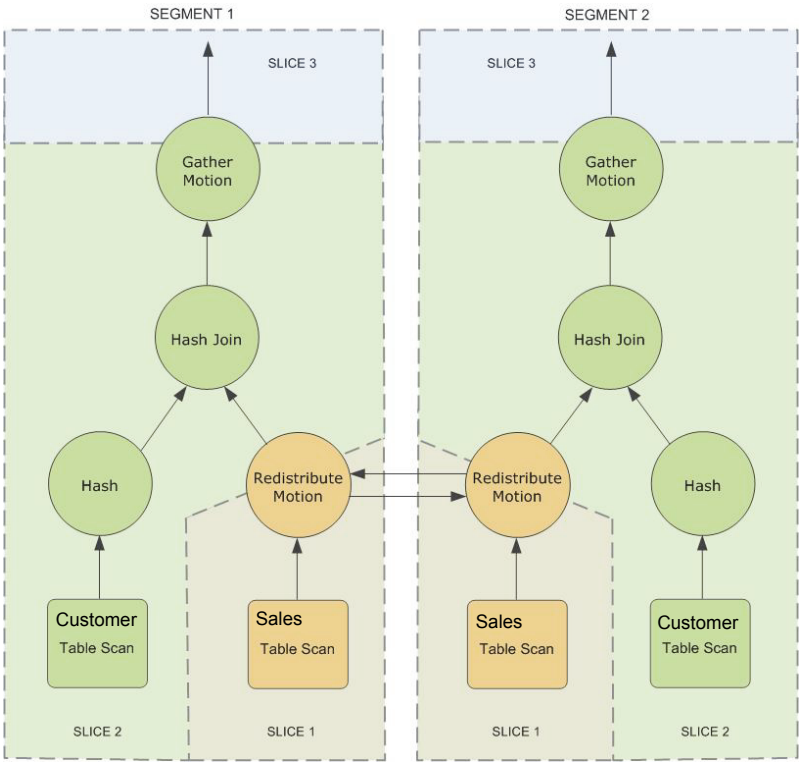


Figure 3.3 Query Slice Plan

For more information on query plans and how to view the plan for a given query with the Greenplum Database `EXPLAIN` command, see "Query Profiling" on page 51.

Understanding Parallel Query Execution

Greenplum 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 Greenplum Database.

Figure 3.4 shows the query worker processes on the master and two segment instances for the query plan illustrated in Figure 3.3.

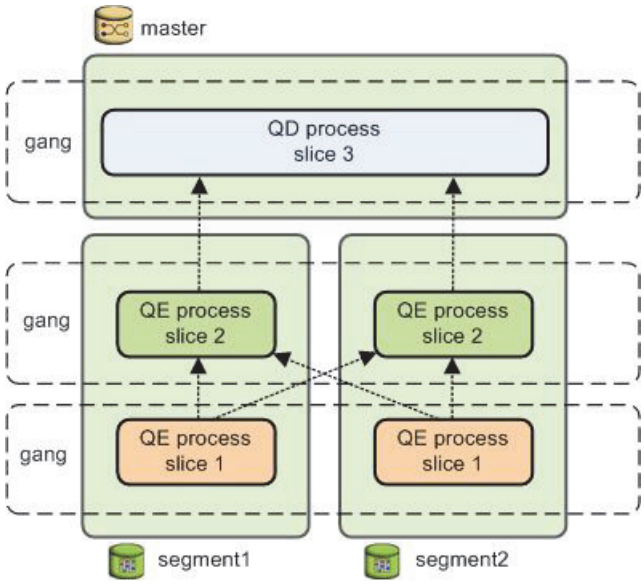


Figure 3.4 Query Worker Processes

4. Database Application Development

In Greenplum Database, you use SQL (Structured Query Language) commands to view, change, and analyze data in a database. You can use built-in functions, including advanced analytic functions with the SQL commands. You can also create user-defined functions with the SQL commands. See Chapter 5, "Querying Data" for information about using SQL and functions in Greenplum Database, and about Greenplum Database built-in functions.

You use PL/pgSQL procedural language to create SQL user-defined functions using pgSQL commands, functions, and operators. PL/pgSQL is installed by default with Greenplum Database. For information about creating user-defined functions, see the User Defined Functions section of the PostgreSQL documentation. The PL/Python language is also installed with Greenplum Database.

With Greenplum Database packages, you can extend Greenplum Database functionality. You can add the following types of functionality that can take advantage of the Greenplum Database MPP architecture:

- Create user-defined in-database functions in languages such as C, Java, and Python.
- Access to R and MADlib analytics functions.
- Access to PostGIS geographic information systems (GIS) functions.
- Access cryptographic functions.
- Access Hadoop file systems.

You can also use Greenplum Database loaders, and APIs, to support and enhance the Greenplum Database functions and applications.

- With Greenplum Database loaders you can perform ETL functions on large amounts of data.
- With the APIs and client software, you can access the data custom analytics functionality available from Greenplum Database.

For information about Greenplum Database tools, see Chapter B, "Greenplum Database Tools".

Enhancing Greenplum Database with Extensions

Greenplum Database extensions add to the capabilities of Greenplum Database. Extensions are available as installable packages from the EMC Download Center. See this article for the latest install and configuration information regarding any supported Greenplum Database packages http://csgateway.emc.com/primus.asp?id=emc288189.

Language Extenstions

You can install the following Greenplum Database extensions to develop database functions with different languages:

- PL/Java Extension - Lets you can access Java methods as database functions from JAR files that have been installed in Greenplum Database.
- PL/Perl Extension - Lets you can create user-defined functions using the Perl language.
- PL/R Extension - Lets you can create user-defined functions using the R language and access R functions from R packages that have been installed in Greenplum Database.

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

Note: If a language is installed into the Greenplum database template1, all subsequently created databases have the language installed automatically. For information about the Greenplum database template1, see "Creating and Managing Databases" in the *Greenplum Database Database Administrator Guide*.

Function Library Extenstions

You can extend Greenplum Database by installing these Greenplum Database extensions.

- MADlib Extension. MADlib is an open-source library for scalable in-database analytics. The MADlib functions provide data-parallel implementations of mathematical, statistical and machine-learning methods for structured and unstructured data.

  MADlib features:

  - Algorithms to support your analytics needs including: classification, regression, recommendation, and cluster analysis.
  - The ability to process directly within your analytic data warehouse, keeping your data secure, preventing difficult to manage data silos, and eliminating costly data exports.
  - The ability to process large amounts of data by utilizing Greenplum Database parallelism.

  For information about MADlib, see http://madlib.net/.
- PostGIS Extension. PostGIS adds support for geographic objects to the Greenplum Database. PostGIS enables a database to be used as a spatial database for geographic information systems (GIS). PostGIS follows the OpenGIS "Simple Features Specification for SQL" and has been certified as compliant with the "Types and Functions" profile.

  The package must be enabled for each database.

  For information about PostGIS, see http://postgis.refractions.net/
- pgcrypto Extension. The pgcrypto library provides cryptographic functions for the Greenplum Database. For information about pgcrypto, see http://www.postgresql.org/docs/8.3/static/pgcrypto.html.
- Oracle Compatibility Functions. Greenplum Database supports SQL functions for Oracle compatibility. The functions are useful for production work.

gNet for Hadoop Extension Package

The Greenplum Database gNet database software connects Greenplum Database and Greenplum supported Hadoop distributions.

Greenplum Database enables high-performance parallel import and export of compressed and uncompressed data from Hadoop clusters using gNet for Hadoop, a parallel communications transport supports direct-query interoperability between Greenplum Database nodes and corresponding Hadoop nodes.

To further streamline resource consumption during load times, custom-format data (binary, Pig, Hive, etc.) in Hadoop can be converted to Greenplum Database format via MapReduce, and then imported into Greenplum Database. This is a high-speed direct integration option that provides an efficient and flexible data exchange between Greenplum Database and Hadoop.

To install the gNet extension, use the Greenplum Database utility `gppkg`.

When you install the gNet package, the gNet Java API documentation is installed in `$GPHOME/docs/javadoc`

Managing Greenplum Database Packages

The Greenplum Database utility `gppkg` manages packages. The utility manges Greenplum Database package extensions along with their dependencies, across an entire cluster.

The `gppkg` option `-q --all` lists the installed packages and their versions.

See the *Greenplum Database Utility Guide* for information about `gppkg`.

Creating and Using Server Functions

You can use Greenplum Database built-in functionality to create user-defined functions that perform common tasks within Greenplum Database to take advantage of the Greenplum Database capability to distribute work among the Greenplum Database hosts. For example, you can create functions to maintain sets of tables or perform complex searches over multiple tables.

Built-in window functions let you easily compose complex online analytical processing (OLAP) queries using standard SQL commands. For example, you can create window expressions from window functions to calculate moving averages or sums over various intervals.

You can also extend Greenplum Database built-in advanced analytic functions by adding Greenplum Database extensions to access MADlib and R analytics functions to create functions that generate statistics and predictive models.

You can also create functions in programming languages such as C, Python, Java, and Perl.
[**MSK**] Is this the correct feature set?

Table 4.1 Procedural Language Feature Support

Feature	SQL	PL/ pgSQL	C	PL/Java	PL/ Python	PL/Perl
IN/INOUT/OUT parameters						
named parameters						
savepoints						
logging						
SQL execution						
complex types						
returning sets						

Greenplum Database Function Classification

To ensure the correct results are generated from SQL statements, a Greenplum Database attribute for functions classifies functions based on the results that are returned by the function. The attribute value can be either `IMMUTABLE`, `STABLE`, or `VOLITILE` to designate consistency of the data returned from the function. The most consistent functions are `IMMUTABLE` functions that always returns the same results, The least consistent are `VOLITILE` functions, where the values returned can change within a single table scan table. For information about the Greenplum Database function classifications, see "Using Functions and Operators" in Chapter 4, "Database Application Development".

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.

Configuring Greenplum Database Hosts

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

- For languages such as Java that requires a JAR file or C that requires a shared object file, the file must be installed on every Greenplum Database host.
- For Greenplum Database extensions that can be modified the modification must be made to each Greenplum Database host. For example, you can add modules to R or MADlib. The module must be installed on every Greenplum Database host.
  [**MSK**] what does `gppkg` handle modified extension? Do you have to install the modifications separately when adding a host?

To add a file or install a module, you can use the Greenplum Database `gpscp` utility. See the *Greenplum Database Utility Guide* for information about the utility.

Other steps might be required when extending Greenplum Database. For example, when you add a JAR file, the Greenplum Database `pljava_classpath` environment variable must be updated. See the documentation on the extension for more information.

Trusted and Untrusted Language

Language extensions can as either *trusted* or *untrusted*. A trusted language is believed not to grant access to anything outside the normal SQL execution environment. An untrusted language does not offer any way of restricting what users can do in it.

Only superusers may create functions in untrusted languages.

For trusted languages certain operations are disabled to preserve security. In general, the operations that are restricted are those that interact with the environment. This includes file handle operations, and access to external modules.

Sometimes it is desirable to write functions that are not restricted. For example, one might want a function that sends mail. To handle these cases, a language can also be installed as an untrusted language. In this case the full language is available.

For untrusted languages, only users with the database superuser privilege can use the language to create new functions.

For example, PL/Java is installed as a trusted language and untrusted language. The `pg_language` system catalog table registers languages in which you can write functions or stored procedures.

Table 4.2 Language and Trust Status

Language	Database Name	Trust Status
SQL	sql	trusted
PL/pgSQL	plpgsql	trusted
C	c	untrusted
PL/Java	java javau	trusted untrusted
PL/Python	plpythonu	untrusted
PL/Perl	plperl plperlu	trusted untrusted
PL/R	plr	untrusted

[**MSK**] is untrusted pl/perl supported?

Notes on Creating and Using User-defined Functions

[**MSK**] Are there limitations on combining user-defined functions created in different languages or combining functions from different extensions?

Accessing External Data Sources

You can access external sources with Greenplum external table functionality. You can access various data sources. The data can be from the following sources:

- Text files with delimited data, for example CSV files.
- Data accessed with the http protocol.
- Data from a Hadoop file system (HDFS) .

See Chapter 17, "External Data Sources" for information about accessing files and external data.

See Chapter 18, "Hadoop Distributed File System" for information about extending Greenplum Database to work with HDFS.

Creating Greenplum Database client applications

Greenplum Database supports the following types of client access:

- Connecting to Greenplum Database with drivers and APIs
  - PostgreSQL ODBC and JDBC database drivers, including DataDirect drivers.
  - APIs
    The C API to PostgreSQL and Greenplum Database libpq.
    The Java API for the PostgreSQL JDBC database driver
    Perl DBI pgperl
    DBI pygresql
- Client tools. The Greenplum Database client tools installer installs the following client tools:
  - For UNIX platforms:
    PostgreSQL Interactive Terminal (psql)
    Greenplum MapReduce Client Program (gpmapreduce)
  - For Windows platforms
    PostgreSQL Interactive Terminal (psql)
- pgAdmin III for Greenplum Database. A GUI client that supports PostgreSQL databases with all standard pgAdmin III features, while adding support for Greenplum-specific features.
- Eclipse IDE
- Data loading utilities. The Greenplum Database load tools installer installs the following data loading tools on Windows and UNIX platforms:
  - Greenplum parallel file distribution program (gpfdist)
  - Greenplum data loading utility (gpload)

Most third-party extract-transform-load (ETL) and business intelligence (BI) tools use standard database interfaces, such as ODBC and JDBC, and can be configured to connect to Greenplum Database. Greenplum has worked with the following tools and is in the process of becoming officially certified:

- Business Objects
- Microstrategy
- Informatica Power Center
- Microsoft SQL Server Integration Services (SSIS) and Reporting Services (SSRS)
- Ascential Datastage
- SAS
- Cognos

Greenplum Professional Services can assist users in configuring their chosen third-party tool for use with Greenplum Database.

Connecting clients to Greenplum Database

To connect to Greenplum Database with client tools configure pg_hba.conf. A default pg_hba.conf file is installed when the data directory is initialized by initdb. It is possible to place the authentication configuration file elsewhere.
Related file pg_ident.conf

For information about client authentication, see
http://www.postgresql.org/docs/8.3/static/client-authentication.html

Performance

[COMMENT] Very rough draft - more information needs to be added

Environment

Recommendations and guidelines for setting up and configuring GPDB

Installation Guide has some information in “Estimating Storage Capacity.”
See also MPP-19003.

Database Administrator Guide has sections on managing workloads and measuring performance.

“Managing Workload and Resources”
“Defining Database Performance”

Performance monitoring

Tools, tips, recommendations

Command Center?

Performance tuning

Tools, tips, recommendations

Security

Access to a Greenplum database and database objects is based on PostgreSQL database roles, roles attributes, and database object privileges.

See the PostgreSQL documentation on roles, privileges, and the pg_hba.conf and pg_ident.conf files.

Also see the Greenplum Database Administration Guide section “Configuring Client Authentication.”

5. Querying Data

This chapter describes how to use SQL (Structured Query Language) in Greenplum Database. You enter SQL commands called *queries* to view, change, and analyze data in a database using the PostgreSQL client `psql` or similar client tools.

Defining Queries

This section describes how to construct SQL queries in Greenplum Database.

Greenplum Database SQL

SQL is a standard language for accessing databases. The language consists of elements that enable data storage, retrieval, analysis, viewing, manipulation, and so on. You use SQL commands to construct queries and commands that the Greenplum Database engine understands. SQL queries consist of a sequence of commands. Commands consist of a sequence of valid tokens in correct syntax order, terminated by a semicolon (`;`). For more information about SQL commands, see the *Greenplum Database Reference Guide*.

Greenplum Database uses the PostgreSQL structure and syntax with some enhancements and exceptions. For more information about SQL rules and concepts in PostgreSQL, see SQL Syntax in the PostgreSQL documentation.

SQL Value Expressions

SQL value expressions consist of one or more values, symbols, operators, SQL functions, and data. The expressions compare data or perform calculations and return a value as the result. Calculations include logical, arithmetic, and set operations.

The following are value expressions:

- An aggregate expression
- An array constructor
- A column reference
- A constant or literal value
- A correlated subquery
- A field selection expression

- A function call
- A new column value in an `INSERT` or `UPDATE`
- An operator invocation column reference
- A positional parameter reference, in the body of a function definition or prepared statement
- A row constructor
- A scalar subquery
- A search condition in a `WHERE` clause
- A target list of a `SELECT` command
- A type cast
- A value expression in parentheses, useful to group sub-expressions and override precedence
- A window expression

SQL constructs such as functions and operators are expressions but do not follow any general syntax rules. Fore more information about these constructs, see "Using Functions and Operators" on page 34.

Column References

A column reference has the form:

```
correlation.columnname
```

Here, *correlation* is the name of a table (possibly qualified with a schema name) or an alias for a table defined with a `FROM` clause or one of the keywords `NEW` or `OLD`. `NEW` and `OLD` can appear only in rewrite rules, but you can use other correlation names in any SQL statement. If the column name is unique across all tables in the query, you can omit the "*correlation.*" part of the column reference.

Positional Parameters

Positional parameters are arguments to SQL statements or functions that you reference by their positions in a series of arguments. For example, `$1` refers to the first argument, `$2` to the second argument, and so on. The values of positional parameters are set from arguments external to the SQL statement or supplied when SQL functions are invoked. Some client libraries support specifying data values separately from the SQL command, in which case parameters refer to the out-of-line data values. A parameter reference has the form:

```
$number
```

For example:

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

Here, the `$1` references the value of the first function argument whenever the function is invoked.

Subscripts

If an expression yields a value of an array type, you can extract a specific element of the array value as follows:

```
expression[subscript]
```

You can extract multiple adjacent elements, called an *array slice*, as follows (including the brackets):

```
expression[lower_subscript:upper_subscript]
```

Each subscript is an expression and yields an integer value.

Array expressions usually must be in parentheses, but you can omit the parentheses when the expression to be subscripted is a column reference or positional parameter. You can concatenate multiple subscripts when the original array is multidimensional. For example (including the parentheses):

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
(arrayfunction(a,b))[42]
```

Field Selection

If an expression yields a value of a composite type (row type), you can extract a specific field of the row as follows:

```
expression.fieldname
```

The row expression usually must be in parentheses, but you can omit these parentheses when the expression to be selected from is a table reference or positional parameter. For example:

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

A qualified column reference is a special case of field selection syntax.

Operator Invocations

Operator invocations have the following possible syntaxes:

expression operator expression (binary infix operator)
operator expression (unary prefix operator)
expression operator (unary postfix operator)

Where *operator* is an operator token, one of the key words `AND`, `OR`, or `NOT`, or qualified operator name in the form:

```
OPERATOR(schema.operatorname)
```

Available operators and whether they are unary or binary depends on the operators that the system or user defines. For more information about built-in operators, see "Built-in Functions and Operators" on page 36.

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)
```

"Built-in Functions and Operators" on page 36 lists the built-in functions. You can add custom functions, too.

Aggregate Expressions

An aggregate expression applies an aggregate function across the rows that a query selects. An aggregate function performs a calculation on a set of values and returns a single value, such as the sum or average of the set of values. The syntax of an aggregate expression is one of the following:

- *aggregate_name* (*expression* [, ...]) [FILTER (WHERE *condition*)]–operates across all input rows for which the expected result value is non-null. ALL is the default.
- *aggregate_name* (ALL *expression* [, ...]) [FILTER (WHERE *condition*)]–operates identically to the first form because ALL is the default
- *aggregate_name* (DISTINCT *expression* [, ...]) [FILTER (WHERE *condition*)]–operates across all distinct non-null values of input rows
- *aggregate_name* (*) [FILTER (WHERE *condition*)]–operates on all rows with values both null and non-null. Generally, this form is most useful for the count(*) aggregate function.

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

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

You can specify a condition with the FILTER clause 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, window function, or outer reference. If you use a user-defined aggregate function, declare the state transition function as STRICT (see CREATE AGGREGATE).

For predefined aggregate functions, see "Aggregate Functions" on page 37. You can also add custom aggregate functions.

Greenplum Database provides the MEDIAN aggregate function, which returns the fiftieth percentile of the PERCENTILE_CONT result and special aggregate expressions for inverse distribution functions as follows:

```
PERCENTILE_CONT(_percentage_) WITHIN GROUP (ORDER BY
_expression_)
PERCENTILE_DISC(_percentage_) WITHIN GROUP (ORDER BY
```

```
_expression_)
```

Currently you can use only these two expressions with the keyword `WITHIN GROUP`.

Limitations of Aggregate Expressions

The following are current limitations of the aggregate expressions:

- Greenplum Database does not support the following keywords: `ALL`, `DISTINCT`, `FILTER` and `OVER`. See Table 5.5, "Advanced Aggregate Functions" on page 40 for more details.
- Greenplum Database does not support the following grouping specifications: `CUBE`, `ROLLUP`, and `GROUPING SETS`.
- An aggregate expression can appear only 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 form. This restriction applies to the query level to which the aggregate belongs.
- When an aggregate expression appears in a subquery, the aggregate is normally evaluated over the rows of the subquery. If the aggregate's arguments contain only outer-level variables, the aggregate belongs to the nearest such outer level and evaluates over the rows of that query. The aggregate expression as a whole is then an outer reference for the subquery in which it appears, and the aggregate expression acts as a constant over any one evaluation of that subquery. See "Scalar Subqueries" on page 30 and "Subquery Expressions" on page 37.
- Greenplum Database does not support `DISTINCT` with multiple input expressions.

Window Expressions

Window expressions allow application developers to more easily compose complex online analytical processing (OLAP) queries using standard SQL commands. For example, with window expressions, users can calculate moving averages or sums over various intervals, reset aggregations and ranks as selected column values change, and express complex ratios in simple terms.

A window expression represents the application of a *window function* applied to a *window frame*, which is defined in a special `OVER()` clause. A *window partition* is a set of rows that are grouped together to apply a window function. Unlike aggregate functions, which return a result value for each group of rows, window functions return a result value for every row, but that value is calculated with respect to the rows in a particular window partition. If no partition is specified, the window function is computed over the complete intermediate result set.

The syntax of a window expression is:

```
window_function ( [expression [, ...]] ) OVER (
window_specification )
```

Where *window_function* is one of the functions listed in "Window Functions" on page 38, *expression* is any value expression that does not contain a window expression, and *window_specification* is:

```
[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 }]]
```

and where *window_frame_bound* can be one of:

```
UNBOUNDED PRECEDING
expression PRECEDING
CURRENT ROW
expression FOLLOWING
UNBOUNDED FOLLOWING
```

A window expression can appear only in the select list of a `SELECT` command. For example:

```
SELECT count(*) OVER(PARTITION BY customer_id), * FROM
sales;
```

The `OVER` clause differentiates window functions from other aggregate or reporting functions. The `OVER` clause defines the *window_specification* to which the window function is applied. A window specification has the following characteristics:

- The `PARTITION BY` clause defines the window partitions to which the window function is applied. If omitted, the entire result set is treated as one partition.
- The `ORDER BY` clause defines the expression(s) for sorting rows within a window partition. The `ORDER BY` clause of a window specification is separate and distinct from the `ORDER BY` clause of a regular query expression. The `ORDER BY` clause is required for the window functions that calculate rankings, as it identifies the measure(s) for the ranking values. For OLAP aggregations, the `ORDER BY` clause is required to use window frames (the `ROWS` | `RANGE` clause).

Note: Columns of data types without 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 a specified 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`

- The `ROWS/RANGE` clause defines a window frame for aggregate (non-ranking) window functions. A window frame defines a set of rows within a window partition. When a window frame is defined, the window function computes on the contents of this moving frame rather than the fixed contents of the entire window partition. Window frames are row-based (`ROWS`) or value-based (`RANGE`).

Type Casts

A type cast specifies a conversion from one data type to another. Greenplum Database accepts two equivalent syntaxes for type casts:

```
CAST ( expression AS type )
expression::type
```

The `CAST` syntax conforms to SQL; the syntax with `::` is historical PostgreSQL usage.

A cast applied to a value expression of a known type is a run-time type conversion. The cast succeeds only if a suitable type conversion function is defined. This differs from the use of casts with constants. A cast applied to a string literal represents the initial assignment of a type to a literal constant value, so it succeeds for any type if the contents of the string literal are acceptable input syntax for the data type.

You can usually omit an explicit type cast if there is no ambiguity about the type a value expression must produce; for example, when it is assigned to a table column, the system automatically applies a type cast. The system applies automatic casting only to casts marked "OK to apply implicitly" in system catalogs. Other casts must be invoked with explicit casting syntax to prevent unexpected conversions from being applied without the user's knowledge.

Scalar Subqueries

A scalar subquery is a `SELECT` query in parentheses that returns exactly one row with one column. Do not use a `SELECT` query that returns multiple rows or columns as a scalar subquery. The query runs and uses the returned value in the surrounding value expression. A correlated scalar subquery contains references to the outer query block.

Correlated Subqueries

A correlated subquery (CSQ) is a `SELECT` query with a `WHERE` clause or target list that contains references to the parent outer clause. CSQs efficiently express results in terms of results of another query. Greenplum Database supports correlated subqueries that provide compatibility with many existing applications. A CSQ is a scalar or table subquery, depending on whether it returns one or multiple rows. Greenplum Database does not support correlated subqueries with skip-level correlations.

Correlated Subquery Examples

Example 1 – Scalar correlated subquery

```
SELECT * FROM t1 WHERE t1.x
            > (SELECT MAX(t2.x) FROM t2 WHERE t2.y = t1.y);
```

Example 2 – Correlated EXISTS subquery

```
SELECT * FROM t1 WHERE
            EXISTS (SELECT 1 FROM t2 WHERE t2.x = t1.x);
```

Greenplum Database uses one of the following methods to run CSQs:

- Unnest the CSQ into join operations--This method is most efficient, and it is how Greenplum Database runs most CSQs, including queries from the TPC-H benchmark.
- Run the CSQ on every row of the outer query--This method is relatively inefficient, and it is how Greenplum Database runs queries that contain CSQs in the `SELECT` list or are connected by `OR` conditions.

The following examples illustrate how to rewrite some of these types of queries to improve performance.

Example 3 - CSQ in the Select List

Original Query

```
SELECT T1.a,
```

```
      (SELECT COUNT(DISTINCT T2.z) FROM t2 WHERE t1.x = t2.y) dt2
FROM t1;
```

Rewrite this query to perform an inner join with `t1` first and then perform a left join with `t1` again. The rewrite applies for only an equijoin in the correlated condition.

Rewritten Query

```
SELECT t1.a, dt2 FROM t1
       LEFT JOIN
       (SELECT t2.y AS csq_y, COUNT(DISTINCT t2.z) AS dt2
             FROM t1, t2 WHERE t1.x = t2.y GROUP BY t1.x)
       ON (t1.x = csq_y);
```

Example 4 - CSQs connected by OR Clauses

Original Query

```
SELECT * FROM t1
WHERE
x > (SELECT COUNT(*) FROM t2 WHERE t1.x = t2.x)
OR x < (SELECT COUNT(*) FROM t3 WHERE t1.y = t3.y)
```

Rewrite this query to separate it into two parts with a union on the `OR` conditions.

Rewritten Query

```
SELECT * FROM t1
WHERE x > (SELECT count(*) FROM t2 WHERE t1.x = t2.x)
UNION
SELECT * FROM t1
WHERE x < (SELECT count(*) FROM t3 WHERE t1.y = t3.y)
```

To view the query plan, use `EXPLAIN SELECT` or `EXPLAIN ANALYZE SELECT`. Subplan nodes in the query plan indicate that the query will run on every row of the outer query, and the query is a candidate for rewriting. For more information about these statements, see "Query Profiling" on page 51.

Advanced Table Functions

Greenplum Database supports table functions with `TABLE` value expressions. You can sort input rows for advanced table functions with an `ORDER BY` clause. You can redistribute them with a `SCATTER BY` clause to specify one or more columns or an expression for which rows with the specified characteristics are available to the same process. This usage is similar to using a `DISTRIBUTED BY` clause when creating a table, but the redistribution occurs when the query runs.

The following command uses a the `TABLE` function with the `SCATTER BY` clause in the the GPText function `gptext.index()` to create an index on the table `messages`:

```
SELECT * FROM gptext.index(TABLE(SELECT * FROM messages
SCATTER BY distrib_id), 'mytest.articles');
```

Note: Based on the distribution of data, Greenplum Database automatically parallelizes table functions with `TABLE` value parameters over the nodes of the cluster.

For information about the function `gptext.index()`, see the Pivotal GPText documentation.

Array Constructors

An array constructor is an expression that builds an array value from values for its member elements. A simple array constructor consists of the key word ARRAY, a left square bracket [, one or more expressions separated by commas for the array element values, and a right square bracket]. For example,

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
```

The array element type is the common type of its member expressions, determined using the same rules as for UNION or CASE constructs.

You can build multidimensional array values by nesting array constructors. In the inner constructors, you can omit the keyword ARRAY. For example, the following two SELECT statements produce the same result:

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
```

Since multidimensional arrays must be rectangular, inner constructors at the same level must produce sub-arrays of identical dimensions.

Multidimensional array constructor elements are not limited to a sub-ARRAY construct; they are anything that produces an array of the proper kind. For example:

```
CREATE TABLE arr(f1 int[], f2 int[]);
INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]],
ARRAY[[5,6],[7,8]]);
SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                     array
------------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
```

You can construct an array from the results of a subquery. Write the array constructor with the keyword ARRAY followed by a subquery in parentheses. For example:

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE
'bytea%');
                          ?column?
------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31}
```

The subquery must return a single column. The resulting one-dimensional array has an element for each row in the subquery result, with an element type matching that of the subquery's output column. The subscripts of an array value built with ARRAY always begin with 1.

Row Constructors

A row constructor is an expression that builds a row value (also called a composite value) from values for its member fields. For example,

```
SELECT ROW(1,2.5,'this is a test');
```

Row constructors have the syntax `rowvalue.*`, which expands to a list of the elements of the row value, as when you use the syntax `.*` at the top level of a `SELECT` list. For example, if table `t` has columns `f1` and `f2`, the following queries are the same:

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

By default, the value created by a `ROW` expression has an anonymous record type. If necessary, it can be cast to a named composite type — either the row type of a table, or a composite type created with `CREATE TYPE AS`. To avoid ambiguity, you can explicitly cast the value if necessary. For example:

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);
CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1'
LANGUAGE SQL;
```

-- In the following query, you do not need to cast the value because there is only one `getf1()` function and therefore no ambiguity:

```
SELECT getf1(ROW(1,2.5,'this is a test'));
 getf1
-------
     1
CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);
CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT
$1.f1' LANGUAGE SQL;
```

-- Now we need a cast to indicate which function to call:

```
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR:  function getf1(record) is not unique
SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
 getf1
-------
     1
SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS
myrowtype));
 getf1
-------
    11
```

You can use row constructors to build composite values to be stored in a composite-type table column or to be passed to a function that accepts a composite parameter.

Expression Evaluation Rules

The order of evaluation of subexpressions is undefined. The inputs of an operator or function are not necessarily evaluated left-to-right or in any other fixed order.

If you can determine the result of an expression by evaluating only some parts of the expression, then other subexpressions might not be evaluated at all. For example, in the following expression:

```
SELECT true OR somefunc();
```

`somefunc()` would probably not be called at all. The same is true in the following expression:

```
SELECT somefunc() OR true;
```

This is not the same as the left-to-right evaluation order that Boolean operators enforce in some programming languages.

Do not use functions with side effects as part of complex expressions, especially in `WHERE` and `HAVING` clauses, because those clauses are extensively reprocessed when developing an execution plan. Boolean expressions (`AND`/`OR`/`NOT` combinations) in those clauses can be reorganized in any manner that Boolean algebra laws allow.

Use a `CASE` construct to force evaluation order. The following example is an untrustworthy way to avoid division by zero in a `WHERE` clause:

```
SELECT ... WHERE x <> 0 AND y/x > 1.5;
```

The following example shows a trustworthy evaluation order:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false
END;
```

This `CASE` construct usage defeats optimization attempts; use it only when necessary.

Using Functions and Operators

Using Functions in Greenplum Database

In Greenplum Database, data is divided up across segments — each segment is a distinct PostgreSQL database. To ensure the correct results are generated from SQL statements, Greenplum Database classifies functions based on the results that are returned by the function. This table describes the Greenplum Database function classifications. For more information about the function classifications and function attributes, see “Built-in Functions and Operators” on page 36.

Table 5.1 Functions in Greenplum Database

Function Type	Greenplum Support	Description	Comments
IMMUTABLE	Yes	Relies only on information directly in its argument list. Given the same argument values, always returns the same result.	
STABLE	Yes, in most cases	Within a single table scan, returns the same result for same argument values, but results change across SQL statements.	Results depend on database lookups or parameter values. `current_timestamp` family of functions is `STABLE`; values do not change within an execution.
VOLATILE	Restricted	Function values can change within a single table scan. For example: `random()`, `currval()`, `timeofday()`.	Any function with side effects is volatile, even if its result is predictable. For example: `setval()`.

In Greenplum Database, data is divided up across segments — each segment is a distinct PostgreSQL database. To prevent inconsistent or unexpected results, do not execute functions classified as `VOLATILE` at the segment level if they contain SQL commands or modify the database in any way. For example, functions such as `setval()` are not allowed to execute on distributed data in Greenplum Database because they can cause inconsistent data between segment instances.

To ensure data consistency, you can safely use `VOLATILE` and `STABLE` functions in statements that are evaluated on and run from the master. For example, the following statements run on the master (statements without a `FROM` clause):

```
SELECT setval('myseq', 201);
SELECT foo();
```

If a statement has a `FROM` clause containing a distributed table and the function in the `FROM` clause returns a set of rows, the statement can run on the segments:

```
SELECT * from foo();
```

Greenplum Database does not support functions that return a table reference (`rangeFuncs`) or functions that use the `refCursor` datatype.

User-Defined Functions

Greenplum Database supports user-defined functions. See Extending SQL in the PostgreSQL documentation for more information.

Use the `CREATE FUNCTION` command to register user-defined functions that are used as described in "Using Functions in Greenplum Database" on page 34. 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.

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

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

Built-in Functions and Operators

The following table lists the categories of built-in functions and operators supported by PostgreSQL. All functions and operators are supported in Greenplum Database as in PostgreSQL with the exception of `STABLE` and `VOLATILE` functions, which are subject to the restrictions noted in "Using Functions in Greenplum Database" on page 34. See the Functions and Operators section of the PostgreSQL documentation for more information about these built-in functions and operators.

Table 5.2 Built-in functions and operators

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
Logical Operators			
Comparison Operators			
Mathematical Functions and Operators	random setseed		
String Functions and Operators	*All built-in conversion functions*	convert pg_client_encoding	
Binary String Functions and Operators			
Bit String Functions and Operators			
Pattern Matching			
Data Type Formatting Functions		to_char to_timestamp	
Date/Time Functions and Operators	timeofday	age current_date current_time current_timestamp localtime localtimestamp now	
Geometric Functions and Operators			

Table 5.2 Built-in functions and operators

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
Network Address Functions and Operators			
Sequence Manipulation Functions	currval lastval nextval setval		
Conditional Expressions			
Array Functions and Operators		*All array functions*	
Aggregate Functions			
Subquery Expressions			
Row and Array Comparisons			
Set Returning Functions	generate_series		
System Information Functions		*All session information functions* *All access privilege inquiry functions* *All schema visibility inquiry functions* *All system catalog information functions* *All comment information functions*	
System Administration Functions	set_config pg_cancel_backend pg_reload_conf pg_rotate_logfile pg_start_backup pg_stop_backup pg_size_pretty pg_ls_dir pg_read_file pg_stat_file	current_setting *All database object size functions*	
XML Functions		xmlagg(xml) xmlexists(text, xml) xml_is_well_formed(text) xml_is_well_formed_document(text) xml_is_well_formed_content(text) xpath(text, xml) xpath(text, xml, text[]) xpath_exists(text, xml) xpath_exists(text, xml, text[]) xml(text) text(xml) xmlcomment(xml) xmlconcat2(xml, xml)	

Window Functions

The following built-in window functions are Greenplum extensions to the PostgreSQL database. All window functions are *immutable*. For more information about window functions, see "Window Expressions" on page 28.

Greenplum Database also supports MADlib and R analytics functions with the GGreenplum Database packages.

Table 5.3 Window functions

Function	Return Type	Full Syntax	Description
`cume_dist()`	`double precision`	`CUME_DIST() OVER ( [PARTITION BY expr] ORDER BY expr )`	Calculates the cumulative distribution of a value in a group of values. Rows with equal values always evaluate to the same cumulative distribution value.
`dense_rank()`	`bigint`	`DENSE_RANK () OVER ( [PARTITION BY expr] ORDER BY expr)`	Computes the rank of a row in an ordered group of rows without skipping rank values. Rows with equal values are given the same rank value.
`first_value(expr)`	same as input *expr* type	`FIRST_VALUE(expr) OVER ( [PARTITION BY expr] ORDER BY expr [ROWS|RANGE frame_expr] )`	Returns the first value in an ordered set of values.
`lag(expr [,offset] [,default])`	same as input *expr* type	`LAG(expr [,offset] [,default]) OVER ( [PARTITION BY expr] ORDER BY expr )`	Provides access to more than one row of the same table without doing a self join. Given a series of rows returned from a query and a position of the cursor, `LAG` provides access to a row at a given physical offset prior to that position. The default `offset` is 1. *`default`* sets the value that is returned if the offset goes beyond the scope of the window. If *`default`* is not specified, the default value is null.
`last_value(expr)`	same as input *expr* type	`LAST_VALUE(expr) OVER ( [PARTITION BY expr] ORDER BY expr [ROWS|RANGE frame_expr] )`	Returns the last value in an ordered set of values.
`lead(expr [,offset] [,default])`	same as input *expr* type	`LEAD(expr [,offset] [,default]) OVER ( [PARTITION BY expr] ORDER BY expr )`	Provides access to more than one row of the same table without doing a self join. Given a series of rows returned from a query and a position of the cursor, `lead` provides access to a row at a given physical offset after that position. If *`offset`* is not specified, the default offset is 1. *`default`* sets the value that is returned if the offset goes beyond the scope of the window. If *`default`* is not specified, the default value is null.

Table 5.3 Window functions

Function	Return Type	Full Syntax	Description
`ntile(expr)`	`bigint`	`NTILE(expr) OVER ( [PARTITION BY expr] ORDER BY expr )`	Divides an ordered data set into a number of buckets (as defined by *expr*) and assigns a bucket number to each row.
`percent_rank( )`	`double precision`	`PERCENT_RANK () OVER ( [PARTITION BY expr] ORDER BY expr )`	Calculates the rank of a hypothetical row *R* minus 1, divided by 1 less than the number of rows being evaluated (within a window partition).
`rank()`	`bigint`	`RANK () OVER ( [PARTITION BY expr] ORDER BY expr )`	Calculates the rank of a row in an ordered group of values. Rows with equal values for the ranking criteria receive the same rank. The number of tied rows are added to the rank number to calculate the next rank value. Ranks may not be consecutive numbers in this case.
`row_number()`	`bigint`	`ROW_NUMBER () OVER ( [PARTITION BY expr] ORDER BY expr )`	Assigns a unique number to each row to which it is applied (either each row in a window partition or each row of the query).

Advanced Analytic Functions

The following built-in advanced analytic functions are Greenplum extensions of the PostgreSQL database. Analytic functions are *immutable*. You can also extend Greenplum Database by adding Greenplum Database extensions to access MADlib and R analytics functions, see Chapter 4, "Database Application Development" for information on Greenplum Database extensions.

.

Table 5.4 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.

Table 5.4 Advanced Analytic Functions

Function	Return Type	Full Syntax	Description
`matrix_transpose(array[])`	Same as input `array` type.	`matrix_transpose( array [[1,1,1],[2,2,2]])`	Transposes a two-dimensional array.
`pinv(array[])`	`smallint[]int[], bigint[], float[]`	`pinv(array[[2.5,0,0],[0,1,0],[0,0,.5]])`	Calculates the Moore-Penrose pseudoinverse of a matrix.
`unnest (array[])`	set of `anyelement`	`unnest( array['one', 'row', 'per', 'item'])`	Transforms a one dimensional array into rows. Returns a set of `anyelement`, a polymorphic pseudotype in PostgreSQL.

Table 5.5 Advanced Aggregate Functions

Function	Return Type	Full Syntax	Description

`MEDIAN (expr)`	`timestamp, timestampz, interval, float`	`MEDIAN (_expression_)`
		Example:
		`SELECT department_id, MEDIAN(salary)`
		`FROM employees`
		`GROUP BY department_id;`
			Can take a two-dimensional array as input. Treats such arrays as matrices.

`PERCENTILE_CONT (expr) WITHIN GROUP (ORDER BY expr [DESC/ASC])`	`timestamp, timestampz, interval, float`	`PERCENTILE_CONT(_percentage_) WITHIN GROUP (ORDER BY _expression_)`
		Example:
		`SELECT department_id,`
		`PERCENTILE_CONT (0.5) WITHIN GROUP (ORDER BY salary DESC)`
		`"Median_cont";`
		`FROM employees GROUP BY department_id;`
			Performs an inverse distirbution function that assumes a continuous distribution model. It takes a percentile value and a sort specification and returns the same datatype as the numeric datatype of the argument. This returned value is a computed result after performing linear interpolation. Null are ignored in this calculation.

`PERCENTILE_DESC (expr) WITHIN GROUP (ORDER BY expr [DESC/ASC])`	`timestamp, timestampz, interval, float`	`PERCENTILE_DESC(_percentage_) WITHIN GROUP (ORDER BY _expression_)`
		Example:
		`SELECT department_id,`
		`PERCENTILE_DESC (0.5) WITHIN GROUP (ORDER BY salary DESC)`
		`"Median_desc";`
		`FROM employees GROUP BY department_id;`
			Performs an inverse distirbution function that assumes a discrete distribution model. It takes a percentile value and a sort specification. This returned value is an element from the set. Null are ignored in this calculation.

Table 5.5 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.
`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.

Table 5.5 Advanced Aggregate Functions

Function	Return Type	Full Syntax	Description
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.

Advanced Analytic Function Examples

These examples illustrate selected advanced analytic functions in queries on simplified example data. They are for the multiple linear regression aggregate functions and for Naive Bayes Classification with `nb_classify`.

Linear Regression Aggregates Example

The following example uses the four linear regression aggregates `mregr_coef`, `mregr_r2`, `mregr_pvalues`, and `mregr_tstats` in a query on the example table `regr_example`. In this example query, all the aggregates take the dependent variable as the first parameter and an array of independent variables as the second parameter.

