PRODUCT DOCUMENTATION

Pivotal™ Greenplum Database®

Version 4.3

Administrator Guide

Rev: A21

Notice

Copyright

Privacy Policy | *Terms of Use*

Pivotal Software, Inc. believes the information in this publication is accurate as of its publication date. The information is subject to change without notice. THE INFORMATION IN THIS PUBLICATION IS PROVIDED "AS IS." PIVOTAL SOFTWARE, INC. ("Pivotal") MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Revised November 2016 (4.3.11.0)

Contents

Preface: About This Guide.. x

About the Greenplum Database Documentation Set.. xi

Document Conventions.. xii

Text Conventions.. xii

Command Syntax Conventions.. xiii

Getting Support.. xiv

Product information and Technical Support.. xiv

Part I: Greenplum Database Concepts.. 15

Chapter 1: About the Greenplum Architecture.............................. 16

About the Greenplum Master.. 17

About the Greenplum Segments.. 17

About the Greenplum Interconnect.. 18

Chapter 2: About Management and Monitoring Utilities................ 19

Chapter 3: About Concurrency Control in Greenplum Database... 21

Example of Managing Transaction IDs.. 25

Chapter 4: About Parallel Data Loading.. 29

Chapter 5: About Redundancy and Failover in Greenplum Database... 30

Chapter 6: About Database Statistics in Greenplum Database.... 33

System Statistics.. 33

Configuring Statistics.. 37

Part II: Managing a Greenplum System.. 40

Chapter 7: Starting and Stopping Greenplum Database................ 41

Starting Greenplum Database.. 41

Restarting Greenplum Database.. 41

Reloading Configuration File Changes Only.. 41

Starting the Master in Maintenance Mode.. 42

Stopping Greenplum Database.. 42

Chapter 8: Accessing the Database.. 43
Establishing a Database Session..44
Supported Client Applications..45
Greenplum Database Client Applications..46
Connecting with psql..47
pgAdmin III for Greenplum Database..48
Installing pgAdmin III for Greenplum Database..49
Documentation for pgAdmin III for Greenplum Database..49
Performing Administrative Tasks with pgAdmin III..49
Using the PgBouncer Connection Pooler..51
Database Application Interfaces..57
Third-Party Client Tools..58
Troubleshooting Connection Problems..59

Chapter 9: Configuring the Greenplum Database System............60
About Greenplum Master and Local Parameters..61
Setting Configuration Parameters..62
Setting a Local Configuration Parameter..62
Setting a Master Configuration Parameter..62
Viewing Server Configuration Parameter Settings..64
Configuration Parameter Categories..65
Runtime Statistics Collection Parameters..65
Automatic Statistics Collection Parameters..65
Lock Management Parameters..66
Workload Management Parameters..66
External Table Parameters..66
Database Table Parameters..66
Database and Tablespace/Filespace Parameters..67
Past PostgreSQL Version Compatibility Parameters..67
Greenplum Master and Segment Mirroring Parameters..67
Greenplum Database Extension Parameters..67
Connection and Authentication Parameters..67
System Resource Consumption Parameters..68
Query Tuning Parameters..69
Error Reporting and Logging Parameters..71
System Monitoring Parameters..71
Client Connection Default Parameters..72
Greenplum Array Configuration Parameters..72

Chapter 10: Enabling High Availability Features...........................74
Overview of Greenplum Database High Availability..75
Overview of Segment Mirroring..77
Overview of Master Mirroring..79
Overview of Fault Detection and Recovery..80
Enabling Mirroring in Greenplum Database..81
Enabling Segment Mirroring..81
Enabling Master Mirroring..82
Detecting a Failed Segment..83
Enabling Alerts and Notifications..84
Checking for Failed Segments..84
Checking the Log Files for Failed Segments..85
Recovering a Failed Segment..86

Recovering From Segment Failures....87
Recovering a Failed Master.... 92
Restoring Master Mirroring After a Recovery....92

Chapter 11: Backing Up and Restoring Databases.... 94
Backup and Restore Overview....95
Backing Up with gpcrondump.... 99
Backing Up a Set of Tables....100
Creating Incremental Backups.... 101
Backup Process and Locks....104
Using Direct I/O....106
Using Named Pipes....107
Backing Up Databases with Data Domain Boost....109
Backing Up Databases with Veritas NetBackup.... 115
Restoring Greenplum Databases.... 119
Restoring a Database Using gpdbrestore....120
Restoring to a Different Greenplum System Configuration.... 122

Chapter 12: Expanding a Greenplum System....124
System Expansion Overview....125
Planning Greenplum System Expansion....128
Preparing and Adding Nodes.... 133
Initializing New Segments.... 136
Redistributing Tables.... 140
Removing the Expansion Schema.... 142

Chapter 13: Migrating Data with gptransfer.... 143

Chapter 14: Monitoring a Greenplum System....149
Monitoring Database Activity and Performance.... 149
Monitoring System State.... 149
Viewing the Database Server Log Files....158
Using gp_toolkit.... 160
Greenplum Database SNMP OIDs and Error Codes....160

Chapter 15: Routine System Maintenance Tasks.... 169
Routine Vacuum and Analyze....169
Routine Reindexing.... 172
Managing Greenplum Database Log Files....172

Chapter 16: Recommended Monitoring and Maintenance Tasks....173
Database State Monitoring Activities....173
Database Alert Log Monitoring....175
Hardware and Operating System Monitoring....176
Catalog Monitoring....177
Data Maintenance....178
Database Maintenance....179
Patching and Upgrading....180

Part III: Managing Greenplum Database Access.................................181

Chapter 17: Configuring Client Authentication............................182

Allowing Connections to Greenplum Database..182
Limiting Concurrent Connections.. 184
Encrypting Client/Server Connections..185
Using LDAP Authentication with TLS/SSL..187
Using Kerberos Authentication..190

Chapter 18: Managing Roles and Privileges................................196

Security Best Practices for Roles and Privileges..196
Creating New Roles (Users)... 197
Role Membership..198
Managing Object Privileges...199
Encrypting Data.. 200
Protecting Passwords in Greenplum Database...201
Time-based Authentication... 202

Chapter 19: Configuring IPsec for Greenplum Database............203

IPsec Overview...203
Installing Openswan.. 204
Configuring Openswan Connections.. 206

Part IV: Working with Databases..211

Chapter 20: Defining Database Objects...................................... 212

Creating and Managing Databases..213
Creating and Managing Tablespaces..215
Creating and Managing Schemas...218
Creating and Managing Tables..220
Choosing the Table Storage Model.. 223
Partitioning Large Tables...234
Creating and Using Sequences.. 247
Using Indexes in Greenplum Database.. 248
Creating and Managing Views.. 252

Chapter 21: Managing Data... 253

About Concurrency Control in Greenplum Database..253
Inserting Rows...254
Updating Existing Rows...255
Deleting Rows..255
Working With Transactions..255
Vacuuming the Database..257