```
SELECT mregr_coef(y, array[1, x1, x2]),
       mregr_r2(y, array[1, x1, x2]),
       mregr_pvalues(y, array[1, x1, x2]),
       mregr_tstats(y, array[1, x1, x2])
from regr_example;
```

Table `regr_example`:

```
 id | y  | x1 | x2
----+----+----+----
  1 |  5 |  2 |  1
  2 | 10 |  4 |  2
  3 |  6 |  3 |  1
  4 |  8|   3 |  1
```

Running the example query against this table yields one row of data with the following values:

`mregr_coef`:

```
{-7.105427357601e-15,2.00000000000003,0.999999999999943}
```

`mregr_r2`:

```
0.86440677966103

mregr_pvalues:

{0.999999999999999,0.454371051656992,0.783653104061216}

mregr_tstats:

{-2.24693341988919e-15,1.15470053837932,0.35355339059327}
```

Greenplum Database returns `NaN` (not a number) if the results of any of these agregates are undefined. This can happen if there is a very small amount of data.

Note: The intercept is computed by setting one of the independent variables to `1`, as shown in the preceding example.

Naive Bayes Classification Examples

The aggregates `nb_classify and nb_probabilities` are used within a larger four-step classification process that involves the creation of tables and views for training data. The following two examples show all the steps. The first example shows a small data set with arbitrary values, and the second example is the Greenplum implementation of a popular Naive Bayes example based on weather conditions.

Overview

The following describes the Naive Bayes classification procedure. In the examples, the value names become the values of the field *attr*:

1. Unpivot the data.

   If the data is not denormalized, create a view with the identification and classification that unpivots all the values. If the data is already in denormalized form, you do not need to unpivot the data.

2. Create a training table.

   The training table shifts the view of the data to the values of the field *attr*.

3. Create a summary view of the training data.

4. Aggregate the data with `nb_classify`, `nb_probabilities`,or both.

Naive Bayes Example 1 – Small Table

This example begins with the normalized data in the example table `class_example` and proceeds through four discrete steps:

Table `class_example`:

```
 id | class | a1 | a2 | a3
----+-------+----+----+----
  1 | C1    |  1 |  2 |  3
  2 | C1    |  1 |  4 |  3
  3 | C2    |  0 |  2 |  2
  4 | C1    |  1 |  2 |  1
  5 | C2    |  1 |  2 |  2
  6 | C2    |  0 |  1 |  3
```

1. Unpivot the data

For use as training data, the data in `class_example` must be unpivoted because the data is in denormalized form. The terms in single quotation marks define the values to use for the new field *attr*. By convention, these values are the same as the field names in the normalized table. In this example, these values are capitalized to highlight where they are created in the command.

```
CREATE view class_example_unpivot AS
SELECT id, class, unnest(array['A1', 'A2', 'A3']) as attr,
unnest(array[a1,a2,a3]) as value FROM class_example;
```

The unpivoted view shows the normalized data. It is not necessary to use this view. Use the command `SELECT * from class_example_unpivot` to see the denormalized data:

```
 id | class | attr | value
----+-------+------+-------
  2 | C1    | A1   |     1
  2 | C1    | A2   |     2
  2 | C1    | A3   |     1
  4 | C2    | A1   |     1
  4 | C2    | A2   |     2
  4 | C2    | A3   |     2
  6 | C2    | A1   |     0
  6 | C2    | A2   |     1
  6 | C2    | A3   |     3
  1 | C1    | A1   |     1
  1 | C1    | A2   |     2
  1 | C1    | A3   |     3
  3 | C1    | A1   |     1
  3 | C1    | A2   |     4
  3 | C1    | A3   |     3
  5 | C2    | A1   |     0
  5 | C2    | A2   |     2
  5 | C2    | A3   |     2
(18 rows)
```

2. Create a training table from the unpivoted data.

   The terms in single quotation marks define the values to sum. The terms in the array passed into `pivot_sum` must match the number and names of classifications in the original data. In the example, C1 and C2:

   ```
   CREATE table class_example_nb_training AS
   SELECT attr, value, pivot_sum(array['C1', 'C2'], class, 1)
   as class_count
   FROM   class_example_unpivot
   GROUP BY attr, value
   DISTRIBUTED by (attr);
   ```

   The following is the resulting training table:

   ```
    attr | value | class_count
   ```

```
------+-------+-------------
 A3   |     1 | {1,0}
 A3   |     3 | {2,1}
 A1   |     1 | {3,1}
 A1   |     0 | {0,2}
 A3   |     2 | {0,2}
 A2   |     2 | {2,2}
 A2   |     4 | {1,0}
 A2   |     1 | {0,1}
(8 rows)
```

3. Create a summary view of the training data.

```
CREATE VIEW class_example_nb_classify_functions AS
SELECT attr, value, class_count, array['C1', 'C2'] as classes,
sum(class_count) over (wa)::integer[] as class_total,
count(distinct value) over (wa) as attr_count
FROM class_example_nb_training
WINDOW wa as (partition by attr);
```

The following is the resulting training table:

```
attr| value | class_count| classes | class_total |attr_count
-----+-------+------------+---------+-------------+---------
 A2  |     2 | {2,2}      | {C1,C2} | {3,3}       |         3
 A2  |     4 | {1,0}      | {C1,C2} | {3,3}       |         3
 A2  |     1 | {0,1}      | {C1,C2} | {3,3}       |         3
 A1  |     0 | {0,2}      | {C1,C2} | {3,3}       |         2
 A1  |     1 | {3,1}      | {C1,C2} | {3,3}       |         2
 A3  |     2 | {0,2}      | {C1,C2} | {3,3}       |         3
 A3  |     3 | {2,1}      | {C1,C2} | {3,3}       |         3
 A3  |     1 | {1,0}      | {C1,C2} | {3,3}       |         3
(8 rows)
```

4. Classify rows with `nb_classify` and display the probability with `nb_probabilities`.

After you prepare the view, the training data is ready for use as a baseline for determining the class of incoming rows. The following query predicts whether rows are of class `C1` or `C2` by using the `nb_classify` aggregate:

```
SELECT nb_classify(classes, attr_count, class_count,
class_total) as class
FROM class_example_nb_classify_functions
where (attr = 'A1' and value = 0) or (attr = 'A2' and value =
2) or (attr = 'A3' and value = 1);
```

Running the example query against this simple table yields one row of data displaying these values:

This query yields the expected single-row result of `C1`.

```
class
```

```
-------
C2
(1 row)
```

Display the probabilities for each class with `nb_probabilities`.

Once the view is prepared, the system can use the training data as a baseline for determining the class of incoming rows. The following query predicts whether rows are of class `C1` or `C2` by using the `nb_probabilities` aggregate:

```
SELECT nb_probabilities(classes, attr_count, class_count,
class_total) as probability
FROM class_example_nb_classify_functions
where (attr = 'A1' and value = 0) or (attr = 'A2' and value =
2) or (attr = 'A3' and value = 1);
```

Running the example query against this simple table yields one row of data displaying the probabilities for each class:

This query yields the expected single-row result showing two probabilities, the first for `C1`, and the second for `C2`.

```
probability
-------------
 {0.4,0.6}
(1 row)
```

You can display the classification and the probabilities with the following query.

```
SELECT nb_classify(classes, attr_count, class_count,
class_total) as class, nb_probabilities(classes, attr_count,
class_count, class_total) as probability FROM
class_example_nb_classify where (attr = 'A1' and value = 0)
or (attr = 'A2' and value = 2) or (attr = 'A3' and value =
1);
```

This query produces the following result:

```
   class | probability
   -------+-------------
    C2    | {0.4,0.6}
(1 row)
```

Actual data in production scenarios is more extensive than this example data and yields better results. Accuracy of classification with `nb_classify and nb_probabilities` improves significantly with larger sets of training data.

Naive Bayes Example 2 – Weather and Outdoor Sports

This example calculates the probabilities of whether the user will play an outdoor sport, such as golf or tennis, based on weather conditions. The table `weather_example` contains the example values. The identification field for the table is *day*. There are two classifications held in the field *play*: `Yes` or `No`. There are four weather attributes, *outlook*, *temperature*, *humidity*, and *wind*. The data is normalized.

```
 day | play | outlook  | temperature | humidity |  wind
-----+------+----------+-------------+----------+--------
   2 | No   | Sunny    | Hot         | High     | Strong
```

```
  4 | Yes  | Rain     | Mild        | High     | Weak
  6 | No   | Rain     | Cool        | Normal   | Strong
  8 | No   | Sunny    | Mild        | High     | Weak
 10 | Yes  | Rain     | Mild        | Normal   | Weak
 12 | Yes  | Overcast | Mild        | High     | Strong
 14 | No   | Rain     | Mild        | High     | Strong
  1 | No   | Sunny    | Hot         | High     | Weak
  3 | Yes  | Overcast | Hot         | High     | Weak
  5 | Yes  | Rain     | Cool        | Normal   | Weak
  7 | Yes  | Overcast | Cool        | Normal   | Strong
  9 | Yes  | Sunny    | Cool        | Normal   | Weak
 11 | Yes  | Sunny    | Mild        | Normal   | Strong
 13 | Yes  | Overcast | Hot         | Normal   | Weak
(14 rows)
```

Because this data is normalized, all four Naive Bayes steps are required.

1. Unpivot the data.

```
CREATE view weather_example_unpivot AS SELECT day, play,
unnest(array['outlook','temperature', 'humidity','wind']) as
attr, unnest(array[outlook,temperature,humidity,wind]) as
value FROM weather_example;
```

Note the use of quotation marks in the command.

The `SELECT * from weather_example_unpivot` displays the denormalized data and contains the following 56 rows.

```
 day | play |    attr     |  value
-----+------+-------------+----------
   2 | No   | outlook     | Sunny
   2 | No   | temperature | Hot
   2 | No   | humidity    | High
   2 | No   | wind        | Strong
   4 | Yes  | outlook     | Rain
   4 | Yes  | temperature | Mild
   4 | Yes  | humidity    | High
   4 | Yes  | wind        | Weak
   6 | No   | outlook     | Rain
   6 | No   | temperature | Cool
   6 | No   | humidity    | Normal
   6 | No   | wind        | Strong
   8 | No   | outlook     | Sunny
   8 | No   | temperature | Mild
   8 | No   | humidity    | High
   8 | No   | wind        | Weak
  10 | Yes  | outlook     | Rain
  10 | Yes  | temperature | Mild
  10 | Yes  | humidity    | Normal
```

```
 10 | Yes  | wind        | Weak
 12 | Yes  | outlook     | Overcast
 12 | Yes  | temperature | Mild
 12 | Yes  | humidity    | High
 12 | Yes  | wind        | Strong
 14 | No   | outlook     | Rain
 14 | No   | temperature | Mild
 14 | No   | humidity    | High
 14 | No   | wind        | Strong
  1 | No   | outlook     | Sunny
  1 | No   | temperature | Hot
  1 | No   | humidity    | High
  1 | No   | wind        | Weak
  3 | Yes  | outlook     | Overcast
  3 | Yes  | temperature | Hot
  3 | Yes  | humidity    | High
  3 | Yes  | wind        | Weak
  5 | Yes  | outlook     | Rain
  5 | Yes  | temperature | Cool
  5 | Yes  | humidity    | Normal
  5 | Yes  | wind        | Weak
  7 | Yes  | outlook     | Overcast
  7 | Yes  | temperature | Cool
  7 | Yes  | humidity    | Normal
  7 | Yes  | wind        | Strong
  9 | Yes  | outlook     | Sunny
  9 | Yes  | temperature | Cool
  9 | Yes  | humidity    | Normal
  9 | Yes  | wind        | Weak
 11 | Yes  | outlook     | Sunny
 11 | Yes  | temperature | Mild
 11 | Yes  | humidity    | Normal
 11 | Yes  | wind        | Strong
 13 | Yes  | outlook     | Overcast
 13 | Yes  | temperature | Hot
 13 | Yes  | humidity    | Normal
 13 | Yes  | wind        | Weak
(56 rows)
```

2. Create a training table.

```
CREATE table weather_example_nb_training AS SELECT attr,
value, pivot_sum(array['Yes','No'], play, 1) as class_count
FROM weather_example_unpivot GROUP BY attr, value
DISTRIBUTED by (attr);
```

The `SELECT * from weather_example_nb_training` displays the training data and contains the following 10 rows.

```
    attr     |  value   | class_count
-------------+----------+-------------
 outlook     | Rain     | {3,2}
 humidity    | High     | {3,4}
 outlook     | Overcast | {4,0}
 humidity    | Normal   | {6,1}
 outlook     | Sunny    | {2,3}
 wind        | Strong   | {3,3}
 temperature | Hot      | {2,2}
 temperature | Cool     | {3,1}
 temperature | Mild     | {4,2}
 wind        | Weak     | {6,2}
(10 rows)
```

3. Create a summary view of the training data.

```
CREATE VIEW weather_example_nb_classify_functions AS SELECT
attr, value, class_count, array['Yes','No'] as
classes,sum(class_count) over (wa)::integer[] as
class_total,count(distinct value) over (wa) as attr_count
FROM weather_example_nb_training WINDOW wa as (partition by
attr);
```

The `SELECT * from weather_example_nb_classify_function` displays the training data and contains the following 10 rows.

```
attr        |  value   | class_count| classes | class_total| attr_count
------------+--------- +------------+---------+------------+-----------
temperature | Mild     | {4,2}      | {Yes,No}| {9,5}      |          3
temperature | Cool     | {3,1}      | {Yes,No}| {9,5}      |          3
temperature | Hot      | {2,2}      | {Yes,No}| {9,5}      |          3
wind        | Weak     | {6,2}      | {Yes,No}| {9,5}      |          2
wind        | Strong   | {3,3}      | {Yes,No}| {9,5}      |          2
humidity    | High     | {3,4}      | {Yes,No}| {9,5}      |          2
humidity    | Normal   | {6,1}      | {Yes,No}| {9,5}      |          2
outlook     | Sunny    | {2,3}      | {Yes,No}| {9,5}      |          3
outlook     | Overcast | {4,0}      | {Yes,No}| {9,5}      |          3
outlook     | Rain     | {3,2}      | {Yes,No}| {9,5}      |          3
(10 rows)
```

4. Aggregate the data with `nb_classify`, `nb_probabilities`,or both.

Decide what to classify. To classify only one record with the following values:

```
 temperature | wind | humidity | outlook
-------------+------+----------+---------
  Cool       | Weak | High     | Overcast
```

Use the following command to aggregate the data. The result gives the classification `Yes` or `No` and the probability of playing outdoor sports under this particular set of conditions.

```
SELECT nb_classify(classes, attr_count, class_count,
```

```
class_total) as class,
       nb_probabilities(classes, attr_count, class_count,
class_total) as probability
FROM weather_example_nb_classify_functions where
  (attr = 'temperature' and value = 'Cool') or
  (attr = 'wind'        and value = 'Weak') or
  (attr = 'humidity'    and value = 'High') or
  (attr = 'outlook'     and value = 'Overcast');
```

The result is a single row.

```
class |              probability
-------+----------------------------------------
 Yes   | {0.858103353920726,0.141896646079274}
(1 row)
```

To classify a group of records, load them into a table. In this example, the table `t1` contains the following records:

```
 day | outlook  | temperature | humidity |  wind
-----+----------+-------------+----------+--------
  15 | Sunny    | Mild        | High     | Strong
  16 | Rain     | Cool        | Normal   | Strong
  17 | Overcast | Hot         | Normal   | Weak
  18 | Rain     | Hot         | High     | Weak
(4 rows)
```

The following command aggregates the data against this table. The result gives the classification `Yes` or `No` and the probability of playing outdoor sports for each set of conditions in the table `t1`. Both the `nb_classify` and `nb_probabilities` aggregates are used.

```
SELECT t1.day,
       t1.temperature, t1.wind, t1.humidity, t1.outlook,
       nb_classify(classes, attr_count, class_count,
class_total) as class,
       nb_probabilities(classes, attr_count, class_count,
class_total) as probability
FROM t1, weather_example_nb_classify_functions
WHERE
  (attr = 'temperature' and value = t1.temperature) or
  (attr = 'wind'        and value = t1.wind) or
  (attr = 'humidity'    and value = t1.humidity) or
  (attr = 'outlook'     and value = t1.outlook)
GROUP BY t1.day, t1.temperature, t1.wind, t1.humidity,
t1.outlook;
```

The result is a four rows, one for each record in `t1`.

```
day| temp| wind   | humidity | outlook  | class |  probability
---+-----+--------+----------+----------+-------+-------------
15 | Mild| Strong | High     | Sunny    | No    | {0.244694132334582,0.755305867665418}
```

```
16 | Cool| Strong | Normal   | Rain     | Yes   | {0.751471997809119,0.248528002190881}
18 | Hot | Weak   | High     | Rain     | No    | {0.446387538890131,0.553612461109869}
17 | Hot | Weak   | Normal   | Overcast | Yes   | {0.9297192642788,0.0702807357212004}
(4 rows)
```

Query Performance

Greenplum Database dynamically eliminates irrelevant partitions in a table and optimally allocates memory for different operators in a query. These enhancements scan less data for a query, accelerate query processing, and support more concurrency.

- Dynamic Partition Elimination

  In the Greenplum Database, values available only when a query runs are used to dynamically prune partitions, which improves query processing speed. Enable or disable dynamic partition elimination by setting the server configuration parameter `gp_dynamic_partition_pruning` to `ON` or `OFF`; it is `ON` by default.

- Memory Optimizations

  Greenplum Database allocates memory optimally for different operators in a query and frees and re-allocates memory during the stages of processing a query.

Greenplum Database server configuration parameters can affect query performance. See "Query Tuning Parameters" in Chapter 3 "Tuning Your Greenplum System" of the *Greenplum Database System Administrator Guide* for a list of query tuning parameters.

Query Profiling

Greenplum Database devises a *query plan* for each query. Choosing the right query plan to match the query and data structure is necessary for good performance. A query plan defines how Greenplum Database will run the query in the parallel execution environment. Examine the query plans of poorly performing queries to identify possible performance tuning opportunities.

The query planner uses data statistics maintained by the database to choose a query plan with the lowest possible cost. Cost is measured in disk I/O, shown as units of disk page fetches. The goal is to minimize the total execution cost for the plan.

View the plan for a given query with the `EXPLAIN` command. `EXPLAIN` shows the query planner's estimated cost for the query plan. For example:

```
EXPLAIN SELECT * FROM names WHERE id=22;
```

`EXPLAIN ANALYZE` runs the statement in addition to displaying its plan. This is useful for determining how close the planner's estimates are to reality. For example:

```
EXPLAIN ANALYZE SELECT * FROM names WHERE id=22;
```

Reading EXPLAIN Output

A query plan is a tree of nodes. Each node in the plan represents a single operation, such as a table scan, join, aggregation, or sort.

Read plans from the bottom to the top: each node feeds rows into the node directly above it. The bottom nodes of a plan are usually table scan operations: sequential, index, or bitmap index scans. If the query requires joins, aggregations, sorts, or other operations on the rows, there are additional nodes above the scan nodes to perform these operations. The topmost plan nodes are usually Greenplum Database motion nodes: redistribute, explicit redistribute, broadcast, or gather motions. These operations move rows between segment instances during query processing.

The output of `EXPLAIN` has one line for each node in the plan tree and shows the basic node type and the following execution cost estimates for that plan node:

- **cost** —Measured in units of disk page fetches. 1.0 equals one sequential disk page read. The first estimate is the start-up cost of getting the first row and the second is the total cost of cost of getting all rows. The total cost assumes all rows will be retrieved, which is not always true; for example, if the query uses `LIMIT`, not all rows are retrieved.
- **rows** —The total number of rows output by this plan node. This number is usually less than the number of rows processed or scanned by the plan node, reflecting the estimated selectivity of any `WHERE` clause conditions. Ideally, the estimate for the topmost node approximates the number of rows that the query actually returns, updates, or deletes.
- **width** —The total bytes of all the rows that this plan node outputs.

Note the following:

- The cost of a node includes the cost of its child nodes. The topmost plan node has the estimated total execution cost for the plan. This is the number the planner intends to minimize.
- The cost reflects only the aspects of plan execution that the query planner takes into consideration. For example, the cost does not reflect time spent transmitting result rows to the client.

EXPLAIN Example

The following example describes how to read an `EXPLAIN` query plan for a 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
```

Read the plan from the bottom to the top. To start, the query planner sequentially scans the *names* table. Notice the `WHERE` clause is applied as a *filter* condition. This means the scan operation checks the condition for each row it scans and outputs only the rows that satisfy the condition.

The results of the scan operation are passed to a *gather motion* operation. In Greenplum Database, a gather motion is when segments send rows to the master. In this example, we have two segment instances that send to one master instance. This operation is working on `slice1` of the parallel query execution plan. A query plan is divided into *slices* so the segments can work on portions of the query plan in parallel.

The estimated startup cost for this plan is `00.00` (no cost) and a total cost of `20.88` disk page fetches. The planner estimates this query will return one row.

Reading EXPLAIN ANALYZE Output

`EXPLAIN ANALYZE` plans and runs the statement. The `EXPLAIN ANALYZE` plan shows the actual execution cost along with the planner's estimates. This allows you to see if the planner's estimates are close to reality. `EXPLAIN ANALYZE` also shows the following:

- The total runtime (in milliseconds) in which the query executed.
- The memory used by each slice of the query plan, as well as the memory reserved for the whole query statement.
- 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 the operation. If multiple segments produce an equal number of rows, `EXPLAIN ANALYZE` shows the segment with the longest <*time*> *to end*.
- The segment id of the segment that produced the most rows for an operation.
- For relevant operations, the amount of memory (`work_mem`) used by the operation. If the `work_mem` was insufficient to perform the operation in memory, the plan shows the amount of data spilled to disk for the lowest-performing segment. For example:

  ```
  Work_mem used: 64K bytes avg, 64K bytes max (seg0).
  Work_mem wanted: 90K bytes avg, 90K byes max (seg0) to lessen
  workfile I/O affecting 2 workers.
  ```

- The time (in milliseconds) in which the segment that produced the most rows retrieved the first row, and the time taken for that segment to retrieve all rows. The result may omit <*time*> *to first row* if it is the same as the <*time*> *to end*.

EXPLAIN ANALYZE Example

This example describes how to read an `EXPLAIN ANALYZE` query plan using the same query. The **`bold`** parts of the plan show actual timing and rows returned for each plan node, as well as memory and time statistics for the whole query.

```
   EXPLAIN ANALYZE SELECT * FROM names WHERE name = 'Joelle';
                          QUERY PLAN
   ------------------------------------------------------------
Gather Motion 2:1  (slice1; segments: 2)  (cost=0.00..20.88 rows=1
width=13)
   Rows out:  1 rows at destination with 0.305 ms to first row,
0.537 ms to end, start offset by 0.289 ms.
   ->  Seq Scan on names  (cost=0.00..20.88 rows=1 width=13)
         Rows out:  Avg 1 rows x 2 workers.  Max 1 rows (seg0) with
0.255 ms to first row, 0.486 ms to end, start offset by 0.968 ms.
               Filter: name = 'Joelle'::text
 Slice statistics:
```

```
   (slice0)    Executor memory: 135K bytes.
(slice1)    Executor memory: 151K bytes avg x 2 workers, 151K bytes
max (seg0).
 Statement statistics:
   Memory used: 128000K bytes
 Total runtime: 22.548 ms
```

Read the plan from the bottom to the top. The total elapsed time to run this query was *22.548* milliseconds.

The *sequential scan* operation had only one segment (*seg0*) that returned rows, and it returned just *1 row*. It took *0.255* milliseconds to find the first row and *0.486* to scan all rows. This result is close to the planner's estimate: the query planner estimated it would return one row for this query. The *gather motion* (segments sending data to the master) received 1 row . The total elapsed time for this operation was *0.537* milliseconds.

Examining Query Plans to Solve Problems

If a query performs poorly, examine its query plan and ask the following questions:

- **Do operations in the plan take an exceptionally long time?** Look for an operation consumes the majority of query processing time. For example, if an index scan takes longer than expected, the index could be out-of-date and need to be reindexed. Or, adjust `enable_<operator>` parameters to see if you can force the planner to choose a different plan by disabling a particular query plan operator for that query.
- **Are the planner's estimates close to reality?** Run `EXPLAIN ANALYZE` and see if the number of rows the planner estimates is close to the number of rows the query operation actually returns. If there is a large discrepancy, collect more statistics on the relevant columns. See the *Greenplum Database Reference Guide* for more information on the `EXPLAIN ANALYZE` and `ANALYZE` commands..
- **Are selective predicates applied early in the plan?** Apply the most selective filters early in the plan so fewer rows move up the plan tree. If the query plan does not correctly estimate query predicate selectivity, collect more statistics on the relevant columns. See the `ANALYZE` command in the *Greenplum Database Reference Guide* for more information collecting statistics. You can also try reordering the `WHERE` clause of your SQL statement.
- **Does the planner choose the best join order?** When you have a query that joins multiple tables, make sure that the planner chooses the most selective join order. Joins that eliminate the largest number of rows should be done earlier in the plan so fewer rows move up the plan tree.

  If the plan is not choosing the optimal join order, set `join_collapse_limit=1` and use explicit `JOIN` syntax in your SQL statement to force the planner to the specified join order. You can also collect more statistics on the relevant join columns. See the `ANALYZE` command in the *Greenplum Database Reference Guide* for more information collecting statistics.

- **Does the planner selectively scan partitioned tables?** If you use table partitioning, is the planner selectively scanning only the child tables required to satisfy the query predicates? Scans of the parent tables should return 0 rows since the parent tables do not contain any data. See "Verifying Your Partition Strategy" in the *Greenplum Database Database Administrator Guide* for an example of a query plan that shows a selective partition scan.
- **Does the planner choose hash aggregate and hash join operations where applicable?** Hash operations are typically much faster than other types of joins or aggregations. Row comparison and sorting is done in memory rather than reading/writing from disk. To enable the query planner to choose hash operations, there must be sufficient memory available to hold the estimated number of rows. Try increasing work memory to improve performance for a query. If possible, run an `EXPLAIN ANALYZE` for the query to show which plan operations spilled to disk, how much work memory they used, and how much memory was required to avoid spilling to disk. For example:

  ```
  Work_mem used:  23430K bytes avg, 23430K bytes max (seg0).
  Work_mem wanted: 33649K bytes avg, 33649K bytes max (seg0) to lessen
  workfile I/O affecting 2 workers.
  ```

  The "bytes wanted" message from `EXPLAIN ANALYZE` is based on the amount of data written to work files and is not exact. The minimum `work_mem` needed can differ from the suggested value.

6. SQL User-Defined Functions

With Greenplum Database you can create user-defined functions using SQL statements, functions, and operators.

When using SQL functions, function attributes affect how Greenplum Database query optimizer creates query plans. See "Using Functions in Greenplum Database" for information about function attributes.

See Chapter 5, "Querying Data" for information about using SQL in Greenplum Database.

Greenplum Database is based on PostgreSQL 8.2 with some features added from later PostgreSQL releases. For information about SQL conformance, see Chapter 11, "Summary of Greenplum Features" in the *Greenplum Database Reference Guide*.

Example User-Defined Functions

The following are examples of SQL user-defined functions.

Example: Simple SQL Functions

This is an SQL function that returns an integer value. The `CREATE FUNCTION` command uses the Postgres dollar quoting syntax ($$):

```
CREATE FUNCTION one() RETURNS integer AS $$
  SELECT 1 AS result;
$$ LANGUAGE SQL;
```

The following is an alternate way of creating the function with the quote syntax:

```
CREATE FUNCTION one() RETURNS integer AS '
  SELECT 1 AS result;
' LANGUAGE SQL;
```

When you run the function from the command line:

```
SELECT one();
```

The result is similar to this:

```
 one
-----
    1
```

The following function does not return any values. The following function deletes the rows from the EMP table where the value in the salary column is less than 0.

```
CREATE FUNCTION clean_emp() RETURNS void AS '
  DELETE FROM emp
    WHERE salary < 0;
```

```
' LANGUAGE SQL;
```

When you run the function:

```
SELECT clean_emp();
```

This function takes two values as input, an account number and a balance, and updated the table BANK. The function returns a number, the updated balance for the account:

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS numeric AS $$
  UPDATE bank
    SET balance = balance - $2
    WHERE accountno = $1;
  SELECT balance FROM bank WHERE accountno = $1;
$$ LANGUAGE SQL;
```

Example: SQL Functions with Composite Types

When writing functions with arguments of composite types, the function must specify which argument and the desired attribute, field, of that argument.

The following example uses this table.

```
CREATE TABLE emp (
   name       text,
   salary     numeric,
   age        integer,
   cubicle    point
);
```

And a row in the table has the following values.

```
INSERT INTO emp values ('Bill', 4200, 40, '(2,1)');
```

This function specifies the salary field from the input row which is a composite type:

```
CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
  SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;
```

When you run the function with this `SELECT` statement

```
SELECT name, double_salary(emp.*) AS dream
  FROM emp
  WHERE emp.cubicle ~= point '(2,1)';
```

The output similar to the following.

```
 name | dream
------+-------
 Bill |  8400
```

A function can also return a composite type. This is an example of a function that returns a composite type, a single emp row based on the emp table:

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
  SELECT text 'None' AS name,
    1000.0 AS salary,
    25 AS age,
    point '(2,2)' AS cubicle;
$$ LANGUAGE SQL;
```

Note two important things about defining a function that returns a composite type such as the `new_emp()` function:

- The select list order in the query must be the same as that in which the columns appear in the table associated with the composite type. In the function, the column names the columns do not need to match.
- The expressions must be typecast to match the definition of the composite type.

This is an alternative function definition to return a single emp row:

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
  SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

When you use a function that returns a composite type, you might want only one field (attribute) from its result. You can do that with syntax like this:

```
SELECT (new_emp()).name;
```

The result is similar to this:

```
name
------
None
```

Example: SQL Functions with Composite Types

An alternative way of describing a function result is to define it with *output parameters*. Output parameters provide a convenient way of defining functions that return several columns. This example specifies two output parameters:

```
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int,
   OUT product int)
  AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

Example: SQL Functions as Table Sources

All SQL functions can be used in the `FROM` clause of a query, but it is particularly useful for functions returning composite types. If the function is defined to return a base type, the table function produces a one-column table. If the function is defined to return a composite type, the table function produces a column for each attribute of the composite type.

Here is an example:

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

```
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');
```

This function returns a a single row.

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

The following `SELECT` statement runs the function:

```
SELECT *, upper(fooname) FROM getfoo(1) AS t1;
```

The output is similar to:

```
 fooid | foosubid | fooname | upper
-------+----------+---------+-------
     1 |        1 | Joe     | JOE

(1 row)
```

As the example shows, we can work with the columns of the function result as if they were columns of a regular table.

This example function returned only one row. To return multiple rows, use the keyword `SETOF`. That is described in the next example.

Example: SQL Functions Returning Sets

When an SQL function is declared as returning `SETOF` some type, the function's final `SELECT` query is executed to completion, and each row it outputs is returned as an element of the result set.

This feature is normally used when calling the function in the `FROM` clause. In this case each row returned by the function becomes a row of the table seen by the query. For example, assume that table foo has the same contents as above, and we say:

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

When you run this query that uses the function:

```
SELECT * FROM getfoo(1) AS t1;
```

The output is similar to this:

```
fooid | foosubid | fooname
------+----------+---------
    1 |        1 | Joe
    1 |        2 | Ed
(2 rows)
```

Example: SQL Polymorphic SQL Functions

SQL functions can be declared to accept and return the polymorphic types `anyelement` and `anyarray`. Here is a polymorphic function `make_array` that builds up an array from two arbitrary data type elements:

```
CREATE FUNCTION make_array(anyelement, anyelement) RETURNS
anyarray AS $$
  SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;
```

The following SELECT statement runs the function:

```
SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b')
AS textarray;
```

The output is similar to the following:

```
 intarray | textarray
----------+-----------
 {1,2}    | {a,b}

(1 row)
```

References

Postgres Query Language (SQL) Functions

http://www.postgresql.org/docs/8.2/static/xfunc-sql.html

See the *Greenplum Database Reference Guide*

CREATE FUNCTION

7. PL/pgSQL Procedural Language

Greenplum Database PL/pgSQL is a loadable procedural language that is installed and registered by default with Greenplum Database. You can create user-defined functions using SQL statements, functions, and operators.

With PL/pgSQL you can group a block of computation and a series of SQL queries inside the database server, thus having the power of a procedural language and the ease of use of SQL. Also, with PL/pgSQL you can use all the data types, operators and functions of Greenplum Database SQL.

Greenplum PL/pqSQL

The PostgreSQL/Greenplum language PL/pgSQL is a subset of Oracle PL/SQL. The Postgres PL/pgSQL documentation is at
http://www.postgresql.org/docs/8.2/static/plpgsql.html

When using PL/pqSQL functions, function attributes affect how Greenplum Database query optimizer creates query plans. See "Using Functions in Greenplum Database" for information about function attributes.

The following are some Greenplum Database SQL limitations

- Triggers are not supported
- Cursors are forward moving only (not scrollable)

For information about SQL conformance, see Chapter 11, "Summary of Greenplum Features" in the *Greenplum Database Reference Guide*.

The PL/pgSQL Language

PL/pgSQL is a block-structured language. The complete text of a function definition must be a block. A block is defined as:

```
[ <<label>> ]
[ DECLARE
   declarations ]

BEGIN
   statements
END [ label ];
```

Each declaration and each statement within a block is terminated by a semicolon (;). A block that appears within another block must have a semicolon after `END`, as shown in the previous block. The `END` that concludes a function body does not require a semicolon.

Important: It is important not to confuse the use of the `BEGIN` and `END` keywords for grouping statements in PL/pgSQL with the database commands for transaction control. The PL/pgSQL `BEGIN` and `END` keywords are only for grouping; they do not start or end a transaction. Functions are always executed within a transaction established by an outer query — they cannot start or commit that transaction, since there would be no context for them to execute in. However, a block containing an

`EXCEPTION` clause effectively forms a subtransaction that can be rolled back without affecting the outer transaction. For more about that see the post the Postgres documentation on error trapping at http://www.postgresql.org/docs/8.2/static/plpgsql-control-structures.html#PLPGSQL-ERROR-TRAPPING.

All key words and identifiers can be written in mixed upper and lower case. Identifiers are implicitly converted to lowercase unless enclosed in double-quotes (").

You can add comments in PL/pgSQL in the following ways:

- A double dash (--) starts a comment that extends to the end of the line.
- A /* starts a block comment that extends to the next occurrence of */.

  Block comments cannot be nested, but double dash comments can be enclosed into a block comment and a double dash can hide the block comment delimiters /* and */.

Any statement in the statement section of a block can be a subblock. Subblocks can be used for logical grouping or to localize variables to a small group of statements.

The variables declared in the declarations section preceding a block are initialized to their default values every time the block is entered, not only once per function call. For example declares the variable `quantity` several times:

```
CREATE FUNCTION somefunc() RETURNS integer AS $$

DECLARE
   quantity integer := 30;

BEGIN
   RAISE NOTICE 'Quantity here is %', quantity;
   -- Quantity here is 30
   quantity := 50;
   --
   -- Create a subblock
   --
   DECLARE
      quantity integer := 80;
   BEGIN
      RAISE NOTICE 'Quantity here is %', quantity;
      -- Quantity here is 80
   END;

   RAISE NOTICE 'Quantity here is %', quantity;
   -- Quantity here is 50

   RETURN quantity;
END;

$$ LANGUAGE plpgsql;
```

PL/pgSQL Examples

The following are examples of PL/pgSQL user-defined functions.

Example: Aliases for Function Parameters

Parameters passed to functions are named with the identifiers such asa `$1`, `$2`. Optionally, aliases can be declared for `$n` parameter names for increased readability. Either the alias or the numeric identifier can then be used to refer to the parameter value.

There are two ways to create an alias. The preferred way is to give a name to the parameter in the `CREATE FUNCTION` command, for example:

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
   RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

You can also explicitly declare an alias, using the declaration syntax:

```
name ALIAS FOR $n;
```

This example, creates the same function with the `DECLARE` syntax.

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Example: Using the Data Type of a Table Column

When delcaring a variable, you can use `%TYPE` to specify the data type of a variable or table column. This is the syntax for decaring a variable with the data type of a table column:

```
name table.column_name%TYPE;
```

You can use this to declare variables that will hold database values. For example, if you have a column named `user_id` in your `users` table. To declare the variable `my_userid` with the same data type as the `users.user_id` column:

```
my_userid users.user_id%TYPE;
```

`%TYPE` is particularly valuable in polymorphic functions, since the data types needed for internal variables may change from one call to the next. Appropriate variables can be created by applying `%TYPE` to the function's arguments or result placeholders.

Example: Composite Type Based on a Table Row

The following syntax declares a composite variable based on table row:

```
name table_name%ROWTYPE;
```

Such a *row variable* can hold a whole row of a `SELECT` or `FOR` query result, so long as that query column set matches the declared type of the variable. The individual fields of the row value are accessed using the usual dot notation, for example `rowvar.column`.

Parameters to a function can be composite types (complete table rows). In that case, the corresponding identifier `$n` will be a row variable, and fields can be selected from it, for `example $1.user_id`.

Only the user-defined columns of a table row are accessible in a row-type variable, not the OID or other system columns. The fields of the row type inherit the table's field size or precision for data types such as `char(n)`.

The next example function uses a row variable composite type. Before creating the function, create the table that is used by the function with this command.

```
CREATE TABLE table1 (
  f1 text,
  f2 numeric,
  f3 integer
) distributed by (f1);
```

This `INSERT` command adds data to the table.

```
INSERT INTO table1 values
  ('test1', 14.1,  3),
  ('test2', 52.5,  2),
  ('test3', 32.22, 6),
  ('test4', 12.1,  4) ;
```

This function uses a `ROWTYPE` composite variable based on `table1`.

```
CREATE OR REPLACE FUNCTION t1_calc( name text) RETURNS    text
AS $$
DECLARE
    t1_row table1%ROWTYPE;
BEGIN
    SELECT * INTO t1_row FROM table1 WHERE table1.f1 = $1 ;
    RETURN t1_row.f2 * t1_row.f3 ;
END;
$$ LANGUAGE plpgsql VOLATILE;
```

The following `SELECT` command uses the function.

```
select t1_calc( 'test1' );
```

References

The Postgres documentation on PL/pgSQL is at
http://www.postgresql.org/docs/8.2/static/plpgsql.html

Also, see the `CREATE FUNCTION` command in the *Greenplum Database Reference Guide.*

For informatin about porting Oracle functions, see:

- http://www.postgresql.org/docs/8.2/static/plpgsql-porting.html
- Chapter 16, “Oracle Compatibility Functions”

8. C Language User-Defined Functions

User-defined functions can be written in C. You write the functions in C and compile the functions into dynamically loadable objects (also called shared libraries). When you run a function, Greenplum Database loads the functions on demand.

This chapter includes the following information:

This chapter describes how to create user-defined functions in the C lanaguage. Creating user-defined function that can be made compatible with C, such as C++ and creating Greenplum Database extensions is beyond the scope of this documentation.

Prerequisites for C Language User-Defined Functions

The following are requirements for creating C Language user-defined functions.

- Greenplum Database installed
- GCC compiler (an ISO/ANSI C compiler.)
- make utility for building C code projects.

[**MSK**] Other requirements?

Registering C Language User-Defined Functions

Use the `CREATE FUNCTION` command to register user-defined functions that are used as described in "Using Functions in Greenplum Database" of Chapter 5, "Querying Data".

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

Registering the user-defined functions in the Greenplum Database template1 database makes the functions available in any new Greenplum databases that you create.

Note: Greenplum Database does not compile a C function automatically. The object file must be compiled before it is referenced in a `CREATE FUNCTION` command. For an example of creating and using C functions, see "Examples."

For general information about user-defined functions, see "Creating and Using Server Functions " in Chapter 4, "Database Application Development".

Running an SQL command in a C UDF requires Server Programming Interface (SPI) functions.
[**MSK**] Are all Postgres SPI functions supported by Greenplum Database?
http://www.postgresql.org/docs/8.2/static/spi.html

Note: In Greenplum Database, C is an untrusted language.

Installing C Language Shared Library Files

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

Copy your shared library file (`.so` file) to `$GPHOME/lib/postgresql/` on to all Greenplum Database hosts. This example uses the Greenplum Database `gpscp` utility to copy the file `funcs.so`:

```
$ gpscp -f gphosts_file funcs.so
    =:/usr/local/greenplum-db/lib/postgresql
```

The file `gphosts_file` contains a list of the Greenplum Database hosts. See the *Greenplum Database Utility Guide* for information about the `gpscp` utility.

Dynamic Loading

The first time a user-defined function in a particular loadable shared library file is called in a session, the dynamic loader loads that object file into memory so that the function can be called. The `CREATE FUNCTION` for a user-defined C function must therefore specify two pieces of information for the function: the name of the loadable object file, and the C name (link symbol) of the specific function to call within that object file. If the C name is not explicitly specified then it is assumed to be the same as the SQL function name.

The following algorithm is used to locate the shared object file based on the name given in the `CREATE FUNCTION` command:

1. If the name is an absolute path, the given file is loaded.
2. If the name starts with the string `$libdir`, that part is replaced by the PostgreSQL package library directory name, which is determined at build time.
3. If the name does not contain a directory part, the file is searched for in the path specified by the configuration variable `dynamic_library_path`.
4. Otherwise (the file was not found in the path, or it contains a non-absolute directory part), the dynamic loader will try to take the name as given, which will most likely fail. (It is unreliable to depend on the current working directory.)

If this sequence does not work, the platform-specific shared library file name extension (often `.so`) is appended to the given name and this sequence is tried again. If that fails, the load fails.

It is recommended to locate shared libraries either relative to `$libdir` or through the dynamic library path. This simplifies version upgrades if the new installation is at a different location. The actual directory that `$libdir` stands for can be found out with the command `pg_config --pkglibdir`.

The user ID the Greenplum Database server runs as must be able to traverse the path to the file you intend to load. Making the file or a higher-level directory not readable or not executable by the postgres user is a common mistake.

In any case, the file name that is given in the `CREATE FUNCTION` command is recorded literally in the system catalogs, so if the file needs to be loaded again the same procedure is applied.

To ensure that a dynamically loaded object file is not loaded into an incompatible server, Greenplum Database checks that the file contains a "magic block" with the appropriate contents. This allows the server to detect obvious incompatibilities, such as code compiled for a different major version of PostgreSQL. A magic block is required as of PostgreSQL 8.2. To include a magic block, write this in one and only one of the module source files, after having included the header `fmgr.h`:

```
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
```

After it is used for the first time, a dynamically loaded object file is retained in memory. Future calls in the same session to the functions in that file incur only the small overhead of a symbol table lookup. If you need to force a reload of an object file, for example after recompiling it, begin a fresh session.

Optionally, a dynamically loaded file can contain initialization and finalization functions. If the file includes a function named `_PG_init`, that function is called immediately after loading the file. The function receives no parameters and should return void. If the file includes a function named `_PG_fini`, that function is called immediately before unloading the file. Likewise, the function receives no parameters and should return void. The function `_PG_fini` is only called during an unload of the file, not during process termination.

Examples

This section contains two examples of C language source code that can be used as Greenplum Database user-defined functions:

- Simple User-Defined Functions
- User-Defined Function to Run SQL Commands

To use your Greenplum Database functions written in C, they must be compiled and linked as a shared library to produce a file that can be dynamically loaded by the Greenplum Database. Shared library files intended as executales are usually not compiled to be dynamically loaded.

For more information about compiling and link C language source code, see the documentation of your operating system, in particular the manual pages for the C compiler, `cc`, and the link editor, `ld`.

Creating shared libraries is generally analogous to linking executables:

1. The source files are compiled into object files
2. The object files are linked together

The object files need to be created as position-independent code (PIC), which conceptually means that they can be placed at an arbitrary location in memory when they are loaded by the executable. The command to link a shared library contains special flags to distinguish it from linking an executable.

This example to compile and link source code uses the source code in a file `funcs.c` and creates a shared library `funcs.so`. The intermediate object file is `funcs.o`. These examples only use a single object file. However, a shared library can contain more than one.

To compile and link C functions for Greenplum Database requires header files that are installed with Greenplum Database. The header files are in `$GPHOME/include`. When creating the shared object, Add the location to your `C_INCLUDE_PATH` environment variable so that your build environment can locate the header files. For example:

```
export C_INCLUDE_PATH=$C_INCLUDE_PATH:
 /usr/local/greenplum_db/include/postgresql/server
```

For Linux, the compiler flag to create PIC is `-fpic`. On some platforms in some situations `-fPIC` must be used if `-fpic` does not work. Refer to the GCC manual for more information. The compiler flag to create a shared library is `-shared`. The following two command commands compile and link the sample code and create the shared library `funcs.so`:

```
cc -fpic -c funcs.c
cc -shared -o funcs.so funcs.o
```

Note: To make compiling and linking easier, you can use a `make` utility. For information about creating and using a make file, see "Make File for funcs.c."

Copy the shared library `funcs.so` to the directory specified by the `$libdir` variable to all Greenplum Database hosts. You can use the Greenplum Database `pg_config` utility to determine the location of `$libdir` with this command:

```
$ pg_config --pkglibdir
```

See "Installing C Language Shared Library Files" on page 67 for an example of copying a shared library to Greenplum Database hosts.

After creating the shared library from `funcs.c` and copying the shared library to the Greenplum Database hosts, you register the user-defined functions in Greenplum Database with these `CREATE FUNCTION` commands. For example:

```
CREATE FUNCTION add_one(integer) RETURNS integer
    AS '$libdir/funcs', 'add_one'
    LANGUAGE C STRICT;
```

With the functions defined in shared library, you can create an overloaded function `add_one()`. This example, creates the function `add_one()` that accepts values of type `double precision`.

```
CREATE FUNCTION add_one(double precision) RETURNS double
```

```
    precision
    AS '$libdir/funcs', 'add_one_float8'
    LANGUAGE C STRICT;
```

These these `CREATE FUNCTION` commands create other user-defined functions that use functions in the shared library.

```
CREATE FUNCTION makepoint(point, point) RETURNS point
    AS '$libdir/funcs', 'makepoint'
    LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
    AS '$libdir/funcs', 'copytext'
    LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
    AS '$libdir/funcs', 'concat_text'
    LANGUAGE C STRICT;
```

Simple User-Defined Functions

This example C language code is a set of simple functions that can be compiled and linked as a shared library. Using the `CREATE FUNCTION` commands at the end of the previous section, you can register the functions in Greenplum Database. To use a make file to compile and link the sample code, see "Make File for funcs.c" on page 72.

funcs.c

The following code is an example set of simple user-defined functions that are written in C.

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"

/* example from http://www.postgresql.org/docs/8.4/static/xfunc-c.html */

/* by value */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/* by reference, fixed length */

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
```

```
    /* The macros for FLOAT8 hide its pass-by-reference nature. */
    float8   arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* Here, the pass-by-reference nature of Point is not hidden. */
    Point    *pointx = PG_GETARG_POINT_P(0);
    Point    *pointy = PG_GETARG_POINT_P(1);
    Point    *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* by reference, variable length */

PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text    *t = PG_GETARG_TEXT_P(0);

    /*
    * VARSIZE is the total size of the struct in bytes.
    */
    text    *new_t = (text *) palloc(VARSIZE(t));
    SET_VARSIZE(new_t, VARSIZE(t));

    /*
    * VARDATA is a pointer to the data region of the struct.
    */
    memcpy((void *) VARDATA(new_t),  /* destination */
        (void *) VARDATA(t),         /* source */
        VARSIZE(t) - VARHDRSZ);      /* how many bytes */
    PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
    text   *arg1 = PG_GETARG_TEXT_P(0);
    text   *arg2 = PG_GETARG_TEXT_P(1);
    int32   new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text   *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
```

```
        VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

Make File for funcs.c

You can use the `make` utility to easily compile and link the code. This example uses `funcs.c` source code. Add the following text to the file `makefile` in the same directory as the source code.

```
all: funcs.so

funcs.so: funcs.o
	cc -shared -o funcs.so funcs.o

funcs.o: funcs.c
	cc -fpic -c funcs.c

clean:
	rm funcs.o funcs.so
```

To build and link the souce code, run the `make` command in the same directory that contains the source code.

```
$ make
```

The following make command cleans the directory so that you can build and link the C code after you have made changes to the source file.

```
$ make clean
```

Greenplum Database supports using the Postgres PGXS `make` option for building simple extensions. In the make file, you set some variables and include the global PGXS makefile. The following example uses the PGXS make option. For example, in the same directory as the `funcs.c` file replace the text in the file `makefile` with the the following text.

```
# This make file uses postgres PGXS make in
# $GPHOME/lib/postgresql/pgxs/src/makefiles/pgxs.mk

MODULES = funcs

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)

all: funcs.so
```

You use the make commands to build and link the C code and to clean the directory.

See your system documenation for information about the `make` utility.

User-Defined Function to Run SQL Commands

You can create a shared library with runsql_test.c source code that can be used as a user-defined function that runs a SQL command. To run the SQL command, the function uses the Postgres Server Programming Interface (SPI).

- See "Make file for runsql_test.c" for the make file to compile and link the source code.

  After you compile and link the source code, copy shared library to the directory `/usr/local/greenplum-db/lib/postgresql` on all Greenplum Database hosts.
- See "CREATE FUNCTION command for runsql_test.so" for the `CREATE FUNCTION` command that register a function that uses the shared library.

runsql_test.c

The following code is an example C language user-defined function that runs an SQL command.

```
#include "postgres.h"
#include "fmgr.h"
#include "executor/spi.h"

/*
 * Converted SPI example to V1 interface.
 *
*/

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

extern Datum execq(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(execq);

Datum
execq(PG_FUNCTION_ARGS)

{
    text *sql;
    int32 cnt = PG_GETARG_INT32(1);

    char *command ;
    int32 ret;
    int32 proc;

    sql = PG_GETARG_TEXT_P(0);

    /* Convert given text object to a C string */
    command = DatumGetCString(DirectFunctionCall1(textout,
        PointerGetDatum(sql)));

    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;

    /*
    * If some rows were fetched, print them via elog(INFO).
    */
    if (ret > 0 && SPI_tuptable != NULL)

    {
       TupleDesc tupdesc = SPI_tuptable->tupdesc;
```

```
        SPITupleTable *tuptable = SPI_tuptable;
        char buf[8192];
        int i, j;

        for (j = 0; j < proc; j++)
        {
           HeapTuple tuple = tuptable->vals[j];

           for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
              snprintf(buf + strlen (buf), sizeof(buf) -
                  strlen(buf), " %s%s",
              SPI_getvalue(tuple, tupdesc, i),
                  i == tupdesc->natts) ? " " : " |");
              elog(INFO, "EXECQ: %s", buf);
        }
    }

    SPI_finish();
    pfree(command);

    PG_RETURN_INT32(proc);
}
```

Make file for runsql_test.c

for `runsql_test.c`, you can create a make file that uses the postgres PGXS make file option. Create the file `makefile` in the same directory that contains the `runsql_test.c` file and add the following text.

```
# MODULES uses postgres PGXS make file in
# $GPHOME/lib/postgresql/pgxs/src/makefiles/pgxs.mk

MODULES = runsql_test
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)

all: runsql_test.so
```

You can use the command `make` to compile and link the code and `make clean` to clean the directory so that you can build and link the C code after you have made changes to the source file.

See "Make File for funcs.c" on page 72 for an example that does not use the Postgres PGXS make file option.

CREATE FUNCTION command for runsql_test.so

After you copy the shared library `runsql_test` to all the Greenplum Database hosts, you can run the following `CREATE FUNCTION` command to create the user-defined function that uses `runsql_test.so`.

```
CREATE OR REPLACE FUNCTION exec_sql(text, integer) RETURNS
  integer
  AS '$libdir/runsql_test', 'execq'
  LANGUAGE C STRICT VOLATILE;
```

See "Installing C Language Shared Library Files" on page 67 for an example of copying a shared library to Greenplum Database hosts.

References

Technical References

For more information about C language user defined functions, see the Postgres documenation at
http://www.postgresql.org/docs/8.2/static/xfunc-c.html

Running SQL commands from a C user-defined function requires Server Programming Interface (SPI) functions. For information about the Postgres SPI functions, see the Postgres documenation at
http://www.postgresql.org/docs/8.2/static/spi.html

[**MSK**] Are all Postgres SPI functions are supported?

For information about the Postgres PGXS make option see the Postgres documenation at
http://www.postgresql.org/docs/8.2/static/xfunc-c.html#XFUNC-C-PGXS

9. PL/Java Extension

With Greenplum Database PL/Java extension, you can write Java methods using your favorite Java IDE and install the JAR files that contain the methods into Greenplum Database. This chapter contains the following information:

About PL/Java

The PL/Java 1.2.0 release of PL/Java provides the following features.

- Ability to write functions using Java 1.6 or higher.
- Standardized utilities (modeled after the SQL 2003 proposal) to install and maintain Java code in the database.
- Standardized mappings of parameters and result. Complex types as well as sets are supported.
- An embedded, high performance, JDBC driver utilizing the internal PostgreSQL SPI routines.
- Metadata support for the JDBC driver. Both DatabaseMetaData and ResultSetMetaData are included.
- The ability to return a ResultSet from a query as an alternative to building a ResultSet row by row
- Full support for savepoints and exception handling.
- The ability to use IN, INOUT, and OUT parameters.
- Two language handlers, one TRUSTED (the default) and one that is not TRUSTED (language tag is `javau` to conform with the de-facto standard)
- Transaction and Savepoint listeners enabling code execution when a transaction or savepoint is committed or rolled back.
- Integration with GNU GCJ on selected platforms.

A function in SQL will appoint a static method in a Java class. In order for the function to execute, the appointed class must be installed in the database. The PL/Java extension adds a set of functions that helps installing and maintaining the java classes. Classes are stored in normal Java archives, JAR files. A JAR file may optionally

contain a deployment descriptor that in turn contains SQL commands to be executed when the JAR is deployed or undeployed. The functions are modeled after the standards proposed for SQL 2003.

PL/Java implements a standardized way of passing parameters and return values. Complex types and sets are passed using the standard JDBC ResultSet class.

A JDBC driver is included in PL/Java. This driver is written with PostgreSQL internal SPI routines. The driver is essential since it is common for functions to reuse the database. When they do, they must use the same transactional boundaries that where used by the caller.

PL/Java is optimized for performance. The Java virtual machine executes within the same process as the backend itself. This vouches for a very low call overhead. PL/Java is designed with the objective to enable the power of Java to the database itself so that database intensive business logic can execute as close to the actual data as possible.

The standard Java Native Interface (JNI) is used when bridging calls between the backend and the Java VM.

Prerequisties for PL/Java

To use PL/Java you must have a JDK 1.6 installation on all Greenplum Database hosts.

Ensure the following Java configuration on all Greenplum Database hosts.

- Java must be accessible to the system user of the Greenplum Database administrator (gpadmin).
- Make sure the `JAVA_HOME` environment variable is set in the environment of the Greenplum Database administrator.
- The Java runtime libraries must be available to the runtime linker.

  On Linux, this involves adding a configuration file for jdk-1.6.0 in `/etc/ld.so.conf.d` and then running ldconfig. For example, you run a command similar to this as root:

  ```
  # echo "$JAVA_HOME/jre/lib/amd64/server" >
  /etc/ld.so.conf.d/libjdk-1.6.0_21.conf
  # /sbin/ldconfig
  ```

  On Solaris, this involves running the crle command and adding the path for the Java runtime libraries to the list of linked files. For example, you run a command similar to this:

  ```
  # crle -64 -c /var/ld/64/ld.config -l
  /lib/64:/usr/lib/64:/usr/sfw/lib/64:/opt/jdk1.6.0_21/jre/lib
  /amd64/server
  ```

Installing PL/Java

For Greenplum Database version 4.2 and later, the PL/Java extension is available as a package. Download the package from the EMC Download Center and then install it with the Greenplum Package Manager (`gppkg`).

The `gppkg` utility installs Greenplum Database extensions, along with any dependencies, on all hosts across a cluster. It also automatically installs extensions on new hosts in the case of system expansion and segment recovery.

For information about `gppkg`, see the *Greenplum Database Utility Guide*.

To install and use PL/Java:

1. Install the Greenplum Database PL/Java extension.
2. Enable the language for each database.
3. Install user-created JAR files containing Java methods on all Greenplum Database hosts.
4. Add the name of the JAR file to the Greenplum Database `pljava_classpath` environment variable. The variable lists the installed JAR files.

Installing the Greenplum PL/Java Extension

Before you install the PL/Java extension, make sure that your Greenplum database is running, you have sourced `greenplum_path.sh`, and that the `$MASTER_DATA_DIRECTORY` and `$GPHOME` variables are set.

1. Download the PL/Java extension package from the EMC Download Center then copy it to the master host.
2. Install the software extension package by running the `gppkg` command. This example installs the PL/Java extension package on a Linux system:

   ```
   $ gppkg -i pljava-1.1-rhel5-x86_64.gppkg
   ```

3. Restart the database.

   ```
   $ gpstop -r
   ```

4. Source the file `$GPHOME/greenplum_path.sh`.

Enabling PL/Java and Installing JAR Files

Perform the following steps as the Greenplum Database administrator gpadmin.

1. Enable PL/Java by running the SQL script `$GPHOME/share/postgresql/pljava/install.sql` in the databases that use PL/Java. For example, his example enables PL/Java on the database mytestdb:

   ```
   $ psql -d mytestdb
     -f $GPHOME/share/postgresql/pljava/install.sql
   ```

   The script `install.sql` registers both the trusted and untrusted PL/Java.

2. Copy your Java archives (JAR files) to `$GPHOME/lib/postgresql/java/` on to all Greenplum Database hosts. This example uses the Greenplum Database `gpscp` utility to copy the file `myclasses.jar`:

   ```
   $ gpscp -f gphosts_file myclasses.jar
   =:/usr/local/greenplum-db/lib/postgresql/java/
   ```

The file `gphosts_file` contains a list of the Greenplum Database hosts.

3. Set the `pljava_classpath` server configuration parameter in the master `postgresql.conf` file. The parameter value is a colon (:) separated list of the JAR files containing the Java classes used in any PL/Java functions. For example:

   ```
   $ gpconfig -c pljava_classpath
   -v \'examples.jar:myclasses.jar\' --masteronly
   ```

4. Restart the database:

   ```
   $ gpstop -r
   ```

5. (optional) Greenplum provides an `examples.sql` file containing sample PL/Java functions that you can use for testing. Run the commands in this file to create the test functions (which use the Java classes in `examples.jar`).

   ```
   $ psql -f $GPHOME/share/postgresql/pljava/examples.sql
   ```

Enabling the PL/Java extension in the template1 database enables PL/Java in any new Greenplum databases.

```
$ psql template1 -f $GPHOME/share/postgresql/pljava/install.sql
```

Uninstalling PL/Java

- Remove PL/Java Support for a Database
- Uninstall the Java JAR files and Software Package

Remove PL/Java Support for a Database

For a database that no long requires the PL/Java language, remove support for PL/Java. Run the `uninstall.sql` file as the gpadmin user. For example, this command disables the PL/Java language in the specified database.

```
$ psql -d mydatabase
  -f $GPHOME/share/postgresql/pljava/uninstall.sql
```

Uninstall the Java JAR files and Software Package

If no databases have PL/Java as a registered language, remove the Java JAR files and uninstall the Greenplum PL/Java extension with the `gppkg` utility.

1. Remove the `pljava_classpath` server configuration parameter in the master `postgresql.conf` file.

2. Remove the JAR files from the `$GPHOME/lib/postgresql/java/` directory of the Greenplum Database hosts.

3. Use the Greenplum `gppkg` utility with the `-r` option to uninstall the PL/Java extension. This example uninstalls the PL/Java extension on a Linux system:

   ```
   $ gppkg -r pljava-1.1
   ```

   The `gppkg` options `-q --all` lists the installed extensions and their versions.

4. After you uninstall the extension, restart the database.

```
$ gpstop -r
```

About Greenplum Database PL/Java

There are a few key differences between the implementation of PL/Java in standard PostgreSQL and Greenplum Database.

FUNCTIONS

The following functions are not supported in Greenplum Database. The classpath is handled differently in a distributed Greenplum Database environment than in the PostgreSQLenvironment.

`sqlj.install_jar`

`sqlj.replace_jar`

`sqlj.remove_jar`

`sqlj.get_classpath`

`sqlj.set_classpath`

Greenplum Database uses the `pljava_classpath` server configuration parameter in place of the `sqlj.set_classpath` function.

SERVER CONFIGURATION PARAMETERS

The following server configuration parameters are used by PL/Java in Greenplum Database. These parameters replace the pljava.* parameters used in the standard PostgreSQL PL/Java implementation:

- `pljava_classpath`
  A colon (:) separated list of the JAR files containing user-created Java classes. The listed JAR files must also be installed on all Greenplum hosts in the following location:

  `$GPHOME/lib/postgresql/java/`
- `pljava_statement_cache_size`
  Sets the size in KB of the Most Recently Used (MRU) cache for prepared statements.
- `pljava_release_lingering_savepoints`
  If TRUE, lingering savepoints will be released on function exit. If FALSE, they will be rolled back.
- `pljava_vmoptions`
  Defines the startup options for the Greenplum Database Java VM.

See the *Greenplum Database Reference Guide* for information about the Greenplum Database parameters.

Writing PL/Java functions

Information about writing functions with PL/Java.

SQL declaration

A Java function is declared with the name of a class and a static method on that class. The class will be resolved using the classpath that has been defined for the schema where the function is declared. If no classpath has been defined for that schema, the public schema is used. If no classpath is found there either, the class is resolved using the system classloader.

The following function can be declared to access the static method `getProperty` on java.lang.System class:

```
CREATE FUNCTION getsysprop(VARCHAR)
  RETURNS VARCHAR
  AS 'java.lang.System.getProperty'
  LANGUAGE java;
```

Run the following command to return the Java `user.home` property:

```
SELECT getsysprop('user.home');
```

Type mapping

Scalar types are mapped in a straight forward way. This table lists the current mappings.

Table 9.1 PL/Java data type mapping

PostgreSQL	Java
bool	boolean
‘char’	byte
int2	short
int4	int
int8	long

Table 9.1 PL/Java data type mapping

PostgreSQL	Java
varchar	java.lang.String
text	java.lang.String
bytea	byte[ ]
date	java.sql.Date
time	java.sql.Time (stored value treated as local time)
timetz	java.sql.Time
timestamp	java.sql.Timestamp (stored value treated as local time)
timestampz	java.sql.Timestamp
complex	java.sql.ResultSet
setof complex	java.sql.ResultSet

All other types are mapped to java.lang.String and will utilize the standard textin/textout routines registered for respective type.

NULL handling

The scalar types that map to Java primitives can not be passed as null values. To enable this, those types can have an alternative mapping. You enable this mapping by explicitly denoting it in the method reference.

```
CREATE FUNCTION trueIfEvenOrNull(integer)
  RETURNS bool
  AS 'foo.fee.Fum.trueIfEvenOrNull(java.lang.Integer)'
  LANGUAGE java;
```

In Java, you would have something like:

```
package foo.fee;
public class Fum
{
  static boolean trueIfEvenOrNull(Integer value)
  {
    return (value == null)
      ? true
      : (value.intValue() % 1) == 0;
  }
}
```

The following two statements both yield true:

```
SELECT trueIfEvenOrNull(NULL);
```

```
SELECT trueIfEvenOrNull(4);
```

In order to return null values from a Java method, you simply use the object type that corresponds to the primitive (i.e. you return java.lang.Integer instead of int). The PL/Java resolve mechanism will find the method regardless. Since Java cannot have different return types for methods with the same name, this does not introduce any ambiguity.

Complex types

A complex type will always be passed as a read-only `java.sql.ResultSet` with exactly one row. The ResultSet will be positioned on its row so no call to `next()` should be made. The values of the complex type are retrieved using the standard getter methods of the ResultSet.

Example:

```
CREATE TYPE complexTest
  AS(base integer, incbase integer, ctime timestamptz);


CREATE FUNCTION useComplexTest(complexTest)
  RETURNS VARCHAR
  AS 'foo.fee.Fum.useComplexTest'
  IMMUTABLE LANGUAGE java;
```

In class `Fum`, we add the following static method:

```
public static String useComplexTest(ResultSet complexTest)
throws SQLException
{
  int base = complexTest.getInt(1);
  int incbase = complexTest.getInt(2);
  Timestamp ctime = complexTest.getTimestamp(3);
  return "Base = \"" + base +
    "\", incbase = \"" + incbase +
    "\", ctime = \"" + ctime + "\"";
}
```

Returning complex types

Java does not stipulate any way to create a ResultSet from scratch. Hence, returning a ResultSet is not an option. The SQL-2003 draft suggests that a complex return value instead is handled as an IN/OUT parameter and PL/Java implements it that way. If you declare a function that returns a complex type, you will need to use a Java method with boolean return type with a last parameter of type `java.sql.ResultSet`. The parameter will be initialized to an empty updateable ResultSet that contains exactly one row.

Assume that the `complexTest` type in previous section has been created.

```
CREATE FUNCTION createComplexTest(int, int)
  RETURNS complexTest
  AS 'foo.fee.Fum.createComplexTest'
  IMMUTABLE LANGUAGE java;
```

The PL/Java method resolve will now find the following method in the `Fum` class:

```
public static boolean complexReturn(int base, int increment,
  ResultSet receiver)
throws SQLException
{
  receiver.updateInt(1, base);
  receiver.updateInt(2, base + increment);
  receiver.updateTimestamp(3, new
    Timestamp(System.currentTimeMillis()));
  return true;
}
```

The return value denotes if the receiver should be considered as a valid tuple (true) or NULL (false).

Functions returning sets

You should not first build a result set and then return it, because large result sets would consume a large amount of resources. It is better to produce one row at a time. Incidentally, that is what the Greenplum Database backend expects a function with SETOF return to do. You can return a SETOF a scalar type such as an int, float or varchar, or you can return a SETOF a complex type.

Returning a SETOF <scalar type>

In order to return a set of a scalar type, you need create a Java method that returns something that implements the `java.util.Iterator` interface. Here is an example of a method that returns a SETOF varchar:

```
CREATE FUNCTION javatest.getSystemProperties()
  RETURNS SETOF varchar
  AS 'foo.fee.Bar.getNames'
  IMMUTABLE LANGUAGE java;
```

The very rudimentary java method that returns an interator:

```
package foo.fee;
import java.util.Iterator;

public class Bar
{
    public static Iterator getNames()
    {
        ArrayList names = new ArrayList();
        names.add("Lisa");
        names.add("Bob");
        names.add("Bill");
        names.add("Sally");
        return names.iterator();
    }
}
```

Returning a SETOF <complex type>

A method returning a SETOF <complex type> must use either the interface `org.postgresql.pljava.ResultSetProvider` or `org.postgresql.pljava.ResultSetHandle`. The reason for having two interfaces is that they cater for optimal handling of two distinct use cases. The former is for cases when you want to dynamically create each row that is to be returned from the SETOF function. The latter makes is in cases where you want to return the result of an executed query.

Using the ResultSetProvider interface

This interface has two methods. The boolean `assignRowValues(java.sql.ResultSet tupleBuilder, int rowNumber)` and the `void close()` method. The Greenplum Database query evaluator will call the assignRowValues repeatedly until it returns false or until the evaluator decides that it does not need any more rows. It will then call close.

You can use this interface the following way:

```
CREATE FUNCTION javatest.listComplexTests(int, int)
  RETURNS SETOF complexTest
  AS 'foo.fee.Fum.listComplexTest'
  IMMUTABLE LANGUAGE java;
```

The function maps to a static java method that returns an instance that implements the ResultSetProvider interface.

```
public class Fum implements ResultSetProvider
{
  private final int m_base;
  private final int m_increment;
  public Fum(int base, int increment)
  {
    m_base = base;
    m_increment = increment;
  }
  public boolean assignRowValues(ResultSet receiver, int
currentRow)
  throws SQLException
  {
    // Stop when we reach 12 rows.
    //
    if(currentRow >= 12)
      return false;
    receiver.updateInt(1, m_base);
    receiver.updateInt(2, m_base + m_increment * currentRow);
    receiver.updateTimestamp(3, new
Timestamp(System.currentTimeMillis()));
    return true;
  }
  public void close()
  {
  // Nothing needed in this example
  }
```

```
  public static ResultSetProvider listComplexTests(int base,
int increment)
  throws SQLException
  {
    return new Fum(base, increment);
  }
}
```

The `listComplextTests` method is called once. It may return null if no results are available or an instance of the `ResultSetProvider`. Here the `Fum` implements this interface so it returns an instance of itself. The method `assignRowValues` will then be called repeatedly until it returns false. At that time, close will be called

Using the ResultSetHandle interface

This interface is similar to the `ResultSetProvider` interface in that it has a `close()` method that will be called at the end. But instead of having the evaluator call a method that builds one row at a time, this method has a method that returns a ResultSet. The query evaluator will iterate over this set and deliver it's contents, one tuple at a time, to the caller until a call to `next()` returns false or the evaluator decides that no more rows are needed.

Here is an example that executes a query using a statement that it obtained using the default connection. The SQL suitable for the deployment descriptor looks like this:

```
CREATE FUNCTION javatest.listSupers()
  RETURNS SETOF pg_user
  AS 'org.postgresql.pljava.example.Users.listSupers'
  LANGUAGE java;
CREATE FUNCTION javatest.listNonSupers()
  RETURNS SETOF pg_user
  AS 'org.postgresql.pljava.example.Users.listNonSupers'
  LANGUAGE java;
```

And in the Java package org.postgresql.pljava.example a class Users is added:

```
public class Users implements ResultSetHandle
{
  private final String m_filter;
  private Statement m_statement;

  public Users(String filter)
  {
    m_filter = filter;
  }

  public ResultSet getResultSet()
  throws SQLException
  {
    m_statement =
      DriverManager.getConnection("jdbc:default:connection").cr
eateStatement();
    return m_statement.executeQuery("SELECT * FROM pg_user
       WHERE " + m_filter);
  }
```

```
  public void close()
  throws SQLException
  {
    m_statement.close();
  }

  public static ResultSetHandle listSupers()
  {
    return new Users("usesuper = true");
  }

  public static ResultSetHandle listNonSupers()
  {
    return new Users("usesuper = false");
  }
}
```

Using JDBC

PL/Java contains a JDBC driver that maps to the PostgreSQL SPI functions. A connection that maps to the current transaction can be obtained using the following statement:

```
Connection conn =
  DriverManager.getConnection("jdbc:default:connection");
```

After obtaining a connection, you can prepare and execute statements similar to other JDBC connections. These are limitations for the PL/Java JDBC driver:

- The transaction cannot be managed in any way. Thus, you cannot use methods on the connection like:
  - `commit()`
  - `rollback()`
  - `setAutoCommit()`
  - `setTransactionIsolation()`
- Savepoints are available with some restrictions. A savepoint cannot outlive the function in which it was set and it must also be rolled back or released by that same function.
- A ResultSet returned from `executeQuery()` are always `FETCH_FORWARD` and `CONCUR_READ_ONLY`.
- Meta-data is only available in PL/Java 1.1 or higher.
- `CallableStatement` (for stored procedures) is not yet implemented.
- Clob or Blob types need more work. byte[] and String works fine for bytea and text respectively. A more efficient mapping is planned where the actual array is not copied.

Exception handling

You can catch and handle an exception in the Greenplum Database backend just like any other exception. The backend ErrorData structure is exposed as a property in a class called `org.postgresql.pljava.ServerException` (derived from `java.sql.SQLException`) and the Java try/catch mechanism is synchronized with the backend mechanism.

Important: You will not be able to continue executing backend functions until your function has returned and the error has been propagated when the backend has generated an exception unless you have used a savepoint. When a savepoint is rolled back, the exceptional condition is reset and you can continue your execution.

Savepoints

Greenplum Database savepoints are exposed using the java.sql.Connection interface. Two restrictions apply.

- A savepoint must be rolled back or released in the function where it was set.
- A savepoint must not outlive the function where it was set

Logging

PL/Java uses the standard Java 1.4 Logger. Hence, you can write things like:

```
Logger.getAnonymousLogger().info( "Time is " + new
Date(System.currentTimeMillis()));
```

At present, the logger is hard wired to a handler that maps the current state of the Greenplum Database configuration setting `log_min_messages` to a valid Logger level and that outputs all messages using the backend function `elog()`. The following mapping apply between the Logger levels and the Greenplum Database backend levels.

Table 9.2 PL/Java Logging Levels

java.util.logging.Level	PostgreSQL level
SEVERE ERROR	ERROR
WARNING	WARNING
INFO	INFO
FINE	DEBUG1
FINER	DEBUG2
FINEST	DEBUG3

Security

- Installation

- Trusted language
- Execution of the deployment descriptor
- Classpath manipulation

Installation

Only a super user can install PL/Java. The PL/Java utility functions are installed using SECURITY DEFINER so that they execute with the access permissions that where granted to the creator of the functions.

Trusted language

PL/Java is a TRUSTED language. PostgreSQL stipulates that a language marked as trusted has no access to the file system and PL/Java enforces this. Any user can create and access functions in a trusted language. PL/Java also installs a language handler for the language `javau`. This version is not trusted and only a superuser can create new functions that use it. Any user can still call the functions.

Execution of the deployment descriptor

The `install_jar`, `replace_jar`, and `remove_jar`, optionally executes commands found in a SQL deployment descriptor. Such commands are executed with the permissions of the caller. In other words, although the utility function is declared with SECURITY DEFINER, it switches back to the session user during execution of the deployment descriptor commands.

Classpath manipulation

The function `set_classpath` requires the caller of the function has been granted `CREATE` permission on the affected schema.

Some PL/Java Issue and Solutions

When writing the PL/Java, mapping the JVM into the same process-space as the Greenplum Database backend code, some concerns have been raised regarding multiple threads, exception handling, and memory management. Here is a brief text explaining how these issues where resolved.

- Multi threading
- Exception handling
- Java Garbage Collector versus palloc() and stack allocation

Multi threading

Java is inherently multi threaded. The Greenplum Database backend is not. There's nothing stopping a developer from utilizing multiple Threads class in the Java code. Finalizers that call out to the backend might have been spawned from a background

Garbage Collection thread. Several third party Java-packages that are likely to be used make use of multiple threads. How can this model coexist with the Greenplum Database backend in the same process without creating havoc?

Solution

The solution is simple. PL/Java defines a special object called the Backend.THREADLOCK. When PL/Java is initialized, the backend will immediately grab this objects monitor (i.e. it will synchronize on this object). When the backend calls a Java function, the monitor is released and then immediately regained when the call returns. All calls from Java out to backend code are synchronized on the same lock. This ensures that only one thread at a time can call the backend from Java, and only at a time when the backend is awaiting the return of a Java function call.

Exception handling

Java makes frequent use of try/catch/finally blocks. Greenplum Database sometimes use an exception mechanism that calls `longjmp` to transfer control to a known state. Such a jump would normally effectively bypass the JVM.

Solution

The backend now allows errors to be caught using the macros PG_TRY/PG_CATCH/PG_END_TRY and in the catch block, the error can be examined using the ErrorData structure. PL/Java implements a java.sql.SQLException subclass called org.postgresql.pljava.ServerException. The ErrorData can be retrieved and examined from that exception. A catch handler is allowed to issue a rollback to a savepoint. After a successful rollback, execution can continue.

Java Garbage Collector versus palloc() and stack allocation

Primitive types are always be passed by value. This includes the `String` type (this is a must since Java uses double byte characters). Complex types are often wrapped in Java objects and passed by reference. For example, a Java object can contain a pointer to a palloc'ed or stack allocated memory and use native JNI calls to extract and manipulate data. Such data will become stale once a call has ended. Further attempts to access such data will at best give very unpredictable results but more likely cause a memory fault and a crash.

Solution

The PL/Java contains code that ensures that stale pointers are cleared when the MemoryContext or stack where they where allocated goes out of scope. The Java wrapper objects might live on but any attempt to use them will result in a stale native handle exception.

Example

The following simple Java example creates a JAR file that contains a single method and runs the method.

Note: The example requires Java SDK to compile the Java file.

The following method returns a substring.

```
{

public static String substring(String text, int beginIndex, int
endIndex)
    {
    return text.substring(beginIndex, endIndex);
    }
}
```

Enter the java code in a text file example.class.

Content of the file manifest.txt:

```
Manifest-Version: 1.0
Main-Class: Example
Specification-Title: "Example"
Specification-Version: "1.0"
Created-By: 1.6.0_35-b10-428-11M3811
Build-Date: 01/20/2013 10:09 AM
```

Compile the java code:

```
javac *.java
```

Create a JAR archive named analytics.jar that contains the class file and the manifest file MANIFEST file in the JAR.

```
jar cfm analytics.jar manifest.txt *.class
```

Upload the jar file to the Greenplum master host.

Run the gpscp utility to copy the jar file to the Greenplum Java directory. Use the -f option to specify the file that contains a list of the master and segment hosts.

```
gpscp -f gphosts_file analytics.jar
=:/usr/local/greenplum-db/lib/postgresql/java/
```

Use the gpconfig utility to set the Greenplum `pljava_classpath` environment variable. The variable lists the installed jar files.

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

Run the gpstop utility with the -u option to reload the configuration files.

```
gpstop -u
```

From the psql command line, run the following command to show the installed jar files.

```
show pljava_classpath
```

The following SQL commands create a table and define a Java function to test the method in the jar file:

```
create table temp (a varchar) distributed randomly;
insert into temp values ('my string');
--Example function
create or replace function java_substring(varchar, int, int)
```

```
returns varchar as 'Example.substring' language java;
--Example execution
select java_substring(a, 1, 5) from temp;
```

You can place the contents in a file, mysample.sql and run the command from a psql command line:

```
\i mysample.sql
```

The output is similar to the following:

```
java_substring
----------------
 y st
(1 row)
```

References

Notes on using PL/Java:

http://pljava.projects.postgresql.org/

10. PL/Python Extension

The Greenplum Database PL/Python extension is installed by default with Greenplum Database.

PL/Python is a loadable procedural language. With the Greenplum Database PL/Python extension, you can write database functions in Python that take advantage of Python features to quickly build robust prototype database applications.

This chapter includes the following information:

Greenplum PL/Python

Greenplum Database installs a version of Python and PL/Python. The default location for Python is:

```
$GPHOME/ext/python/bin/
```

Greenplum Database PL/Python Limitations

Greenplum Database does not support PL/Python triggers.

PL/Python is available only as an untrusted language.

Enabling and Removing PL/Python support

PL/Python is installed with Greenplum Database.

Enabling PL/Python Support

For each database that requires its use, register the PL/Python language with the SQL command `CREATE LANGUAGE` or the utility `createlang`. For example, following command run as the gpadmin user registers the language for the database `testdb`:

```
$ createlang plpythonu -d testdb
```

PL/Python is registered as an untrusted language.

Removing PL/Python Support

For a database that no long requires the PL/Python language, remove support for PL/Python with the SQL command `DROP LANGUAGE` or the Greenplum Database `droplang` utility. For example, running this command run as the gpadmin removes support for the PL/Python from the database `testdb`:

```
$ droplang plpythonu -d testdb
```

When you remove support for PL/Python, the PL/Python routines that you created in the database will no longer work.

Developing Functions with PL/Python

The body of a PL/Python function is a Python script. When the function is called, its arguments are passed as elements of the array args[]; named arguments are also passed as ordinary variables to the Python script. The result is returned from the Python code in the usual way, with `return` or `yield` (in case of a result-set statement).

The PL/Python language module automatically imports the Python module `plpy`. The module `plpy` implements the functions `plpy.debug(`*msg*`)`, `plpy.log(`*msg*`)`, `plpy.info(`*msg*`)`, `plpy.notice(`*msg*`)`, `plpy.warning(`*msg*`)`, `plpy.error(`*msg*`)`, and `plpy.fatal(`*msg*`)`. `plpy.error` and `plpy.fatal` actually raise a Python exception which, if uncaught, propagates out to the calling query, causing the current transaction or subtransaction to be aborted. `raise plpy.ERROR(msg)` and `raise plpy.FATAL(msg)` are equivalent to calling `plpy.error` and `plpy.fatal`, respectively. The other functions only generate messages of different priority levels.

Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the `log_min_messages` and `client_min_messages` configuration variables.

Accessing a database

The PL/Python `plpy` module provides two functions called `execute` and `prepare`. Calling `plpy.execute` with a query string and an optional limit argument causes that query to be run and the result to be returned in a result object. The result object emulates a list or dictionary object. The result object can be accessed by row number and column name. It has these additional methods: `nrows` which returns the number of rows returned by the query, and `status` which is the `SPI_execute()` return value. The result object can be modified.

For example, this statement can be in a PL/Python user-defined function.

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

It returns up to 5 rows from `my_table`. If `my_table` has a column `my_column`, it would be accessed as:

```
foo = rv[i]["my_column"]
```

The second function, `plpy.prepare`, prepares the execution plan for a query. It is called with a query string and a list of parameter types, if you have parameter references in the query. For example, this statement can be in a PL/Python user-defined function:

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE
  first_name = $1", [ "text" ])
```

`text` is the type of the variable you will be passing for $1. After preparing a statement, you use the function `plpy.execute` to run it:

```
rv = plpy.execute(plan, [ "name" ], 5)
```

The third argument is the limit for the number of rows returned and is optional.

When you prepare a plan using the PL/Python module it is automatically saved. See the Postgres SPI documentation for a description of what this means.

http://www.postgresql.org/docs/8.2/static/spi.html

In order to make effective use of this across function calls one needs to use one of the persistent storage dictionaries SD or GD.

The global dictionary SD is available to store data between function calls. This variable is private static data. The global dictionary GD is public data, available to all Python functions within a session. Use GD with care.

Each function gets its own execution environment in the Python interpreter, so that global data and function arguments from `myfunc` are not available to `myfunc2`. The exception is the data in the GD dictionary, as mentioned previously.

The following example uses the SD dictionary:

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
  if SD.has_key("plan"):
    plan = SD["plan"]
  else:
    plan = plpy.prepare("SELECT 1")
    SD["plan"] = plan

  # rest of function

$$ LANGUAGE plpythonu;
```

Example

You can create a PL/Python function to return the maximum of two integers:

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
    return None
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

You can use the STRICT property to perform the null handling instead of using the two conditional statements.

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer AS $$
return max(a,b)
$$ LANGUAGE plpythonu STRICT ;
```

You can run the user-defined function `pymax`.

```
select ( pymax(123, 43));
```

```
column1
---------
     123

(1 row)
```

References

Technical References

For information about PL/Python see the PostgreSQL documentation at http://www.postgresql.org/docs/8.2/static/plpython.html.

http://www.postgresql.org/docs/8.2/static/plpython.html

Useful Reading

For information about the Perl language, see http://www.python.org/.

11. PL/Perl Extension

PL/Perl is a loadable procedural language. With the Greenplum Database PL/Perl extension, you can write database functions in Perl.

With PL/Perl you can create stored functions that can take advantage of string manipulation operators and functions available for Perl. For example, you can more easily create functions that parse complex strings in Perl than similar functions created in PL/pgSQL.This chapter includes the following information:

- Installing PL/Perl
- Uninstalling PL/Perl
- Installing and Using Perl Modules
- Examples
- References

Greenplum PL/Perl

The Greenplum Database PL/Perl utilizes the native operating system Perl library.

The PostgreSQL documentation describes limitations and missing features from PL/Perl. See http://www.postgresql.org/docs/8.2/static/plperl-missing.html.

For general information about Greenplum Database user-defined functions, see "Creating and Using Server Functions " in Chapter 4, "Database Application Development".

Greenplum Database PL/Perl Limitations

Greenplum Database does not support PL/Perl triggers.

Notes on using PL/Perl

For limitations and missing features from PL/Perl, see:

http://www.postgresql.org/docs/8.2/static/plperl-missing.html

The PostgreSQL documentation on PL/Perl is available here:

http://www.postgresql.org/docs/8.2/static/plperl.html

[**MSK**] does Greenplum Database support both trusted and untrusted PL/Perl?

For security reasons, the trusted version of PL/Perl executes functions called by any one SQL role in a separate Perl interpreter for that role.

Installing PL/Perl

There is a Greenplum PL/Perl extension built for Red Hat 5, Red Hat 6, and SUSE 11. Download and install the corresponding PL/Perl extension that matches your Linux based operating system. To utilize the PL/Perl extension, an official Perl package from the supported distribution must be used. This table lists the PL/Perl extension versions supported by Greenplum Database 4.2.x. For the versions of the PL/Perl extension supported by your version of Greenplum Database, see the Greenplum Database release notes:

Table 11.1 PL/Perl Language Extension Versions

PL/Perl Extension Version	Perl Version	Supported Platforms
PL/Perl 1.2	5.12.4 5.8.8	RHEL 6.x, RHEL 5.x, SUSE 10
PL/Perl 1.1	5.12.4	RHEL 5.x, SUSE 10
PL/Perl 1.0	5.12.4	RHEL 5.x, SUSE 10

For Greenplum Database version 4.2 and later, the PL/Perl extension is available as a package. Download the package from the EMC Download Center and install it with the Greenplum Database Package Manager utility (`gppkg`).

The `gppkg` utility installs Greenplum Database extensions, along with any dependencies, on all hosts across a cluster. It also automatically installs extensions on new hosts in the case of system expansion and segment recovery.

For information about `gppkg`, see the *Greenplum Database Utility Guide*.

Installing the PL/Perl Software Package

Before you install the PL/Perl software package, make sure that your Greenplum database is running, you have sourced `greenplum_path.sh`, and that the `$MASTER_DATA_DIRECTORY` and `$GPHOME` variables are set.

1. Download the Greenplum Database PL/Perl package from the EMC Download Center then copy it to the master host.
2. Install the software package by running the `gppkg` command. This example installs the PL/Perl package on a Linux system:

   ```
   $ gppkg -i plperl-1.1-rhel5-x86_64.gppkg
   ```

Installing PL/Perl installs Perl.

Enabling PL/Perl Support

For each database that requires its use, register the PL/Perl language with the SQL command `CREATE LANGUAGE` or the utility `createlang`. For example, following command run as the gpadmin user registers the language for the database `testdb`:

```
$ createlang plperl -d testdb
```

This command registers PL/Perl as an untrusted language:

```
$ createlang plperlu -d testdb
```

Uninstalling PL/Perl

Remove PL/Perl Support

Uninstall the Software Package

When you remove PL/Perl language support from a database, the PL/Perl routines that you created in the database will no longer work.

Remove PL/Perl Support

For a database that no long requires the PL/Perl language, remove support for PL/Perl with the SQL command `DROP LANGUAGE` or the Greenplum Database `droplang` utility. For example, running this command run as the gpadmin user removes support for PL/Perl from the database `testdb`:

```
$ droplang plperl -d testdb
```

When you remove PL/Perl language support from a database, the PL/Perl routines that you created in the database will no longer work.

Uninstall the Software Package

If no databases have PL/Perl as a registered language, uninstall the Greenplum PL/Perl extension with the `gppkg` utility and the `-r` option. This example uninstalls PL/R extension Version 1.0

```
$ gppkg -r plperl-1.0
```

The `gppkg` options `-q --all` lists the installed extensions and their versions.

Use the Greenplum gppkg utility with the -r option to uninstall the PL/Perl package. When removing the package you must specify the package and version. This example uninstalls PL/Perl Package Version 1.0

```
$ gppkg -r plperl-1.0
```

The `gppkg` option `-q --all` lists the installed packages and their versions.

After you uninstall the package, restart the database.

```
$ gpstop -r
```

Installing and Using Perl Modules

[MSK] Is this supported for GPDB? If so, how is it done?

Optional Perl modules extend Perl functionality. For information about installing Perl modules, see http://www.postgresql.org/docs/9.0/static/plperl-under-the-hood.html.

The modules must be installed on all the Greenplum Database hosts.

Data Values in PL/Perl

Within the PL/Perl function, all values are text strings. The argument values supplied to a PL/Perl function are the input arguments converted to text form (just as if they had been displayed by a SELECT statement). Conversely, the return command will accept any string that is acceptable input format for the function's declared return type.

Global Values in PL/Perl

You can use the global hash `%_SHARED` to store data, including code references, between function calls for the lifetime of the current session.

[**MSK**] Is the following supported?
For the trusted PL/Perl, functions called by any one SQL role run in a Perl interpreter for that role. Two PL/Perl functions will share the same value of `%_SHARED` if and only if they are executed by the same SQL role. To ensure that PL/Perl functions can share `%_SHARED` data, make sure that functions that should communicate are owned by the same user, and specify `SECURITY DEFINER` when creating or altering the function.