Chapter 22: Loading and Unloading Data....................................258

Working with File-Based External Tables.. 259
Accessing File-Based External Tables...259

file:// Protocol........ 260
gpfdist:// Protocol........ 260
gpfdists:// Protocol........ 261
gphdfs:// Protocol........ 262
s3:// Protocol........ 262
Using a Custom Protocol........ 269
Handling Errors in External Table Data........ 269
Using the Greenplum Parallel File Server (gpfdist)........ 270
About gpfdist Setup and Performance........ 270
Controlling Segment Parallelism........ 271
Installing gpfdist........ 271
Starting and Stopping gpfdist........ 271
Troubleshooting gpfdist........ 272
Using Hadoop Distributed File System (HDFS) Tables........ 273
One-time HDFS Protocol Installation........ 273
About gphdfs JVM Memory........ 280
Using Amazon EMR with Greenplum Database installed on AWS........ 280
Support for Avro Files........ 281
Support for Parquet Files........ 289
Creating and Using Web External Tables........ 296
Command-based Web External Tables........ 296
URL-based Web External Tables........ 296
Loading Data Using an External Table........ 298
Loading and Writing Non-HDFS Custom Data........ 299
Using a Custom Format........ 299
Using a Custom Protocol........ 300
Creating External Tables - Examples........ 302
Example 1—Single gpfdist instance on single-NIC machine........ 302
Example 2—Multiple gpfdist instances........ 302
Example 3—Multiple gpfdists instances........ 302
Example 4—Single gpfdist instance with error logging........ 302
Example 5—TEXT Format on a Hadoop Distributed File Server........ 303
Example 6—Multiple files in CSV format with header rows........ 303
Example 7—Readable Web External Table with Script........ 303
Example 8—Writable External Table with gpfdist........ 304
Example 9—Writable External Web Table with Script........ 304
Example 10—Readable and Writable External Tables with XML Transformations.... 304
Handling Load Errors........ 305
Define an External Table with Single Row Error Isolation........ 305
Capture Row Formatting Errors and Declare a Reject Limit........ 306
Viewing Bad Rows in the Error Table or Error Log........ 306
Identifying Invalid CSV Files in Error Table Data........ 307
Moving Data between Tables........ 307
Loading Data with gpload........ 308
Loading Data with COPY........ 309
Running COPY in Single Row Error Isolation Mode........ 310
Optimizing Data Load and Query Performance........ 311
Unloading Data from Greenplum Database........ 312
Defining a File-Based Writable External Table........ 312
Defining a Command-Based Writable External Web Table........ 313
Unloading Data Using a Writable External Table........ 314
Unloading Data Using COPY........ 314
Transforming XML Data........ 315
Determine the Transformation Schema........ 315
Write a Transform........ 316
Write the gpfdist Configuration........ 316

Load the Data.......... 318
Transfer and Store the Data.......... 318
XML Transformation Examples.......... 321
Formatting Data Files.......... 324
Formatting Rows.......... 324
Formatting Columns.......... 324
Representing NULL Values.......... 324
Escaping.......... 324
Character Encoding.......... 326
Example Custom Data Access Protocol.......... 327
Installing the External Table Protocol.......... 327

Chapter 23: Querying Data.......... 333
About Greenplum Query Processing.......... 334
About the Pivotal Query Optimizer.......... 338
Overview of the Pivotal Query Optimizer.......... 338
Enabling the Pivotal Query Optimizer.......... 339
Considerations when Using the Pivotal Query Optimizer.......... 340
Pivotal Query Optimizer Features and Enhancements.......... 341
Changed Behavior with the Pivotal Query Optimizer.......... 344
Pivotal Query Optimizer Limitations.......... 345
Determining the Query Optimizer that is Used.......... 346
About Uniform Multi-level Partitioned Tables.......... 348
Defining Queries.......... 350
Using Functions and Operators.......... 359
Query Performance.......... 376
Managing Spill Files Generated by Queries.......... 377
Query Profiling.......... 378

Part V: Managing Performance.......... 383

Chapter 24: Defining Database Performance.......... 384
Understanding the Performance Factors.......... 384
Determining Acceptable Performance.......... 385

Chapter 25: Common Causes of Performance Issues.......... 386
Identifying Hardware and Segment Failures.......... 386
Managing Workload.......... 386
Avoiding Contention.......... 387
Maintaining Database Statistics.......... 387
Optimizing Data Distribution.......... 388
Optimizing Your Database Design.......... 388

Chapter 26: Workload Management with Resource Queues.......... 389
Overview of Memory Usage in Greenplum Database.......... 389
Overview of Managing Workloads with Resource Queues.......... 392
Steps to Enable Workload Management.......... 397
Configuring Workload Management.......... 397
Creating Resource Queues.......... 399
Assigning Roles (Users) to a Resource Queue.......... 400
Modifying Resource Queues.......... 401

Checking Resource Queue Status......401

Chapter 27: Investigating a Performance Problem......405

Checking System State......405
Checking Database Activity......405
Troubleshooting Problem Queries......406
Investigating Error Messages......406

About This Guide

This guide describes system and database administration tasks for Greenplum Database. The guide consists of five sections:

- *Greenplum Database Concepts* describes Greenplum Database architecture and components. It introduces administration topics such as mirroring, parallel data loading, and Greenplum management and monitoring utilities.
- *Managing a Greenplum System* contains information about everyday Greenplum Database system administration tasks. Topics include starting and stopping the server, client front-ends to access the database, configuring Greenplum, enabling high availability features, backing up and restoring databases, expanding the system by adding nodes, monitoring the system, and regular maintenance tasks.
- *Managing Greenplum Database Access* covers configuring Greenplum Database authentication, managing roles and privileges, and setting up Kerberos athentication.
- *Working with Databases* contains information about creating and managing databases, schemas, tables and other database objects. It describes how to view database metadata, insert, update, and delete data in tables, load data from external files, and run queries in a database.
- *Managing Performance* describes how to monitor and manage system performance. It discusses how to define performance in a parallel environment, how to diagnose performance problems, workload and resource administration, and performance troubleshooting.

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

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

This guide provides information for system administrators responsible for administering a Greenplum Database system.

- *About the Greenplum Database Documentation Set*
- *Document Conventions*
- *Getting Support*

About the Greenplum Database Documentation Set

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

Table 1: Greenplum Database documentation set

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

Document Conventions

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

- *Text Conventions*
- *Command Syntax Conventions*

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 column 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 3: Command Syntax Conventions

Text Convention	Usage	Examples
`{ }`	Within command syntax, curly braces group related command options. Do not type the curly braces.	`FROM { 'filename' | STDIN }`
`[ ]`	Within command syntax, square brackets denote optional arguments. Do not type the brackets.	`TRUNCATE [ TABLE ] name`
`...`	Within command syntax, an ellipsis denotes repetition of a command, variable, or option. Do not type the ellipsis.	`DROP TABLE name [, ... ]`
`|`	Within command syntax, the pipe symbol denotes an "OR" relationship. Do not type the pipe symbol.	`VACUUM [ FULL | FREEZE ]`
`$` system_command `#` root_system_command `=>` gpdb_command `=#` su_gpdb_command	Denotes a command prompt - do not type the prompt symbol. `$` and `#` denote terminal command prompts. `=>` and `=#` denote Greenplum Database interactive program command prompts (`psql` or `gpssh`, for example).	$ `createdb mydatabase` # `chown gpadmin -R /datadir` => `SELECT * FROM mytable;` =# `SELECT * FROM pg_database;`

Getting Support

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

Product information and Technical Support

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

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

Part I

Greenplum Database Concepts

This section provides an overview of Greenplum Database components and features such as high availability, parallel data loading features, and management utilities.

This section contains the following topics:

- *About the Greenplum Architecture*
- *About Management and Monitoring Utilities*
- *About Parallel Data Loading*
- *About Redundancy and Failover in Greenplum Database*
- *About Database Statistics in Greenplum Database*

Chapter 1

About the Greenplum Architecture

Pivotal Greenplum Database is a massively parallel processing (MPP) database server with an architecture specially designed to manage large-scale analytic data warehouses and business intelligence workloads.

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 based on PostgreSQL open-source technology. It 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, 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.

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.