For untrusted PL/Perl, functions executed in a given session run in a single Perl interpreter which is not any of the ones used for trusted PL/Perl functions. This allows untrusted PL/Perl functions to share data freely, but no communication can occur between trusted and untrusted PL/Perl functions.

Examples

You can create a PL/Perl function:

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS
$$

    my ($x,$y) = @_;
    if (! defined $x) {
       if (! defined $y) { return undef; }
       return $y;
    }

       if (! defined $y) { return $x; }
       if ($x > $y) { return $x; }
       return $y;
$$ LANGUAGE plperl;
```

You can run the function to return the maximum of two integers.

```
select ( perl_max(123, 43));
column1
---------
      123

(1 row)
```

Accessing a database

You use the database with the PL/Perl command `spi_exec_query`. PL/Perl also inludes other functions to work with queries and fetching data from queries. See http://www.postgresql.org/docs/8.2/static/plperl-database.html.

[**MSK**] are all spi_* functions supported?

The following example runs a `SELECT` statement and modifies the returned data.

This table is used in the example:

```
CREATE TABLE test (
    i int,
    v varchar
);
```

```
INSERT INTO test (i, v) VALUES (1, 'first line');
INSERT INTO test (i, v) VALUES (2, 'second line');
INSERT INTO test (i, v) VALUES (3, 'third line');
INSERT INTO test (i, v) VALUES (4, 'immortal');
```

This PL/Perl function runs the `spi_exec_query` function with this `SELECT` statement:

```
select i, v from test
```

The function modifies the data returned data. It adds 200 to the the data returned for the first column and changes the text in the second column to upper case.

```
CREATE OR REPLACE FUNCTION test_plfunc() RETURNS SETOF test AS $$
    my $rv = spi_exec_query('select i, v from test;');
    my $status = $rv->{status};
    my $nrows = $rv->{processed};

    foreach my $rn (0 .. $nrows - 1) {
       my $row = $rv->{rows}[$rn];
       $row->{i} += 200 if defined($row->{i});
       $row->{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row->{v}));
       return_next($row);
    }

    return undef;
$$ LANGUAGE plperl;
```

This `SELECT` statement runs the function.

```
SELECT * FROM test_plfunc();
```

References

Technical References

For information about PL/Perl see the PostgreSQL documentation at http://www.postgresql.org/docs/8.2/static/plperl.html.

Useful Reading

For information about the Perl language, see http://www.perl.org/.

12. PL/R Extension

PL/R is a procedural language. With the Greenplum Database PL/R extension you can write database functions in the R programming language and use R packages that contain R functions and data sets.

This chapter contains the following information:

Installing PL/R

For Greenplum Database version 4.2 and later, the PL/R extension is available as a package. Download the package from the EMC Download Center and install it with the Greenplum Package Manager (`gppkg`).

The `gppkg` utility installs Greenplum Database extensions, along with any dependencies, on all hosts across a cluster. It also automatically installs extensions on new hosts in the case of system expansion and segment recovery.

For information about `gppkg`, see the *Greenplum Database Utility Guide*.

Installing the Software Extension Package

Before you install the PL/R extension, make sure that your Greenplum database is running, you have sourced `greenplum_path.sh`, and that the `$MASTER_DATA_DIRECTORY` and `$GPHOME` variables are set.

1. Download the PL/R extension package from the EMC Download Center then copy it to the master host.
2. Install the software extension package by running the `gppkg` command. This example installs the PL/R extension on a Linux system:

   ```
   $ gppkg -i plr-1.0-rhel5-x86_64.gppkg
   ```

3. Restart the database.

   ```
   $ gpstop -r
   ```

4. Source the file `$GPHOME/greenplum_path.sh`.

The extension and the R environment is installed in this directory:

```
$GPHOME/ext/R-2.13.0/
```

Enabling PL/R Language Support

For each database that requires its use, register the PL/R language with the SQL command `CREATE LANGUAGE` or the utility `createlang`. For example, this command registers the language for the database testdb:

```
$ createlang plr -d testdb
```

PL/R is registered as an untrusted language.

Uninstalling PL/R

- Remove PL/R Support for a Database
- Uninstall the Software Package

When you remove PL/R language support from a database, the PL/R routines that you created in the database will no longer work.

Remove PL/R Support for a Database

For a database that no long requires the PL/R language, remove support for PL/R with the SQL command `DROP LANGUAGE` or the Greenplum Database `droplang` utility. For example, running this command run as the gpadmin user removes support for PL/R from the database testdb:

```
$ droplang plr -d testdb
```

Uninstall the Software Package

If no databases have PL/R as a registered language, uninstall the Greenplum PL/R extension with the `gppkg` utility. This example uninstalls PL/R Package Version 1.0

```
$ gppkg -r plr-1.0
```

The `gppkg` options `-q --all` lists the installed extensions and their versions.

Restart the database.

```
$ gpstop -r
```

Examples

The following are simple PL/R examples.

Example 1: Using PL/R for single row operators

This function generates an array of numbers with a normal distribution using the R function `rnorm()`.

```
CREATE OR REPLACE FUNCTION r_norm(n integer, mean float8,
  std_dev float8) RETURNS float8[ ] AS
$$
```

```
  x<-rnorm(n,mean,std_dev)
  return(x)
$$
LANGUAGE 'plr';
```

The following CREATE TABLE command uses the r_norm function to populate the table. The rnorm function creates an array of 10 numbers.

```
CREATE TABLE test_norm_var
  AS SELECT id, r_norm(10,0,1) as x
  FROM (SELECT generate_series(1,30:: bigint) AS ID) foo
  DISTRIBUTED BY (id);
```

Example 2: Returning PL/R data.frames in Tabular Form

Assuming your PL/R function returns an R `data.frame` as its output, unless you want to use arrays of arrays, some work is required to see your `data.frame` from PL/R as a simple SQL table:

- Create a `TYPE` in a Greenplum database with the same dimensions as your R `data.frame`:
  ```
  CREATE TYPE t1 AS ...
  ```
- Use this `TYPE` when defining your PL/R function
  ```
  .. RETURNS SET OF t1 AS ...
  ```

Sample SQL for this is given in the next example.

Example 3: Hierarchical Regression using PL/R

The SQL below defines a `TYPE` and runs hierarchical regression using PL/R:

```
--Create TYPE to store model results
DROP TYPE IF EXISTS wj_model_results CASCADE;
CREATE TYPE wj_model_results AS (
cs text, coefext float, ci_95_lower float, ci_95_upper
float, ci_90_lower, float, ci_90_upper float, ci_80_lower,
float, ci_80_upper float);

--Create PL/R function to run model in R
DROP FUNCTION wj.plr.RE(response float [ ], cs text [ ])
RETURNS SETOF wj_model_results AS
$$
   library(arm)
   y<- log(response)
   cs<- cs
   d_temp<- data.frame(y,cs)
   m0 <- lmer (y ~ 1 + (1 | cs), data=d_temp)
   cs_unique<- sort(unique(cs))
```

```
    n_cs_unique<- length(cs_unique)
    temp_m0<- data.frame(matrix0,n_cs_unique, 7))
    for (i in 1:n_cs_unique){temp_m0[i,]<-
      c(exp(coef(m0)$cs[i,1] + c(0,-1.96,1.96,-1.65,1.65
      -1.28,1.28)*se.ranef(m0)$cs[i]))}
    names(temp_m0)<- c("Coefest", "CI_95_Lower",
      "CI_95_Upper", "CI_90_Lower", "CI_90_Upper",
      "CI_80_Lower", "CI_80_Upper")
    temp_m0_v2<- data.frames(cs_unique, temp_m0)
    return(temp_m0_v2)
$$
LANGUAGE 'plr';

--Run modeling plr function and store model results in a
--table
DROP TABLE IF EXISTS wj_model_results_roi;
CREATE TABLE wj_model_results_roi AS SELECT * FROM
wj.plr_RE((SELECT wj.droi2_array), (SELECT cs FROM
wj.droi2_array));
```

Displaying R Library Information

You can use the R command line to display information about the installed libraries and functions. You can also add and remove libraries from the R installation. To startt the R command line, run the script R from the directory `$GPHOME/ext/R-2.13.0/bin`.

This R function lists the available R packages from the R command line:

```
> library()
```

Display the documentation for a particular R package

```
> library(help="package_name")
> help(package="package_name")
```

Display the help file for an R function:

```
> help("function_name")
> ?function_name
```

To see what packages are installed, use the R command `installed.packages()`. This will return a matrix with a row for each package that has been installed. Below, we look at the first 5 rows of this matrix.

```
> installed.packages()
```

Any package that does not appear in the installed packages matrix must be installed and loaded before its functions can be used.

An R package can be installed with `install.packages()`:

```
install.packages("package_name")
install.packages("mypkg", dependencies = TRUE, type="source")
```

Load a package from the R command line

```
> library("package_name")
```

An R package can be removed with remove.packages

```
> remove.packages("package_name")
```

Downloading and Installing R Packages

R packages are modules that contain R functions and data sets. You can install R packages to extend R and PL/R functionality in a Greenplum database.

1. For a given R package, identify all dependent R packages and each package web url. This can be found by selecting the given package from the following navigation page:

   http://cran.r-project.org/web/packages/available_packages_by_name.html

   For example, from the page for the `arm` library, you can see that this library requires the following R libraries: `Matrix`, `lattice`, `lme4`, `R2WinBUGS`, `coda`, `abind`, `foreign`, and `MASS`.

   R CMD INSTALL --build <package.tar.gz>

2. From the command line, use `wget` to download the `tar.gz` files for the required libraries to the master node:

   ```
   wget
   http://cran.r-project.org/src/contrib/arm_1.5-03.tar.gz
   wget
   http://cran.r-project.org/src/contrib/Archive/Matrix/Matrix_1.0-1.tar.gz
   wget
   http://cran.r-project.org/src/contrib/Archive/lattice/lattice_0.19-33.tar.gz
   wget
   http://cran.r-project.org/src/contrib/lme4_0.999375-42.tar.gz
   wget
   http://cran.r-project.org/src/contrib/R2WinBUGS_2.1-18.tar.gz
   wget
   http://cran.r-project.org/src/contrib/coda_0.14-7.tar.gz
   wget
   ```

```
http://cran.r-project.org/src/contrib/abind_1.4-0.tar.gz
wget
http://cran.r-project.org/src/contrib/foreign_0.8-49.tar.
gz
wget
http://cran.r-project.org/src/contrib/MASS_7.3-17.tar.gz
```

3. Using `gpscp` and the hostname file, copy the `tar.gz` files to the same directory on all nodes of the Greenplum cluster. You might require root access to do this.

```
gpscp -f /home/gpadmin/hosts_all lattice_0.19-33.tar.gz
=:/home/gpadmin
gpscp -f /home/gpadmin/hosts_all Matrix_1.0-1.tar.gz
=:/home/gpadmin
gpscp -f /home/gpadmin/hosts_all abind_1.4-0.tar.gz
=:/home/gpadmin
gpscp -f /home/gpadmin/hosts_all coda_0.14-7.tar.gz
=:/home/gpadmin
gpscp -f /home/gpadmin/hosts_all R2WinBUGS_2.1-18.tar.gz
=:/home/gpadmin
gpscp -f /home/gpadmin/hosts_all lme4_0.999375-42.tar.gz
=:/home/gpadmin
gpscp -f /home/gpadmin/hosts_all MASS_7.3-17.tar.gz
=:/home/gpadmin
gpscp -f /home/gpadmin/hosts_all arm_1.5-03.tar.gz
=:/home/gpadmin
```

4. Use `R CMD INSTALL` to install the packages from the command line. You might require root access to do this.

```
R CMD INSTALL lattice_0.19-33.tar.gz Matrix_1.0-1.tar.gz
abind_1.4-0.tar.gz coda_0.14-7.tar.gz
R2WinBUGS_2.1-18.tar.gz lme4_0.999375-42.tar.gz
MASS_7.3-17.tar.gz arm_1.5-03.tar.gz
```

References

Tutorial for using R and PL/R with Greenplum Database. This tutorial also contains information on the R package PivotalR.

http://zimmeee.github.io/gp-r/

R Functions and Arguments

- See `http://www.joeconway.com/plr/doc/plr-funcs.html`

Passing Data Values in R

- See `http://www.joeconway.com/plr/doc/plr-data.html`

Aggregate Functions in R

- See `http://www.joeconway.com/plr/doc/plr-aggregate-funcs.html`

R documentation is installed with the Greenplum R package:

$GPHOME/ext/R-2.13.0/lib64/R/doc

13. MADlib Extension

MADlib is an open-source library for scalable in-database analytics. With the Greenplum Database MADlib extension, you can use MADlib functionality in a Greenplum Database.

MADlib provides data-parallel implementations of mathematical, statistical and machine-learning methods for structured and unstructured data. It provides an suite of SQL-based algorithms for machine learning, data mining and statistics that run at scale within a database engine, with no need for data import/export to other tools.

MADlib serves a role for scalable database systems that is similar to the CRAN library for R: a community repository of statistical methods. MADlib modules are written with scalability and parallelism in mind.

This chapter includes the following information:

Installing MADlib

To install MADlib on Greenplum Database, you install the Greenplum MADlib package on Greenplum Database and then install the MADlib function libraries on the databases that use MADlib. For the versions of the MADlib extension supported by your version of Greenplum Database, see the Greenplum Database release notes.

For Greenplum Database version 4.2 and later, the MADlib extension is available as a package. Download the package from the EMC Download Center and install it with the Greenplum Package Manager (`gppkg`).

The `gppkg` utility installs Greenplum Database extensions, along with any dependencies, on all hosts across a cluster. It also automatically installs extensions on new hosts in the case of system expansion segment recovery.

For information about `gppkg`, see the *Greenplum Database Utility Guide*.

Installing the Greenplum Database MADlib Package

Before you install the MADlib package, make sure that your Greenplum database is running, you have sourced `greenplum_path.sh`, and that the `$MASTER_DATA_DIRECTORY` and `$GPHOME` variables are set.

1. Download the Greenplum Database MADlib package from the EMC Download Center then copy it to the master host.
2. Install the software package by running the `gppkg` command. This example installs the MADlib package version 1.4 on a Linux system:

```
$ gppkg -i madlib-1.4-rhel5-x86_64.gppkg
```

Adding MADlib Functions to a Database

After installing the MADlib package, run the `madpack` command to add MADlib functions to Greenplum database. `madpack` is in `$GPHOME/madlib/bin`.

```
$ madpack install [-s schema_name] -p greenplum -c
   user@host:port/database
```

For example, this command creates MADlib functions in the Greenplum database testdb running on server mdw on port 5432. The madpack command logs in as the user gpadmin and prompts for password. The target schema is madlib.

```
$ madpack install -s madlib -p greenplum -c
   gpadmin@mdw:5432/testdb
```

The madpack `--help` option lists the madpack options:

```
$ madpack --help
```

The madpack `install-check` option checks the MADlib installation:

```
$ madpack install-check -p greenplum
```

Uninstalling MADlib

- Remove MADlib objects from the database
- Uninstall the Greenplum Database MADlib Package

When you remove MADlib support from a database, routines that you created in the database that use MADlib functionality will no longer work.

Remove MADlib objects from the database

Use the madpack uninstall command to remove MADlib objects from a Greenplum database. For example, this command removes MADlib objects from the database testdb.

```
$ madpack uninstall -s madlib -p greenplum -c
   gpadmin@mdw:5432/testdb
```

Uninstall the Greenplum Database MADlib Package

Use the Greenplum `gppkg` utility with the `-r` option to uninstall the MADlib package. When removing the package you must specify the package and version. This example uninstalls MADlib Package Version 1.4.

```
$ gppkg -r madlib-1.4
```

The `gppkg` option `-q --all` lists the installed packages and their versions.

After you uninstall the package, restart the database.

```
$ gpstop -r
```

Example

[MSK] Caleb suggests using a different example. Where can I get a simple MADlib example?

Support vector machines (SVMs) and related kernel methods is a popular and well-studied machine learning technique. The following example uses SVM classification to predict if a stock will go up or down.

- The ID of each vector is the number of days after 2000-01-01.
- The data to learn from is a set of 2 dimensional vectors. The first element is the 5 day moving avg slope of the test stock price. The 2nd element is the temperature of New York City at 9:00 AM.
- The value +1 is assigned to the class of up and the value -1 to the class of down.

A set of three vectors looks like this:

```
[3, 0.001, 28.5, +1]

[33, 0.002, 38.5, -1]

[333, -0.001, 30.5, +1]
```

The vector that is used for a prediction looks like this:

```
[444, 0.0015, 40.2]
```

The following SQL commands create a table and generates a prediction for whether a the test stock will go up or down based on the three vectors.

```
create table test_stock(id int, ind float8[], label float8)
  distributed by (id ) ;

insert into test_stock values (3,array[0.001,28.5],1),
  (33,array[0.002,38.5],-1), (333,array[-0.001,30.5],1);

select madlib.svm_classification('test_stock',
  'test_stock',false, 'madlib.svm_dot');

select madlib.svm_predict('test_stock', '{0.0015,40.2}') > 0;
```

References

Technical References

MADlib documentation is at http://doc.madlib.net.

The MADlib documentation is also installed locally in `$GPHOME/doc`

14. PostGIS Extension

With the Greenplum Database PostGIS extension you can store and manage GIS (Geographic Information Systems) information in a Greenplum database. PostGIS includes support for GiST-based R-Tree spatial indexes and functions for analysis and processing of GIS objects.

Greenplum PostGIS

The Greenplum PostGIS extension is available from the EMC Download Center. You can install it using the Greenplum Package Manager (`gppkg`). For details, see `gppkg` in the *Greenplum Database Utility Guide*.

- Greenplum Database 4.2.6 and later supports PostGIS extension package version 1.0 and 2.0 (PostGIS 1.4 and 2.0.3)

  Only one version of the PostGIS extension package, either 1.0 or 2.0, can be installed on an installation of Greenplum Database.
- Greenplum Database prior to 4.2.6 supports PostGIS extension package version 1.0 (PostGIS 1.4).

Table 14.1 PostGIS Component Version

PostGIS Extension Package	PostGIS	Geos	Proj
2.0	2.0.3	3.3.8	4.8.0
1.0	1.4.2	3.2.2.	4.7.0

Major enhancements and changes in 2.0.3 from 1.4.2 include:

- Support for geographic coordinates (latitude and longitude) with a GEOGRAPHY type and related functions.
- Input format support for these formats: GML, KML, and JSON
- Unknown SRID changed from -1 to 0
- 3D relationship and measurement support functions
- Making spatial indexes 3D aware
- KNN GiST centroid distance operator
- Many deprecated functions are removed

- Performance improvements

See the PostGIS documentation for a list of changes.
http://postgis.net/docs/manual-2.0/release_notes.html

Warning: PostGIS 2.0 removed many functions that were deprecated but available in PostGIS 1.4. Functions and applications written with functions that were deprecated in PostGIS 1.4 might need to be rewritten. See the PostGIS documentation for a list of new, enhanced, or changed functions.
http://postgis.net/docs/manual-2.0/PostGIS_Special_Functions_Index.html#NewFunctions

Greenplum PostGIS Limitations

The Greenplum PostGIS extension does not support the following features:

- `estimated_extent` functions
- PostGIS long transaction support
- Topology (PostGIS 2.0)
- Raster (PostGIS 2.0)
- Geometry and geography type modifier (PostGIS 2.0)

Installing Greenplum PostGIS Extension

After you install the Greenplum PostGIS extension on Greenplum Database, you can create PostGIS objects in the databases that are used as a a spatial database for geographic information systems (GIS).

For Greenplum Database version 4.2 and higher, PostGIS extension is available as a package. Download from the EMC Download Center and then install it using the Greenplum Package Manager (`gppkg`).

The `gppkg` utility installs Greenplum Database extensions, along with any dependencies, on all hosts across a cluster. It also automatically installs extensions on new hosts in the case of system expansion and segment recovery.

For information about `gppk`, see the *Greenplum Database Utility Guide*.

Installing the Greenplum Database PostGIS Package

Before you install the Greenplum Database PostGIS extension, make sure that your Greenplum database is running, you have sourced `greenplum_path.sh`, and that the `$MASTER_DATA_DIRECTORY` and `$GPHOME` variables are set.

1. Download the Greenplum Database PostGIS extension package from the EMC Download Center then copy it to the master host.
2. Install the software extension package by running the `gppkg` command. For example this command installs the PostGIS 2.0 package on a Linux system:

   ```
   $ gppkg -i postgis-2.0-rhel5-x86_64.gppkg
   ```

Enabling PostGIS Support For a Database

You must enable PostGIS support for each database that requires its use. To enable the support, run enabler SQL scripts that are supplied with the PostGIS package in your target database.

For PosgGIS 1.4 the enabler script `postgis.sql`. This `psql` command runs the script in the database mytestdb.

```
$ psql -f $GPHOME/share/postgresql/contrib/postgis.sql
   -d mytestdb
```

The script creates database objects used by PostGIS.

Your database is now spatially enabled.

For PostGIS 2.0.3, run two SQL scripts `postgis.sql` and `spatial_ref_sys.sql` in your target database. For example, these `psql` commands run the script in the database mytestdb

```
$ psql -d mytestdb
  -f $GPHOME/share/postgresql/contrib/postgis-2.0/postgis.sql
```

```
$ psql -d mytestdb
  -f $GPHOME/share/postgresql/contrib/postgis-2.0/
  spatial_ref_sys.sql
```

Note: `spatial_ref_sys.sql` populates the `spatial_ref_sys` table with EPSG coordinate system definition identifiers. If you have overridden standard entries and want to use those overrides, do not load the `spatial_ref_sys.sql` file when creating the new database.

Your database is now spatially enabled.

To verify that postGIS is enabled for the database, you can run the PostGIS function `postgis_full_version()`. This psql command retrieves the postGIS version from the database mytestdb:

```
$ psql -d mytestdb -c 'SELECT postgis_full_version();'
```

For a complete set of EPSG coordinate system definition identifiers, you can also load the `spatial_ref_sys.sql` definitions file and populate the `spatial_ref_sys` table. This will permit you to perform `ST_Transform()` operations on geometries.

Upgrading a Greenplum Database to PostGIS 2.0

To migrate a PostGIS-enabled database from 1.4 to 2.0 you must perform a PostGIS hard upgrade. A hard upgrade consists of dumping a database that is enabled with PostGIS 1.4 and loading the database the data to a new database that is enabled with PostGIS 2.0.

For information about a PostGIS hard upgrade procedure, see the PostGIS documentation:
http://postgis.net/docs/manual-2.0/postgis_installation.html#hard_upgrade

Uninstalling PostGIS

- Remove PostGIS Support From a Database
- Uninstall the Software Package

When you remove PostGIS support from a database, routines that you created in the database that use PostGIS object or functionality will no longer work.

Remove PostGIS Support From a Database

The SQL script uninstall_postgis.sql removes PostGIS objects from the database. This example command removes the PostGIS 2.0 objects from the database mytestdb.

```
$ psql -d mytestdb
  -f $GPHOME/share/postgresql/contrib/postgis-2.0/
  uninstall_postgis.sql
```

Uninstall the Software Package

Use the Greenplum `gppkg` utility with the `-r` option to uninstall the PostGIS package. When removing the package you must specify the package and version. This example uninstalls PostGIS Package Version 2.0

```
$ gppkg -r postgis-2.0
```

The `gppkg` option `-q --all` lists the installed packages and their versions.

After you uninstall the extension, restart the database.

```
$ gpstop -r
```

Examples

The following example SQL commands create a non-OpenGIS table and geometries.

```
CREATE TABLE geom_test ( gid int4, geom geometry,
  name varchar(25) );
INSERT INTO geom_test ( gid, geom, name )
  VALUES ( 1, 'POLYGON((0 0 0,0 5 0,5 5 0,5 0 0,0 0 0))',
  '3D Square');
INSERT INTO geom_test ( gid, geom, name )
  VALUES ( 2, 'LINESTRING(1 1 1,5 5 5,7 7 5)', '3D Line' );
INSERT INTO geom_test ( gid, geom, name )
  VALUES ( 3, 'MULTIPOINT(3 4,8 9)', '2D Aggregate Point' );
```

The following `SELECT` command selects the objects from the table that overlap with a three dimensional box with the lower-left bottom corner at (2,2,0) and the upper-right top corner at (3,3,0).

```
SELECT * from geom_test WHERE geom &&
  Box3D(ST_GeomFromEWKT('LINESTRING(2 2 0, 3 3 0)'));
```

The following example SQL commands create a table and add data. The `SELECT` command adds a geometry column to the table with a SRID integer value that references an entry in the `SPATIAL_REF_SYS` table. The `INSERT` commands add to geopoints to the table.

```
CREATE TABLE geotest2 (id INT4, name VARCHAR(32) );
SELECT AddGeometryColumn('geotest','geopoint',
  4326,'POINT',2);
INSERT INTO geotest2 (id, name, geopoint)
  VALUES (1, 'Olympia',
  ST_GeometryFromText('POINT(-122.90 46.97)', 4326));
INSERT INTO geotest2 (id, name, geopoint)|
  VALUES (2, 'Renton',
  ST_GeometryFromText('POINT(-122.22 47.50)', 4326));

SELECT name,ST_AsText(geopoint) FROM geotest2;
```

========NEED TO UPDATE ? =======

The following example SQL commands creates an OpenGIS entry in the postGIS `SPATIAL_REF_SYS` table and a geometry column in the `GEOTEST` table. The entry in `SPATIAL_REF_SYS` table contains the Spatial Reference System Identifier (SRID). A SRID value identifies a spatial coordinate system definition. When you add a geometry column with `AddGeometryColumn()`, the information you add to the geotest table be valid spatial data and must be associated with a SRID in the `SPATIAL_REF_SYS` table.

```
INSERT INTO SPATIAL_REF_SYS
  ( SRID, AUTH_NAME, AUTH_SRID, SRTEXT ) VALUES
  ( 1, 'EPSG', 4269,
    'GEOGCS["NAD83",
      DATUM[
        "North_American_Datum_1983",
        SPHEROID[
        "GRS 1980",
        6378137,
        298.257222101
      ]
    ],
    PRIMEM["Greenwich",0],
    UNIT["degree",0.0174532925199433]]'
);
```

Create the table GEOTEST3 with two columns, ID and NAME.

```
CREATE TABLE geotest3 (
    id INT4,
    name VARCHAR(32)
    );
```

Add a geometry column GEOPOINT, that contains geometric POINT information.

```
SELECT
AddGeometryColumn('public','geotest','geopoint',1,'POINT',2)
;
```

Add two points to the table.

```
INSERT INTO geotest3 (id, name, geopoint)
    VALUES (1, 'Olympia', GeometryFromText('POINT(-122.90
46.97)',1));
INSERT INTO geotest3 (id, name, geopoint)
    VALUES (2, 'Renton', GeometryFromText('POINT(-122.22
47.50)',1));
```

Get the point information from the table:

```
SELECT name,AsText(geopoint) FROM geotest3;
```

Spatial Indexes

PostgreSQL provides support for GiST spatial indexing. Spatial indexing creates indexes the bounding boxes of the geometric features being indexed. The GiST scheme offers indexing even on large objects. It uses a system of lossy indexing in which smaller objects act as proxies for larger ones in the index. In the PostGIS indexing system, all objects use their bounding boxes as proxies in the index.

Building a Spatial Index

You can build a GiST index as follows:

```
CREATE INDEX indexname
ON tablename
USING GIST ( geometryfield );
```

References

The PostGIS web site is at http://postgis.refractions.net/.

The documentation for PostGIS Version 2.0 s at http://postgis.net/docs/manual-2.0/.

The documentation for PostGIS Version 1.4 is at http://postgis.net/docs/manual-1.4/.

A PostGIS tutorial created by OpenGeo is at http://workshops.opengeo.org/postgis-intro.

Note: Some exercises in the tutorial use features that are not available with the Greenplum Database PostGIS extenstion. See “Greenplum PostGIS Limitations” for a list of limitations.

15. pgcrypto Extension

With the Greenplum Database pgcrypto extension, you can use pgcrypto functions to store columns of data in encrypted form. The encryption adds an extra layer of protection for sensitive data, as data stored in Greenplum Database in encrypted form cannot be read by users who do not have the encryption key, nor be read directly from the disks.

It is important to note that the pgcrypto functions run inside database server. That means that all the data and passwords move between pgcrypto and the client application in clear-text. For optimal security, consider also using SSL connections between the client and the Greenplum master server.

pgcrypto has various levels of encryption ranging from basic to advanced built-in functions.

This chapter includes the following information:

About Encryption

When encrypting data generally speaking, the harder you make it to keep people out of your data, the easier it is for protect sensitive data and make it more secure.

Not only does encryption makes it difficult to read data, it also comes with a cost of consuming resources to encrypt and decrypt.

Keeping the above basics in mind, it's important to pick your encryption strategies based on the sensitivity of the data and performance needs.

Installing pgcrypto Functions

For Greenplum Database version 4.2 and later, pgcrypto is available as an extension package. Download the package from the EMC Download Center and then install it with the Greenplum Package Manager (`gppkg`).

The `gppkg` utility installs Greenplum Database extensions, along with any dependencies, on all hosts across a cluster. It also automatically installs extensions on new hosts in the case of system expansion and segment recovery.

For information about `gppkg`, see the *Greenplum Database Utility Guide*.

Installing the pgcrypto Extension Package

Before you install the pgcrypto extension, make sure that your Greenplum database is running, you have sourced `greenplum_path.sh`, and that the `$MASTER_DATA_DIRECTORY` and `$GPHOME` variables are set.

1. Download the PostGIS extension package from the EMC Download Center then copy it to the master host.
2. Install the software extension package by running the `gppkg` command. This example installs the pgcrypto extension on a Linux system:

   ```
   $ gppkg -i pgcrypto-1.0-rhel5-x86_64.gppkg
   ```

Enabling pgcrypto Support For a Database

You must enable pgcrypto support for each database that requires its use. To enable the support, run the SQL script `pgcrypto.sql`. The script pgcrypto.sql contains all the pgcrypto functions.

For example, this `psql` command runs the script in the database mytestdb.

```
$ psql -d mytestdb
   -f $GPHOME/share/postgresql/contrib/pgcrypto.sql
```

Uninstalling pgcrypto

- Remove pgcrypto Database Objects
- Uninstall the Software Package

When you remove pgcrypto support from a database, routines that you created in the database that use pgrcypto functions will no longer work.

Remove pgcrypto Database Objects

For a database that no long requires the pgcrypto support, remove the pgcrypto functions with the SQL script `uninstall_pgcrypto.sql`. For example, this `psql` command runs the script in the database mytestdb.

```
$ psql -d mytestdb
  -f $GPHOME/share/postgresql/contrib/uninstall_pgcrypto.sql
```

Uninstall the Software Package

If no databases have pgcrypto support installed, uninstall the Greenplum pgcrypto extension with the `gppkg` utility. Use the Greenplum `gppkg` utility with the `-r` option to uninstall the pgcrypto package. This example uninstalls pgcrypto Package Version 1.0

```
$ gppkg -r pgcrypto-1.0
```

The `gppkg` option `-q --all` lists the installed packages and their versions.

After you uninstall the package, restart the database.

```
$ gpstop -r
```

pgcrypto Functions

See also
http://www.postgresql.org/docs/8.4/static/pgcrypto.html

General hashing functions

digest()

```
digest(data text, type text) returns bytea
digest(data bytea, type text) returns bytea
```

Computes a binary hash of the given data. The `type` is the algorithm to use. Standard algorithms are md5, sha1, sha224, sha256, sha384 and sha512. If pgcrypto was built with OpenSSL, more algorithms are available as detailed in this table.

Table 15.1 Summary of functionality with and without OpenSSL

Functionality	Built-in	With OpenSSL
MD5	yes	yes
SHA1	yes	yes
SHA224/256/384/512	yes	yes (Note 1)
Other digest algorithms	no	yes (Note 2)
Blowfish	yes	yes
AES	yes	yes (Note 3)
DES/3DES/CAST5	no	yes
Raw encryption	yes	yes
PGP Symmetric encryption	yes	yes
PGP Public-Key encryption	yes	yes

NOTES:

1. SHA2 algorithms were added to OpenSSL in pgcrypto version 0.9.8. For previous versions, pgcrypto will use built-in code.
2. Any digest algorithm OpenSSL supports is automatically picked up. This is not possible with ciphers, which need to be supported explicitly.
3. AES is included in OpenSSL since pgcrypto version 0.9.7. For previous versions, pgcrypto will use built-in code.

hmac()

```
hmac(data text, key text, type text) returns bytea
hmac(data bytea, key text, type text) returns bytea
```

Calculates hashed MAC for data with `key` key. The `type` is the same as in `digest()`.

This is similar to `digest()` but the hash can only be recalculated knowing the key. This prevents the scenario of someone altering data and also changing the hash to match.

If the key is larger than the hash block size it will first be hashed and the result will be used as key.

Password hashing functions

The functions `crypt()` and `gen_salt()` are specifically designed for hashing passwords. `crypt()` does the hashing and `gen_salt()` prepares algorithm parameters for it.

`crypt()`

```
crypt(password text, salt text) returns text
```

Calculates a crypt(3)-style hash of password. When storing a new password, you must use `gen_salt()` to generate a new salt value. To check a password, pass the stored hash value as salt, and test whether the result matches the stored value.

Table 15.2 Supported algorithms for crypt()

Algorithm	Max password length	Adaptive?	Salt bits	Description
bf	72	yes	128	Blowfish-based, variant 2a
md5	unlimited	no	48	MD5-based crypt
xdes	8	yes	24	Extended DES
des	8	no	12	Original UNIX crypt

`gen_salt()`

```
gen_salt(type text [, iter_count integer ]) returns text
```

Generates a new random salt string for use in `crypt()`. The salt string also tells `crypt()` which algorithm to use.

The type parameter specifies the hashing algorithm. The accepted types are: des, xdes, md5, and bf.

The `iter_count` parameter lets the user specify the iteration count for algorithms that have one. The higher the count, the more time it takes to hash the password and therefore the more time to break it. Although, with a very high a count, the time to calculate a hash may be several years. If the `iter_count` parameter is omitted, the default iteration count is used. Allowed values for `iter_count` depend on the algorithm and are shown in this table:

Table 15.3 Iteration counts for crypt()

Algorithm	Default	Min	Max
xdes	725	1	16777215
bf	6	4	31

For `xdes` there is an additional limitation that the iteration count must be an odd number.

PGP encryption functions

The functions in this section implement the encryption part of the OpenPGP (RFC 4880) standard. Supported are both symmetric-key and public-key encryption. Some of the functions support options. The description of the option parameters and values are described in "Options for PGP functions" on page 124.

pgp_sym_encrypt()

```
pgp_sym_encrypt(data text, psw text [, options text ])
  returns bytea
pgp_sym_encrypt_bytea(data bytea, psw text [, options text ])
  returns bytea
```

Encrypt `data` with a symmetric PGP key `psw`.

The options parameter can contain the following option settings:

`compress-algo`
`compress-level`
`convert-crlf`
`disable-mdc`
`enable-session-key`
`s2k-mode`
`s2k-digest-algo`
`s2k-cipher-algo`
`unicode-mode`

pgp_sym_decrypt()

```
pgp_sym_decrypt(msg bytea, psw text [ options text ])
  returns text
pgp_sym_decrypt_bytea(msg bytea, psw text [ options text ])
   returns bytea
```

Decrypt a symmetric-key-encrypted PGP message.

Decrypting bytea data with `pgp_sym_decrypt()` is disallowed. This is to avoid the output of invalid character data. Decrypting originally textual data with `pgp_sym_decrypt_bytea()` is allowed.

The options parameter can contain the `convert-crlf` option

pgp_pub_encrypt()

```
pgp_pub_encrypt(data text, key bytea [, options text ])
  returns bytea
pgp_pub_encrypt_bytea(data bytea, key bytea [, options text ])
  returns bytea
```

Encrypt `data` with a public PGP key `key`. Giving this function a secret key will produce a error.

The options parameter can contain the following option settings:

`cipher-algo`
`compress-algo`

```
compress-level
convert-crlf
disable-mdc
unicode-mode
```

pgp_pub_decrypt()

```
pgp_pub_decrypt(msg bytea, key bytea [, psw text
  [ options text ]]) returns text
pgp_pub_decrypt_bytea(msg bytea, key bytea [, psw text
  [ options text ]]) returns bytea
```

Decrypt a public-key-encrypted message. The `key` must be the secret key corresponding to the public key that was used to encrypt. If the secret key is password protected, you must specify the password in `psw`. If there is no password, but you want to specify options, specify an empty password.

Decrypting bytea data with `pgp_pub_decrypt()` is disallowed. This avoids the output of invalid character data. Decrypting originally textual data with `pgp_pub_decrypt_bytea()` is allowed.

The options parameter can contain the `convert-crlf` option

pgp_key_id()

```
pgp_key_id(bytea) returns text
```

`pgp_key_id()` extracts the key ID of a PGP public or secret key. Or it gives the key ID that was used for encrypting the data, if given an encrypted message. It can return 2 special key IDs:

- SYMKEY
  The message is encrypted with a symmetric key.
- ANYKEY
  The message is public-key encrypted, but the key ID has been removed.
  With the key ID removed, you will need to try all your secret keys on it to see which one decrypts it. pgcrypto does not produce such messages.

Different keys might have the same ID. This is rare but a normal event. The client application should then try to decrypt with each one, to see which fits. This is similar to ANYKEY.

armor(), dearmor()

```
armor(data bytea) returns text
dearmor(data text) returns bytea
```

These functions wrap and unwrap binary data into PGP Ascii Armor format, which is basically Base64 with CRC and additional formatting.

Options for PGP functions

Options are named to be similar to GnuPG. An option value should be given after an equal sign; separate options from each other with commas. For example:

```
pgp_sym_encrypt(data, psw, 'compress-algo=1,
   cipher-algo=aes256')
```

All of the options except `convert-crlf` apply only to encrypt functions. Decrypt functions get the parameters from the PGP data.

cipher-algo

Which cipher algorithm to use.

Values: bf, aes128 (the default), aes192, aes256 (OpenSSL-only: 3des, cast5)

compress-algo

Which compression algorithm to use. Only available if PostgreSQL was built with zlib. Values:

0 - no compression (the default)

1 - ZIP compression

2 - ZLIB compression (ZIP plus meta-data and block CRCs)

compress-level

How much to compress. Higher levels compress smaller but are slower. 0 disables compression.

Values: 0, 1-9.

Default: 6

convert-crlf

Whether to convert \n into \r\n when encrypting and \r\n to \n when decrypting. RFC 4880 specifies that text data should be stored using \r\n line-feeds. Use this to get fully RFC-compliant behavior.

Values: 0 (the default), 1

disable-mdc

Do not protect data with SHA-1. Use this option is to achieve compatibility with ancient PGP products, predating the addition of SHA-1 protected packets to RFC 4880.

Values: 0 (the default), 1

enable-session-key

Use separate session key. Public-key encryption always uses a separate session key; this is for symmetric-key encryption, which by default uses the S2K key directly.

Values: 0 (the default) , 1

s2k-mode

Which S2K algorithm to use. Values:

0 - Without salt. Not recommended.

1 - With salt but with fixed iteration count.

3 - Variable iteration count (the default).

s2k-digest-algo

Which digest algorithm to use in S2K calculation.

Values: md5, sha1 (the default).

s2k-cipher-algo

Which cipher to use for encrypting a separate session key.

Values: bf, aes, aes128, aes192, aes256

Default: use cipher-algo

unicode-mode

Whether to convert textual data from database internal encoding to UTF-8 and back. If your database already is UTF-8, no conversion is done, but the message will be tagged as UTF-8. Without this option, it is not tagged.

Values: 0, 1

Default: 0

Examples

The following examples use pgcrypto encryption.

One-way Encryption

For one-way encryption, the `crypt` function packaged with pgcrypto provides an added level of security beyond MD5 encryption.

When using MD5 encryption, you can see who has the same password because there is no salt so all people with the same password will have the same encoded MD5 string.

With `crypt`, they will be different.

To demonstrate this, we will create a table with two users who have chosen the same password.

```
CREATE TABLE testusers(username varchar(100) PRIMARY KEY,
cryptpwd text, md5pwd bytea);
INSERT INTO testusers(username, cryptpwd, md5pwd)
   VALUES ('robby', crypt('test', gen_salt('md5')),
   digest('test', 'md5')),
   ('artoo', crypt('test',gen_salt('md5')), digest('test',
   'md5'));


SELECT username, cryptpwd, md5pwd FROM testusers;
```

This returns:

```
username | cryptpwd | md5pwd
---------+----------------------------------+---------
```

```
robby | $1$BCLDXk/7$zOAlROGeaQ3Cvwu9uxJk1/ |
\011\217k\315F!\323s\312\336N\203&'\264\366
artoo | $1$wtJcn2Gf$D/zCtsebGf36YsMkqLptU0 |
\011\217k\315F!\323s\312\336N\203&'\264\366
```

Note that both users have chosen the same password `test`. The md5 version is the same for both, but the crypted password is different although they are the same password.

When any log in attempted, we do this test.

```
--successful login
SELECT username FROM testusers WHERE username = 'robby' AND
   cryptpwd = crypt('test', cryptpwd);
--successful login
SELECT username FROM testusers WHERE username = 'artoo' AND
   cryptpwd = crypt('test', cryptpwd);
--unsuccessful login
SELECT username FROM testusers WHERE username = 'artoo' AND
   cryptpwd = crypt('artoo', cryptpwd);
-- using md5 SELECT username FROM testusers WHERE
    username = 'robby' and md5pwd = md5('test');
```

PGP Encryption

```
CREATE TABLE testuserscards(card_id SERIAL PRIMARY KEY,
username varchar(100), cc bytea);
```

To encrypt the data:

```
INSERT INTO testuserscards(username, cc)
SELECT robotccs.username, pgp_pub_encrypt(robotccs.cc,
keys.pubkey) As cc
FROM (VALUES ('robby', '4111111111111111'),
    ('artoo', '4111111111111112') ) As robotccs(username,
cc)
     CROSS JOIN (SELECT dearmor(' -----BEGIN PGP PUBLIC KEY
BLOCK-----
Version: GnuPG v2.0.17 (GNU/Linux)

mQGiBE7AfUoRBACpupjE5tG9Fh1dWe2kb/yX+lNlMLpwMj1hjTrJo1cYmSYi
xkGX
Si90ZIjn0IOSU7XOkFai8btpbFGyGSdaB9BQK7s8ItN/wx9IHcnB83Lbex3a
F/VS
hN81VummzKQ0YB+Crwp1mu1l76UrTg6sPnY+wHj3jPOleXcX9L9UAAzOnwCg
i4OS
JoRzR/pPiWtW0Nk5qnYhuZMD/RyNYbKkoNVO4WUnfOFMqm2zIqRXmMnkXS6g
NPsd
RNVXb4ByFSzugsZKW5ez9+zS0G0aarySQIuGgPGKSeZezYtwKR3DH676Mmdn
NSvx
```

```
GiGDQW+hSXBOiBOmxhZfBK8H6JfmEtUpZwA8tkzD0u6ikZjQZR0cRux/tdut
zTuZ
YGyaA/4tWzKtQP+WDi5tUPNO1/7EcBphYMvZDfNzYUn5ZwXzw5B5YSi0rdY6
ZLSP
H3X8hrHbSmDrD8KseLtl9E4YvaOWd0BZCg9QwUcVrR+9sYtyNy/ztX++vVOt
FjQ6
b19rj0853fwSgv9gHoNelmBXs0jTDGaKSBwzTD8GYtusQcu3lbQZYWFhYWEg
PGFh
YWFhQGV4YW1wbGUuY29tPohiBBMRAgAiBQJOwH1KAhsDBgsJCAcDAgYVCAIJ
CgsE
FgIDAQIeAQIXgAAKCRCNcpg0BUjyDOKPAJ4viutaojyBhV0ICJED09ArUXgZ
7ACf
U6CX156L6i6x8UzRLFxsvVKHXIK5AQ0ETsB9ShAEAMDqwXmBeJGqWgXrtVKh
6XIw
uanQtl/lIhktVcAYa/FHnvleL9RqI6JpiVWuvLfOdDcUQmh3MvsmD6h6plVm
g/bz
/y1ZGnWANjCazmSWDjTfuIX+wuWo4TKSRhXzUd5tw5bgaeC0Hvy+rlgswRIL
FYL1
5I0/NTm+fFkB0McY9E2fAAMHBACgpmaAW/VR4IGn+j74GCzn2W06UnnWSK7A
0GPJ
kUiJa37mv04yCeIqmoTVkl5rnz8dZZUwJVKYwlRvvLB/omIdzRkouhK/QWio
RQ+M
B5qPXjRNrcUnruWVzC3XfhZ6sImI8bh2tHpN1/r0hHXFb/5078Bv2d4Cq2Wd
MZJo
oGDxBIhJBBgRAgAJBQJOwH1KAhsMAAoJEI1ymDQFSPIM7RcAn22lbnWNWiGb
y9SU
mEQSkrE34O8+AKCFFPLQiCs3/EL3+2DsplWOnEcSuQ==
=Q6Oq
-----END PGP PUBLIC KEY BLOCK-----') As pubkey) As keys;
```

Now if we select our data

```
SELECT username, cc
FROM testuserscards;
```

We will see a whole bunch of encrypted stuff in the cc column that is way too hard to print on this page.

Now we can use pgp_keyid to verify which public key we used to encrypt our data.