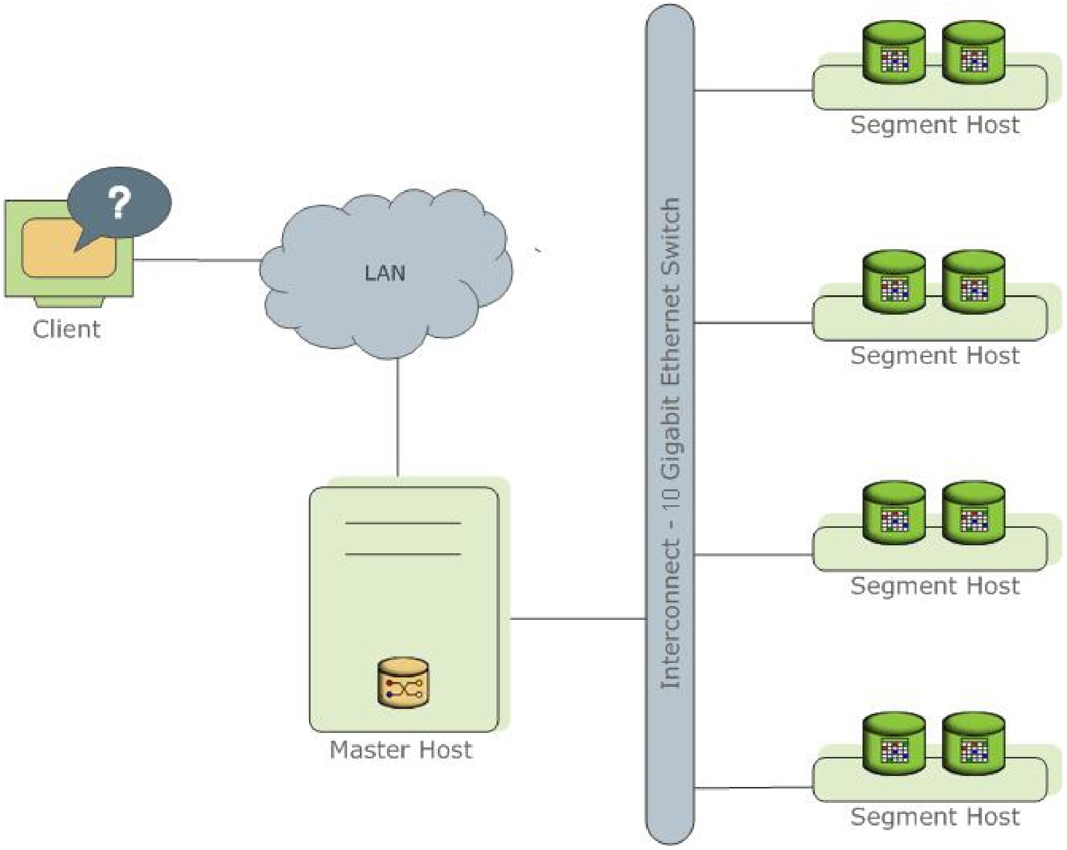


Figure 1: High-Level Greenplum Database Architecture

The following topics describe the components that make up a Greenplum Database system and how they work together.

About the Greenplum Master

The Greenplum Database master is the entry to the Greenplum Database system, accepting client connections and SQL queries, and distributing work to the segment instances.

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 workloads among segments, coordinates the results returned by each segment, and presents the final results to the client program.

About the Greenplum Segments

Greenplum Database segment instances are independent PostgreSQL databases that each store a portion of the data and perform the majority of query processing.

When a user connects to the database via the Greenplum master and issues a query, processes are created in each segment database to handle the work of that query. For more information about query processes, see *About Greenplum Query Processing*.

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.

Segments run on a servers called *segment hosts*. A segment host typically executes from two to eight Greenplum segments, depending on the CPU cores, RAM, storage, network interfaces, and workloads. Segment hosts are expected to be identically configured. The key to obtaining the best performance from Greenplum Database is to distribute data and workloads *evenly* across a large number of equally capable segments so that all segments begin working on a task simultaneously and complete their work at the same time.

About the Greenplum Interconnect

The interconect is the networking layer of the Greenplum Database architecture.

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 10-Gigabit Ethernet switching fabric.

By default, the interconnect uses User Datagram Protocol (UDP) with flow control for interconnect traffic 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. For information about the types of interconnect supported by Greeplum Database, see server configuration parameter `gp_interconnect_type` in the *Greenplum Database Reference Guide*.

Chapter 2

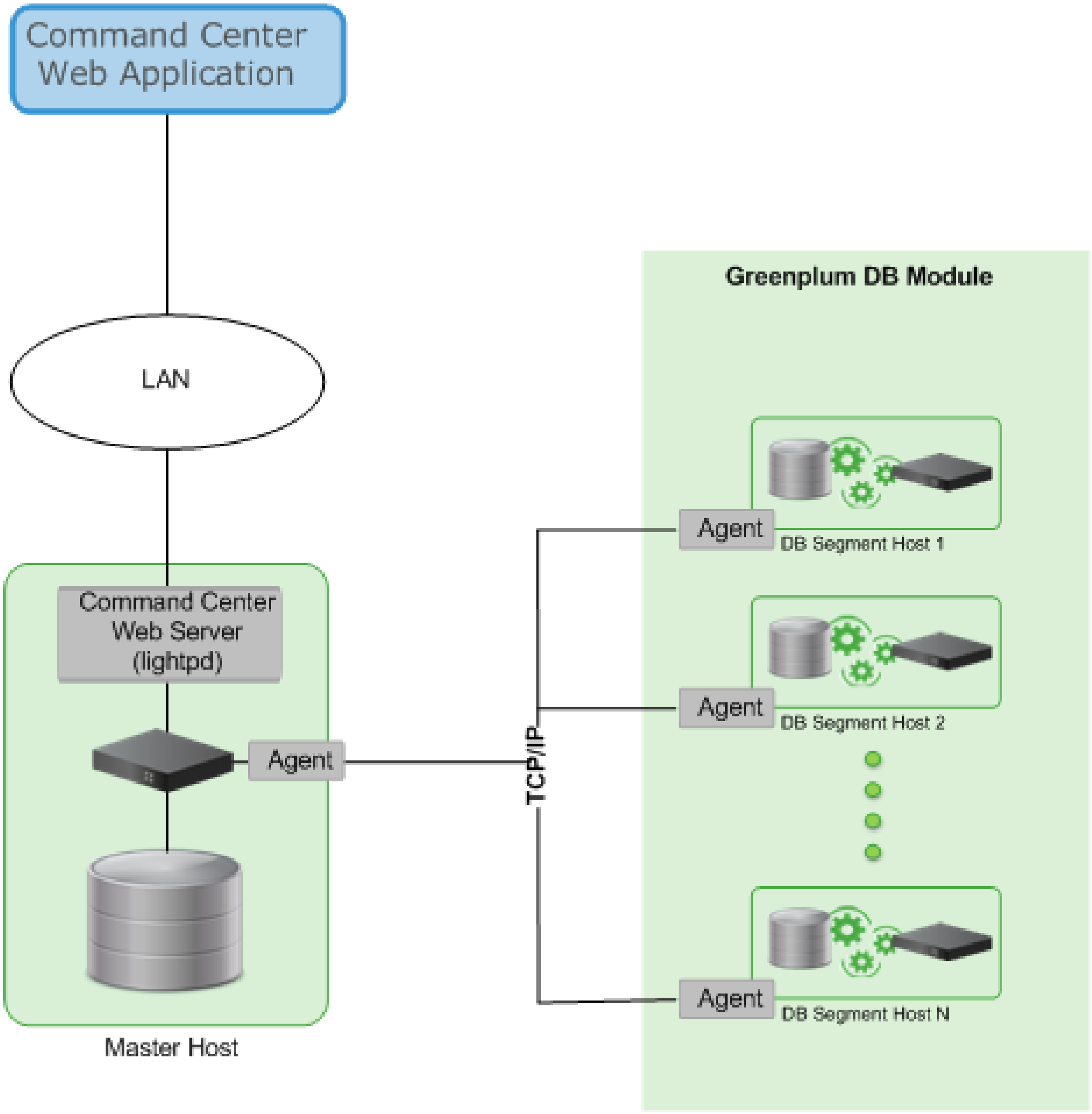
About Management and Monitoring Utilities

Greenplum Database provides standard command-line utilities for performing common monitoring and administration tasks.

Greenplum command-line utilities are located in the `$GPHOME/bin` directory and are executed on the master host. 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
- Transferring data between Greenplum databases
- 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.


Command Center Web Application
LAN
Command Center Web Server (lightpd)
Agent
Master Host
TCP/IP
Greenplum DB Module
Agent
DB Segment Host 1
Agent
DB Segment Host 2
Agent
DB Segment Host N

Chapter 3

About Concurrency Control in Greenplum Database

Greenplum Database uses the PostgreSQL Multiversion Concurrency Control (MVCC) model to manage concurrent transactions for heap tables.

Concurrency control in a database management system allows concurrent queries to complete with correct results while ensuring the integrity of the database. Traditional databases use a two-phase locking protocol that prevents a transaction from modifying data that has been read by another concurrent transaction and prevents any concurrent transaction from reading or writing data that another transaction has updated. The locks required to coordinate transactions add contention to the database, reducing overall transaction throughput.

Greenplum Database uses the PostgreSQL Multiversion Concurrency Control (MVCC) model to manage concurrency for heap tables. With MVCC, each query operates on a snapshot of the database when the query starts. While it executes, a query cannot see changes made by other concurrent transactions. This ensures that a query sees a consistent view of the database. Queries that read rows can never block waiting for transactions that write rows. Conversely, queries that write rows cannot be blocked by transactions that read rows. This allows much greater concurrency than traditional database systems that employ locks to coordinate access between transactions that read and write data.

> **Note:** Append-optimized tables are managed with a different concurrency control model than the MVCC model discussed in this topic. They are intended for "write-once, read-many" applications that never, or only very rarely, perform row-level updates.

Snapshots

The MVCC model depends on the system's ability to manage multiple versions of data rows. A query operates on a snapshot of the database at the start of the query. A snapshot is the set of rows that are visible at the beginning of a statement or transaction. The snapshot ensures the query has a consistent and valid view of the database for the duration of its execution.

Each transaction is assigned a unique *transaction ID* (XID), an incrementing 32-bit value. When a new transaction starts, it is assigned the next XID. A SQL statement that is not enclosed in a transaction is treated as a single-statement transaction—the `BEGIN` and `COMMIT` are added implicitly. This is similar to autocommit in some database systems.

When a transaction inserts a row, the XID is saved with the row in the `xmin` system column. When a transaction deletes a row, the XID is saved in the `xmax` system column. Updating a row is treated as a delete and an insert, so the XID is saved to the `xmax` of the current row and the `xmin` of the newly inserted row. The `xmin` and `xmax` columns, together with the transaction completion status, specify a range of transactions for which the version of the row is visible. A transaction can see the effects of all transactions less than `xmin`, which are guaranteed to be committed, but it cannot see the effects of any transaction greater than or equal to `xmax`.

Multi-statement transactions must also record which command within a transaction inserted a row (`cmin`) or deleted a row (`cmax`) so that the transaction can see changes made by previous commands in the transaction. The command sequence is only relevant during the transaction, so the sequence is reset to 0 at the beginning of a transaction.

XID is a property of the database. Each segment database has its own XID sequence that cannot be compared to the XIDs of other segment databases. The master coordinates distributed transactions with the segments using a cluster-wide *session ID number*, called `gp_session_id`. The segments maintain a mapping of distributed transaction IDs with their local XIDs. The master coordinates distributed transactions across all of the segment with the two-phase commit protocol. If a transaction fails on any one segment, it is rolled back on all segments.

You can see the xmin, xmax, cmin, and cmax columns for any row with a SELECT statement:

```
SELECT xmin, xmax, cmin, cmax, * FROM tablename;
```

Because you run the SELECT command on the master, the XIDs are the distributed transactions IDs. If you could execute the command in an individual segment database, the xmin and xmax values would be the segment's local XIDs.

Transaction ID Wraparound

The MVCC model uses transaction IDs (XIDs) to determine which rows are visible at the beginning of a query or transaction. The XID is a 32-bit value, so a database could theoretically execute over four billion transactions before the value overflows and wraps to zero. However, Greenplum Database uses *modulo* 2^{32} arithmetic with XIDs, which allows the transaction IDs to wrap around, much as a clock wraps at twelve o'clock. For any given XID, there could be about two billion past XIDs and two billion future XIDs. This works until a version of a row persists through about two billion transactions, when it suddenly appears to be a new row. To prevent this, Greenplum has a special XID, called FrozenXID, which is always considered older than any regular XID it is compared with. The xmin of a row must be replaced with FrozenXID within two billion transactions, and this is one of the functions the VACUUM command performs.

Vacuuming the database at least every two billion transactions prevents XID wraparound. Greenplum Database monitors the transaction ID and warns if a VACUUM operation is required.

A warning is issued when a significant portion of the transaction IDs are no longer available and before transaction ID wraparound occurs:

```
WARNING: database "database_name" must be vacuumed within number_of_transactions
 transactions
```

When the warning is issued, a VACUUM operation is required. If a VACUUM operation is not performed, Greenplum Database stops creating transactions to avoid possible data loss when it reaches a limit prior to when transaction ID wraparound occurs and issues this error:

```
FATAL: database is not accepting commands to avoid wraparound data loss in database
 "database_name"
```

See *Recovering from a Transaction ID Limit Error* for the procedure to recover from this error.

The server configuration parameters xid_warn_limit and xid_stop_limit control when the warning and error are displayed. The xid_warn_limit parameter specifies the number of transaction IDs before the xid_stop_limit when the warning is issued. The xid_stop_limit parameter specifies the number of transaction IDs before wraparound would occur when the error is issued and new transactions cannot be created.

Transaction Isolation Modes

The SQL standard describes three phenomena that can occur when database transactions run concurrently:

- *Dirty read* – a transaction can read uncommitted data from another concurrent transaction.
- *Non-repeatable read* – a row read twice in a transaction can change because another concurrent transaction committed changes after the transaction began.
- *Phantom read* – a query executed twice in the same transaction can return two different sets of rows because another concurrent transaction added rows.

The SQL standard defines four transaction isolation modes that database systems must support:

Table 4: Transaction Isolation Modes

Level	Dirty Read	Non-Repeatable	Phantom Read
Read Uncommitted	Possible	Possible	Possible
Read Committed	Impossible	Possible	Possible
Repeatable Read	Impossible	Impossible	Possible
Serializable	Impossible	Impossible	Impossible

The Greenplum Database SQL commands allow you to request `READ UNCOMMITTED`, `READ COMITTED`, or `SERIALIZABLE`. Greenplum Database treats `READ UNCOMMITTED` the same as `READ COMMITTED`. Requesting `REPEATABLE READ` produces an error; use `SERIALIZABLE` instead. The default isolation mode is `READ COMMITTED`.

The difference between `READ COMMITTED` and `SERIALIZABLE` is that in `READ COMMITTED` mode, each statement in a transaction sees only rows committed before the *statement* started, while in `SERIALIZABLE` mode, all statements in a transaction see only rows committed before the *transaction* started.

The `READ COMMITTED` isolation mode permits greater concurrency and better performance than the `SERIALIZABLE` mode. It allows *non-repeatable reads*, where the values in a row retrieved twice in a transaction can differ because another concurrent transaction has committed changes since the transaction began. `READ COMMITTED` mode also permits *phantom reads*, where a query executed twice in the same transaction can return two different sets of rows.

The `SERIALIZABLE` isolation mode prevents both non-repeatable reads and phantom reads, but at the cost of concurrency and performance. Each concurrent transaction has a consistent view of the database taken at the beginning of execution. A concurrent transaction that attempts to modify data modified by another transaction is rolled back. Applications that execute transactions in `SERIALIZABLE` mode must be prepared to handle transactions that fail due to serialization errors. If `SERIALIZABLE` isolation mode is not required by the application, it is better to use `READ COMMITTED` mode.