```
SELECT pgp_key_id(dearmor('-----BEGIN PGP PUBLIC KEY
BLOCK-----
super publickey goobly gook goes here
-----END PGP PUBLIC KEY BLOCK-----'));
```

-- gives you something like

-- where last set of characters is the public key id you got in gpg

E0B086C2999DEFG

--verify our data was encrypted with the above public key

```
SELECT username, pgp_key_id(cc) As keyweused
FROM testuserscards;
```

Returns

```
 username |    keyweused
----------+------------------
 robby    | E0B086C2999DEFG
 artoo    | E0B086C2999DEFG
```

To decrypt the data we pull from our chest of private keys matching the public key we used to encrypt with.

To decrypt the data

```
SELECT username, pgp_pub_decrypt(cc, keys.privkey) As
ccdecrypt
FROM testuserscards
    CROSS JOIN
    (SELECT dearmor('-----BEGIN PGP PRIVATE KEY BLOCK-----
super private key gobbly gook goes here
-----END PGP PRIVATE KEY BLOCK-----') As privkey) As keys;
```

-- We get --

```
 username |  ccdecrypt
----------+------------------
 robby    | 41111111111111111
 artoo    | 41111111111111112
```

Raw Encryption

Example?

References

pgcrypto Modules

The following table lists pgcrypto modules

Table 16 pgcrypto Modules

Type	Module	Purpose
Function	`text2bytea.sql`	Converts text to bytea. Called from `decrypt_sqlserver` and `encrypt_sqlserver` functions
Function	`bytea2text.sql`	Converts bytea to text. Called from `decrypt_sqlserver` and `encrypt_sqlserver` functions

Table 16 pgcrypto Modules

Type	Module	Purpose
Function	cr_func_decrypt_sqlserver.sql	Function that decrypts encrypted data along with key. Called from view statement / script
Function	cr_func_encrypt_sqlserver.sql	Function that encrypts data when decrypted data is passed along with the key. Called from Load statement or Transform job
Table	cr_tbl_decryptextendedrepresentative.sql	External web table script that calls the remote SQLSERVER encryption package
Table	cr_tbl_encryptextendedrepresentative.sql	Create table script to store encrypted data in Greenplum
Load/Transform Job	load_encryptextendedrepresentative.sql	Load job / Transform Job to load data from external web table to Greenplum encrypt table. Calls external web table when OS Tranform job is executed.
View	cr_vw_extendedrepresentative.sql	View to show decrypted data from encryptedextendedrepresentative. Permissions will be only provided to DBA / Authorized user
Table	manage_keys.sql	Access privileges to keys object will be only provided to gpadmin user / GPDBA

The following table lists the implementation details of the pgcrypto modules.

Table 17 Implementation Details

Module	Code
text2bytea.sql	CREATE OR REPLACE FUNCTION ext.text2bytea(text) RETURNS bytea AS $_$ BEGIN return $1; END $_$ LANGUAGE plpgsql;

bytea2text.sql	CREATE OR REPLACE FUNCTION
	ext.bytea2text(bytea) RETURNS text AS $_$
	BEGIN
	return $1;
	END;
	$_$
	LANGUAGE plpgsql;

Table 17 Implementation Details

Module	Code

cr_func_decrypt_sqlserver.sql	CREATE OR REPLACE FUNCTION decrypt_sqlserver(p_data bytea, p_key varchar) RETURNS varchar
	as $$
	BEGIN
	return bytea2text(decrypt(p_data,decode(p_key,'escape'),'3des'::text));
	END;
	$$
	LANGUAGE plpgsql security definer;

cr_func_encrypt_sqlserver.sql	CREATE OR REPLACE FUNCTION encrypt_sqlserver(p_data varchar, p_key varchar) returns bytea
	as $$
	BEGIN
	return encrypt(text2bytea(p_data), text2bytea(p_key),'3des'::text);
	END;
	$$
	LANGUAGE plpgsql security definer;

cr_tbl_decryptextendedrepresentative.sql	DROP EXTERNAL TABLE orders.decryptextendedrepresentative; CREATE EXTERNAL WEB TABLE orders.decryptextendedrepresentative (extendedrepresentativeid integer, representativeid integer, dateofbirth timestamp, residentstatusid integer, decryptedfield1 varchar(11)) EXECUTE E'java -classpath /data/os_2_1/Outsourcer.jar:/data/os_2_1/sqljdbc_3.0/enu/sqljdbc4.jar:/data/os_2_1/ojdbc6.jar SQLServer 10.10.240.22 INT emc p1ttsburgh "use orders;select extendedrepresentativeid, representativeid, dateofbirth, residentstatusid, convert(varchar(11), decryptbykeyautocert(cert_id(\'OrdersECert\'), NULL, EncryptedField1)) as decryptedfield1 from orders.dbo.EncryptExtendedRepresentative"' ON MASTER FORMAT 'text' (delimiter '|' null 'null' escape E'\\') ENCODING 'UTF8';

Table 17 Implementation Details

Module	Code
cr_tbl_encryptextendedrepresentative.sql	DROP TABLE orders.encryptextendedrepresentative; create table orders.encryptextendedrepresentative (extendedrepresentativeid integer, representativeid integer, dateofbirth timestamp, residentstatusid integer, encryptedfield bytea) distributed by (extendedrepresentativeid);
load_encryptextendedrepresentative.sql	INSERT INTO os.job (refresh_type, sql_text) values ('transform','insert into orders.encryptextendedrepresentative (extendedrepresentativeid, representativeid, dateofbirth, residentstatusid, encryptedfield) select d.extendedrepresentativeid, d.representativeid, d.dateofbirth, d.residentstatusid, encrypt_sqlserver(d.decryptedfield1, k.value) from orders.decryptextendedrepresentative d cross join keys k where k.name = \'sqlserver\'');
cr_vw_extendedrepresentative.sql	CREATE VIEW orders.vw_extendedrepresentative as select d.extendedrepresentativeid, d.representativeid, d.dateofbirth, d.residentstatusid, decrypt_sqlserver(d.encryptedfield, k.value) from orders.encryptextendedrepresentative d cross join keys k where k.name = 'sqlserver';
manage_keys.sql	CREATE TABLE KEYS (name varchar, value varchar) distributed by (name); insert into keys values ('sqlserver', 'gr33np1um');

Useful Reading

- The GNU Privacy Handbook: http://www.gnupg.org/gph/en/manual.html
- Description of the crypt-blowfish algorithm: http://www.openwall.com/crypt/
- How to chose a good password: http://www.stack.nl/~galactus/remailers/passphrase-faq.html
- More information about picking passwords: http://world.std.com/~reinhold/diceware.html
- Description of good and bad cryptography: http://www.interhack.net/people/cmcurtin/snake-oil-faq.htm

Technical References

- OpenPGP message format: http://www.ietf.org/rfc/rfc2440.txt

- New version of RFC2440: http://www.imc.org/draft-ietf-openpgp-rfc2440bis
- The MD5 Message-Digest Algorithm: http://www.ietf.org/rfc/rfc1321.txt
- HMAC: Keyed-Hashing for Message Authentication:http://www.ietf.org/rfc/rfc2104.txt
- Comparison of crypt-des, crypt-md5 and bcrypt algorithms: http://www.usenix.org/events/usenix99/provos.html
- Standards for DES, 3DES and AES:
- http://csrc.nist.gov/cryptval/des.htm
- Description of Fortuna CSPRNG: http://en.wikipedia.org/wiki/Fortuna_(PRNG)
- Jean-Luc Cooke Fortuna-based /dev/random driver for Linux: http://jlcooke.ca/random/
- Collection of cryptology pointers: http://www.cs.ut.ee/~helger/crypto/

16. Oracle Compatibility Functions

The Oracle Compatibility SQL functions extend Greenplum Database functionality in a production environment. These functions target PostgreSQL.

This chapter includes the following information:

Installing Oracle Compatibility Functions

Before using any Oracle Compatibility Functions, run the installation script `$GPHOME/share/postgresql/contrib/orafunc.sql` once for each database. For example, to install the functions in database `testdb`, use the command

```
$ psql -d testdb -f \
$GPHOME/share/postgresql/contrib/orafunc.sql
```

To uninstall Oracle Compatibility Functions, run the `uinstall_orafunc.sql` script:

`$GPHOME/share/postgresql/contrib/uninstall_orafunc.sql`.

The following functions are available by default and do not require running the Oracle Compatibility installer:

- sinh
- tanh
- cosh
- decode

Note: The Oracle Compatibility Functions reside in the `oracompat` schema. To access them, prefix the schema name (`oracompat`) or alter the database search path to include the schema name. For example:

```
ALTER DATABASE db_name SET search_path = $user, public,
oracompat;
```

If you alter the database search path, you must restart the database.

Oracle and Greenplum Implementation Differences

There are some differences in the implementation of these compatibility functions in the Greenplum Database from the Oracle implementation. If you use validation scripts, the output may not be exactly the same as in Oracle. Some of the differences are as follows:

- Oracle performs a decimal round off, Greenplum Database does not. 2.00 becomes 2 in Oracle and remains 2.00 in Greenplum Database.

- The provided Oracle Compatibility functions handle implicit type conversions differently. For example, using the `decode` function

  ```
  decode(expression, value, return [,value, return]...
              [, default])
  ```

  Oracle automatically converts `expression` and each `value` to the datatype of the first `value` before comparing. Oracle automatically converts `return` to the same datatype as the first result.

  The Greenplum implementation restricts `return` and `default` to be of the same data type. The `expression` and `value` can be different types if the data type of `value` can be converted into the data type of the `expression`. This is done implicitly. Otherwise, `decode` fails with an `invalid input syntax` error. For example:

  ```
  SELECT decode('M',true,false);
  CASE
  ------
   f
  (1 row)

  SELECT decode(1,'M',true,false);
  ERROR: Invalid input syntax for integer:"M"
  LINE 1: SELECT decode(1,'M',true,false);
  ```

- Numbers in `bigint` format are displayed in scientific notation in Oracle, but not in Greenplum Database. 9223372036854775 displays as 9.2234E+15 in Oracle and remains 9223372036854775 in Greenplum Database.
- The default date and timestamp format in Oracle is different than the default format in Greenplum Database. If the following code is executed

  ```
  CREATE TABLE TEST(date1 date, time1 timestamp, time2
                    timestamp with timezone);
  INSERT INTO TEST VALUES ('2001-11-11','2001-12-13
                   01:51:15','2001-12-13 01:51:15 -08:00');
  SELECT DECODE(date1, '2001-11-11', '2001-01-01') FROM TEST;
  ```

  Greenplum Database returns the row, but Oracle does not return any rows.

  Note: The correct syntax in Oracle is

  ```
  SELECT DECODE(to_char(date1, 'YYYY-MM-DD'), '2001-11-11',
                '2001-01-01') FROM TEST
  ```

  which returns the row.

Oracle Compatibility Functions Reference

The following are the Oracle Compatibility Functions.

- add_months
- bitand
- concat
- cosh
- decode
- dump
- instr
- last_day
- listagg
- listagg (2)
- lnnvl
- months_between
- nanvl
- next_day
- next_day
- nlssort
- nvl
- nvl2
- oracle.substr
- reverse
- round
- sinh
- tanh
- trunc

add_months

Oracle-compliant function to add a given number of months to a given date.

Synopsis

```
add_months(date_expression, months_to_add)
```

Description

This Oracle-compatible function adds `months_to_add` to a *`date_expression`* and returns a `DATE`.

If the *`date_expression`* specifies the last day of the month, or if the resulting month has fewer days than the *`date_expression`*, then the returned value is the last day of the resulting month. Otherwise, the returned value has the same day of the month as the *`date_expression`*.

Parameters

`date_expression`

The starting date. This can be any expression that can be implicitly converted to `DATE`.

`months_to_add`

The number of months to add to the *`date_expression`*. This is an integer or any value that can be implicitly converted to an integer. This parameter can be positive or negative.

Example

```
SELECT name, phone, nextcalldate FROM clientdb
WHERE nextcalldate >= add_months(CURRENT_DATE,6);
```

Returns `name`, `phone`, and `nextcalldate` for all records where `nextcalldate` is at least six months in the future.

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

bitand

Oracle-compliant function that computes a logical `AND` operation on the bits of two non-negative values.

Synopsis

```
bitand(expr1, expr2)
```

Description

This Oracle-compatible function returns an integer representing an AND operation on the bits of two non-negative values (`expr1` and `expr2`). 1 is returned when the values are the same. 0 is returned when the values are different. Only significant bits are compared. For example, an `AND` operation on the integers 5 (binary 101) and 1 (binary 001 or 1) compares only the rightmost bit, and results in a value of 1 (binary 1).

The types of `expr1` and `expr2` are `NUMBER`, and the result is of type `NUMBER`. If either argument is `NULL`, the result is `NULL`.

The arguments must be in the range -(2(n-1)) .. ((2(n-1))-1). If an argument is out of this range, the result is undefined.

Notes:

- The current implementation of BITAND defines n = 128.
- PL/SQL supports an overload of BITAND for which the types of the inputs and of the result are all BINARY_INTEGER and for which n = 32.

Parameters

`expr1`

A non-negative integer expression.

`expr2`

A non-negative integer expression.

Example

```
SELECT bitand(expr1, expr2)
FROM ClientDB;
```

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

concat

Oracle-compliant function to concatenate two strings together.

Synopsis

```
concat (string1, string2)
```

Description

This Oracle-compatible function concatenates two strings (*string1* and *string2*) together.

The string returned is in the same character set as *string1*. Its datatype depends on the datatypes of the arguments.

In concatenations of two different datatypes, the datatype returned is the one that results in a lossless conversion. Therefore, if one of the arguments is a LOB, then the returned value is a LOB. If one of the arguments is a national datatype, then the returned value is a national datatype. For example:

```
concat(CLOB, NCLOB) returns NCLOB
concat(NCLOB, NCHAR) returns NCLOB
concat(NCLOB, CHAR) returns NCLOB
concat(NCHAR, CLOB) returns NCLOB
```

This function is equivalent to the concatenation operator (||).

Parameters

string1/string2

The two strings to concatenate together.

Both *string1* and *string2* can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB.

Example

```
SELECT concat(concat(last_name, '''s job category is '),
      job_id)
FROM employees
```

Returns 'Smith's job category is 4B'

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

cosh

Oracle-compliant function to return the hyperbolic cosine of a given number.

Synopsis

```
cosh(float8)
```

Description

This Oracle-compatible function returns the hyperbolic cosine of the floating 8 input number (*float8)*.

Note: This function is available by default and can be accessed without running the Oracle Compatibility installer.

Parameters

float8

The input number.

Example

```
SELECT cosh(0.2)
FROM ClientDB;
```

Returns `'1.02006675561908''` (hyperbolic cosine of 0.2)

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

decode

Oracle-compliant function to transform a data value to a specified return value. This function is a way to implement a set of `CASE` statements.

Note: `decode` is converted into a reserved word in Greenplum Database. If you want to use the Postgres two-argument `decode` function that decodes binary strings previously encoded to ASCII-only representation, you must invoke it by using the full schema-qualified syntax, `pg_catalog.decode()`, or by enclosing the function name in quotes `"decode" ()`.

Note: Greenplum's implementation of this function transforms `decode` into `case`. This results in the following type of output:

```
gptest=# select decode(a, 1, 'A', 2, 'B', 'C') from
decodetest;
 case
------
 C
 A
 C
 B
 C
(5 rows)
```

This also means that if you deparse your view with `decode`, you will see `case` expression instead.

Greenplum recommends you use the `case` function instead of `decode`.

Synopsis

```
decode(expression, value, return [,value, return]...
       [, default])
```

Description

The Oracle-compatible function decode searches for a value in an expression. If the value is found, the function returns the specified value.

Note: This function is available by default and can be accessed without running the Oracle Compatibility installer.

Parameters

`expression`

The expression to search.

`value`

The value to find in the expression.

return

What to return if expression matches value.

default

What to return if expression does not match any of the values.

Only one `expression` is passed to the function. Multiple `value`/`return` pairs can be passed.

The `default` parameter is optional. If `default` is not specified and if `expression` does not match any of the passed `value` parameters, decode returns `null`. The Greenplum implementation restricts `return` and `default` to be of the same data type. The `expression` and `value` can be different types if the data type of `value` can be converted into the data type of the `expression`. This is done implicitly. Otherwise, `decode` fails with an `invalid input syntax` error.

Examples

In the following code, `decode` searches for a value for `company_id` and returns a specified value for that company. If `company_id` not one of the listed values, the default value `Other` is returned.

```
SELECT decode(company_id, 1, 'EMC',
                          2, 'Greenplum',
                          'Other')
FROM suppliers;
```

The following code using `CASE` statements to produce the same result as the example using `decode`.

```
SELECT CASE company_id
WHEN IS NOT DISTINCT FROM 1 THEN 'EMC'
WHEN IS NOT DISTINCT FROM 2 THEN 'Greenplum'
ELSE 'Other'
END
FROM suppliers;
```

Notes

To assign a range of values to a single return value, either pass an expression for each value in the range, or pass an expression that evaluates identically for all values in the range. For example, if a fiscal year begins on August 1, the quarters are shown in the following table.

Table 16.1 Months and Quarters for Fiscal Year Beginning on August 1

Range (Alpha)	Range (Numeric)	Quarter
August — October	8 — 10	Q1
November — January	11 — 1	Q2

Table 16.1 Months and Quarters for Fiscal Year Beginning on August 1

Range (Alpha)	Range (Numeric)	Quarter
February — April	2 — 4	Q3
May — July	5 — 7	Q4

The table contains a numeric field `curr_month` that holds the numeric value of a month, 1 – 12. There are two ways to use decode to get the quarter.

Method 1 - Include 12 values in the `decode` function

```
SELECT decode(curr_month, 1, 'Q2',
                          2, 'Q3',
                          3, 'Q3',
                          4, 'Q3',
                          5, 'Q4',
                          6, 'Q4',
                          7, 'Q4',
                          8, 'Q1',
                          9, 'Q1',
                         10, 'Q1',
                         11, 'Q2',
                         12, 'Q2')
FROM suppliers;
```

Method 2 - Use an expression that defines a unique value to decode

```
SELECT decode((1+MOD(curr_month+4,12)/3)::int, 1, 'Q1',
                                               2, 'Q2',
                                               3, 'Q3',
                                               4, 'Q4',
FROM suppliers;
```

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

See Also

PostgreSQL decode (not compatible with Oracle)

dump

Oracle-compliant function that returns a text value that includes the datatype code, the length in bytes, and the internal representation of the expression.

Synopsis

```
dump(expression [,integer])
```

Description

This Oracle-compatible function returns a text value that includes the datatype code, the length in bytes, and the internal representation of the expression.

Parameters

expression

Any expression

integer

The number of characters to return

Example

`dump('Tech')` returns `'Typ=96 Len=4: 84,101,99,104'`

`dump ('tech')` returns `'Typ-96 Len=4: 84,101,99,104'`

`dump('Tech', 10)` returns `'Typ=96 Len=4: 84,101,99,104'`

`dump('Tech', 16)` returns `'Typ=96 Len=4: 54,65,63,68'`

`dump('Tech', 1016)` returns `'Typ=96 Len=4 CharacterSet=US7ASCII: 54,65,63,68'`

`dump('Tech', 1017)` returns `'Typ=96 Len=4 CharacterSet=US7ASCII: T,e,c,h'`

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

instr

Oracle-compliant function to return the location of a substring in a string.

Synopsis

```
instr(string, substring, [position[,occurrence]])
```

Description

This Oracle-compatible function searches for a *substring* in a *string*. If found, it returns an integer indicating the position of the *substring* in the *string*, if not found, the function returns 0.

Optionally you can specify that the search starts at a given *position* in the string, and only return the nth *occurrence* of the *substring* in the *string*.

`instr` calculates strings using characters as defined by the input character set.

The value returned is of `NUMBER` datatype.

Parameters

string

The string to search.

substring

The substring to search for in *string*.

Both *string* and *substring* can be any of the datatypes `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`.

position

The position is a nonzero integer in *string* where the search will start. If not specified, this defaults to 1. If this value is negative, the function counts backwards from the end of *string* then searches towards to beginning from the resulting position.

occurrence

Occurrence is an integer indicating which occurrence of the *substring* should be searched for. The value of occurrence must be positive.

Both *position* and *occurrence* must be of datatype `NUMBER`, or any datatype that can be implicitly converted to `NUMBER`, and must resolve to an integer. The default values of both *position* and *occurrence* are 1, meaning that the search begins at the first character of *string* for the first occurrence of *substring*. The return value is relative to the beginning of *string*, regardless of the value of *position*, and is expressed in characters.

Examples

```
SELECT instr('Greenplum', 'e')
FROM ClientDB;
```

Returns 3; the first occurrence of 'e'

```
SELECT instr('Greenplum', 'e',1,2)
FROM ClientDB;
```

Returns 4; the second occurrence of 'e'

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

last_day

Oracle-compliant function to return the last day in a given month.

Synopsis

```
last_day(date_expression)
```

Description

This Oracle-compatible function returns the last day of the month specified by a *date_expression*.

The return type is always `DATE`, regardless of the datatype of *date_expression*.

Parameters

date_expression

The date value used to calculate the last day of the month. This can be any expression that can be implicitly converted to `DATE`.

Example

```
SELECT name, hiredate, last_day(hiredate) "Option Date"
FROM employees;
```

Returns the `name`, `hiredate`, and `last_day` of the month of `hiredate` labeled "`Option Date`."

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

listagg

Oracle-compliant function that aggregates text values into a string.

Note: This function is an overloaded function. There are two Oracle-compliant `listagg` functions, one that takes one argument, the text to be aggregated (see below), and one that takes two arguments, the text to be aggregated and a delimiter (see next page).

Synopsis

```
listagg(text)
```

Description

This Oracle-compatible function aggregates text values into a string.

Parameters

`text`

The text value to be aggregated into a string.

Example

```
SELECT listagg(t) FROM (VALUES('abc'), ('def')) as l(t)
```

Returns: `abcdef`

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

listagg (2)

Oracle-compliant function that aggregates text values into a string, separating each by the separator specified in a second argument.

Note: This function is an overloaded function. There are two Oracle-compliant `listagg` functions, one that takes one argument, the text to be aggregated (see previous page), and one that takes two arguments, the text to be aggregated and a delimiter (see below).

Synopsis

```
listagg(text, separator)
```

Description

This Oracle-compatible function aggregates text values into a string, separating each by the separator specified in a second argument (*separator*).

Parameters

text

The text value to be aggregated into a string.

separator

The separator by which to delimit the text values.

Example

```
SELECT oracompat.listagg(t, '.') FROM (VALUES('abc'),
('def')) as l(t)
```

Returns: `abc.def`

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

lnnvl

Oracle-compliant function that returns `true` if the argument is false or NULL, or `false`.

Synopsis

```
listagg(condition)
```

Description

This Oracle-compatible function takes as an argument a condition and returns `true` if the condition is false or NULL and `false` if the condition is true.

Parameters

`condition`

Any condition that evaluates to `true`, `false`, or `null`.

Example

```
SELECT lnnvl(true)
```

Returns: `false`

```
SELECT lnnvl(NULL)
```

Returns: `true`

```
SELECT lnnvl(false)
```

Returns: `true`

```
SELECT (3=5)
```

Returns: `true`

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

months_between

Oracle-compliant function to evaluate the number of months between two given dates.

Synopsis

```
months_between(date_expression1, date_expression2)
```

Description

This Oracle-compatible function returns the number of months between `date_expression1` and `date_expression2`.

If *`date_expression1`* is later than *`date_expression2`*, then the result is positive.

If *`date_expression1`* is earlier than *`date_expression2`*, then the result is negative.

If *`date_expression1`* and *`date_expression2`* are either the same days of the month or both last days of months, then the result is always an integer. Otherwise the function calculates the fractional portion of the month based on a 31-day month.

Parameters

`date_expression1, date_expression2`

The date values used to calculate the number of months. This can be any expression that can be implicitly converted to `DATE`.

Examples

```
SELECT months_between
    (to_date ('2003/07/01', 'yyyy/mm/dd'),
    to_date ('2003/03/14', 'yyyy/mm/dd'));
```

Returns the number of months between July 1, 2003 and March 14, 2014.

```
SELECT * FROM employees
    where months_between(hire_date, leave_date) <12;
```

Returns the number of months between `hire_date` and `leave_date`.

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

nanvl

Oracle-compliant function to substitute a value for a floating point number when a non-number value is encountered.

Synopsis

```
nanvl(float1, float2)
```

Description

This Oracle-compatible function evaluates a floating point number (*float1)* such as `BINARY_FLOAT` or `BINARY_DOUBLE`. If it is a non-number ('not a number', NaN), the function returns *float2*. This function is most commonly used to convert non-number values into either NULL or 0.

Parameters

float1

The `BINARY_FLOAT` or `BINARY_NUMBER` to evaluate.

float2

The value to return if `float1` is not a number.

float1 and *float2* can be any numeric datatype or any nonnumeric datatype that can be implicitly converted to a numeric datatype. The function determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that datatype, and returns that datatype.

Example

```
SELECT nanvl(binary1, 0)
FROM MyDB;
```

Returns 0 if the `binary1` field contained a non-number value. Otherwise, it would return the `binary1` value.

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

next_day

Oracle-compliant function to return the date of the next specified weekday after a date.

This section describes using this function with a string argument; see the following page for details about using this function with an integer argument.

Note: This function is an overloaded function. There are two Oracle-compliant `next_day` functions, one that takes a date and a day of the week as its arguments (see below), and one that takes a date and an integer as its arguments (see next page).

Synopsis

```
next_day(date_expression, day_of_the_week)
```

Description

This Oracle-compatible function returns the first `day_of_the_week` (Tuesday, Wednesday, etc.) to occur after a *`date_expression`*.

The weekday must be specified in English.

The case of the weekday is irrelevant.

The return type is always `DATE`, regardless of the datatype of *`date_expression`*.

Parameters

`date_expression`

The starting date. This can be any expression that can be implicitly converted to `DATE`.

`day_of_the_week`

A string containing the name of a day, in English; for example ‘Tuesday’.

`Day_of_the_week` is case-insensitive.

Example

```
SELECT name, next_day(hiredate,"MONDAY") "Second Week Start"
FROM employees;
```

Returns the `name` and the `date` of the next Monday after `hiredate` labeled `"Second Week Start."`

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

next_day

Oracle-compliant function to add a given number of days to a date and returns the date of the following day.

Note: This function is an overloaded function. There are two Oracle `next_day` functions, one that takes a date and a day of the week as its arguments (see previous page), and one that takes a date and an integer as its arguments (see below).

Synopsis

```
next_day(date_expression, days_to_add)
```

Description

This Oracle-compatible function adds the number of *`days_to_add`* to a *`date_expression`* and returns the date of the day after the result.

The return type is always `DATE`, regardless of the datatype of *`date_expression`*.

Parameters

`date_expression`

The starting date. This can be any expression that can be implicitly converted to `DATE`.

`days_to_add`

The number of days to be add to the *`date_expression`*. This is an integer or any value that can be implicitly converted to an integer. This parameter can be positive or negative.

Example

```
SELECT name, next_day(hiredate,90) "Benefits Eligibility
Date"
FROM EMPLOYEES;
```

Returns the `name` and the `date` that is 90 days after `hiredate` labeled "`Benefits Eligibility Date.`"

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

nlssort

Oracle-compliant function that sorts data according to a specific collation.

Synopsis

```
nlssort (variable, collation)
```

Description

This Oracle-compatible function sorts data according to a specific collation.

Parameters

variable

The data to sort.

collation

The collation type by which to sort.

Example

```
CREATE TABLE test (name text);
INSERT INTO test VALUES('Anne'), ('anne'), ('Bob'), ('bob');
SELECT * FROM test ORDER BY nlssort(name, 'en_US.UTF-8');
 anne
 Anne
 bob
 Bob

SELECT * FROM test ORDER BY nlssort(name, 'C');
 Anne
 Bob
 anne
 bob
```

In the first example, the UTF-8 collation rules are specified. This groups characters together regardless of case.

In the second example, ASCII (C) collation is specified. This sorts according to ASCII order. The result is that upper case characters are sorted ahead of lower case ones.

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

nvl

Oracle-compliant function to substitute a specified value when an expression evaluates to `null`.

Note: This function is analogous to PostgreSQL `coalesce` function.

Synopsis

```
nvl(expression_to_evaluate, null_replacement_value)
```

Description

This Oracle-compatible function evaluates *`expression_to_evaluate`*. If it is `null`, the function returns *`null_replacement_value`*; otherwise, it returns *`expression_to_evaluate`*.

Parameters

`expression_to_evaluate`

The expression to evaluate for a null value.

`null_replacement_value`

The value to return if *`expression_to_evaluate`* is `null`.

Both *`expression_to_evaluate`* and *`null_replacement_value`* must be the same data type.

Examples

```
SELECT nvl(contact_name,'None')
FROM clients;
SELECT nvl(amount_past_due,0)
FROM txns;
SELECT nvl(nickname, firstname)
FROM contacts;
```

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

nvl2

Oracle-compliant function that returns alternate values for both null and non-null values.

Synopsis

```
nvl2(expression_to_evaluate, non_null_replacement_value,
     null_replacement_value)
```

Description

This Oracle-compatible function evaluates *expression_to_evaluate*. If it is not `null`, the function returns *non_null_replacement_value*; otherwise, it returns *null_replacement_value*.

Parameters

expression_to_evaluate

The expression to evaluate for a null value.

non_null_replacement_value

The value to return if *expression_to_evaluate* is not `null`.

null_replacement_value

The value to return if *expression_to_evaluate* is `null`.

Example

```
select nvl2(unit_number,'Multi Unit','Single Unit')
from clients;
```

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

See Also

nvl

oracle.substr

This Oracle-compliant function extracts a portion of a string.

Synopsis

```
oracle.substr(string, [start [,char_count]])
```

Description

This Oracle-compatible function extract a portion of a string.

If *start* is 0, it is evaluated as 1.

If *start* is negative, the starting position is negative, the starting position is *start* characters moving backwards from the end of string.

If *char_count* is not passed to the function, all characters from start to the end of string are returned.

If *char_count* is less than 1, null is returned.

If *start* or *char_count* is a number, but not an integer, the values are resolved to integers.

Parameters

string

The string from which to extract.

start

An integer specifying the starting position in the string.

char_count

An integer specifying the number of characters to extract.

Example

```
oracle.substr(name,1,15)
```

Returns the first 15 characters of `name`.

```
oracle.substr("Greenplum",-4,4)
```

Returns "`plum`."

```
oracle.substr(name,2)
```

Returns all characters of `name`, beginning with the second character.

Compatibility

PostgreSQL substr (not compatible with Oracle)

reverse

Oracle-compliant function to return the input string in reverse order.

Synopsis

```
reverse (string)
```

Description

This Oracle-compatible function returns the input string (*string*) in reverse order.

Parameters

string

The input string.

Example

```
SELECT reverse('gnirts')
FROM ClientDB;
```

Returns `'string'`

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

round

Oracle-compliant function to round a date to a specific unit of measure (day, week, etc.).

Note: This function is an overloaded function. It shares the same name with the Postgres `round` mathematical function that rounds numeric input to the nearest integer or optionally to the nearest x number of decimal places.

Synopsis

```
round (date_time_expression, [unit_of_measure])
```

Description

This Oracle-compatible function rounds a *`date_expression`* to the nearest *`unit_of_measure`* (day, week, etc.). If a *`unit_of_measure`* is not specified, the *`date_expression`* is rounded to the nearest day. It operates according to the rules of the Gregorian calendar.

If the *`date_time_expression`* datatype is `TIMESTAMP`, the value returned is always of datatype `TIMESTAMP`.

If the *`date_time_expression`* datatype is `DATE`, the value returned is always of datatype `DATE`.

Parameters

`date_time_expression`

The date to round. This can be any expression that can be implicitly converted to `DATE` or `TIMESTAMP`.

`unit_of_measure`

The unit of measure to apply for rounding. If not specified, then the *`date_time_expression`* is rounded to the nearest day. Valid parameters are:

Table 16.2 Valid Parameters

Unit	Valid parameters	Rounding Rule
Year	SYYYY, YYYY, YEAR, SYEAR, YYY, YY, Y	Rounds up on July 1st
ISO Year	IYYY, IY, I	
Quarter	Q	Rounds up on the 16th day of the second month of the quarter
Month	MONTH, MON, MM, RM	Rounds up on the 16th day of the month
Week	WW	Same day of the week as the first day of the year
IW	IW	Same day of the week as the first day of the ISO year

Table 16.2 Valid Parameters

Unit	Valid parameters	Rounding Rule
`W`	`W`	Same day of the week as the first day of the month
`Day`	`DDD, DD, J`	Rounds to the nearest day
`Start day of the week`	`DAY, DY, D`	Rounds to the nearest start (sunday) day of the week
`Hour`	`HH, HH12, HH24`	Rounds to the next hour
`Minute`	`MI`	Rounds to the next minute

Example

```
SELECT round(TO_DATE('27-OCT-00','DD-MON-YY'), 'YEAR')
FROM ClientDB;
```

Returns '01-JAN-01' (27 Oct 00 rounded to the first day of the following year (`YEAR`))

```
SELECT round('startdate','Q')
FROM ClientDB;
```

Returns '01-JUL-92' (the `startdate` rounded to the first day of the quarter (`Q`))

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

See Also

PostgreSQL round (not compatible with Oracle)

sinh

Oracle-compliant function to return the hyperbolic sine of a given number.

Synopsis

```
sinh(float8)
```

Description

This Oracle-compatible function returns the hyperbolic sine of the floating 8 input number (*float8*).

Note: This function is available by default and can be accessed without running the Oracle Compatibility installer.

Parameters

float8

The input number.

Example

```
SELECT sinh(3)
FROM ClientDB;
```

Returns `'10.0178749274099'`(hyperbolic sine of 3)

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

tanh

Oracle-compliant function to return the hyperbolic tangent of a given number.

Synopsis

```
tanh(float8)
```

Description

This Oracle-compatible function returns the hyperbolic tangent of the floating 8 input number (*float8)*.

Note: This function is available by default and can be accessed without running the Oracle Compatibility installer.

Parameters

float8

The input number.

Example

```
SELECT tanh(3)
FROM ClientDB;
```

Returns `'0.99505475368673''` (hyperbolic tangent of 3)

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

trunc

Oracle-compliant function to truncate a date to a specific unit of measure (day, week, hour, etc.).

Note: This function is an overloaded function. It shares the same name with the Postgres `trunc` and the Oracle `trunc` mathematical functions. Both of these truncate numeric input to the nearest integer or optionally to the nearest x number of decimal places.

Synopsis

```
trunc(date__time_expression, [unit_of_measure])
```

Description

This Oracle-compatible function truncates a *date_time_expression* to the nearest *unit_of_measure* (day, week, etc.). If a *unit_of_measure* is not specified, the *date_time_expression* is truncated to the nearest day. It operates according to the rules of the Gregorian calendar.

If the *date_time_expression* datatype is `TIMESTAMP`, the value returned is always of datatype `TIMESTAMP`, truncated to the hour/min level.

If the *date_time_expression* datatype is `DATE`, the value returned is always of datatype `DATE`.

Parameters

date_time_expression

The date to truncate. This can be any expression that can be implicitly converted to `DATE` or `TIMESTAMP`.

unit_of_measure

The unit of measure to apply for truncating. If not specified, then *date_time_expression* is truncated to the nearest day. Valid formats are:

Table 16.3 Valid Format Parameters

Unit	Valid parameters
Year	SYYYY, YYYY, YEAR, SYEAR, YYY, YY, Y
ISO Year	IYYY, IY, I
Quarter	Q
Month	MONTH, MON, MM, RM
Week	WW
IW	IW
W	W

Table 16.3 Valid Format Parameters

Unit	Valid parameters
Day	DDD, DD, J
Start day of the week	DAY, DY, D
Hour	HH, HH12, HH24
Minute	MI

Examples

```
SELECT TRUNC(TO_DATE('27-OCT-92','DD-MON-YY'), 'YEAR')
FROM ClientDB;
```

Returns '01-JAN-92' (27 Oct 92 truncated to the first day of the year (YEAR))

```
SELECT TRUNC(startdate,'Q')
FROM ClientDB;
```

Returns '1992-07-01' (the startdate truncated to the first day of the quarter (Q), depending on the date_style setting)

Compatibility

This command is compatible with Oracle syntax and is provided for convenience.

See Also

PostgreSQL trunc (not compatible with Oracle)

17. External Data Sources

You access external data as regular database tables with the CREATE EXTERNAL TABLE command and the Greenplum parallel file distribution utility `gpfdist`. This chapter covers the following topics:

[COMMENT] Load/Unload introduction moved to "New - Loading and Unloading Data"

External Tables

The forms of the command CREATE EXTERNAL TABLE lets you create a readable or writable external table definition in Greenplum Database.

The CREATE EXTERNAL WEB TABLE form of the command creates an external web table. 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.

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.

You can also use external tables to load and unload data.

When you define an external table in Greenplum Database. You specify one of these protocols

- file://
- gpdist://
- gpdists://
- gphdfs://
- http:// supported by external web tables.

You can also create your own custom protocol, see "Creating and Declaring a Custom Protocol" on page 171

External tables enable accessing external files as if they are regular database tables. When used with theGreenplum Database utility `gpfdist`, external tables provide full parallelism by using the resources of all Greenplum segments to load or unload data. When working with Hadoop Distributed File System, Greenplum Database leverages the parallel architecture of the Hadoop Distributed File System to access files on that system.

You can query external table data directly and in parallel using SQL commands such as `SELECT`, `JOIN`, or `SORT EXTERNAL TABLE DATA`, and you can create views for external tables.

Greenplum Database provides readable and writable external tables:

- Readable external tables for data loading. Readable external tables support basic extraction, transformation, and loading (ETL) tasks common to data warehousing. Greenplum Database segment instances read external table data in parallel to optimize large load operations. You cannot modify readable external tables.
- Writable external tables for data unloading. Writable external tables support:
  - Selecting data from database tables to insert into the writable external table.
  - Sending data to an application as a stream of data. For example, unload data from Greenplum Database and send it to an application that connects to another database or ETL tool to load the data elsewhere.
  - Receiving output from Greenplum parallel MapReduce calculations.

  Writable external tables allow only `INSERT` operations.

External tables can be file-based or web-based. External tables using the `file://` protocol are read-only tables.

- Regular (file-based) external tables access static flat files. Regular external tables are rescannable: the data is static while the query runs.
- Web (web-based) external tables access dynamic data sources, either on a web server with the `http://` protocol or by executing OS commands or scripts. Web external tables are not rescannable: the data can change while the query runs.

Dump and restore operate only on external and web external table definitions, not on the data sources.

Creating and Using External Tables

The steps for using external tables are:

1. Define the external table with the CREATE EXTERNAL TABLE statement.
2. Do one of the following:
   - Start the Greenplum files servers if you plan to use the `gpfdist` or `gpdists` protocols.
   - Verify that you have already set up the required one-time configuration for `gphdfs`.
3. Place the data files in the correct location.
4. Query the external table with SQL commands.

[COMMENT] gpload moved to "gpload"

[COMMENT] COPY moved to "COPY"

[COMMENT] Loading Data into Greenplum Database moved to "Loading Data into Greenplum Database"

External Table Protocols that Access Data Sources

[COMMENT] Duplicate in "Accessing File-Based External Tables"

When you create an external table definition, you specify the format of your input files and the location of your external data sources. For information about input file formats, see "Specifying the Format of Data Files" on page 188.

Use one of the following protocols to access external table data sources. You cannot mix protocols in `CREATE EXTERNAL TABLE` statements.

- `gpfdist:` points to a directory on the file host and serves external data files to all Greenplum Database segments in parallel.
- `gpfdists:` the secure version of `gpfdist`.
- `gphdfs:` accesses files on a Hadoop Distributed File System (HDFS).
- `file://` accesses external data files on a segment host that the Greenplum superuser (`gpadmin`) can access.

The `gpfdist` and `gpfdists` protocols require a one-time setup during table creation.

gpfdist

The `gpfdist` protocol uses the `gpfdist` utility. The utility serves external data files from a directory on the file host to all Greenplum Database segments in parallel. `gpfdist` uncompresses `gzip (.gz)` and `bzip2 (.bz2)` files automatically. Run `gpfdist` on the host on which the external data files reside.

All primary segments access the external files in parallel, subject to the number of segments set in `gp_external_max_segments`. Use multiple `gpfdist` data sources in a `CREATE EXTERNAL TABLE` statement to scale the external table's scan performance. For more information about configuring this, see "Controlling Segment Parallelism" on page 174.

You can use the wildcard character (*) or other C-style pattern matching to denote multiple files to get. The files specified are assumed to be relative to the directory that you specified when you started `gpfdist`.

The `gpfdist` utility is located in `$GPHOME/bin` on your Greenplum Database master host and on each segment host. See the `gpfdist` reference documentation for more information about using `gpfdist` with external tables.

gpfdists

`The gpfdists` protocol is a secure version of gpfdist. `gpfdists` enables encrypted communication and secure identification of the file server and the Greenplum Database to protect against attacks such as eavesdropping and man-in-the-middle attacks.

The `gpfdists` protocol implements SSL security in a client/server scheme as follows.

- Client certificates are required.
- Multilingual certificates are not supported.
- A Certificate Revocation List (CRL) is not supported.
- The `TLSv1` protocol is used with the `TLS_RSA_WITH_AES_128_CBC_SHA` encryption algorithm.
- SSL parameters cannot be changed.
- SSL renegotiation is supported.
- The SSL ignore host mismatch parameter is set to `false`.
- Private keys containing a passphrase are not supported for the `gpfdist` file server (server.key) and for the Greenplum Database (client.key).
- Issuing certificates that are appropriate for the operating system in use is the user's responsibility. Generally, converting certificates as shown in https://www.sslshopper.com/ssl-converter.html is supported.

Note: A server started with the `gpfdist --ssl` option can only communicate with the *gpfdists* protocol. A server that was started with `gpfdist` without the `--ssl` option can only communicate with the *gpfdist* protocol.

Use one of the following methods to invoke the *gpfdists* protocol.

- Run `gpfdist` with the `--ssl` option and then use the *gpfdists* protocol in the `LOCATION` clause of a `CREATE EXTERNAL TABLE` statement.
- Use a YAML Control File with the `SSL` option set to true and run `gpload`. Running `gpload` starts the `gpfdist` server with the `--ssl` option, then uses the *gpfdists* protocol.

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

The `gpfdists` protocol requires that the following client certificates reside in the `$PGDATA/gpfdists` directory on each segment.

  - The client certificate file, `client.crt`
  - The client private key file, `client.key`
  - The trusted certificate authorities, `root.crt`

For an example of loading data into an external table securely, see "Example 3—Multiple gpfdists instances" on page 176.

file

The `file://` protocol requires that the external data files reside on a segment host in a location accessible by the Greenplum superuser (`gpadmin`). The number of URIs that you specify corresponds to the number of segment instances that will work in parallel to access the external table. For example, if you have a Greenplum Database system with 8 primary segments and you specify 2 external files, only 2 of the 8 segments will access the external table in parallel at query time. The number of external files per segment host cannot exceed the number of primary segment instances on that host. For example, if your array has 4 primary segment instances per segment host, you can place 4 external files on each segment host. The host name used in the URI must match the segment host name as registered in the *gp_segment_configuration* system catalog table. Tables based on the `file://` protocol can only be readable tables.

The system view `pg_max_external_files` shows how many external table files are permitted per external table. This view lists the available file slots per segment host when using the `file://` protocol. The view is only applicable for the `file://` protocol. For example:

```
SELECT * FROM pg_max_external_files;
```

gphdfs

The gphdfs protocol specifies a path that can contain wild card characters on a Hadoop Distributed File System. `TEXT` and custom formats are allowed for `HDFS` files.

See "Hadoop Distributed File System" for information about accessing HDFS data.

Creating and Declaring a Custom Protocol

Greenplum provides protocols such as `gpfdist`, `http`, and `file` for accessing data over a network, or you can author a custom protocol. You can use the standard data formats, `TEXT` and `CSV`, or a custom data format with custom protocols.

You can create a custom protocol whenever the available built-in protocols do not suffice for a particular need. For example, if you need to connect Greenplum Database in parallel to another system directly, and stream data from one to the other without the need to materialize the system data on disk or use an intermdiate process such as `gpfdist`.

The following steps describe authoring and using a custom protocol.

1. Author the send, receive, and (optionally) validator functions in C, with a predefined API. These functions are compiled and registered with the Greenplum Database. For an example custom protocol, see "Example Custom Data Access Protocol" on page 203.

2. After writing and compiling the read and write functions into a shared object (.so), declare a database function that points to the .so file and function names.

   The following examples use the compiled import and export code.