The SQL standard specifies that concurrent serializable transactions produce the same database state they would produce if executed sequentially. The MVCC snapshot isolation model prevents dirty reads, non-repeatable reads, and phantom reads without expensive locking, but there are other interactions that can occur between some `SERIALIZABLE` transactions in Greenplum Database that prevent them from being truly serializable. These anomalies can often be attributed to the fact that Greenplum Database does not perform *predicate locking*, which means that a write in one transaction can affect the result of a previous read in another concurrent transaction.

Transactions that run concurrently should be examined to identify interactions that are not prevented by disallowing concurrent updates of the same data. Problems identified can be prevented by using explicit table locks or by requiring the conflicting transactions to update a dummy row introduced to represent the conflict.

The SQL `SET TRANSACTION ISOLATION LEVEL` statement sets the isolation mode for the current transaction. The mode must be set before any `SELECT`, `INSERT`, `DELETE`, `UPDATE`, or `COPY` statements:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
...
COMMIT;
```

The isolation mode can also be specified as part of the `BEGIN` statement:

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

The default transaction isolation mode can be changed for a session by setting the *default_transaction_isolation* configuration property.

Removing Dead Rows from Tables

Updating or deleting a row leaves an expired version of the row in the table. When an expired row is no longer referenced by any active transactions, it can be removed and the space it occupied can be reused. The `VACUUM` command removes expired rows from tables.

When expired rows accumulate in a table, the disk files must be extended to accommodate new rows. Performance suffers due to the increased disk I/O required to execute queries. This condition is called *bloat* and it should be managed by regularly vacuuming tables.

The `VACUUM` command (without `FULL`) can run concurrently with other queries. It removes expired rows from pages, and repacks the remaining rows to consolidate the free space. If the amount of remaining free space is significant, it adds the page to the table's free space map. When Greenplum Database later needs space for new rows, it first consults the table's free space map to find pages with available space. If none are found, new pages will be appended to the file.

`VACUUM` (without `FULL`) does not consolidate pages or reduce the size of the table on disk. The space it recovers is only available through the free space map. To prevent disk files from growing, it is important to run `VACUUM` often enough, at least once per day, to ensure that the available free space can be found through the free space map. It is also important to run `VACUUM` after running a transaction that updates or deletes a large number of rows.

The `VACUUM FULL` command rewrites the table without expired rows, reducing the table to its minimum size. There must be sufficient disk space to create the new table, and the table is locked until `VACUUM FULL` completes. This is very expensive compared to the regular `VACUUM` command, and can be avoided or postponed by vacuuming regularly. It is best to run `VACUUM FULL` during a maintenance period. An alternative to `VACUUM FULL` is to recreate the table with a `CREATE TABLE AS` statement and then drop the old table.

The free space map resides in shared memory and keeps track of free space for all tables and indexes. Each table or index uses about 60 bytes of memory and each page with free space consumes six bytes. Two system configuration parameters configure the size of the free space map:

max_fsm_pages

> Sets the maximum number of disk pages that can be added to the shared free space map. Six bytes of shared memory are consumed for each page slot. The default is 200000. This parameter must be set to at least 16 times the value of *max_fsm_relations*.

max_fsm_relations

> Sets the maximum number of relations that will be tracked in the shared memory free space map. This parameter should be set to a value larger than the total number of *tables* + *indexes* + *system tables*. The default is 1000. About 60 bytes of memory are consumed for each relation per segment instance. It is better to set the parameter too high than too low.

If the free space map is undersized, some disk pages with available space will not be added to the map, and that space cannot be reused until at least the next `VACUUM` command runs. This causes files to grow.

You can run `VACUUM VERBOSE` *`tablename`* to get a report, by segment, of the number of dead rows removed, the number of pages affected, and the number of pages with usable free space.

Query the `pg_class` system table to find out how many pages a table is using across all segments. Be sure to `ANALYZE` the table first to get accurate data.

```
SELECT relname, relpages, reltuples FROM pg_class WHERE relname='tablename';
```

Another useful tool is the `gp_bloat_diag` view in the `gp_toolkit` schema, which identifies bloat in tables by comparing the actual number of pages used by a table to the expected number. See "The gp_toolkit Administrative Schema" in the *Greenplum Database Reference Guide* for more about `gp_bloat_diag`.

Example of Managing Transaction IDs

For Greenplum Database, the transaction ID (XID) value an incrementing 32-bit (2^{32}) value. The maximum unsigned 32-bit value is 4,294,967,295, or about four billion. The XID values restart at 0 after the maximum is reached. Greenplum Database handles the limit of XID values with two features:

- Calculations on XID values using modulo-2^{32} arithmetic that allow Greenplum Database to reuse XID values. The modulo calculations determine the order of transactions, whether one transaction has occurred before or after another, based on the XID.

  Every XID value can have up to two billion (2^{31}) XID values that are considered previous transactions and two billion (2^{31} -1) XID values that are considered newer transactions. The XID values can be considered a circular set of values with no endpoint similar to a 24 hour clock.

  Using the Greenplum Database modulo calculations, as long as two XIDs are within 2^{31} transactions of each other, comparing them yields the correct result.
- A frozen XID value that Greenplum Database uses as the XID for current (visible) data rows. Setting a row's XID to the frozen XID performs two functions.
  - When Greenplum Database compares XIDs using the modulo calculations, the frozen XID is always smaller, earlier, when compared to any other XID. If a row's XID is not set to the frozen XID and 2^{31} new transactions are executed, the row appears to be executed in the future based on the modulo calculation.
  - When the row's XID is set to the frozen XID, the original XID can be used, without duplicating the XID. This keeps the number of data rows on disk with assigned XIDs below (2^{32}).

Simple MVCC Example

This is a simple example of the concepts of a MVCC database and how it manages data and transactions with transaction IDs. This simple MVCC database example consists of a single table:

- The table is a simple table with 2 columns and 4 rows of data.
- The valid transaction ID (XID) values are from 0 up to 9, after 9 the XID restarts at 0.
- The frozen XID is -2. This is different than the Greenplum Database frozen XID.
- Transactions are performed on a single row.
- Only insert and update operations are performed.
- All updated rows remain on disk, no operations are performed to remove obsolete rows.

The example only updates the amount values. No other changes to the table.

The example shows these concepts.

- *How transaction IDs are used to manage multiple, simultaneous transactions on a table.*
- *How transaction IDs are managed with the frozen XID*
- *How the modulo calculation determines the order of transactions based on transaction IDs*

Managing Simultaneous Transactions

This table is the initial table data on disk with no updates. The table contains two database columns for transaction IDs, `xmin` (transaction that created the row) and `xmax` (transaction that updated the row). In the table, changes are added, in order, to the bottom of the table.

Table 5: Example Table

item	amount	xmin	xmax
widget	100	0	null
giblet	200	1	null
sprocket	300	2	null
gizmo	400	3	null

The next table shows the table data on disk after some updates on the amount values have been performed.

xid = 4: `update tbl set amount=208 where item = 'widget'`
xid = 5: `update tbl set amount=133 where item = 'sprocket'`
xid = 6: `update tbl set amount=16 where item = 'widget'`

In the next table, the bold items are the current rows for the table. The other rows are obsolete rows, table data that on disk but is no longer current. Using the xmax value, you can determine the current rows of the table by selecting the rows with `null` value. Greenplum Database uses a slightly different method to determine current table rows.

Table 6: Example Table with Updates

item	amount	xmin	xmax
widget	100	0	4
giblet	**200**	1	**null**
sprocket	300	2	5
gizmo	**400**	3	**null**
widget	208	4	6
sproket	**133**	5	**null**
widget	**16**	6	**null**

The simple MVCC database works with XID values to determine the state of the table. For example, both these independent transactions execute concurrently.

- `UPDATE` command changes the sprocket amount value to `133` (xmin value `5`)
- `SELECT` command returns the value of sprocket.

During the `UPDATE` transaction, the database returns the value of sprocket `300`, until the `UPDATE` transaction completes.

Managing XIDs and the Frozen XID

For this simple example, the database is close to running out of available XID values. When Greenplum Database is close to running out of available XID values, Greenplum Database takes these actions.

- Greenplum Database issues a warning stating that the database is running out of XID values.

```
WARNING: database "database_name" must be vacuumed within number_of_transactions
 transactions
```

- Before the last XID is assigned, Greenplum Database stops accepting transactions to prevent assigning an XID value twice and issues this message.

```
FATAL: database is not accepting commands to avoid wraparound data loss in
 database "database_name"
```

To manage transaction IDs and table data that is stored on disk, Greenplum Database provides the VACUUM command.

- A VACUUM operation frees up XID values so that a table can have more than 10 rows by changing the xmin values to the frozen XID.
- A VACUUM operation manages obsolete or deleted table rows on disk. This database's VACUUM command changes the XID values obsolete to indicate obsolete rows. A Greenplum Database VACUUM operation, without the FULL option, deletes the data opportunistically to remove rows on disk with minimal impact to performance and data availability.

For the example table, a VACUUM operation has been performed on the table. The command updated table data on disk. This version of the VACUUM command performs slightly differently than the Greenplum Database command, but the concepts are the same.

- For the widget and sprocket rows on disk that are no longer current, the rows have been marked as obsolete.
- For the giblet and gizmo rows that are current, the xmin has been changed to the frozen XID.

  The values are still current table values (the row's xmax value is null). However, the table row is visible to all transactions because the xmin value is frozen XID value that is older than all other XID values when modulo calculations are performed.

After the VACUUM operation, the XID values 0, 1, 2, and 3 available for use.

Table 7: Example Table after VACUUM

item	amount	xmin	xmax
widget	100	obsolete	obsolete
giblet	**200**	-2	**null**
sprocket	300	obsolete	obsolete
gizmo	**400**	-2	**null**
widget	208	4	6
sproket	**133**	5	**null**
widget	**16**	6	**null**

When a row disk with the xmin value of -2 is updated, the xmax value is replaced with the transaction XID as usual, and the row on disk is considered obsolete after any concurrent transactions that access the row have completed.

Obsolete rows can be deleted from disk. For Greenplum Database, the VACUUM command, with FULL option, does more extensive processing to reclaim disk space.

Example of XID Modulo Calculations

The next table shows the table data on disk after more UPDATE transactions. The XID values have rolled over and start over at 0. No additional VACUUM operations have been performed.

Table 8: Example Table with Wrapping XID

item	amount	xmin	xmax
widget	100	obsolete	obsolete
giblet	200	-2	1
sprocket	300	obsolete	obsolete
gizmo	400	-2	9
widget	208	4	6
sproket	**133**	5	**null**
widget	16	6	7
widget	**222**	7	**null**
giblet	233	8	0
gizmo	**18**	9	**null**
giblet	88	0	1
giblet	**44**	1	**null**

When performing the modulo calculations that compare XIDs, Greenplum Database, considers the XIDs of the rows and the current range of available XIDs to determine if XID wrapping has occurred between row XIDs.

For the example table XID wrapping has occurred. The XID `1` for giblet row is a later transaction than the XID `7` for widget row based on the modulo calculations for XID values even though the XID value `7` is larger than `1`.

For the widget and sprocket rows, XID wrapping has not occurred and XID `7` is a later transaction than XID `5`.

Chapter 4

About Parallel Data Loading

This topic provides a short introduction to Greenplum Database data loading features.

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.

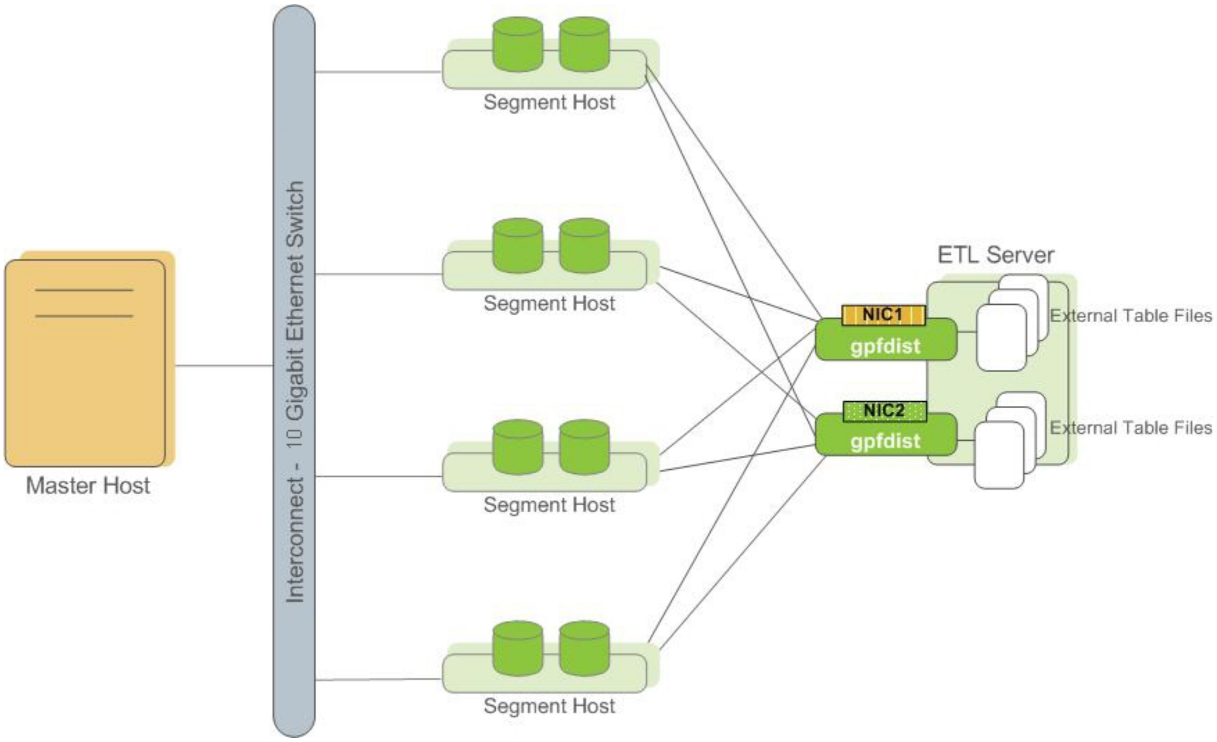


Figure 2: External Tables Using Greenplum Parallel File Server (gpfdist)

Another Greenplum utility, `gpload`, runs a load task that you specify in a YAML-formatted control file. You describe the source data locations, format, transformations required, participating hosts, database destinations, and other particulars in the control file and `gpload` executes the load. This allows you to describe a complex task and execute it in a controlled, repeatable fashion.

Chapter 5

About Redundancy and Failover in Greenplum Database

This topic provides a high-level overview of Greenplum Database high availability features.

You can deploy Greenplum Database without a single point of failure by mirroring components. The following sections describe the strategies for mirroring the main components of a Greenplum system. For a more detailed overview of Greenplum high availability features, see *Overview of Greenplum Database High Availability*.

About Segment Mirroring

When you deploy your Greenplum Database system, you can configure *mirror* segments. Mirror segments allow database queries to fail over to a backup segment if the primary segment becomes unavailable. Mirroring is strongly recommended for production systems and required for Pivotal support.

The secondary (mirror) segment must always reside on a different host than its primary segment to protect against a single host failure. Mirror segments can be arranged over the remaining hosts in the cluster in configurations designed to maximize availability or minimize the performance degradation when hosts or multiple primary segments fail.

Two standard mirroring configurations are available when you initialize or expand a Greenplum system. The default configuration, called *group mirroring*, places all the mirrors for a host's primary segments on one other host in the cluster. The other standard configuration, *spread mirroring*, can be selected with a command-line option. Spread mirroring spreads each host's mirrors over the remaining hosts and requires that there are more hosts in the cluster than primary segments per host.

Figure 3: Spread Mirroring in Greenplum Database shows how table data is distributed across segments when spread mirroring is configured.

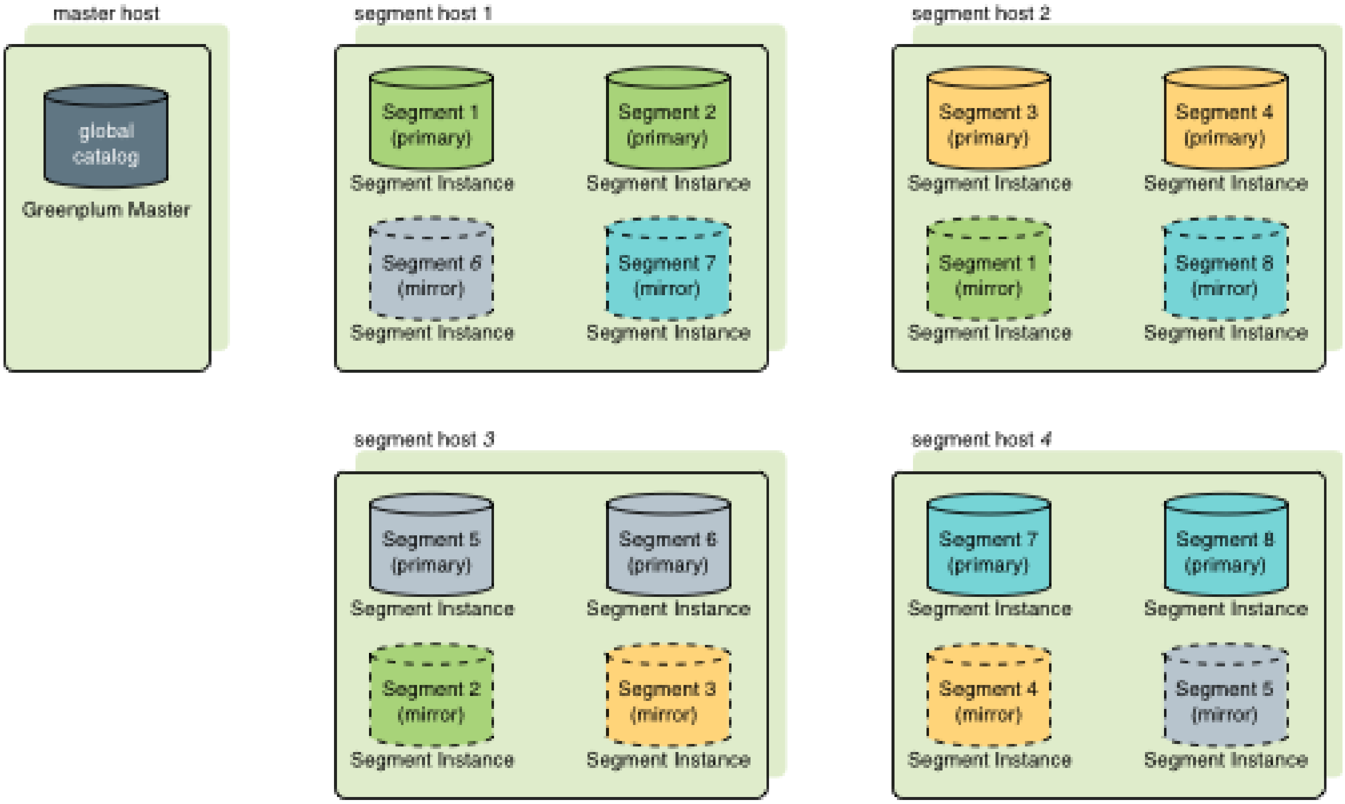


Figure 3: Spread Mirroring in Greenplum Database

Segment Failover and Recovery

When mirroring is enabled in a Greenplum Database system, the system will automatically fail over to the mirror segment 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.

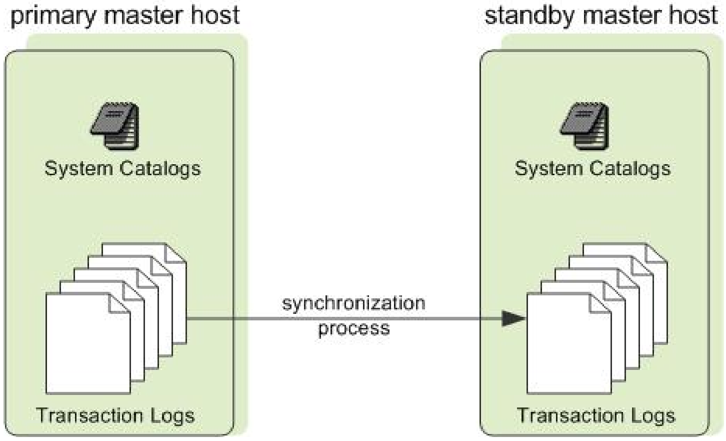


Figure 4: Master Mirroring in Greenplum Database

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 using by deploying dual 10-Gigabit Ethernet switches on your network and redundant 10-Gigabit connections to the Greenplum Database host (master and segment) servers.

Chapter 6

About Database Statistics in Greenplum Database

An overview of statistics gathered by the `ANALYZE` command in Greenplum Database.

Statistics are metadata that describe the data stored in the database. The query optimizer needs up-to-date statistics to choose the best execution plan for a query. For example, if a query joins two tables and one of them must be broadcast to all segments, the optimizer can choose the smaller of the two tables to minimize network traffic.

The statistics used by the optimizer are calculated and saved in the system catalog by the `ANALYZE` command. There are three ways to initiate an analyze operation:

- You can run the `ANALYZE` command directly.
- You can run the `analyzedb` management utility outside of the database, at the command line.
- An automatic analyze operation can be triggered when DML operations are performed on tables that have no statistics or when a DML operation modifies a number of rows greater than a specified threshold.

These methods are described in the following sections. The `VACUUM ANALYZE` command is another way to initiate an analyze operation, but its use is discouraged because vacuum and analyze are different operations with different purposes.

Calculating statistics consumes time and resources, so Greenplum Database produces estimates by calculating statistics on samples of large tables. In most cases, the default settings provide the information needed to generate correct execution plans for queries. If the statistics produced are not producing optimal query execution plans, the administrator can tune configuration parameters to produce more accurate stastistics by increasing the sample size or the granularity of statistics saved in the system catalog. Producing more accurate statistics has CPU and storage costs and may not produce better plans, so it is important to view explain plans and test query performance to ensure that the additional statistics-related costs result in better query performance.