```
CREATE FUNCTION myread() RETURNS integer
as '$libdir/gpextprotocol.so', 'myprot_import'
LANGUAGE C STABLE;

CREATE FUNCTION mywrite() RETURNS integer
as '$libdir/gpextprotocol.so', 'myprot_export'
LANGUAGE C STABLE;
```

The format of the optional function is:

```
CREATE OR REPLACE FUNCTION myvalidate() RETURNS void
AS '$libdir/gpextprotocol.so', 'myprot_validate'
LANGUAGE C STABLE;
```

3. Create a protocol that accesses these functions. `Validatorfunc` is optional.

```
CREATE TRUSTED PROTOCOL myprot(
   writefunc='mywrite'
   readfunc='myread',
   validatorfunc='myvalidate');
```

4. Grant access to any other users, as necessary

```
GRANT ALL ON PROTOCOL myprot TO otheruser
```

5. Use the protocol in readable or writable external tables.

```
CREATE WRITABLE EXTERNAL TABLE ext_sales(LIKE sales)
LOCATION ('myprot://<meta>/<meta>/…')
FORMAT 'TEXT';

CREATE READABLE EXTERNAL TABLE ext_sales(LIKE sales)
LOCATION('myprot://<meta>/<meta>/…')
FORMAT 'TEXT';
```

Declare custom protocols with the SQL command `CREATE TRUSTED PROTOCOL`, then use the `GRANT` command to grant access to your users. For example:

- Allow a user to create a readable external table with a trusted protocol

```
GRANT SELECT ON PROTOCOL <protocol name> TO <user name>
```

- Allow a user to create a writable external table with a trusted protocol

```
GRANT INSERT ON PROTOCOL <protocol name> TO <user name>
```

- Allow a user to create readable and writable external tables with a trusted protocol

```
GRANT ALL ON PROTOCOL <protocol name> TO <user name>
```

Using the Greenplum Parallel File Server (gpfdist)

The `gpfdist` utility provides the best performance and is the easiest to set up. `gpfdist` ensures optimum use of all segments in your Greenplum Database system for external table reads.

This section describes the setup and management tasks for using `gpfdist` with external tables.

- About gpfdist Setup and Performance
- Controlling Segment Parallelism
- Installing the gpfdist utility
- Starting and Stopping gpfdist
- Troubleshooting gpfdist

About gpfdist Setup and Performance

Consider the following scenarios for optimizing your ETL network performance.

- Allow network traffic to use all ETL host Network Interface Cards (NICs) simultaneously. Run one instance of `gpfdist` on the ETL host, then declare the host name of each NIC in the `LOCATION` clause of your external table definition (see "Defining External Tables - Examples" on page 176).

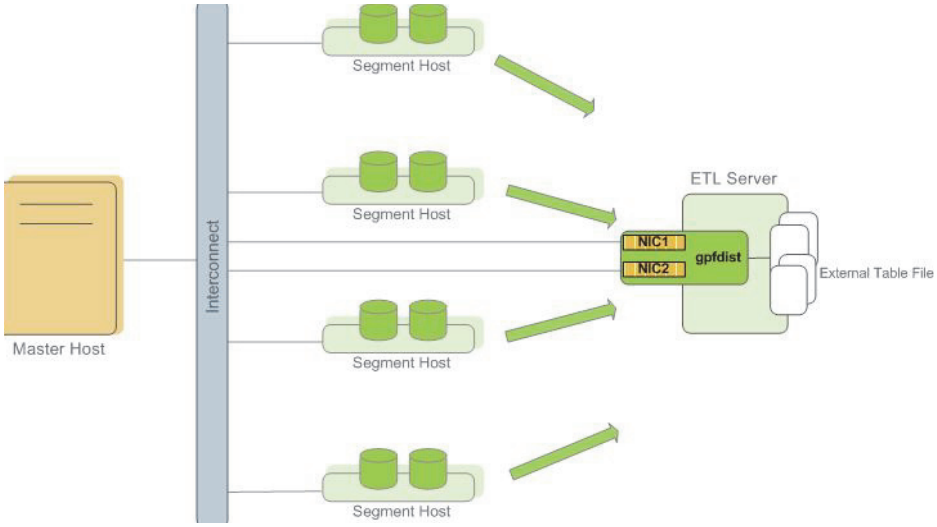


Figure 17.1 External Table Using Single gpfdist Instance with Multiple NICs

- Divide external table data equally among multiple `gpfdist` instances on the ETL host. For example, on an ETL system with two NICs, run two `gpfdist` instances (one on each NIC) to optimize data load performance and divide the external table data files evenly between the two `gpfdist`s.

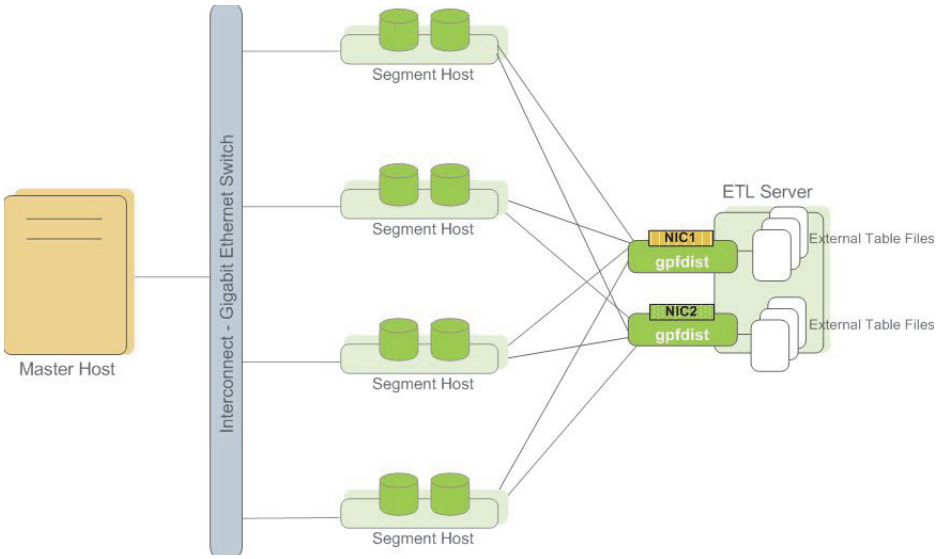


Figure 17.2 External Tables Using Multiple gpfdist Instances with Multiple NICs

Note: Use pipes (|) to separate formatted text when you submit files to `gpfdist`. Greenplum Database encloses comma-separated text strings in single or double quotes. `gpfdist` has to remove the quotes to parse the strings. Using pipes to separate formatted text avoids the extra step and improves performance.

Controlling Segment Parallelism

The `gp_external_max_segs` server configuration parameter controls the number of segment instances that can access a single `gpfdist` instance simultaneously. 64 is the default. You can set the number of segments such that some segments process external data files and some perform other database processing. Set this parameter in the `postgresql.conf` file of your master instance.

Installing the gpfdist utility

The Greenplum Database utility `gpfdist` is installed in `$GPHOME/bin` of your Greenplum Database master host installation.

You can run `gpfdist` from a machine other than the Greenplum Database master, such as on a machine devoted to ETL processing. If you want to install gpfdist on your ETL server, get it from the *Greenplum Load Tools* package and follow its installation instructions.

Starting and Stopping gpfdist

You can start gpfdist in your current directory location or in any directory that you specify. The default port is `8080`.

From your current directory, type:

```
& gpfdist
```

From a different directory, specify the directory from which to serve files, and optionally, the HTTP port to run on.

To start `gpfdist` in the background and log output messages and errors to a log file:

```
$ gpfdist -d /var/load_files -p 8081 -l /home/gpadmin/log &
```

For multiple `gpfdist` instances on the same ETL host (see Figure 17.2), use a different base directory and port for each instance. For example:

```
$ gpfdist -d /var/load_files1 -p 8081 -l /home/gpadmin/log1 &
$ gpfdist -d /var/load_files2 -p 8082 -l /home/gpadmin/log2 &
```

To stop `gpfdist` when it is running in the background:

First find its process id:

```
$ ps -ef | grep gpfdist
```

Then kill the process, for example (where 3456 is the process ID in this example):

```
$ kill 3456
```

Troubleshooting gpfdist

The segments access `gpfdist` at runtime. Ensure that the Greenplum segment hosts have network access to `gpfdist`. `gpfdist` is a web server: test connectivity by running the following command from each host in the Greenplum array (segments and master):

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

The `CREATE EXTERNAL TABLE` definition must have the correct host name, port, and file names for `gpfdist`. Specify file names and paths relative to the directory from which `gpfdist` serves files (the directory path specified when `gpfdist` started). See "Defining External Tables - Examples" on page 176.

[COMMENT] Moved Using Hadoop Distributed File System (HDFS) Tables "Hadoop Distributed File System"

Accessing External Table Data

Duplicate in "Loading Data Using an External Table"

Use SQL commands such as `INSERT` and `SELECT` to query a readable external table, the same way that you query a regular database table. For example, to load travel expense data from an external table, *ext_expenses*, into a database table, *expenses_travel*:

```
=# INSERT INTO expenses_travel
    SELECT * from ext_expenses where category='travel';
```

To load all data into a new database table:

```
=# CREATE TABLE expenses AS SELECT * from ext_expenses;
```

Defining External Tables - Examples

The following examples show how to define external data with different protocols. Each `CREATE EXTERNAL TABLE` command can contain only one protocol.

Note: When using IPv6, always enclose the numeric IP addresses in square brackets.

Start `gpfdist` before you create external tables with the `gpfdist` protocol. The following code starts the `gpfdist` file server program in the background on port *8081* serving files from directory */var/data/staging*. The logs are saved in */home/gpadmin/log*.

```
gpfdist -p 8081 -d /var/data/staging -l /home/gpadmin/log &
```

The `CREATE EXTERNAL TABLE` SQL command defines external tables, the location and format of the data to load, and the protocol to use to load the data, but does not load data into the table. For example, the following command creates an external table, `ext_expenses`, from pipe ('|') delimited text data located on `etlhost-1:8081 and etlhost-2:8081`. See the *Greenplum Database Reference Guide* for information about `CREATE EXTERNAL TABLE`.

Example 1—Single gpfdist instance on single-NIC machine

Creates a readable external table, *ext_expenses,* using the `gpfdist` protocol. The files are formatted with a pipe (|) as the column delimiter.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date, amount float4, category text, desc1 text )
   LOCATION ('gpfdist://etlhost-1:8081/*',
             'gpfdist://etlhost-1:8082/*')
   FORMAT 'TEXT' (DELIMITER '|');
```

Example 2—Multiple gpfdist instances

Creates a readable external table, *ext_expenses,* using the `gpfdist` protocol from all files with the *txt* extension. The column delimiter is a pipe (|) and NULL (' ') is a space.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date,  amount float4, category text, desc1 text )
   LOCATION ('gpfdist://etlhost-1:8081/*.txt',
             'gpfdist://etlhost-2:8081/*.txt')
   FORMAT 'TEXT' ( DELIMITER '|' NULL ' ') ;
```

Example 3—Multiple gpfdists instances

Creates a readable external table, *ext_expenses,* from all files with the *txt* extension using the `gpfdists` protocol. The column delimiter is a pipe (|) and NULL (' ') is a space. For information about the location of security certificates, see "gpfdists" on page 170.

1. Run gpfdist with the `--ssl` option.
2. Run the following command.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
```

```
    date date,  amount float4, category text, desc1 text )
    LOCATION ('gpfdists://etlhost-1:8081/*.txt',
              'gpfdists://etlhost-2:8082/*.txt')
    FORMAT 'TEXT' ( DELIMITER '|' NULL ' ') ;
```

Example 4—Single gpfdist instance with error logging

Uses the gpfdist protocol to create a readable external table, *ext_expenses,* from all files with the *txt* extension. The column delimiter is a pipe (|) and NULL (‘ ‘) is a space.

Access to the external table is single row error isolation mode. Input data formatting errors are written to the error table, `err_customer`, with a description of the error. Query `err_customer` to see the errors, then fix the issues and reload the rejected data. If the error count on a segment is greater than five (the `SEGMENT REJECT LIMIT` value), the entire external table operation fails and no rows are processed.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date,  amount float4, category text, desc1 text )
   LOCATION ('gpfdist://etlhost-1:8081/*.txt',
             'gpfdist://etlhost-2:8082/*.txt')
   FORMAT 'TEXT' ( DELIMITER '|' NULL ' ')
   LOG ERRORS INTO err_customer SEGMENT REJECT LIMIT 5;
```

To create the readable *ext_expenses* table from CSV-formatted text files:

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date,  amount float4, category text, desc1 text )
   LOCATION ('gpfdist://etlhost-1:8081/*.txt',
             'gpfdist://etlhost-2:8082/*.txt')
   FORMAT 'CSV' ( DELIMITER ',' )
   LOG ERRORS INTO err_customer SEGMENT REJECT LIMIT 5;
```

Example 5—TEXT Format on a Hadoop Distributed File Server

Creates a readable external table, *ext_expenses,* using the `gphdfs` protocol. The column delimiter is a pipe (|).

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date,  amount float4, category text, desc1 text )
   LOCATION ('gphdfs://hdfshost-1:8081/data/filename.txt')
   FORMAT 'TEXT' (DELIMITER '|');
```

Note: gphdfs requires only one data path.

For examples of reading and writing custom formatted data on a Hadoop Distributed File System, see “Reading and Writing Custom-Formatted HDFS Data” on page 215.

Example 6—Multiple files in CSV format with header rows

Creates a readable external table, *ext_expenses,* using the `file` protocol. The files are `CSV` format and have a header row.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
```

```
date date,  amount float4, category text, desc1 text )
LOCATION ('file://filehost:5432/data/international/*',
          'file://filehost:5432/data/regional/*'
          'file://filehost:5432/data/supplement/*.csv')
FORMAT 'CSV' (HEADER);
```

Example 7—Readable Web External Table with Script

Creates 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 '|');
```

Example 8—Writable External Table with gpfdist

Creates a writable external table, *sales_out,* that uses `gpfdist` to write output data to the file *sales.out*. The column delimiter is a pipe (|) and NULL (‘ ‘) is a space. The file will be created in the directory specified when you started the gpfdist file server.

```
=# CREATE WRITABLE EXTERNAL TABLE sales_out (LIKE sales)
   LOCATION ('gpfdist://etl1:8081/sales.out')
   FORMAT 'TEXT' ( DELIMITER '|' NULL ' ')
   DISTRIBUTED BY (txn_id);
```

Example 9—Writable External Web Table with Script

Creates a writable external web table, *campaign_out*, that pipes output data received by the segments to an executable script, *to_adreport_etl.sh*:

```
=# CREATE WRITABLE EXTERNAL WEB TABLE campaign_out
   (LIKE campaign)
   EXECUTE '/var/unload_scripts/to_adreport_etl.sh'
   FORMAT 'TEXT' (DELIMITER '|');
```

Example 10—Readable and Writable External Tables with XML Transformations

Greenplum Database can read and write XML data to and from external tables with gpfdist. For information about setting up an XML transform, see “Transforming XML Data” on page 191.

Creating External Web Tables

The SQL command `CREATE EXTERNAL WEB TABLE` creates a web table definition. Web external tables allow Greenplum Database to treat dynamic data sources like regular database tables. Because web table data can change as a query runs, the data is not rescannable.

You can define command-based or URL-based web external tables. The definition forms are distinct: you cannot mix command-based and URL-based definitions.

Command-based Web External Tables

The output of a shell command or script defines command-based web table data. Specify the command in the EXECUTE clause of CREATE EXTERNAL WEB TABLE. The data is current as of the time the command runs. The EXECUTE clause runs the shell command or script on the specified master, and/or segment host or hosts. The command or script must reside on the hosts corresponding to the hosts defined in the EXECUTE clause.

By default, the command is run on segment hosts when active segments have output rows to process. For example, if each segment host runs four primary segment instances that have output rows to process, the command runs four times per segment host. You can optionally limit the number of segment instances that execute the web table command. All segments included in the web table definition in the ON clause run the command in parallel.

The command that you specify in the external table definition executes from the database and cannot access environment variables from .bashrc or .profile. Set environment variables in the EXECUTE clause. For example:

```
=# CREATE EXTERNAL WEB TABLE output (output text)
    EXECUTE 'PATH=/home/gpadmin/programs; export PATH;
myprogram.sh'
    FORMAT 'TEXT';
```

Scripts must be executable by the gpadmin user and reside in the same location on the master or segment hosts.

The following command defines a web table that runs a script. The script runs on each segment host where a segment has output rows to process.

```
=# CREATE EXTERNAL WEB TABLE log_output
    (linenum int, message text)
    EXECUTE '/var/load_scripts/get_log_data.sh' ON HOST
    FORMAT 'TEXT' (DELIMITER '|');
```

URL-based Web External Tables

A URL-based web table accesses data from a web server using the HTTP protocol. Web table data is dynamic: the data is not rescannable.

Specify the LOCATION of files on a web server with the http protocol http://. The web data files must reside on a web server that Greenplum segment hosts can access. The number of URLs specified corresponds to the number of segment instances that work in parallel to access the web table. For example, if you specify 2 external files to a Greenplum Database system with 8 primary segments, 2 of the 8 segments access the web table in parallel at query runtime.

The following sample command defines a web table that gets data from several URLs.

```
=# CREATE EXTERNAL WEB TABLE ext_expenses (name text,
  date date, amount float4, category text, description text)
  LOCATION (
  'http://intranet.company.com/expenses/sales/file.csv',
  'http://intranet.company.com/expenses/exec/file.csv',
```

```
'http://intranet.company.com/expenses/finance/file.csv',
'http://intranet.company.com/expenses/ops/file.csv',
'http://intranet.company.com/expenses/marketing/file.csv',
'http://intranet.company.com/expenses/eng/file.csv'
 )
FORMAT 'CSV' ( HEADER );
```

Moving Data from External Tables

Readable external tables are commonly used to select data to load into regular database tables. You use `CREATE TABLE AS` or `INSERT...SELECT` to load external and web external table data into another (non-external) database table. The data is loaded in parallel according to the external or web external table definition.

If an external table file or web external table data source has an error, one of the following occurs, depending on the isolation mode used:

- **Tables without error isolation mode**: any operation that reads from that table fails. Loading from external and web external tables without error isolation mode is an all or nothing operation.
- **Tables with error isolation mode**: the entire file will be loaded, except for the problematic rows (subject to the configured REJECT_LIMIT)

[COMMENT] Moved Loading data to "Loading Data"

Optimizing Data Load and Query Performance

[COMMENT] Duplicate in "Optimizing Data Load and Query Performance"

Use the following tips to help optimize your data load and subsequent query performance.

- Drop indexes before loading data into existing tables.

  Creating an index on pre-existing data is faster than updating it incrementally as each row is loaded. You can temporarily increase the *maintenance_work_mem* server configuration parameter to help speed up `CREATE INDEX` commands, though load performance is affected. Drop and recreate indexes only when there are no active users on the system.
- Create indexes last when loading data into new tables. Create the table, load the data, and create any required indexes.
- Run `ANALYZE` after loading data. If you significantly altered the data in a table, run `ANALYZE` or `VACUUM ANALYZE` to update table statistics for the query planner. Current statistics ensure that the planner makes the best decisions during query planning and avoids poor performance due to inaccurate or nonexistent statistics.
- Run `VACUUM` after load errors. If the load operation does not run in single row error isolation mode, the operation stops at the first error. The target table contains the rows loaded before the error occurred. You cannot access these rows, but they occupy disk space. Use the `VACUUM` command to recover the wasted space.

Errors in External Table Data

By default when you load data from an external table, if external table data contains an error, the command fails and no data loads into the target database table. You can define the external table with single row error handling to enable loading correctly-formatted rows and to isolate data errors in external table data. See "Handling Errors from External Tables".

The `gpfdist` file server uses the `HTTP` protocol. External table queries that use `LIMIT` end the connection after retrieving the rows, causing an `HTTP` socket error. If you use `LIMIT` in queries of external tables that use the `gpfdist://` or `http://` protocols, ignore these errors – data is returned to the database as expected.

Handling Errors from External Tables

Readable external tables are most commonly used to select data to load into regular database tables. You use the `CREATE TABLE AS SELECT` or `INSERT INTO` commands to query the external table data. By default, if the data contains an error, the entire command fails and the data is not loaded into the target database table.

The `SEGMENT REJECT LIMIT` clause allows you to isolate format errors in external table data and to continue loading correctly formatted rows. Use `SEGMENT REJECT LIMIT` to set an error threshold, specifying the reject limit *`count`* as number of `ROWS` (the default) or as a `PERCENT` of total rows (1-100).

The entire external table operation is aborted, and no rows are processed, if the number of error rows reaches the `SEGMENT REJECT LIMIT`. The limit of error rows is per-segment, not per entire operation. The operation processes all good rows, and it discards or logs any erroneous rows into an error table (if you specified an error table), if the number of error rows does not reach the `SEGMENT REJECT LIMIT`.

The `LOG ERRORS INTO` clause allows you to keep error rows for further examination. Use `LOG ERRORS INTO` to declare an error table in which to write error rows.

When you set `SEGMENT REJECT LIMIT`, Greenplum scans the external data in single row error isolation mode. Single row error isolation mode applies to external data rows with format errors such as extra or missing attributes, attributes of a wrong data type, or invalid client encoding sequences. Greenplum does not check constraint errors, but you can filter constraint errors by limiting the `SELECT` from an external table at runtime. For example, to eliminate duplicate key errors:

```
=# INSERT INTO table_with_pkeys
    SELECT DISTINCT * FROM external_table;
```

Define an External Table with Single Row Error Isolation

The following example creates an external table, *ext_expenses*, sets an error threshold of 10 errors, and writes error rows to the table *err_expenses*.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date,  amount float4, category text, desc1 text )
   LOCATION ('gpfdist://etlhost-1:8081/*',
             'gpfdist://etlhost-2:8082/*')
```

```
FORMAT 'TEXT' (DELIMITER '|')
LOG ERRORS INTO err_expenses SEGMENT REJECT LIMIT 10
  ROWS;
```

Create an Error Table and Declare a Reject Limit

The following SQL fragment creates an error table, *err_expenses*, and declares a reject limit of 10 rows.

```
LOG ERRORS INTO err_expenses SEGMENT REJECT LIMIT 10 ROWS
```

Viewing Bad Rows in the Error Table

If you use single row error isolation (see "Define an External Table with Single Row Error Isolation" on page 181), any rows with formatting errors are logged into an error table. The error table has the following columns:

Table 17.1 Error Table Format

column	type	description
cmdtime	timestampz	Timestamp when the error occurred.
relname	text	The name of the external table or the target table of a COPY command.
filename	text	The name of the load file that contains the error.
linenum	int	If COPY was used, the line number in the load file where the error occurred. For external tables using file:// protocol or gpfdist:// protocol and CSV format, the file name and line number is logged.
bytenum	int	For external tables with the gpfdist:// protocol and data in TEXT format: the byte offset in the load file where the error occurred. gpfdist parses TEXT files in blocks, so logging a line number is not possible. CSV files are parsed a line at a time so line number tracking is possible for CSV files.
errmsg	text	The error message text.
rawdata	text	The raw data of the rejected row.
rawbytes	bytea	In cases where there is a database encoding error (the client encoding used cannot be converted to a server-side encoding), it is not possible to log the encoding error as *rawdata*. Instead the raw bytes are stored and you will see the octal code for any non seven bit ASCII characters.

You can use SQL commands to query the error table and view the rows that did not load. For example:

```
=# SELECT * from err_expenses;
```

Identifying Invalid CSV Files in Error Table Data

If a CSV file contains invalid formatting, the *rawdata* field in the error table can contain several combined rows. For example, if a closing quote for a specific field is missing, all the following newlines are treated as embedded newlines. When this happens, Greenplum stops parsing a row when it reaches 64K, puts that 64K of data into the error table as a single row, resets the quote flag, and continues. If this happens three times during load processing, the load file is considered invalid and the entire load fails with the message "`rejected N or more rows`". See "Escape Characters in CSV Formatted Files" on page 190 for more information on the correct use of quotes in CSV files.

Writable External Tables

[COMMENT] Duplicate in "Unloading Data from Greenplum Database"

A writable external table allows you to select rows from other database tables and output the rows to files, named pipes, to applications, or as output targets for Greenplum parallel MapReduce calculations. You can define file-based and web-based writable external tables.

This section describes how to unload data from Greenplum Database using parallel unload (writable external tables).

- Writable External Tables
- Defining a Command-Based Writable External Web Table
- Writing Data to a Writable External Table

For information about writing to the HDFS, see Hadoop Distributed File System.

You can also and non-parallel unload (`COPY`). See the DBA Guide for information.

Writable External Tables

You can use writable external tables and writeable external web tables to output data to files with the Greenplum parallel file server program, `gpfdist`, or the Hadoop Distributed File System interface, `gphdfs`.

Use the `CREATE WRITABLE EXTERNAL TABLE` command to define the external table and Use the `CREATE WRITABLE EXTERNAL WEB TABLE` command to define the external web table. Specify the location and format of the output files. See "Using the Greenplum Parallel File Server (gpfdist)" on page 172 for instructions on setting up `gpfdist` for use with an external table and Chapter 18, "Hadoop Distributed File System" for information about working with `gphdfs` and external tables.

- With a writable external table using the `gpfdist` protocol, the Greenplum segments send their data to `gpfdist`, which writes the data to the named file. `gpfdist` must run on a host that the Greenplum segments can access over the network. `gpfdist` points to a file location on the output host and writes data received from the Greenplum segments to the file. To divide the output data among multiple files, list multiple `gpfdist` URIs in your writable external table definition.
- A writable external web table sends data to an application as a stream of data. For example, unload data from Greenplum Database and send it to an application that connects to another database or ETL tool to load the data elsewhere. Writable external web tables use the `EXECUTE` clause to specify a shell command, script, or application to run on the segment hosts and accept an input stream of data. See "Defining a Command-Based Writable External Web Table" for more information about using `EXECUTE` commands in a writable external table definition.

You can optionally declare a distribution policy for your writable external tables. By default, writable external tables use a random distribution policy. If the source table you are exporting data from has a hash distribution policy, defining the same distribution key columns for the writable external table improves unload performance by eliminating the requirement to move rows over the interconnect. If you unload data from a particular table, you can use the `LIKE` clause to copy the column definitions and distribution policy from the source table.

Example 1—Greenplum file server (gpfdist)

```
=# CREATE WRITABLE EXTERNAL TABLE unload_expenses
   ( LIKE expenses )
   LOCATION ('gpfdist://etlhost-1:8081/expenses1.out',
             'gpfdist://etlhost-2:8081/expenses2.out')
 FORMAT 'TEXT' (DELIMITER ',')
 DISTRIBUTED BY (exp_id);
```

Defining a Command-Based Writable External Web Table

You can define writable external web tables to send output rows to an application or script. The application must accept an input stream, reside in the same location on all of the Greenplum segment hosts, and be executable by the `gpadmin` user. All segments in the Greenplum system run the application or script, whether or not a segment has output rows to process.

Use `CREATE WRITABLE EXTERNAL WEB TABLE` to define the external table and specify the application or script to run on the segment hosts. Commands execute from within the database and cannot access environment variables (such as `$PATH`). Set environment variables in the `EXECUTE` clause of your writable external table definition. For example:

```
=# CREATE WRITABLE EXTERNAL WEB TABLE output (output text)
    EXECUTE 'export PATH=$PATH:/home/gpadmin/programs;
    myprogram.sh'
```

```
    FORMAT 'TEXT'
    DISTRIBUTED RANDOMLY;
```

The following Greenplum Database variables are available for use in OS commands executed by a web or writable external table. Set these variables as environment variables in the shell that executes the commands. They can be used to identify a set of requests made by an external table statement across the Greenplum Database array of hosts and segment instances.

Table 17.2 External Table EXECUTE Variables

Variable	Description
$GP_CID	Command count of the session executing the external table statement.
$GP_DATABASE	The database in which the external table definition resides.
$GP_DATE	The date on which the external table command ran.
$GP_MASTER_HOST	The host name of the Greenplum master host from which the external table statement was dispatched.
$GP_MASTER_PORT	The port number of the Greenplum master instance from which the external table statement was dispatched.
$GP_SEG_DATADIR	The location of the data directory of the segment instance executing the external table command.
$GP_SEG_PG_CONF	The location of the `postgresql.conf` file of the segment instance executing the external table command.
$GP_SEG_PORT	The port number of the segment instance executing the external table command.
$GP_SEGMENT_COUNT	The total number of primary segment instances in the Greenplum Database system.
$GP_SEGMENT_ID	The ID number of the segment instance executing the external table command (same as `dbid` in `gp_segment_configuration`).
$GP_SESSION_ID	The database session identifier number associated with the external table statement.
$GP_SN	Serial number of the external table scan node in the query plan of the external table statement.
$GP_TIME	The time the external table command was executed.
$GP_USER	The database user executing the external table statement.
$GP_XID	The transaction ID of the external table statement.

Disabling EXECUTE for Web or Writable External Tables

There is a security risk associated with allowing external tables to execute OS commands or scripts. To disable the use of `EXECUTE` in web and writable external table definitions, set the `gp_external_enable_exec` server configuration parameter to `off` in your master `postgresql.conf` file:

```
gp_external_enable_exec = off
```

Writing Data to a Writable External Table

[COMMENT] Duplicate in "Unloading Data Using a Writable External Table"

Writable external tables allow only `INSERT` operations. You must grant `INSERT` permission on a table to enable access to users who are not the table owner or a superuser. For example:

```
GRANT INSERT ON writable_ext_table TO admin;
```

To unload data using a writable external table, select the data from the source tables and insert it into the writable external table. The resulting rows are output to the writable external table. For example:

```
INSERT INTO writable_ext_table SELECT * FROM regular_table;
```

[COMMENT] Moved Unloading Data Using COPY to "Unloading Data Using COPY"

Accessing and Writing Data with Custom Formats

Greenplum supports `TEXT` and `CSV` formats for importing and exporting data. You can load and write the data in other formats by defining and using a custom format or custom protocol.

For information about importing custom data from HDFS, see "Reading and Writing Custom-Formatted HDFS Data" on page 215.

You can also create a custom protocol to access data in custom formats. See Creating and Declaring a Custom Protocol for information about creating custom protocols.

Using a Custom Format

You specify a custom data format in the `FORMAT` clause of `CREATE EXTERNAL TABLE`.

```
FORMAT 'CUSTOM' (formatter=format_function,
key1=val1,...keyn=valn)
```

Where the `'CUSTOM'` keyword indicates that the data has a custom format and `formatter` specifies the function to use to format the data, followed by comma-separated parameters to the formatter function.

Greenplum Database provides functions for formatting fixed-width data, but you must author the formatter functions for variable-width data. The steps are as follows.

1. Author and compile input and output functions as a shared library.
2. Specify the shared library function with `CREATE FUNCTION` in Greenplum Database.
3. Use the `formatter` parameter of `CREATE EXTERNAL TABLE's FORMAT` clause to call the function.

Importing and Exporting Fixed Width Data

Specify custom formats for fixed-width data with the Greenplum Database functions `fixedwith_in` and `fixedwidth_out`. These functions already exist in the file *$GPHOME\share\postgresql\cdb_external_extensions.sql*. The following example declares a custom format, then calls the `fixedwidth_in` function to format the data.

```
CREATE READABLE EXTERNAL TABLE students (
name varchar(20), address varchar(30), age int)
LOCATION ('file://<host>/file/path/')
FORMAT 'CUSTOM' (formatter=fixedwidth_in,
        name='20', address='30', age='4');
```

The following options specify how to import fixed width data.

- Read all the data.

  To load all the fields on a line of fixed with data, you must load them in their physical order. You must specify the field length, but cannot specify a starting and ending position. The fields names in the fixed width arguments must match the order in the field list at the beginning of the `CREATE TABLE` command.

- Set options for blank and null characters.

  Trailing blanks are trimmed by default. To keep trailing blanks, use the `preserve_blanks=on` option.You can reset the trailing blanks option to the default with the `preserve_blanks=off` option.

  Use the null=`'null_string_value'` option to specify a value for null characters.

  - If you specify `preserve_blanks=on`, you must also define a value for null characters.
  - If you specify `preserve_blanks=off`, null is not defined, and the field contains only blanks, Greenplum writes a null to the table. If null is defined, Greenplum writes an empty string to the table.

  Use the `line_delim='line_ending'` parameter to specify the line ending character. The following examples cover most cases. The E specifies an escape string constant.

  ```
  line_delim=E'\n'
  line_delim=E'\r'
  line_delim=E'\r\n'
  line_delim='abc'
  ```

Examples: Read Fixed-Width Data

The following examples show how to read fixed-width data.

Example 1 – Loading a table with all fields defined

```
CREATE READABLE EXTERNAL TABLE students (
name varchar(20), address varchar(30), age int)
LOCATION ('file://<host>/file/path/')
```

```
FORMAT 'CUSTOM' (formatter=fixedwidth_in,
        name=20, address=30, age=4);
```

Example 2 – Loading a table with PRESERVED_BLANKS ON

```
CREATE READABLE EXTERNAL TABLE students (
name varchar(20), address varchar(30), age int)
LOCATION ('gpfdist://<host>:<portNum>/file/path/')
FORMAT 'CUSTOM' (formatter=fixedwidth_in,
        name=20, address=30, age=4,
        preserve_blanks='on',null='NULL');
```

Example 3 – Loading data with no line delimiter

```
CREATE READABLE EXTERNAL TABLE students (
name varchar(20), address varchar(30), age int)
LOCATION ('file://<host>/file/path/')
FORMAT 'CUSTOM' (formatter=fixedwidth_in,
        name='20', address='30', age='4', line_delim='?@')
```

Example 4 – Create a writable external table with a \r\n line delimiter

```
CREATE WRITABLE EXTERNAL TABLE students_out (
name varchar(20), address varchar(30), age int)
LOCATION ('gpfdist://<host>:<portNum>/file/path/')
FORMAT 'CUSTOM' (formatter=fixedwidth_out,
        name=20, address=30, age=4, line_delim=E'\r\n');
```

Specifying the Format of Data Files

When you use the Greenplum tools for loading and unloading data, you must specify how your data is formatted. `COPY`, `CREATE EXTERNAL TABLE`, and `gpload` have clauses that allow you to specify how your data is formatted. Data can be delimited text (`TEXT`) or comma separated values (`CSV`) format. External data must be formatted correctly to be read by Greenplum Database. This section explains the format of data files expected by Greenplum Database.

- Row Separator
- Column Formatting
- Representing NULL Values
- Escape Characters
- Character Encoding

Row Separator

Greenplum Database expects rows of data to be separated by the `LF` character (Line feed, `0x0A`), `CR` (Carriage return, `0x0D`), or `CR` followed by `LF` (`CR+LF`, `0x0D 0x0A`). `LF` is the standard newline representation on UNIX or UNIX-like operating systems. Operating systems such as Windows or Mac OS 9 use `CR` or `CR+LF`. All of these representations of a newline are supported by Greenplum Database as a row delimiter. For more information, see "Importing and Exporting Fixed Width Data" on page 187.

Column Formatting

The default column or field delimiter is the horizontal `TAB` character (`0x09`) for text files and the comma character (`0x2C`) for CSV files. You can declare a single character delimiter using the `DELIMITER` clause of `COPY`, `CREATE EXTERNAL TABLE` or `gpload` when you define your data format. The delimiter character must appear between any two data value fields. Do not place a delimiter at the beginning or end of a row. For example, if the pipe character (|) is your delimiter:

```
data value 1|data value 2|data value 3
```

The following command shows the use of the pipe character as a column delimiter:

```
=# CREATE EXTERNAL TABLE ext_table (name text, date date)
   LOCATION ('gpfdist://<hostname>/filename.txt')
   FORMAT 'TEXT' (DELIMITER '|');
```

Representing NULL Values

`NULL` represents an unknown piece of data in a column or field. Within your data files you can designate a string to represent null values. The default string is `\N` (backslash-N) in `TEXT` mode, or an empty value with no quotations in `CSV` mode. You can also declare a different string using the `NULL` clause of `COPY`, `CREATE EXTERNAL TABLE` or `gpload` when defining your data format. For example, you can use an empty string if you do not want to distinguish nulls from empty strings. When using the Greenplum Database loading tools, any data item that matches the designated null string is considered a null value.

Escape Characters

There are two reserved characters that have special meaning to Greenplum Database:

- The designated delimiter character separates columns or fields in the data file.
- The newline character designates a new row in the data file.

If your data contains either of these characters, you must escape the character so that Greenplum 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.

Escaping in Text Formatted Files

By default, the escape character is a \ (backslash) for text-formatted files. You can declare a different escape character in the `ESCAPE` clause of `COPY`, `CREATE EXTERNAL TABLE` or `gpload`. If your escape character appears in your data, use it to escape itself.

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

- `backslash = \`
- `vertical bar = |`
- `exclamation point = !`

Your designated delimiter character is `|` (pipe character), and your designated escape character is `\` (backslash). The formatted row in your data file looks like this:

```
backslash = \\ | vertical bar = \| | exclamation point = !
```

Notice how the backslash character that is part of the data is escaped with another backslash character, and the pipe character that is part of the data is escaped with a backslash character.

You can use the escape character to escape octal and hexidecimal sequences. The escaped value is converted to the equivalent character when loaded into Greenplum Database. For example, to load the ampersand character (`&`), use the escape character to escape its equivalent hexidecimal (`\0x26`) or octal (`\046`) representation.

You can disable escaping in `TEXT`-formatted files using the `ESCAPE` clause of `COPY`, `CREATE EXTERNAL TABLE` or `gpload` as follows:

```
ESCAPE 'OFF'
```

This is useful for input data that contains many backslash characters, such as web log data.

Escape Characters in CSV Formatted Files

By default, the escape character is a " (double quote) for CSV-formatted files. If you want to use a different escape character, use the `ESCAPE` clause of `COPY`, `CREATE EXTERNAL TABLE` or `gpload` to declare a different escape character. In cases where your selected escape character is present in your data, you can use it to escape itself.

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

- `Free trip to A,B`
- `5.89`
- `Special rate "1.79"`

Your designated delimiter character is `,` (comma), and your designated escape character is " (double quote). The formatted row in your data file looks like this:

```
"Free trip to A,B","5.89","Special rate ""1.79"""
```

The data value with a comma character that is part of the data is enclosed in double quotes. The double quotes that are part of the data are escaped with a double quote even though the field value is enclosed in double quotes.

Embedding the entire field inside a set of double quotes guarantees preservation of leading and trailing whitespace characters:

```
"Free trip to A,B ","5.89 ","Special rate ""1.79"" "
```

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

Character Encoding

Character encoding systems consist of a code that pairs each character from a character set with something else, such as a sequence of numbers or octets, to facilitate data transmission and storage. Greenplum Database supports 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. Clients can use all supported character sets transparently, but a few are not supported for use within the server as a server-side encoding.

Data files must be in a character encoding recognized by Greenplum Database. See the Greenplum Database Reference Guide for the supported character sets. Data files that contain invalid or unsupported encoding sequences encounter errors when loading into Greenplum Database.

Note: On data files generated on a Microsoft Windows operating system, run the `dos2unix` system command to remove any Windows-only characters before loading into Greenplum Database.

Transforming XML Data

The Greenplum Database data loader *gpfdist* provides transformation features to load XML data into a table and to write data from the Greenplum Database to XML files. The following diagram shows *gpfdist* performing an XML transform.

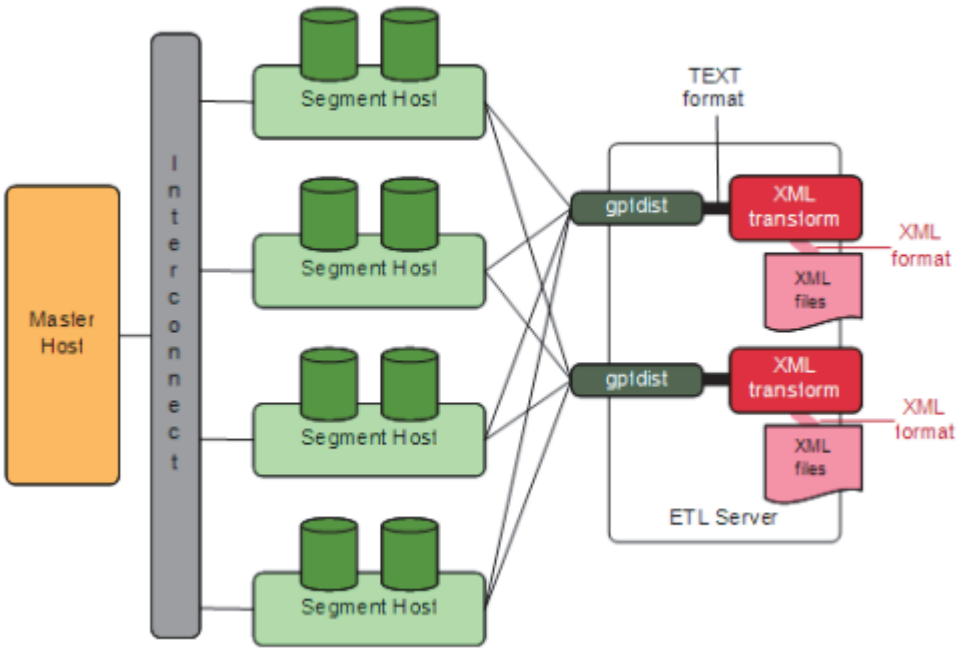


Figure 17.3 External Tables using XML Transformations

To load or extract XML data:

- Determine the Transformation Schema
- Write a Transform
- Write the gpfdist Configuration
- Load the Data
- Transfer and Store the Data

The first three steps comprise most of the development effort. The last two steps are straightforward and repeatable, suitable for production.

Determine the Transformation Schema

To prepare for the transformation project:

1. Determine the goal of the project, such as indexing data, analyzing data, combining data, and so on.
2. Examine the XML file and note the file structure and element names.
3. Choose the elements to import and decide if any other limits are appropriate.

For example, the following XML file, *prices.xml*, is a simple, short file that contains price records. Each price record contains two fields: an item number and a price.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<prices>
  <pricerecord>
    <itemnumber>708421</itemnumber>
    <price>19.99</price>
```

```
  </pricerecord>
  <pricerecord>
    <itemnumber>708466</itemnumber>
    <price>59.25</price>
  </pricerecord>
  <pricerecord>
    <itemnumber>711121</itemnumber>
    <price>24.99</price>
  </pricerecord>
</prices>
```

The goal is to import all the data into a Greenplum Database table with an integer `itemnumber` column and a decimal `price` column.

Write a Transform

The transform specifies what to extract from the data.You can use any authoring environment and language appropriate for your project. For XML transformations Greenplum suggests choosing a technology such as XSLT, Joost (STX), Java, Python, or Perl, based on the goals and scope of the project.

In the price example, the next step is to transform the XML data into a simple two-column delimited format.

```
708421|19.99
708466|59.25
711121|24.99
```

The following STX transform, called *input_transform.stx*, completes the data transformation.

```
<?xml version="1.0"?>
<stx:transform version="1.0"
   xmlns:stx="http://stx.sourceforge.net/2002/ns"
   pass-through="none">

  <!-- declare variables -->

  <stx:variable name="itemnumber"/>
  <stx:variable name="price"/>

  <!-- match and output prices as columns delimited by | -->

  <stx:template match="/prices/pricerecord">
    <stx:process-children/>
    <stx:value-of select="$itemnumber"/>
<stx:text>|</stx:text>
    <stx:value-of select="$price"/>     <stx:text>
</stx:text>
  </stx:template>

  <stx:template match="itemnumber">
    <stx:assign name="itemnumber" select="."/>
  </stx:template>
```

```
  <stx:template match="price">
    <stx:assign name="price" select="."/>
  </stx:template>

</stx:transform>
```

This STX transform declares two temporary variables, `itemnumber` and `price`, and the following rules.

1. When an element that satisfies the XPath expression `/prices/pricerecord` is found, examine the child elements and generate output that contains the value of the `itemnumber` variable, a `|` character, the value of the price variable, and a newline.
2. When an `<itemnumber>` element is found, store the content of that element in the variable `itemnumber`.
3. When a <price> element is found, store the content of that element in the variable `price`.

Write the gpfdist Configuration

The `gpfdist` configuration is specified as a YAML 1.1 document. It specifies rules that `gpfdist` uses to select a Transform to apply when loading or extracting data.

This example `gpfdist` configuration contains the following items:

- the `config.yaml` file defining `TRANSFORMATIONS`
- the `input_transform.sh` wrapper script, referenced in the `config.yaml` file
- the `input_transform.stx` joost transformation, called from `input_transform.sh`

Aside from the ordinary YAML rules, such as starting the document with three dashes (`---`), a `gpfdist` configuration must conform to the following restrictions:

1. a `VERSION` setting must be present with the value `1.0.0.1`.
2. a `TRANSFORMATIONS` setting must be present and contain one or more mappings.
3. Each mapping in the `TRANSFORMATION` must contain:
   - a `TYPE` with the value 'input' or 'output'
   - a `COMMAND` indicating how the transform is run.
4. Each mapping in the `TRANSFORMATION` can contain optional `CONTENT`, `SAFE`, and `STDERR` settings.

The following `gpfdist` configuration called `config.YAML` applies to the prices example. The initial indentation on each line is significant and reflects the hierarchical nature of the specification. The name `prices_input` in the following example will be referenced later when creating the table in SQL.

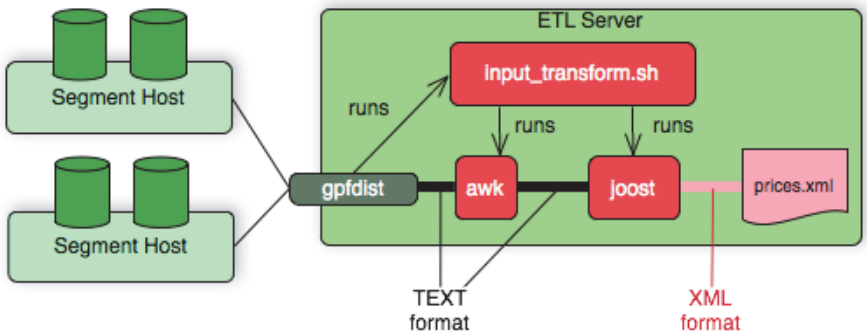
```
---
VERSION: 1.0.0.1

TRANSFORMATIONS:
  prices_input:
    TYPE:     input
    COMMAND:  /bin/bash input_transform.sh %filename%
```

The `COMMAND` setting uses a wrapper script called `input_transform.sh` with a `%filename%` placeholder. When `gpfdist` runs the `prices_input` transform, it invokes `input_transform.sh` with `/bin/bash` and replaces the `%filename%` placeholder with the path to the input file to transform. The wrapper script called `input_transform.sh` contains the logic to invoke the STX transformation and return the output.

If Joost is used, the Joost STX engine must be installed.

```
#!/bin/bash
# input_transform.sh - sample input transformation,
# demonstrating use of Java and Joost STX to convert XML into
# text to load into Greenplum Database.
# java arguments:
#   -jar joost.jar         joost STX engine
#   -nodecl                don't generate a <?xml?> declaration
#   $1                        filename to process
#   input_transform.stx    the STX transformation
#
# the AWK step eliminates a blank line joost emits at the end
java \
    -jar joost.jar \
    -nodecl \
    $1 \
    input_transform.stx \
 | awk 'NF>0
```

The `input_transform.sh` file uses the Joost STX engine with the AWK interpreter. The following diagram shows the process flow as `gpfdist` runs the transformation.



Load the Data

Create the tables with SQL statements based on the appropriate schema.

There are no special requirements for the Greenplum Database tables that hold loaded data. In the prices example, the following command creates the appropriate table.

```
CREATE TABLE prices (
  itemnumber integer,
  price       decimal
)
DISTRIBUTED BY (itemnumber);
```

Transfer and Store the Data

Use one of the following approaches to transform the data with `gpfdist`.

- `GPLOAD` supports only input transformations, but is easier to implement in many cases.
- `INSERT INTO SELECT FROM` supports both input and output transformations, but exposes more details.

Transforming with GPLOAD

Remove this section?

Transforming data with `GPLOAD` requires that the settings `TRANSFORM` and `TRANSFORM_CONFIG`. appear in the `INPUT` section of the `GPLOAD` control file. For more information about the syntax and placement of these settings in the `GPLOAD` control file, see the Greenplum Database Reference Guide.

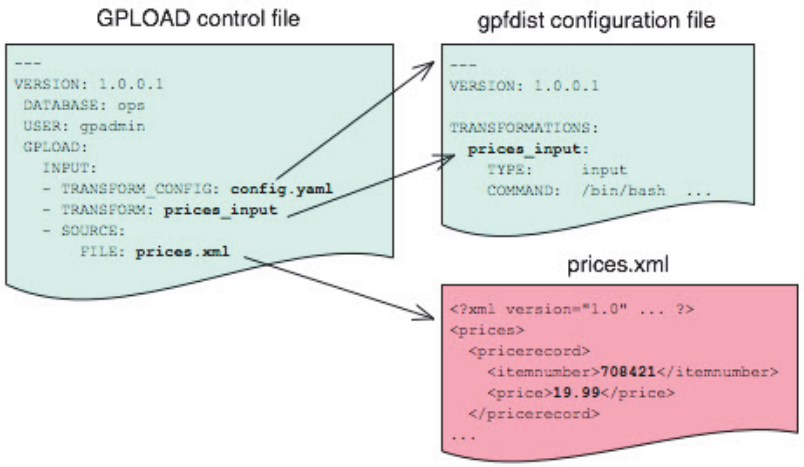
- `TRANSFORM_CONFIG` specifies the name of the `gpfdist` configuration file.
- The `TRANSFORM` setting indicates the name of the transformation that is described in the file named in `TRANSFORM_CONFIG`.

```
VERSION: 1.0.0.1
   DATABASE: ops
   USER: gpadmin
   GPLOAD:
      INPUT:
      - TRANSFORM_CONFIG: config.yaml
      - TRANSFORM: prices_input
      - SOURCE:
         FILE: prices.xml
```

The transformation name must appear in two places: in the `TRANSFORM` setting of the `gpfdist` configuration file and in the `TRANSFORMATIONS` section of the file named in the `TRANSFORM_CONFIG` section.

In the `GPLOAD` control file, the optional parameter `MAX_LINE_LENGTH` specifies the maximum length of a line in the XML transformation data that is passed to gpload.

The following diagram shows the relationships between the `GPLOAD` control file, the `gpfdist` configuration file, and the XML data file.



Transforming with INSERT INTO SELECT FROM

Specify the transformation in the `CREATE EXTERNAL TABLE` definition's `LOCATION` clause. For example, the transform is shown in bold in the following command. (Run `gpfdist` first, using the command `gpfdist -c config.yaml`).

```
CREATE READABLE EXTERNAL TABLE prices_readable (LIKE prices)
   LOCATION ('gpfdist://hostname:8080/prices.xml#transform=prices_input')
   FORMAT 'TEXT' (DELIMITER '|')
   LOG ERRORS INTO prices_errortable SEGMENT REJECT LIMIT 10;
```

In the command above, change `hostname` to your host name. `Prices_input` comes from the configuration file.

The following query loads data into the **`prices`** table.

```
INSERT INTO prices SELECT * FROM prices_readable;
```

Configuration File Format

The `gpfdist` configuration file uses the YAML 1.1 document format and implements a schema for defining the transformation parameters. The configuration file must be a valid YAML document.

The `gpfdist` program processes the document in order and uses indentation (spaces) to determine the document hierarchy and relationships of the sections to one another. The use of white space is significant. Do not use white space for formatting and do not use tabs.

The following is the basic structure of a configuration file.

```
---
VERSION: 1.0.0.1

TRANSFORMATIONS:
  transformation_name1:
     TYPE:      input | output
     COMMAND:   command
     CONTENT:   data | paths
     SAFE:      posix-regex
     STDERR:    server | console

  transformation_name2:
     TYPE:      input | output
     COMMAND:   command
     ...
```

VERSION

Required. The version of the `gpfdist` configuration file schema. The current version is 1.0.0.1.

TRANSFORMATIONS

Required. Begins the transformation specification section. A configuration file must have at least one transformation. When `gpfdist` receives a transformation request, it looks in this section for an entry with the matching transformation name.

TYPE

Required. Specifies the direction of transformation. Values are `input` or `output`.

- `input`: `gpfdist` treats the standard output of the transformation process as a stream of records to load into Greenplum Database.
- `output`: `gpfdist` treats the standard input of the transformation process as a stream of records from Greenplum Database to transform and write to the appropriate output.

COMMAND

Required. Specifies the command `gpfdist` will execute to perform the transformation.

For input transformations, `gpfdist` invokes the command specified in the `CONTENT` setting. The command is expected to open the underlying file(s) as appropriate and produce one line of `TEXT` for each row to load into Greenplum Database. The input transform determines whether the entire content should be converted to one row or to multiple rows.

For output transformations, `gpfdist` invokes this command as specified in the `CONTENT` setting. The output command is expected to open and write to the underlying file(s) as appropriate. The output transformation determines the final placement of the converted output.

CONTENT

Optional. The values are `data` and `paths`. The default value is `data`.

- When `CONTENT` specifies `data`, the text `%filename%` in the `COMMAND` section is replaced by the path to the file to read or write.
- When `CONTENT` specifies `paths`, the text `%filename%` in the `COMMAND` section is replaced by the path to the temporary file that contains the list of files to read or write.

The following is an example of a `COMMAND` section showing the text `%filename%` that is replaced.

```
COMMAND:  /bin/bash input_transform.sh %filename%
```

SAFE

Optional. A `POSIX` regular expression that the paths must match to be passed to the transformation. Specify `SAFE` when there is a concern about injection or improper interpretation of paths passed to the command. The default is no restriction on paths.

STDERR

Optional.The values are `server` and `console`.

This setting specifies how to handle standard error output from the transformation. The default, `server`, specifies that `gpfdist` will capture the standard error output from the transformation in a temporary file and send the first 8k of that file to Greenplum Database as an error message. The error message will appear as a SQL error. `Console` specifies that `gpfdist` does not redirect or transmit the standard error output from the transformation.

XML Transformation Examples

The following examples demonstrate the complete process for different types of XML data and STX transformations. Files and detailed instructions associated with these examples are in `demo/gpfdist_transform.tar.gz`. Read the README file in the *Before You Begin* section before you run the examples. The README file explains how to download the example data file used in the examples.

Example 1 - DBLP Database Publications (In demo Directory)

This example demonstrates loading and extracting database research data. The data is in the form of a complex XML file downloaded from the University of Washington. The DBLP information consists of a top level <dblp> element with multiple child elements such as <article>, <proceedings>, <mastersthesis>, <phdthesis>, and so on, where the child elements contain details about the associated publication. For example, the following is the XML for one publication.

```
<?xml version="1.0" encoding="UTF-8"?>
<mastersthesis key="ms/Brown92">
<author>Kurt P. Brown</author>
<title>PRPL: A Database Workload Language, v1.3.</title>
<year>1992</year>
<school>Univ. of Wisconsin-Madison</school>
</mastersthesis>
```

The goal is to import these `<mastersthesis>` and `<phdthesis>` records into the Greenplum Database. The sample document, *dblp.xml*, is about 130MB in size uncompressed. The input contains no tabs, so the relevent information can be converted into tab-delimited records as follows:

```
ms/Brown92 tab masters tab Kurt P. Brown tab PRPL: A Database
Workload Specification Language, v1.3. tab 1992 tab Univ. of
Wisconsin-Madison newline
```

With the columns:

```
key        text, -- e.g. ms/Brown92
type       text, -- e.g. masters
author     text, -- e.g. Kurt P. Brown
title      text, -- e.g. PRPL: A Database Workload Language, v1.3.
year       text, -- e.g. 1992
school     text, -- e.g. Univ. of Wisconsin-Madison
```

Then, load the data into Greenplum Database.

After the data loads, verify the data by extracting the loaded records as XML with an output transformation.

Example 2 - IRS MeF XML Files (In demo Directory)

This example demonstrates loading a sample IRS Modernized eFile tax return using a joost STX transformation. The data is in the form of a complex XML file.

The U.S. Internal Revenue Service (IRS) made a significant commitment to XML and specifies its use in its Modernized e-File (MeF) system. In MeF, each tax return is an XML document with a deep hierarchical structure that closely reflects the particular form of the underlying tax code.

XML, XML Schema and stylesheets play a role in their data representation and business workflow. The actual XML data is extracted from a ZIP file attached to a MIME “transmission file” message. For more information about MeF, see Modernized e-File (Overview) on the IRS web site.

The sample XML document, *RET990EZ_2006.xml*, is about 350KB in size with two elements:

- ReturnHeader
- ReturnData

The <ReturnHeader> contains general details about the tax return such as the taxpayer's name, the tax year of the return, and the preparer. The <ReturnData> contains multiple sections with specific details about the tax return and associated schedules.

The following is an abridged sample of the XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<Return returnVersion="2006v2.0"
   xmlns="http://www.irs.gov/efile"
   xmlns:efile="http://www.irs.gov/efile"
   xsi:schemaLocation="http://www.irs.gov/efile"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <ReturnHeader binaryAttachmentCount="1">
     <ReturnId>AAAAAAAAAAAAAAAAAAAA</ReturnId>
     <Timestamp>1999-05-30T12:01:01+05:01</Timestamp>
     <ReturnType>990EZ</ReturnType>
     <TaxPeriodBeginDate>2005-01-01</TaxPeriodBeginDate>
     <TaxPeriodEndDate>2005-12-31</TaxPeriodEndDate>
     <Filer>
       <EIN>011248772</EIN>
       ... more data ...
     </Filer>
     <Preparer>
       <Name>Percy Polar</Name>
       ... more data ...
     </Preparer>
     <TaxYear>2005</TaxYear>
   </ReturnHeader>
   ... more data ..
```

The goal is to import all the data into a Greenplum database. First, convert the XML document into text with newlines “escaped”, with two columns: `ReturnId` and a single column on the end for the entire MeF tax return. For example:

```
AAAAAAAAAAAAAAAAAAAA|<Return returnVersion="2006v2.0"...
```

Load the data into Greenplum Database.

Example 3 - WITSML™ Files (In demo Directory)

This example demonstrates loading sample data describing an oil rig using a joost STX transformation. The data is in the form a complex XML file downloaded from energistics.org.

The Wellsite Information Transfer Standard Markup Language (WITSML™) is an oil industry initiative to provide open, non-proprietary, standard interfaces for technology and software to share information among oil companies, service companies, drilling contractors, application vendors, and regulatory agencies. For more information about WITSML™, see http://www.witsml.org.

The oil rig information consists of a top level `<rigs>` element with multiple child elements such as `<documentInfo>`, `<rig>`, and so on. The following excerpt from the file shows the type of information in the `<rig>` tag.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="../stylesheets/rig.xsl" type="text/xsl"
media="screen"?>
<rigs
xmlns="http://www.witsml.org/schemas/131"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.witsml.org/schemas/131  ../obj_rig.xsd"
   version="1.3.1.1">
   <documentInfo>
          ... misc data ...
   </documentInfo>
   <rig uidWell="W-12" uidWellbore="B-01" uid="xr31">
        <nameWell>6507/7-A-42</nameWell>
        <nameWellbore>A-42</nameWellbore>
        <name>Deep Drill #5</name>
        <owner>Deep Drilling Co.</owner>
        <typeRig>floater</typeRig>
        <manufacturer>Fitsui Engineering</manufacturer>
        <yearEntService>1980</yearEntService>
        <classRig>ABS Class A1 M CSDU AMS ACCU</classRig>
        <approvals>DNV</approvals>
      ... more data ...
```

The goal is to import the information for this rig into Greenplum Database.

The sample document, *rig.xml*, is about 11KB in size. The input does not contain tabs so the relevant information can be converted into records delimited with a pipe (|).

```
W-12|6507/7-A-42|xr31|Deep Drill #5|Deep Drilling Co.|John
Doe|John.Doe@his-ISP.com|<?xml version="1.0" encoding="UTF-8" ...
```

With the columns:

```
well_uid    text,  -- e.g. W-12
well_name   text,  -- e.g. 6507/7-A-42
rig_uid     text,  -- e.g. xr31
rig_name    text,  -- e.g. Deep Drill #5
rig_owner   text,  -- e.g. Deep Drilling Co.
rig_contact text,  -- e.g. John Doe
rig_email   text,  -- e.g. John.Doe@his-ISP.com
doc         xml
```

Then, load the data into Greenplum Database.

Example Custom Data Access Protocol

The following is the API for the Greenplum Database custom data access protocol. The example protocol implementation gpextprotocal.c is written in C and shows how the API can be used. For information about accessing a custom data access protocol, see "Creating and Declaring a Custom Protocol" on page 171.

```
/* ---- Read/Write function API ------*/

CALLED_AS_EXTPROTOCOL(fcinfo)

EXTPROTOCOL_GET_URL(fcinfo)(fcinfo)
EXTPROTOCOL_GET_DATABUF(fcinfo)
EXTPROTOCOL_GET_DATALEN(fcinfo)
EXTPROTOCOL_GET_SCANQUALS(fcinfo)
EXTPROTOCOL_GET_USER_CTX(fcinfo)
EXTPROTOCOL_IS_LAST_CALL(fcinfo)
EXTPROTOCOL_SET_LAST_CALL(fcinfo)
EXTPROTOCOL_SET_USER_CTX(fcinfo, p)


/* ------ Validator function API ------*/

CALLED_AS_EXTPROTOCOL_VALIDATOR(fcinfo)

EXTPROTOCOL_VALIDATOR_GET_URL_LIST(fcinfo)
EXTPROTOCOL_VALIDATOR_GET_NUM_URLS(fcinfo)
EXTPROTOCOL_VALIDATOR_GET_NTH_URL(fcinfo, n)
EXTPROTOCOL_VALIDATOR_GET_DIRECTION(fcinfo)
```

Notes

The protocol corresponds to the example described in "Creating and Declaring a Custom Protocol" on page 171. The source code file name and shared object are `gpextprotocol.c` and `gpextprotocol.so`.

The protocol has the following properties:

- The name defined for the protocol is `myprot`.
- The protocol has the following simple form: the protocol name and a path, separated by `://`.

  `myprot://`*`path`*
- Three functions are implemented:
  - `myprot_import()` a read function
  - `myprot_export()` a write function
  - `myprot_validate_urls()` a validation function

  These functions are referenced in the `CREATE PROTOCOL` statement when the protocol is created and declared in the database.

The example implementation gpextprotocal.c uses `fopen()` and `fread()` to simulate a simple protocol that reads local files. In practice, however, the protocol would implement functionality such as a remote connection to some process over the network.

Installing the External Table Protocol

To use the example external table protocol, you use the C compiler `cc` to compile and link the source code to create a shared object that can be dynamically loaded by Greenplum Database. The commands to compile and link the source code on a Linux system are similar to this:

```
cc -fpic -c gpextprotocal.c
cc -shared -o gpextprotocal.so gpextprotocal.o
```

The option `-fpic` specifies creating position-independent code (PIC) and the `-c` option compiles the source code without linking and creates an object file. The object file needs to be created as position-independent code (PIC) so that it can be loaded at any arbitrary location in memory by Greenplum Database.

The flag `-shared` specifies creating a shared object (shared library) and the `-o` option specifies the shared object file name `gpextprotocal.so`. Refer to the GCC manual for more information on the `cc` options.

The header files that are declared as include files in `gpextprotocal.c` are located in subdirectories of `$GPHOME/include/postgresql/`.

For more information on compiling and linking dynamically-loaded functions and examples of compiling C source code to create a shared library on other operating systems, see the Postgres documentation at http://www.postgresql.org/docs/8.4/static/xfunc-c.html#DFUNC.

The manual pages for the C compiler `cc` and the link editor `ld` for your operating system also contain information on compiling and linking source code on your system.

The compiled code (shared object file) for the custom protocol must be placed in the same location on every host in your Greenplum Database array (master and all segments). This location must also be in the `LD_LIBRARY_PATH` so that the server can locate the files. It is recommended to locate shared libraries either relative to `$libdir` (which is located at `$GPHOME/lib`) or through the dynamic library path (set by the `dynamic_library_path` server configuration parameter) on all master segment instances in the Greenplum array. You can use the Greenplum Database utilities `gpssh` and `gpscp` to update segments.

gpextprotocal.c

```
#include "postgres.h"
#include "fmgr.h"
#include "funcapi.h"

#include "access/extprotocol.h"
#include "catalog/pg_proc.h"
#include "utils/array.h"
#include "utils/builtins.h"
#include "utils/memutils.h"
```

```
/* Our chosen URI format. We can change it however needed */
typedef struct DemoUri
{
   char     *protocol;
   char     *path;
}  DemoUri;

static DemoUri *ParseDemoUri(const char *uri_str);
static void FreeDemoUri(DemoUri* uri);


/* Do the module magic dance */
PG_MODULE_MAGIC;
PG_FUNCTION_INFO_V1(demoprot_export);
PG_FUNCTION_INFO_V1(demoprot_import);
PG_FUNCTION_INFO_V1(demoprot_validate_urls);

Datum demoprot_export(PG_FUNCTION_ARGS);
Datum demoprot_import(PG_FUNCTION_ARGS);
Datum demoprot_validate_urls(PG_FUNCTION_ARGS);


/* A user context that persists across calls. Can be declared in
any other way */

typedef struct {
  char    *url;
  char    *filename;
  FILE    *file;
} extprotocol_t;

/*
* The read function - Import data into GPDB.
*/
Datum
myprot_import(PG_FUNCTION_ARGS)
{
  extprotocol_t   *myData;
  char            *data;
  int             datlen;
  size_t          nread = 0;

  /* Must be called via the external table format manager */
  if (!CALLED_AS_EXTPROTOCOL(fcinfo))
    elog(ERROR, "myprot_import: not called by external
       protocol manager");

  /* Get our internal description of the protocol */
  myData = (extprotocol_t *) EXTPROTOCOL_GET_USER_CTX(fcinfo);

  if(EXTPROTOCOL_IS_LAST_CALL(fcinfo))
  {
    /* we're done receiving data. close our connection */
    if(myData && myData->file)
      if(fclose(myData->file))
        ereport(ERROR,
            (errcode_for_file_access(),
             errmsg("could not close file \"%s\": %m",
                 myData->filename)));
```

```
  PG_RETURN_INT32(0);
}
if (myData == NULL)
{
  /* first call. do any desired init */

  const char    *p_name = "myprot";
  DemoUri       *parsed_url;
  char          *url = EXTPROTOCOL_GET_URL(fcinfo);
  myData        = palloc(sizeof(extprotocol_t));
  myData->url   = pstrdup(url);
  parsed_url    = ParseDemoUri(myData->url);
  myData->filename = pstrdup(parsed_url->path);
  if(strcasecmp(parsed_url->protocol, p_name) != 0)
    elog(ERROR, "internal error: myprot called with a
        different protocol (%s)",
          parsed_url->protocol);
  FreeDemoUri(parsed_url);
  /* open the destination file (or connect to remote server in
       other cases) */
  myData->file = fopen(myData->filename, "r");
  if (myData->file == NULL)
    ereport(ERROR,
        (errcode_for_file_access(),
         errmsg("myprot_import: could not open file \"%s\"
            for reading: %m",
             myData->filename),
         errOmitLocation(true)));

  EXTPROTOCOL_SET_USER_CTX(fcinfo, myData);
}
/* ========================================
 *          DO THE IMPORT
 * ======================================== */
data    = EXTPROTOCOL_GET_DATABUF(fcinfo);
datlen  = EXTPROTOCOL_GET_DATALEN(fcinfo);
/* read some bytes (with fread in this example, but normally
in some other method over the network) */
if(datlen > 0)
{
  nread = fread(data, 1, datlen, myData->file);
  if (ferror(myData->file))
    ereport(ERROR,
        (errcode_for_file_access(),
         errmsg("myprot_import: could not write to file
           \"%s\": %m",
             myData->filename)));
}
```

```
  PG_RETURN_INT32((int)nread);
}

/*
 * Write function - Export data out of GPDB
 */
Datum
myprot_export(PG_FUNCTION_ARGS)
{
  extprotocol_t  *myData;
  char           *data;
  int            datlen;
  size_t         wrote = 0;

  /* Must be called via the external table format manager */

  if (!CALLED_AS_EXTPROTOCOL(fcinfo))
    elog(ERROR, "myprot_export: not called by external
       protocol manager");

  /* Get our internal description of the protocol */
  myData = (extprotocol_t *) EXTPROTOCOL_GET_USER_CTX(fcinfo);

  if(EXTPROTOCOL_IS_LAST_CALL(fcinfo))
  {
    /* we're done sending data. close our connection */
    if(myData && myData->file)
      if(fclose(myData->file))
        ereport(ERROR,
            (errcode_for_file_access(),
             errmsg("could not close file \"%s\": %m",
                 myData->filename)));

    PG_RETURN_INT32(0);
  }

  if (myData == NULL)
  {
    /* first call. do any desired init */

    const char *p_name = "myprot";
    DemoUri    *parsed_url;
    char       *url = EXTPROTOCOL_GET_URL(fcinfo);

    myData           = palloc(sizeof(extprotocol_t));

    myData->url      = pstrdup(url);
    parsed_url       = ParseDemoUri(myData->url);
    myData->filename = pstrdup(parsed_url->path);

    if(strcasecmp(parsed_url->protocol, p_name) != 0)
      elog(ERROR, "internal error: myprot called with a
         different protocol (%s)",
            parsed_url->protocol);

    FreeDemoUri(parsed_url);

    /* open the destination file (or connect to remote server in
       other cases) */
```

```
    myData->file = fopen(myData->filename, "a");

    if (myData->file == NULL)
      ereport(ERROR,
          (errcode_for_file_access(),
           errmsg("myprot_export: could not open file \"%s\"
             for writing: %m",
               myData->filename),
           errOmitLocation(true)));

    EXTPROTOCOL_SET_USER_CTX(fcinfo, myData);
  }

  /* ========================================
   *      DO THE EXPORT
   * ======================================== */

  data   = EXTPROTOCOL_GET_DATABUF(fcinfo);
  datlen   = EXTPROTOCOL_GET_DATALEN(fcinfo);

  if(datlen > 0)
  {
    wrote = fwrite(data, 1, datlen, myData->file);
    if (ferror(myData->file))
      ereport(ERROR,
          (errcode_for_file_access(),
           errmsg("myprot_import: could not read from file
              \"%s\": %m",
               myData->filename)));
  }

  PG_RETURN_INT32((int)wrote);
}

Datum
myprot_validate_urls(PG_FUNCTION_ARGS)
{
  List         *urls;
  int          nurls;
  int          i;
  ValidatorDirection  direction;

  /* Must be called via the external table format manager */
  if (!CALLED_AS_EXTPROTOCOL_VALIDATOR(fcinfo))
    elog(ERROR, "myprot_validate_urls: not called by external
       protocol manager");

  nurls       = EXTPROTOCOL_VALIDATOR_GET_NUM_URLS(fcinfo);
  urls        = EXTPROTOCOL_VALIDATOR_GET_URL_LIST(fcinfo);
  direction   = EXTPROTOCOL_VALIDATOR_GET_DIRECTION(fcinfo);

  /*
   * Dumb example 1: search each url for a substring
   * we don't want to be used in a url. in this example
   * it's 'secured_directory'.
   */
  for (i = 1 ; i <= nurls ; i++)
```

```
  {
    char *url = EXTPROTOCOL_VALIDATOR_GET_NTH_URL(fcinfo, i);

    if (strstr(url, "secured_directory") != 0)
    {
            ereport(ERROR,
                    (errcode(ERRCODE_PROTOCOL_VIOLATION),
                     errmsg("using 'secured_directory' in a url
                       isn't allowed ")));
    }
  }

  /*
   * Dumb example 2: set a limit on the number of urls
   * used. In this example we limit readable external
   * tables that use our protocol to 2 urls max.
   */
  if(direction == EXT_VALIDATE_READ && nurls > 2)
  {
        ereport(ERROR,
                (errcode(ERRCODE_PROTOCOL_VIOLATION),
                 errmsg("more than 2 urls aren't allowed in
this protocol ")));
  }

  PG_RETURN_VOID();
}

/* --- utility functions --- */

static
DemoUri *ParseDemoUri(const char *uri_str)
{
  DemoUri *uri = (DemoUri *) palloc0(sizeof(DemoUri));
  int     protocol_len;

   uri->path = NULL;
   uri->protocol = NULL;

  /*
   * parse protocol
   */
  char *post_protocol = strstr(uri_str, "://");

  if(!post_protocol)
  {
    ereport(ERROR,
        (errcode(ERRCODE_SYNTAX_ERROR),
         errmsg("invalid protocol URI \'%s\'", uri_str),
         errOmitLocation(true)));
  }

  protocol_len = post_protocol - uri_str;
  uri->protocol = (char *)palloc0(protocol_len + 1);
  strncpy(uri->protocol, uri_str, protocol_len);

  /* make sure there is more to the uri string */
  if (strlen(uri_str) <= protocol_len)
```

```
    ereport(ERROR,
        (errcode(ERRCODE_SYNTAX_ERROR),
    errmsg("invalid myprot URI \'%s\' : missing path",
      uri_str),
    errOmitLocation(true)));

  /* parse path */
  uri->path = pstrdup(uri_str + protocol_len + strlen("://"));

  return uri;
}

static
void FreeDemoUri(DemoUri *uri)
{
  if (uri->path)
    pfree(uri->path);
  if (uri->protocol)
    pfree(uri->protocol);

  pfree(uri);
}
```

18. Hadoop Distributed File System

Greenplum Database can be extended to work with the Hadoop distributed file system (HDFS). Greenplum supports several Hadoop distributions. See the *Greenplum Database Release Notes* for the supported distributions.

You access HDFS data with the CREATE EXTERNAL TABLE command and the gphdfs protocol.

This chapter contains the following information:

- About HDFS
- About the gphdfs protocol
- Examples
- References
  See "Loading Data into Greenplum Database" in the *Greenplum Database Database Administrator Guide* for information about using external tables to load and unload data.

gphdfs protocol

This protocol specifies a path that can contain wild card characters on a Hadoop Distributed File System. `TEXT` and custom formats are allowed for `HDFS` files.

When Greenplum links with `HDFS` files, all the data is read in parallel from the `HDFS` data nodes into the Greenplum segments for rapid processing. Greenplum determines the connections between the segments and nodes.

Each Greenplum segment reads one set of Hadoop data blocks. For writing, each Greenplum segment writes only the data contained on it.

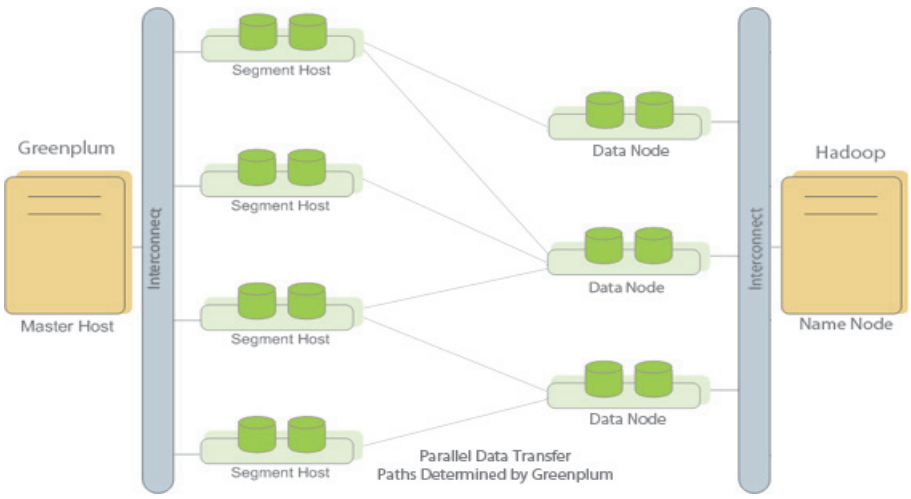


Figure 18.1 External Table Located on a Hadoop Distributed File System

The FORMAT clause describes the format of the external table files. Valid file formats are similar to the formatting options available with the PostgreSQL COPY command and user-defined formats for the gphdfs protocol.

- Delimited text (TEXT) for all protocols.
- Comma separated values (CSV) format for gpfdist, gpfdists, 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 Greenplum Database reads the data in the external file correctly.

The gphdfs protocol requires a one-time setup. See "One-time HDFS Protocol Installation".

Using Hadoop Distributed File System (HDFS) Tables

Greenplum Database leverages the parallel architecture of a Hadoop Distributed File System to read and write data files efficiently with the gphdfs protocol. There are three components to using HDFS:

- One-time setup
- Grant privileges for the HDFS protocol
- Specify Hadoop Distributed File System data in an external table definition

One-time HDFS Protocol Installation

Install and configure Hadoop for use with gphdfs as follows.

1. Install Java 1.6 or later on **all** segments.
2. Greenplum Database includes the following Greenplum HD targets:
   - The Greenplum HD target (gphd-1.0, gphd-1.1, and gphd-1.2)
     The default target. To use any other target, set the Server Configuration Parameter to one of the values shown in Table 18.1, "Server Configuration Parameters for Hadoop Targets" on page 213.
   - The Greenplum MR target for MR v. 1.0 or 1.2 (gpmr-1.0 and gpmr-1.2)
     If you are using gpmr-1.0 or gpmr-1.2, install the Greenplum MR client program. See the *Setting up the Client* section of the *Greenplum HD MR Administrator Guide* available from Support Zone.
   - The Greenplum Cloudera Hadoop Connect (cdh3u2 and cdh4.1).
     For CDH 4.1, only CDH4 with MRv1 is supported.
3. After installation, ensure that the Greenplum system user (gpadmin) has read and execute access to the Hadoop libraries or to the Greenplum MR client.
4. Set the following environment variables on all segments.
   JAVA_HOME – the Java home directory

HADOOP_HOME – the Hadoop home directory

For example, add lines such as the following to the gpadmin user's .bashrc profile. **Note:** The variables must be set in .bashrc because Greenplum Database always uses SSH.

```
export JAVA_HOME=/opt/jdk1.6.0_21
export HADOOP_HOME=/bin/hadoop
```

5. Set the following server configuration Pprameters.

Table 18.1 Server Configuration Parameters for Hadoop Targets

Configuration Parameter	Description	Default Value	Set Classifications
gp_hadoop_target_version	The Hadoop target. Choose one of the following. gphd-1.0 gphd-1.1 gphd-1.2 gpmr-1.0 gpmr-1.2 cdh3u2 cdh4.1	gphd-1.1	master session reload
gp_hadoop_home	Same value as HADOOP_HOME.	NULL	master session reload

6. Restart the database.

Grant Privileges for the HDFS Protocol

To enable privileges required to create external tables that access files on HDFS:

1. Grant the following privileges on gphdfs to the owner of the external table.
   - Grant SELECT privileges to enable creating readable external tables on HDFS.
   - Grant INSERT privileges to enable creating writable external tables on HDFS.

   Use the GRANT command to grant read privileges (SELECT) and, if needed, write privileges (INSERT) on HDFS to the Greenplum system user (gpadmin).

   ```
   GRANT INSERT ON PROTOCOL gphdfs TO gpadmin;
   ```

2. Greenplum Database uses Greenplum OS credentials to connect to HDFS. Grant read privileges and, if needed, write privileges to HDFS to the Greenplum administrative user (gpadmin OS user).

Specify HDFS Data in an External Table Definition

`CREATE EXTERNAL TABLE's LOCATION` option for Hadoop files has the following format:

```
LOCATION ('gphdfs://hdfs_host[:port]/path/filename.txt')
```

- For any connector except `gpmr-1.0-gnet-1.0.0.1`, specify a name node port. Do not specify a port with the `gpmr-1.0-gnet-1.0.0.1` connector.

Restrictions for `HDFS` files are as follows.

- You can specify one path for a readable external table with `gphdfs`. Wildcard characters are allowed. If you specify a directory, the default is all files in the directory.

  You can specify only a directory for writable external tables.
- Format restrictions are as follows.
  - `TEXT` format is allowed for readable and writable external tables.
  - Only the `gphdfs_import` formatter is allowed for readable external tables with a custom format.
  - Only the `gphdfs_export` formatter is allowed for writable external tables with a custom format.
- You can set compression only for writable external tables. Compression settings are automatic for readable external tables.

Setting Compression Options for Hadoop Writable External Tables

Compression options for Hadoop Writable External Tables use the form of a URI query and begin with a question mark. Specify multiple compression options with an ampersand (`&`).

Table 18.2 Compression Options

Compression Option	Values	Default Value
compress	`true` or `false`	`false`
compression_type	`BLOCK` or `RECORD`	`RECORD`
codec	Codec class name	`GzipCodec` for text format and `DefaultCodec` for `gphdfs_export` format.

Place compression options in the query portion of the URI.

HDFS Readable and Writable External Table Examples

The following code defines a readable external table for an HDFS file named *filename.txt* on port 8081.

```
=# CREATE EXTERNAL TABLE ext_expenses ( name text,
   date date,  amount float4, category text, desc1 text )
   LOCATION ('gphdfs://hdfshost-1:8081/data/filename.txt')
   FORMAT 'TEXT' (DELIMITER ',');
```

Note: Omit the port number when using the `gpmr-1.0-gnet-1.0.0.1` connector.

The following code defines a set of readable external tables that have a custom format located in the same HDFS directory on port 8081.

```
=# CREATE EXTERNAL TABLE ext_expenses
   LOCATION ('gphdfs://hdfshost-1:8081/data/custdat*.dat')
   FORMAT 'custom' (formatter='gphdfs_import');
```

Note: Omit the port number when using the `gpmr-1.0-gnet-1.0.0.1` connector.

The following code defines an HDFS directory for a writable external table on port 8081 with all compression options specified.

```
=# CREATE WRITABLE EXTERNAL TABLE ext_expenses
   LOCATION    ('gphdfs://hdfshost-1:8081/data/
   ?compress=true&compression_type=RECORD
   &codec=org.apache.hadoop.io.compress.DefaultCodec')
   FORMAT 'custom' (formatter='gphdfs_export');
```

Note: Omit the port number when using the `gpmr-1.0-gnet-1.0.0.1` connector.

Because the previous code uses the default compression options for `compression_type` and `codec`, the following command is equivalent.

```
=# CREATE WRITABLE EXTERNAL TABLE ext_expenses
   LOCATION    ('gphdfs://hdfshost-1:8081/data?compress=true')
   FORMAT 'custom' (formatter='gphdfs_export');
```

Note: Omit the port number when using the `gpmr-1.0-gnet-1.0.0.1` connector.

Reading and Writing Custom-Formatted HDFS Data

Use MapReduce and the `CREATE EXTERNAL TABLE` command to read and write data with custom formats on HDFS.

To read custom-formatted data:

1. Author and run a MapReduce job that creates a copy of the data in a format accessible to Greenplum Database.
2. Use `CREATE EXTERNAL TABLE` to read the data into Greenplum Database.

See "Example 1 - Read Custom-Formatted Data from HDFS" on page 216.

To write custom-formatted data:

1. Write the data.
2. Author and run a MapReduce program to convert the data to the custom format and place it on the Hadoop Distributed File System.

See "Example 2 - Write Custom-Formatted Data from Greenplum Database to HDFS" on page 218.

MapReduce is written in Java. Greenplum provides Java APIs for use in the MapReduce code. The Javadoc is available in the `$GPHOME/docs` directory. To view the Javadoc, expand the file gphd-xnet-1.0.0.0-javadoc.tgz and open `index.html`. The Javadoc documents the following packages:

```
com.emc.greenplum.gpdb.hadoop.io
com.emc.greenplum.gpdb.hadoop.mapred
com.emc.greenplum.gpdb.hadoop.mapreduce.lib.input
com.emc.greenplum.gpdb.hadoop.mapreduce.lib.output
```

The HDFS cross-connect packages contain the Java library, which contains the packages `GPDBWritable`, `GPDBInputFormat`, and `GPDBOutputFormat`. The Java packages are available in `$GPHOME/lib/hadoop`. For Greenplum HD, compile and run the MapReduce job with `gphd_xnet_1.0.0.0.jar`. For Greenplum MR, compile and run the MapReduce job with `gpmr_xnet_1.0.0.0.jar`.

To make the Java library available to all Hadoop users, the Hadoop cluster administrator should place the corresponding `gphdfs` connector jar in the `$HADOOP_HOME/lib` directory and restart the job tracker. If this is not done, a Hadoop user can still use the `gphdfs` connector jar; but with the *distributed cache* technique.

Example —Hadoop file server (gphdfs)

```
=# CREATE WRITABLE EXTERNAL TABLE unload_expenses
    ( LIKE expenses )
    LOCATION ('gphdfs://hdfslhost-1:8081/path')
  FORMAT 'TEXT' (DELIMITER ',')
  DISTRIBUTED BY (exp_id);
```

You can only specify a directory for a writable external table with the `gphdfs` protocol. (You can only specify one file for a readable external table with the `gphdfs` protocol)

Note: The default port number is 9000.

Example 1 - Read Custom-Formatted Data from HDFS

The sample code makes the following assumptions.

- The data is contained in HDFS directory `/demo/data/temp` and the name node is running on port 8081.
- This code writes the data in Greenplum Database format to `/demo/data/MRTest1` on HDFS.
- The data contains the following columns, in order.
  1. A long integer
  2. A Boolean
  3. A text string

Sample MapReduce Code

```
import com.emc.greenplum.gpdb.hadoop.io.GPDBWritable;
import com.emc.greenplum.gpdb.hadoop.mapreduce.lib.input.GPDBInputFormat;
import
com.emc.greenplum.gpdb.hadoop.mapreduce.lib.output.GPDBOutputFormat;

import java.io.*;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.output.*;
import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.util.*;

public class demoMR {
    /*
     * Helper routine to create our generic record. This section shows the
     * format of the data. Modify as necessary.
     */
     public static GPDBWritable generateGenericRecord() throws
      IOException{
        int[] colType = new int[3];
        colType[0] = GPDBWritable.BIGINT;
        colType[1] = GPDBWritable.BOOLEAN;
        colType[2] = GPDBWritable.VARCHAR;

    /*
     * This section passes the values of the data. Modify as necessary.
     */
        GPDBWritable gw = new GPDBWritable(colType);
        gw.setLong   (0, (long)12345);
        gw.setBoolean(1, true);
        gw.setString (2, "abcdef");

        return gw;
    }

    /*
     * DEMO Map/Reduce class test1
     * -- Regardless of the input, this section dumps the generic record
     *    into GPDBFormat/
     */
    public static class Map_test1 extends Mapper<LongWritable, Text,
     LongWritable, GPDBWritable> {
    private LongWritable word = new LongWritable(1);

    public void map(LongWritable key, Text value, Context context) throws
     IOException {
        try {
              GPDBWritable gw = generateGenericRecord();
              context.write(word, gw);
        } catch (Exception e) { throw new IOException (e.getMessage()); }
    }
  }
```

```
public static void runTest1() throws Exception{
Configuration conf = new Configuration(true);
   Job job = new Job(conf, "test1");

   job.setJarByClass(demoMR.class);

   job.setInputFormatClass(TextInputFormat.class);

   job.setOutputKeyClass   (LongWritable.class);
   job.setOutputValueClass (GPDBWritable.class);
   job.setOutputFormatClass(GPDBOutputFormat.class);

   job.setMapperClass(Map_test1.class);

   FileInputFormat.setInputPaths (job, new Path("/demo/data/tmp"));
   GPDBOutputFormat.setOutputPath(job, new Path("/demo/data/MRTest1"));

   job.waitForCompletion(true);
}
```

Run CREATE EXTERNAL TABLE

The Hadoop location corresponds to the output path in the MapReduce job.

```
=# CREATE EXTERNAL TABLE demodata
   LOCATION ('gphdfs://hdfshost-1:8081/demo/data/MRTest1')
   FORMAT 'custom' (formatter='gphdfs_import');
```

Note: Omit the port number when using the `gpmr-1.0-gnet-1.0.0.1` connector.

Example 2 - Write Custom-Formatted Data from Greenplum Database to HDFS

The sample code makes the following assumptions.

- The data in Greenplum Database format is located on the Hadoop Distributed File System on `/demo/data/writeFromGPDB_42` on port 8081.
- This code writes the data to `/demo/data/MRTest2` on port 8081.

1. Run a SQL command to create the writable table.

```
=# CREATE WRITABLE EXTERNAL TABLE demodata
   LOCATION ('gphdfs://hdfshost-1:8081/demo/data/MRTest2')
   FORMAT 'custom' (formatter='gphdfs_export');
```

2. Author and run code for a MapReduce job. Use the same import statements shown in "Example 1 - Read Custom-Formatted Data from HDFS" on page 216.

Note: Omit the port number when using the `gpmr-1.0-gnet-1.0.0.1` connector.

MapReduce Sample Code

```
/*
 * DEMO Map/Reduce class test2
 * -- Convert GPDBFormat back to TEXT
 */

public static class Map_test2 extends Mapper<LongWritable, GPDBWritable,
       Text, NullWritable> {
```

```
  public void map(LongWritable key, GPDBWritable value, Context context )
         throws IOException {
    try {
      context.write(new Text(value.toString()), NullWritable.get());
    } catch (Exception e) { throw new IOException (e.getMessage()); }
  }
}

public static void runTest2() throws Exception{
Configuration conf = new Configuration(true);
    Job job = new Job(conf, "test2");

    job.setJarByClass(demoMR.class);

    job.setInputFormatClass(GPDBInputFormat.class);

    job.setOutputKeyClass  (Text.class);
    job.setOutputValueClass(NullWritable.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    job.setMapperClass(Map_test2.class);

    GPDBInputFormat.setInputPaths (job,
     new Path("/demo/data/writeFromGPDB_42"));
    GPDBOutputFormat.setOutputPath(job, new Path("/demo/data/MRTest2"));

    job.waitForCompletion(true);
}
```

A. Greenplum MapReduce

Greenplum Database supports MapReduce jobs with the Greenplum utility `gpmapreduce`. This chapter describes the document format and schema for defining Greenplum MapReduce jobs.

MapReduce is a programming model developed by Google for processing and generating large data sets on an array of commodity servers. Greenplum MapReduce allows programmers who are familiar with the MapReduce model to write map and reduce functions and submit them to the Greenplum Database parallel engine for processing.

To enable Greenplum to process MapReduce functions, the define the functions in a document, then pass the document to the Greenplum MapReduce utility, `gpmapreduce`, for execution by the Greenplum Database parallel engine. The Greenplum Database system distributes the input data, executes the program across a set of machines, handles machine failures, and manages the required inter-machine communication.

See the *Greenplum Database Utility Guide* for information about the `gpmapreduce`.

See "Reading and Writing Custom-Formatted HDFS Data" in Chapter 18, "Hadoop Distributed File System" for examples of MapReduce Java applications.

MapReduce is written in Java. Greenplum Database provides Java APIs for use in the MapReduce code. The Javadoc is available in the `$GPHOME/docs/javadoc` directory. To view the Javadoc, untar the file `gnet-1.1-javadoc.tar` and open `index.html`.

Greenplum MapReduce Document Format

This section explains some basics of the Greenplum MapReduce document format to help you get started creating your own Greenplum MapReduce documents. Greenplum uses the YAML 1.1 document format and then implements its own schema for defining the various steps of a MapReduce job.

All Greenplum MapReduce files must first declare the version of the YAML specification they are using. After that, three dashes (---) denote the start of a document, and three dots (. . .) indicate the end of a document without starting a new one. Comment lines are prefixed with a pound symbol (#). It is possible to declare multiple Greenplum MapReduce documents in the same file:

```
%YAML 1.1
---
# Begin Document 1
# ...
---
# Begin Document 2
# ...
```

Within a Greenplum MapReduce document, there are three basic types of data structures or *nodes*: *scalars*, *sequences* and *mappings*.

A *scalar* is a basic string of text indented by a space. If you have a scalar input that spans multiple lines, a preceding pipe (|) denotes a *literal* style, where all line breaks are significant. Alternatively, a preceding angle bracket (>) folds a single line break to a space for subsequent lines that have the same indentation level. If a string contains characters that have reserved meaning, the string must be quoted or the special character must be escaped with a backslash (\ ).

```
# Read each new line literally
somekey: |
   this value contains two lines
   and each line is read literally
# Treat each new line as a space
anotherkey: >
   this value contains two lines
   but is treated as one continuous line
# This quoted string contains a special character
ThirdKey: "This is a string: not a mapping"
```

A *sequence* is a list with each entry in the list on its own line denoted by a dash and a space (-). Alternatively, you can specify an inline sequence as a comma-separated list within square brackets. A sequence provides a set of data and gives it an order. When you load a list into the Greenplum MapReduce program, the order is kept.

```
# list sequence
- this
- is
- a list
- with
- five scalar values
# inline sequence
[this, is, a list, with, five scalar values]
```

A *mapping* is used to pair up data values with identifiers called *keys*. Mappings use a colon and space (:) for each `key: value` pair, or can also be specified inline as a comma-separated list within curly braces. The *key* is used as an index for retrieving data from a mapping.

```
# a mapping of items
title: War and Peace
author: Leo Tolstoy
date: 1865
# same mapping written inline
{title: War and Peace, author: Leo Tolstoy, date: 1865}
```

Keys are used to associate meta information with each node and specify the expected node type (*scalar*, *sequence* or *mapping*). See "Greenplum MapReduce Document Schema" on page 223 for the keys expected by the Greenplum MapReduce program.

The Greenplum MapReduce program processes the nodes of a document in order and uses indentation (spaces) to determine the document hierarchy and the relationships of the nodes to one another. The use of white space is significant. White space should not be used simply for formatting purposes, and tabs should not be used at all.

Greenplum MapReduce Document Schema

Greenplum MapReduce uses the YAML document framework and implements its own YAML schema. The basic structure of a Greenplum MapReduce document is:

```
%YAML 1.1
---
VERSION: 1.0.0.2
DATABASE: dbname
USER: db_username
HOST: master_hostname
PORT: master_port

DEFINE:

  - INPUT:
     NAME: input_name
     FILE:
      - hostname:/path/to/file
     SSL: true | false
     CERTIFICATES_PATH: /path/to/certificates
     GPFDIST:
       - hostname:port:/file_pattern
     TABLE: table_name
     QUERY: SELECT_statement
     EXEC: command_string
     COLUMNS:
       - field_name data_type
     FORMAT: TEXT | CSV
     DELIMITER: delimiter_character
     ESCAPE: escape_character
     NULL: null_string
     QUOTE: csv_quote_character
     ERROR_LIMIT: integer
     ENCODING: database_encoding
```

```
- OUTPUT:
    NAME: output_name
    FILE: file_path_on_client
    TABLE: table_name
    KEYS:
      - column_name
    MODE: REPLACE | APPEND

- MAP:
    NAME: function_name
    FUNCTION: function_definition
    LANGUAGE: perl | python | java | c
    LIBRARY: /path/filename.so
    PARAMETERS:
      - name type
    RETURNS:
      - name type
    OPTIMIZE: STRICT IMMUTABLE
    MODE: SINGLE | MULTI

- TRANSITION | CONSOLIDATE | FINALIZE:
    NAME: function_name
    FUNCTION: function_definition
    LANGUAGE: perl | python | java | c
    LIBRARY: /path/filename.so
    PARAMETERS:
      - name type
    RETURNS:
      - name type
    OPTIMIZE: STRICT IMMUTABLE
    MODE: SINGLE | MULTI
```

```
  - REDUCE:
     NAME: reduce_job_name
     TRANSITION: transition_function_name
     CONSOLIDATE: consolidate_function_name
     FINALIZE: finalize_function_name
     INITIALIZE: value
     KEYS:
       - key_name

  - TASK:
     NAME: task_name
     SOURCE: input_name
     MAP: map_function_name
     REDUCE: reduce_function_name

EXECUTE:

  - RUN:
     SOURCE: input_or_task_name
     TARGET: output_name
     MAP: map_function_name
     REDUCE: reduce_function_name
...
```

VERSION

Required. The version of the Greenplum MapReduce YAML specification. Current versions are 1.0.0.1.

DATABASE

Optional. Specifies which database in Greenplum to connect to. If not specified, defaults to the default database or `$PGDATABASE` if set.

USER

Optional. Specifies which database role to use to connect. If not specified, defaults to the current user or `$PGUSER` if set. You must be a Greenplum superuser to run functions written in untrusted Python and Perl. Regular database users can run functions written in trusted Perl. You also must be a database superuser to run MapReduce jobs that contain FILE, GPFDIST or EXEC input types.

HOST

Optional. Specifies Greenplum master host name. If not specified, defaults to localhost or `$PGHOST` if set.

PORT

Optional. Specifies Greenplum master port. If not specified, defaults to 5432 or `$PGPORT` if set.

DEFINE

Required. A sequence of definitions for this MapReduce document. The `DEFINE` section must have at least one `INPUT` definition.

INPUT

Required. Defines the input data. Every MapReduce document must have at least one input defined. Multiple input definitions are allowed in a document, but each input definition can specify only one of these access types: a file, a `gpfdist` file distribution program, a table in the database, an SQL command, or an operating system command. See the *Greenplum Database Utility Guide* for information about `gpfdist`.

NAME

A name for this input. Names must be unique with regards to the names of other objects in this MapReduce job (such as map function, task, reduce function and output names). Also, names cannot conflict with existing objects in the database (such as tables, functions or views).

FILE

A sequence of one or more input files in the format: *`seghostname:/path/to/filename`*. You must be a Greenplum Database superuser to run MapReduce jobs with `FILE` input. The file must reside on a Greenplum segment host.

SSL

Optional. Specifies usage of SSL encryption.

CERTIFICATES_PATH

Required when SSL is true; otherwise, optional.The certificate path is required. 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 in `certificate_path`.

GPFDIST

A sequence of one or more running `gpfdist` file distribution programs in the format: *`hostname`*`[:`*`port`*`]/`*`file_pattern`*. You must be a Greenplum Database superuser to run MapReduce jobs with `GPFDIST` input, unless the server configuration parameter `gp_external_grant_privileges` is set to `on`.

TABLE

The name of an existing table in the database.

QUERY

An SQL `SELECT` command to run within the database.

EXEC

An operating system command to run on the Greenplum segment hosts. The command is run by all segment instances in the system by default. For example, if you have four segment instances per segment host, the command will be run four times on each host. You must be a Greenplum Database superuser to run MapReduce jobs with `EXEC` input and the server configuration parameter `gp_external_enable_exec` is set to `on`.

COLUMNS

Optional. Columns are specified as: *`column_name`* `[`*`data_type`*`]`. If not specified, the default is `value text`. The `DELIMITER` character is what separates two data value fields (columns). A row is determined by a line feed character (`0x0a`).

FORMAT

Optional. Specifies the format of the data - either delimited text (`TEXT`) or comma separated values (`CSV`) format. If the data format is not specified, defaults to `TEXT`.

DELIMITER

Optional for `FILE`, `GPFDIST` and `EXEC` inputs. Specifies a single character that separates data values. The default is a tab character in `TEXT` mode, a comma in `CSV` mode.The delimiter character must only appear between any two data value fields. Do not place a delimiter at the beginning or end of a row.

ESCAPE

Optional for `FILE`, `GPFDIST` and `EXEC` inputs. Specifies the single character that is used for C escape sequences (such as `\n`,`\t`,`\100`, and so on) and for escaping data characters that might otherwise be taken as row or column delimiters. Make sure to choose an escape character that is not used anywhere in your actual column data. The default escape character is a \ (backslash) for text-formatted files and a " (double quote) for csv-formatted files, however it is possible to specify another character to represent an escape. It is also

possible to disable escaping by specifying the value `'OFF'` as the escape value. This is very useful for data such as text-formatted web log data that has many embedded backslashes that are not intended to be escapes.

NULL

Optional for `FILE`, `GPFDIST` and `EXEC` inputs. Specifies the string that represents a null value. The default is `\N` in `TEXT` format, and an empty value with no quotations in `CSV` format. You might prefer an empty string even in `TEXT` mode for cases where you do not want to distinguish nulls from empty strings. Any input data item that matches this string will be considered a null value.

QUOTE

Optional for `FILE`, `GPFDIST` and `EXEC` inputs. Specifies the quotation character for `CSV` formatted files. The default is a double quote (`"`). In `CSV` formatted files, data value fields must be enclosed in double quotes if they contain any commas or embedded new lines. Fields that contain double quote characters must be surrounded by double quotes, and the embedded double quotes must each be represented by a pair of consecutive double quotes. It is important to always open and close quotes correctly in order for data rows to be parsed correctly.

ERROR_LIMIT

If the input rows have format errors they will be discarded provided that the error limit count is not reached on any Greenplum segment instance during input processing. If the error limit is not reached, all good rows will be processed and any error rows discarded.

ENCODING

Character set encoding to use for the data. Specify a string constant (such as `'SQL_ASCII'`), an integer encoding number, or `DEFAULT` to use the default client encoding. See “Character Set Support” in the *Greenplum Database Reference Guide* for more information.

OUTPUT

Optional. Defines where to output the formatted data of this MapReduce job. If output is not defined, the default is `STDOUT` (standard output of the client). You can send output to a file on the client host or to an existing table in the database.

NAME

A name for this output. The default output name is `STDOUT`. Names must be unique with regards to the names of other objects in this MapReduce job (such as map function, task, reduce function and input names). Also, names cannot conflict with existing objects in the database (such as tables, functions or views).

FILE

Specifies a file location on the MapReduce client machine to output data in the format: */path/to/filename*

TABLE

Specifies the name of a table in the database to output data. If this table does not exist prior to running the MapReduce job, it will be created using the distribution policy specified with KEYS.

KEYS

Optional for TABLE output. Specifies the column(s) to use as the Greenplum Database distribution key. If the EXECUTE task contains a REDUCE definition, then the REDUCE keys will be used as the table distribution key by default. Otherwise, the first column of the table will be used as the distribution key.

MODE

Optional for TABLE output. If not specified, the default is to create the table if it does not already exist, but error out if it does exist. Declaring APPEND adds output data to an existing table (provided the table schema matches the output format) without removing any existing data. Declaring REPLACE will drop the table if it exists and then recreate it. Both APPEND and REPLACE will create a new table if one does not exist.

MAP

Required. Each MAP function takes data structured in (*key*, *value*) pairs, processes each pair, and generates zero or more output (*key*, *value*) pairs. The Greenplum MapReduce framework then collects all pairs with the same key from all output lists and groups them together. This output is then passed to the REDUCE task, which is comprised of TRANSITION | CONSOLIDATE | FINALIZE functions. There is one predefined MAP function named IDENTITY that returns (*key*, *value*) pairs unchanged. Although (*key*, *value*) are the default parameters, you can specify other prototypes as needed.

TRANSITION | CONSOLIDATE | FINALIZE

TRANSITION, CONSOLIDATE and FINALIZE are all component pieces of REDUCE. A TRANSITION function is required. CONSOLIDATE and FINALIZE functions are optional. By default, all take state as the first of their input PARAMETERS, but other prototypes can be defined as well.

A TRANSITION function iterates through each value of a given key and accumulates values in a state variable. When the transition function is called on the first value of a key, the state is set to the value specified by INITIALIZE of a REDUCE job (or the default state value for the data type). A transition takes two arguments as input; the current state of the key reduction, and the next value, which then produces a new state.

If a `CONSOLIDATE` function is specified, `TRANSITION` processing is performed at the segment-level before redistributing the keys across the Greenplum interconnect for final aggregation (two-phase aggregation). Only the resulting `state` value for a given key is redistributed, resulting in lower interconnect traffic and greater parallelism. `CONSOLIDATE` is handled like a `TRANSITION`, except that instead of `(state + value) => state`, it is `(state + state) => state`.

If a `FINALIZE` function is specified, it takes the final `state` produced by `CONSOLIDATE` (if present) or `TRANSITION` and does any final processing before emitting the final result. `TRANSITION` and `CONSOLIDATE` functions cannot return a set of values. If you need a `REDUCE` job to return a set, then a `FINALIZE` is necessary to transform the final state into a set of output values.

NAME

Required. A name for the function. Names must be unique with regards to the names of other objects in this MapReduce job (such as function, task, input and output names). You can also specify the name of a function built-in to Greenplum Database. If using a built-in function, do not supply `LANGUAGE` or a `FUNCTION` body.

FUNCTION

Optional. Specifies the full body of the function using the specified `LANGUAGE`. If `FUNCTION` is not specified, then a built-in database function corresponding to `NAME` is used.

LANGUAGE

Required when `FUNCTION` is used. Specifies the implementation language used to interpret the function. This release has language support for `perl`, `python`, `java`, and `C`. If calling a built-in database function, `LANGUAGE` should not be specified.

LIBRARY

Required when `LANGUAGE` is C (not allowed for other language functions). To use this attribute, `VERSION` must be 1.0.0.2. The specified library file must be installed prior to running the MapReduce job, and it must exist in the same file system location on all Greenplum hosts (master and segments).

PARAMETERS

Optional. Function input parameters. The default type is `text`.

`MAP` default - `key text`, `value text`

`TRANSITION` default - `state text`, `value text`

`CONSOLIDATE` default - `state1 text`, `state2 text` (must have exactly two input parameters of the same data type)

`FINALIZE` default - `state text` (single parameter only)

RETURNS

Optional. The default return type is `text`.

`MAP` default - `key text`, `value text`

`TRANSITION` default - `state text` (single return value only)

`CONSOLIDATE` default - `state text` (single return value only)

`FINALIZE` default - `value text`

OPTIMIZE

Optional optimization parameters for the function:

`STRICT` - function is not affected by `NULL` values

`IMMUTABLE` - function will always return the same value for a given input

MODE

Optional. Specifies the number of rows returned by the function.

`MULTI` - returns 0 or more rows per input record. The return value of the function must be an array of rows to return, or the function must be written as an iterator using `yield` in Python or `return_next` in Perl. `MULTI` is the default mode for `MAP` and `FINALIZE` functions.

`SINGLE` - returns exactly one row per input record. `SINGLE` is the only mode supported for `TRANSITION` and `CONSOLIDATE` functions. When used with `MAP` and `FINALIZE` functions, `SINGLE` mode can provide modest performance improvement.

REDUCE

Required. A `REDUCE` definition names the `TRANSITION | CONSOLIDATE | FINALIZE` functions that comprise the reduction of (`key`, `value`) pairs to the final result set. There are also several predefined `REDUCE` jobs you can execute, which all operate over a column named `value`:

`IDENTITY` - returns (key, value) pairs unchanged

`SUM` - calculates the sum of numeric data

`AVG` - calculates the average of numeric data

`COUNT` - calculates the count of input data

`MIN` - calculates minimum value of numeric data

`MAX` - calculates maximum value of numeric data

NAME

Required. The name of this `REDUCE` job. Names must be unique with regards to the names of other objects in this MapReduce job (function, task, input and output names). Also, names cannot conflict with existing objects in the database (such as tables, functions or views).

TRANSITION

Required. The name of the `TRANSITION` function.

CONSOLIDATE

Optional. The name of the `CONSOLIDATE` function.

FINALIZE

Optional. The name of the `FINALIZE` function.

INITIALIZE

Optional for `text` and `float` data types. Required for all other data types. The default value for text is `''` . The default value for float is `0.0` . Sets the initial `state` value of the `TRANSITION` function.

KEYS

Optional. Defaults to `[key, *]`. When using a multi-column reduce it may be necessary to specify which columns are key columns and which columns are value columns. By default, any input columns that are not passed to the `TRANSITION` function are key columns, and a column named `key` is always a key column even if it is passed to the `TRANSITION` function. The special indicator `*` indicates all columns not passed to the `TRANSITION` function. If this indicator is not present in the list of keys then any unmatched columns are discarded.

TASK

Optional. A `TASK` defines a complete end-to-end `INPUT/MAP/REDUCE` stage within a Greenplum MapReduce job pipeline. It is similar to `EXECUTE` except it is not immediately executed. A task object can be called as `INPUT` to further processing stages.

NAME

Required. The name of this task. Names must be unique with regards to the names of other objects in this MapReduce job (such as map function, reduce function, input and output names). Also, names cannot conflict with existing objects in the database (such as tables, functions or views).

SOURCE

The name of an `INPUT` or another `TASK`.

MAP

Optional. The name of a `MAP` function. If not specified, defaults to `IDENTITY`.

REDUCE

Optional. The name of a `REDUCE` function. If not specified, defaults to `IDENTITY`.

EXECUTE

Required. `EXECUTE` defines the final `INPUT`/`MAP`/`REDUCE` stage within a Greenplum MapReduce job pipeline.

RUN

SOURCE

Required. The name of an `INPUT` or `TASK`.

TARGET

Optional. The name of an `OUTPUT`. The default output is `STDOUT`.

MAP

Optional. The name of a `MAP` function. If not specified, defaults to `IDENTITY`.

REDUCE

Optional. The name of a `REDUCE` function. Defaults to `IDENTITY`.

Examples

The section ”HDFS Readable and Writable External Table Examples” in the *Greenplum Database Database Administrator Guide* contains examples using MapReduce.

Example Greenplum MapReduce Document

```
# This example MapReduce job processes documents and looks for keywords in them.
# It takes two database tables as input:
#   - documents (doc_id integer, url text, data text)
#   - keywords  (keyword_id integer, keyword text)#
# The documents data is searched for occurences of keywords and returns results of
# url, data and keyword (a keyword can be multiple words, such as "high performance
# computing")
```

```
%YAML 1.1
---


VERSION: 1.0.0.1
# Connect to Greenplum Database using this database and role
DATABASE: webdata
USER: jsmith
```

```
# Begin definition section
DEFINE:
  # Declare the input, which selects all columns and rows from the
  # 'documents' and 'keywords' tables.
  - INPUT:
      NAME:  doc
      TABLE: documents


  - INPUT:
      NAME:  kw
      TABLE: keywords
```

```
# Define the map functions to extract terms from documents and keyword
# This example simply splits on white space, but it would be possible
# to make use of a python library like nltk (the natural language toolkit)
# to perform more complex tokenization and word stemming.
```

```
- MAP:
    NAME:     doc_map
    LANGUAGE: python
    FUNCTION: |
      i = 0            # the index of a word within the document
      terms = {}       # a hash of terms and their indexes within the document
     # Lower-case and split the text string on space
      for term in data.lower().split():
        i = i + 1      # increment i (the index)
      # Check for the term in the terms list:
      # if stem word already exists, append the i value to the array entry
      # corresponding to the term. This counts multiple occurances of the word.
      # If stem word does not exist, add it to the dictionary with position i.
      # For example:
        #   data: "a computer is a machine that manipulates data"
        #   "a" [1, 4]
        #   "computer" [2]
        #   "machine" [3]
        #   …
        if term in terms:
          terms[term] += ','+str(i)
        else:
          terms[term] = str(i)
      # Return multiple lines for each document. Each line consists of
      # the doc_id, a term and the positions in the data where the term appeared.
      # For example:
      #   (doc_id => 100, term => "a", [1,4]
      #   (doc_id => 100, term => "computer", [2]
      #    …
      for term in terms:
        yield([doc_id, term, terms[term]])
    OPTIMIZE: STRICT IMMUTABLE
    PARAMETERS:
      - doc_id integer
      - data text
    RETURNS:
      - doc_id integer
      - term text
      - positions text
```

```
  # The map function for keywords is almost identical to the one for documents
  # but it also counts of the number of terms in the keyword.
  - MAP:
      NAME: kw_map
      LANGUAGE: python
      FUNCTION: |
        i = 0
        terms = {}

        for term in keyword.lower().split():
          i = i + 1
          if term in terms:
            terms[term] += ','+str(i)
          else:
            terms[term] = str(i)
        # output 4 values including i (the total count for term in terms):
          yield([keyword_id, i, term, terms[term]])
      OPTIMIZE: STRICT IMMUTABLE
      PARAMETERS:
        - keyword_id integer
        - keyword text
      RETURNS:
        - keyword_id integer
        - nterms integer
        - term text
        - positions text
```

```
# A TASK is an object that defines an entire INPUT/MAP/REDUCE stage
# within a Greenplum MapReduce pipeline. It is like EXECUTION, but it is
# executed only when called as input to other processing stages.
# Identify a task called 'doc_prep' which takes in the 'doc' INPUT defined earlier
# and runs the 'doc_map' MAP function which returns doc_id, term, [term_position]
  - TASK:
      NAME: doc_prep
      SOURCE: doc
      MAP: doc_map
```

```
# Identify a task called 'kw_prep' which takes in the 'kw' INPUT defined earlier
# and runs the kw_map MAP function which returns kw_id, term, [term_position]
  - TASK:
      NAME: kw_prep
      SOURCE: kw
      MAP: kw_map
```

```
# One advantage of Greenplum MapReduce is that MapReduce tasks can be
# used as input to SQL operations and SQL can be used to process a MapReduce task.
# This INPUT defines a SQL query that joins the output of the 'doc_prep'
# TASK to that of the 'kw_prep' TASK. Matching terms are output to the 'candidate'
# list (any keyword that shares at least one term with the document).
  - INPUT:
      NAME: term_join
      QUERY: |
        SELECT doc.doc_id, kw.keyword_id, kw.term, kw.nterms,
               doc.positions as doc_positions,
               kw.positions as kw_positions
          FROM doc_prep doc INNER JOIN kw_prep kw ON (doc.term = kw.term)
```

```
# In Greenplum MapReduce, a REDUCE function is comprised of one or more functions.
# A REDUCE has an initial 'state' variable defined for each grouping key. that is
# A TRANSITION function adjusts the state for every value in a key grouping.
# If present, an optional CONSOLIDATE function combines multiple
# 'state' variables.  This allows the TRANSITION function to be executed locally at
# the segment-level and only redistribute the accumulated 'state' over
# the network. If present, an optional FINALIZE function can be used to perform
# final computation on a state and emit one or more rows of output from the state.
#
# This REDUCE function is called 'term_reducer' with a TRANSITION function
# called 'term_transition' and a FINALIZE function called 'term_finalizer'
```

```
- REDUCE:
      NAME: term_reducer
      TRANSITION: term_transition
      FINALIZE:  term_finalizer
```

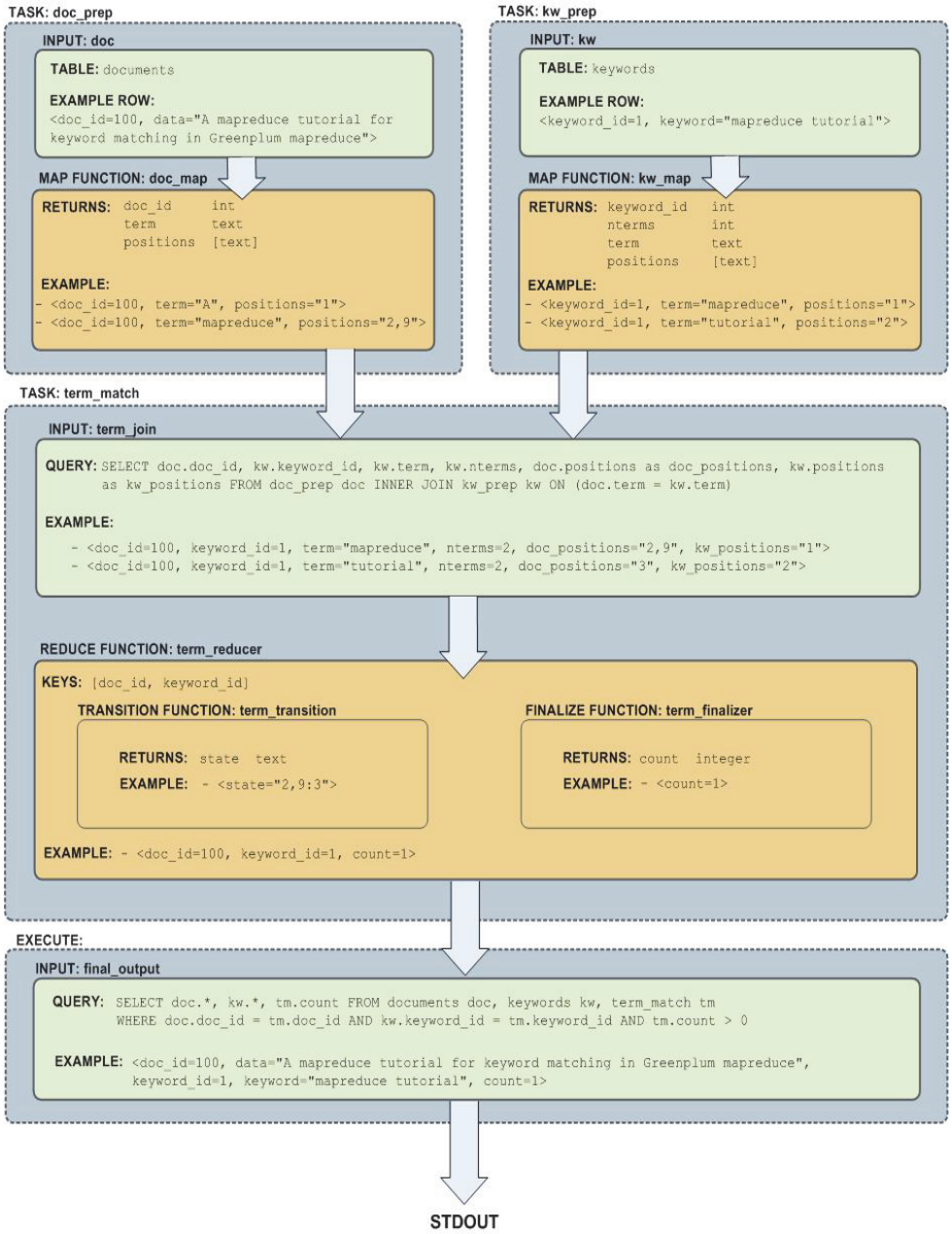
```
- TRANSITION:
      NAME: term_transition
      LANGUAGE: python
      PARAMETERS:
        - state text
        - term text
        - nterms integer
        - doc_positions text
        - kw_positions text
      FUNCTION: |
       # 'state' has an initial value of '' and is a colon delimited set
       # of keyword positions. keyword positions are comma delimited sets of
       # integers. For example, '1,3,2:4:'
       # If there is an existing state, split it into the set of keyword positions
       # otherwise construct a set of 'nterms' keyword positions - all empty
       if state:
          kw_split = state.split(':')
        else:
          kw_split = []
          for i in range(0,nterms):
            kw_split.append('')
       # 'kw_positions' is a comma delimited field of integers indicating what
       # position a single term occurs within a given keyword.
       # Splitting based on ',' converts the string into a python list.
       # add doc_positions for the current term
        for kw_p in kw_positions.split(','):
          kw_split[int(kw_p)-1] = doc_positions
       # This section takes each element in the 'kw_split' array and strings
       # them together placing a ':' in between each element from the array.
       # For example: for the keyword "computer software computer hardware",
       # the 'kw_split' array matched up to the document data of
       # "in the business of computer software software engineers"
       # would look like: ['5', '6,7', '5', '']
       # and the outstate would look like: 5:6,7:5:
        outstate = kw_split[0]
        for s in kw_split[1:]:
          outstate = outstate + ':' + s
        return outstate
```

```
- FINALIZE:
    NAME: term_finalizer
    LANGUAGE: python
    RETURNS:
      - count integer
    MODE: MULTI
    FUNCTION: |
      if not state:
        return 0
      kw_split = state.split(':')
      # This function does the following:
      # 1) Splits 'kw_split' on ':'
      #    for example, 1,5,7:2,8 creates '1,5,7' and '2,8'
      # 2) For each group of positions in 'kw_split', splits the set on ','
      #    to create ['1','5','7'] from Set 0: 1,5,7 and
      #    eventually ['2', '8'] from Set 1: 2,8
      # 3) Checks for empty strings
      # 4) Adjusts the split sets by subtracting the position of the set
      #      in the 'kw_split' array
      #      ['1','5','7'] - 0 from each element = ['1','5','7']
      #      ['2', '8'] - 1 from each element = ['1', '7']
      # 5) Resulting arrays after subtracting the offset in step 4 are
      #    intersected and their overlaping values kept:
      #    ['1','5','7'].intersect['1', '7'] = [1,7]
      # 6) Determines the length of the intersection, which is the number of
      #    times that an entire keyword (with all its pieces) matches in the
      #    document data.
      previous = None
      for i in range(0,len(kw_split)):
        isplit = kw_split[i].split(',')
        if any(map(lambda(x): x == '', isplit)):
          return 0
        adjusted = set(map(lambda(x): int(x)-i, isplit))
        if (previous):
          previous = adjusted.intersection(previous)
        else:
          previous = adjusted
      # return the final count
      if previous:
        return len(previous)
      return 0
```

```
   # Define the 'term_match' task which is then executed as part
   # of the 'final_output' query. It takes the INPUT 'term_join' defined
   # earlier and uses the REDUCE function 'term_reducer' defined earlier

  - TASK:
      NAME: term_match
      SOURCE: term_join
      REDUCE: term_reducer

  - INPUT:
      NAME: final_output
      QUERY: |
        SELECT doc.*, kw.*, tm.count
        FROM documents doc, keywords kw, term_match tm
        WHERE doc.doc_id = tm.doc_id
          AND kw.keyword_id = tm.keyword_id
          AND tm.count > 0

# Execute this MapReduce job and send output to STDOUT

EXECUTE:
  - RUN:
      SOURCE: final_output
      TARGET: STDOUT
```

MapReduce Flow Diagram

The following diagram shows the job flow of the MapReduce job defined in the example:



B. Greenplum Database Tools

Greenplum Database tools support developing Greenplum Database applications, extending Greenplum Database functionality, and managing and administering Greenplum databases.

Greenplum Database application development tools

Separate extensions that extend Greenplum Database functionality:

- Extensions for languages to support creating database routines:
  - PL/Java
  - PL/Perl
  - PL/R (R is a statistical computing language)

  Note: By default, the PL/Python and PL/pgSQL languages are installed with Greenplum Database.
- MADlib is an open-source library for scalable in-database analytics.
- R is an open-source language that is used for statistical computing.
- PostGIS adds in-database support for geographic objects to the Greenplum Database for geographic information systems (GIS). PostGIS follows the OpenGIS "Simple Features Specification for SQL" and has been certified as compliant with the "Types and Functions" profile.
- pgcrypto provides cryptographic functions for the Greenplum Database.
- **GPText** - an add-on package to Greenplum Database. GPText includes full-text search as well as text analytics applications such as k-means clustering. GPText combines Greenplum Database with Greenplum Solr enterprise search to support the next generation of data warehousing and large-scale analytics processing.
- Greenplum **Partner Connector** - a library that allows easy deployment of partner applications on the Greenplum database without requiring application recompilation between database versions.
- PivotalR is a first class R package that enables users to interact with data resident in Greenplum Database using the R client.

  This package allows R users to leverage the scalability and performance of in-database analytics without leaving the R command line. All of the computational heavy lifting is executed in-database, while the end user benefits from a familiar R interface. Compared with respective native R functions, we observe a dramatic increase in scalability and a decrease in running time. Furthermore, data movement -- which can take hours for big data -- is eliminated via PivotalR.

  Using PivotalR requires that MADlib is installed on the Greenplum Database.

  The top-line benefit of this release is faster ramp up and adoption of Pivotal Big Data Platform for Data Science.

  Key features:

- Explore and manipulate data in the database through R transparently(SQL translation is done by PivotalR
- Use familiar R syntax for predictive analytics algorithms, specifically linear and logistic regression. Seamlessly translated to MADlib in-database analytics function calls
- Comprehensive documentation package with examples in standard R format accessible from R client.
- PivotalR package also supports acces to the following MADlib functionality.

  Linear regression

  Logisistic regression

  Table summary

[MSK] Is this the right link?
For information about PivotalR, see
http://madlib-internal.github.io/PivotalR/

Access to external data

Greenplum Database supports access to external data with external tables. Greenplum Database also supports Hadoop integration:

- **gNet for Hadoop** - a package that enables high performance parallel import and export of compressed and uncompressed data from Hadoop clusters.
- **Client tools** are utilities to access Greenplum Database from client machines.
  [**Ivan**] I would consider mentioning an example such as PSQL client
- **Connectivity tools** are database drivers and a C API for connecting to Greenplum Database.
  [**Ivan**] I would consider mentioning ODBC and JDBC drives and also we should talk to Oak to clarify the difference between our own ODBC and JDBC drivers, and then ones provided and supported by Data Direct which may be better
- **Load tools** are Greenplum Database data loading programs that are run on client machines.
- **PowerExchange for Greenplum** - an Informatica-Greenplum adaptor. The adaptor allows customers to use their existing Informatica PowerCenter environments to bulk/parallel load into Greenplum database system. The adaptor allows bulk loading utilizing gpfdist & gpload
- **Pivotal HD** - a 100 percent open-source certified and supported version of the Apache Hadoop stack. Pivotal HD includes HDFS, MapReduce, Hive, Pig, Hbase and Zookeeper.
- **Greenplum MR** - Based on the MapR M5 Distribution, Greenplum MR enables you to increase the performance and availability of Hadoop via breakthrough innovations. With Greenplum MR, Hadoop is faster, more dependable, and easier to use.
  [**Ivan**] Is this being phased out?

Applications and Solutions

Pivotal applications and solutions that support Greenplum Database and Greenplum analytics development:

- **Greenplum Command Center** - a unified management console that allows administrators to monitor their cluster, whether it is a software-only or appliance installation of Greenplum Database.
- **Pivotal Chorus** - enables Big Data agility for your data science team. Greenplum Chorus provides an analytic productivity platform that enables the team to search, explore, visualize, and import data from anywhere in the organization with rich social network features.
  http://www.gopivotal.com/pivotal-products/pivotal-data-fabric/pivotal-chorus
- **Pivotal UAP** - Unified Analytics Platform (UAP) that delivers:
  - Greenplum Database for structured data
  - Pivotal HD for the analysis and processing of unstructured data
  - Pivotal Chorus the productivity layer for the data science team

  http://www.gopivotal.com/pivotal-products/pivotal-expert-services/pivotal-data-science-labs
- **Greenplum Data Computing Appliance (DCA)** - a unified Big Data analytics appliance. DCA offers the power of a massively parallel processing (MPP) architecture, while delivering the fastest data-loading rate and the best price/performance ratio in the industry—without the complexity and constraints of proprietary hardware.
- **Greenplum Analytics Lab** - Analytics Lab is a package of services, technology or training delivered by Greenplum's team of leading Data Scientists. During the engagement, your analytics stakeholders and data platform leadership work in partnership with Greenplum's team of statisticians and modelers to solve real business problems using big data advanced analytics.

  http://www.gopivotal.com/pivotal-products/pivotal-expert-services/pivotal-data-science-labs

[**MSK**] How do we want to reference third-party tools.

- **pgAdmin III for Greenplum Database** - a GUI client that supports PostgreSQL databases with all standard pgAdmin III features, while adding support for Greenplum-specific features.
- **Alpine Predictive Analytics for Pivotal** - delivers deep insights and models from all your data in a simple web-based application that combines the power of big data processing with the sophistication of predictive analytics. This solution moves beyond traditional business intelligence by delivering in-database analytics that allow you to unlock the full potential of your data.
  http://www.gopivotal.com/pivotal-solutions/alpine-predictive-analytics-for-pivotal

[**Ivan**] comments about support and availability.

- For pgAdmin I would mention we don't support it directly. Perhaps we should not include pgAdmin in the guide?

- Greenplum UAP i would not include in the doc? Is that shippable? What is UAP?
- for Greenplum Chorus: Is this supported by us or by Alpine now? Should we mention that?
- Apline Miner, should we mention who supports that?
- MapR: do we still support MapR? Are we phasing it out?

Backup and Disaster Recovery

- **Data Domain** - Greenplum Database can use Data Domain systems for backup and disaster recovery.

  Data Domain Boost is a module for Greenplum Database that makes backup more simple, and scalable with Data Domain. Data Domain Boost dramatically increases aggregate throughput, reduces costs by reducing the amount of data transferred over the network and increases simplicity by eliminating the need to create and manage virtual drives.

  You can also connect to Data Domain with NFS shares to perform backups. Using NFS does not require Data Domain Boost.