System Statistics

Table Size

The query planner seeks to minimize the disk I/O and network traffic required to execute a query, using estimates of the number of rows that must be processed and the number of disk pages the query must access. The data from which these estimates are derived are the `pg_class` system table columns `reltuples` and `relpages`, which contain the number of rows and pages at the time a `VACUUM` or `ANALYZE` command was last run. As rows are added or deleted, the numbers become less accurate. However, an accurate count of disk pages is always available from the operating system, so as long as the ratio of `reltuples` to `relpages` does not change significantly, the optimizer can produce an estimate of the number of rows that is sufficiently accurate to choose the correct query execution plan.

When the `reltuples` column differs significantly from the row count returned by `SELECT COUNT(*)`, an analyze should be performed to update the statistics.

When a `REINDEX` command finishes recreating an index, the `relpages` and `reltuples` columns are set to zero. The `ANALYZE` command should be run on the base table to update these columns.

The pg_statistic System Table and pg_stats View

The `pg_statistic` system table holds the results of the last `ANALYZE` operation on each database table. There is a row for each column of every table. It has the following columns:

starelid

The object ID of the table or index the column belongs to.

staatnum

The number of the described column, beginning with 1.

stanullfrac

The fraction of the column's entries that are null.

stawidth

The average stored width, in bytes, of non-null entries.

stadistinct

A positive number is an estimate of the number of distinct values in the column; the number is not expected to vary with the number of rows. A negative value is the number of distinct values divided by the number of rows, that is, the ratio of rows with distinct values for the column, negated. This form is used when the number of distinct values increases with the number of rows. A unique column, for example, has an `n_distinct` value of -1.0. Columns with an average width greater than 1024 are considered unique.

stakind*N*

A code number indicating the kind of statistics stored in the *N*th slot of the `pg_statistic` row.

staop*N*

An operator used to derive the statistics stored in the *N*th slot. For example, a histogram slot would show the < operator that defines the sort order of the data.

stanumbers*N*

float4 array containing numerical statistics of the appropriate kind for the *N*th slot, or NULL if the slot kind does not involve numerical values.

stavalues*N*

Column data values of the appropriate kind for the *N*th slot, or NULL if the slot kind does not store any data values. Each array's element values are actually of the specific column's data type, so there is no way to define these columns' types more specifically than *anyarray*.

The statistics collected for a column vary for different data types, so the `pg_statistic` table stores statistics that are appropriate for the data type in four *slots*, consisting of four columns per slot. For example, the first slot, which normally contains the most common values for a column, consists of the columns `stakind1`, `staop1`, `stanumbers1`, and `stavalues1`.

The `stakindN` columns each contain a numeric code to describe the type of statistics stored in their slot. The `stakind` code numbers from 1 to 99 are reserved for core PostgreSQL data types. Greenplum Database uses code numbers 1, 2, and 3. A value of 0 means the slot is unused. The following table describes the kinds of statistics stored for the three codes.

Table 9: Contents of pg_statistic "slots"

stakind Code	Description
1	*Most CommonValues (MCV) Slot* • `staop` contains the object ID of the "=" operator, used to decide whether values are the same or not. • `stavalues` contains an array of the *K* most common non-null values appearing in the column. • `stanumbers` contains the frequencies (fractions of total row count) of the values in the `stavalues` array. The values are ordered in decreasing frequency. Since the arrays are variable-size, *K* can be chosen by the statistics collector. Values must occur more than once to be added to the `stavalues` array; a unique column has no MCV slot.
2	*Histogram Slot* – describes the distribution of scalar data. • `staop` is the object ID of the "<" operator, which describes the sort ordering. • `stavalues` contains *M* (where *M*>=2) non-null values that divide the non-null column data values into *M*-1 bins of approximately equal population. The first `stavalues` item is the minimum value and the last is the maximum value. • `stanumbers` is not used and should be null. If a Most Common Values slot is also provided, then the histogram describes the data distribution after removing the values listed in the MCV array. (It is a *compressed histogram* in the technical parlance). This allows a more accurate representation of the distribution of a column with some very common values. In a column with only a few distinct values, it is possible that the MCV list describes the entire data population; in this case the histogram reduces to empty and should be omitted.
3	*Correlation Slot* – describes the correlation between the physical order of table tuples and the ordering of data values of this column. • `staop` is the object ID of the "<" operator. As with the histogram, more than one entry could theoretically appear. • `stavalues` is not used and should be NULL. • `stanumbers` contains a single entry, the correlation coefficient between the sequence of data values and the sequence of their actual tuple positions. The coefficient ranges from +1 to -1.

The `pg_stats` view presents the contents of `pg_statistic` in a friendlier format. The `pg_stats` view has the following columns:

schemaname	The name of the schema containing the table.
tablename	The name of the table.
attname	The name of the column this row describes.
null_frac	The fraction of column entries that are null.
avg_width	The average storage width in bytes of the column's entries, calculated as `avg(pg_column_size(`*`column_name`*`))`.
n_distinct	A positive number is an estimate of the number of distinct values in the column; the number is not expected to vary with the number of rows. A

	negative value is the number of distinct values divided by the number of rows, that is, the ratio of rows with distinct values for the column, negated. This form is used when the number of distinct values increases with the number of rows. A unique column, for example, has an `n_distinct` value of -1.0. Columns with an average width greater than 1024 are considered unique.
most_common_vals	An array containing the most common values in the column, or null if no values seem to be more common. If the `n_distinct` column is -1, `most_common_vals` is null. The length of the array is the lesser of the number of actual distinct column values or the value of the `default_statistics_target` configuration parameter. The number of values can be overridden for a column using `ALTER TABLE` *`table`* `SET COLUMN` *`column`* `SET STATISTICS` *`N`*.
most_common_freqs	An array containing the frequencies of the values in the `most_common_vals` array. This is the number of occurrences of the value divided by the total number of rows. The array is the same length as the `most_common_vals` array. It is null if `most_common_vals` is null.
histogram_bounds	An array of values that divide the column values into groups of approximately the same size. A histogram can be defined only if there is a `max()` aggregate function for the column. The number of groups in the histogram is the same as the `most_common_vals` array size.
correlation	Greenplum Database does not calculate the correlation statistic.

Newly created tables and indexes have no statistics. You can check for tables with missing statistics using the `gp_stats_missing` view, which is in the `gp_toolkit` schema:

```
SELECT * from gp_toolkit.gp_stats_missing;
```

Sampling

When calculating statistics for large tables, Greenplum Database creates a smaller table by sampling the base table. If the table is partitioned, samples are taken from all partitions.

If the number of rows in the base table is estimated to be less than the value of the `gp_statistics_sampling_threshold` configuration parameter, the entire base table is used to calculate the statistics.

If a sample table is created, the number of rows in the sample is calculated to provide a maximum acceptable relative error. The amount of acceptable error is specified with the `gp_analyze_relative_error` system configuration parameter, which is set to .25 (25%) by default. This is usually sufficiently accurate to generate correct query plans. If `ANALYZE` is not producing good estimates for a table column, you can increase the sample size by setting the `gp_analyze_relative_error` configuration parameter to a lower value. Beware that setting this parameter to a low value can lead to a very large sample size and dramatically increase analyze time.

Updating Statistics

Running ANALYZE with no arguments updates statistics for all tables in the database. This could take a very long time, so it is better to analyze tables selectively after data has changed. You can also analyze a subset of the columns in a table, for example columns used in joins, WHERE clauses, SORT clauses, GROUP BY clauses, or HAVING clauses.

Analyzing a severely bloated table can generate poor statistics if the sample contains empty pages, so it is good practice to vacuum a bloated table before analyzing it.

See the *SQL Command Reference* in the *Greenplum Database Reference Guide* for details of running the ANALYZE command.

Refer to the *Greenplum Database Management Utility Reference* for details of running the analyzedb command.

Analyzing Partitioned and Append-Optimized Tables

When the ANALYZE command is run on a partitioned table, it analyzes each leaf-level subpartition, one at a time. You can run ANALYZE on just new or changed partition files to avoid analyzing partitions that have not changed. If a table is partitioned, you can analyze just new or changed partitions.

The analyzedb command-line utility, introduced in Greenplum Database 4.3.5 skips unchanged partitions automatically. It also runs concurrent sessions so it can analyze several partitions concurrently. It runs five sessions by default, but the number of sessions can be set from 1 to 10 with the -p command-line option. Each time analyzedb runs, it saves state information for append-optimized tables and partitions in the db_analyze directory in the master data directory. The next time it runs, analyzedb compares the current state of each table with the saved state and skips analyzing a table or partition if it is unchanged. Heap tables are always analyzed.

If the Pivotal Query Optimizer is enabled, you also need to run ANALYZE ROOTPARTITION to refresh the root partition statistics. The Pivotal Query Optimizer requires statistics at the root level for partitioned tables. The legacy optimizer does not use these statistics. Enable the Pivotal Query Optimizer by setting both the optimizer and optimizer_analyze_root_partition system configuration parameters to on. The root level statistics are then updated when you run ANALYZE or ANALYZE ROOTPARTITION. The time to run ANALYZE ROOTPARTITION is similar to the time to analyze a single partition since ANALYZE ROOTPARTITION. The analyzedb utility updates root partition statistics by default but you can add the the --skip_root_stats option to leave root partition statistics empty if you do not use the Pivotal Query Optimizer.

Configuring Statistics

There are several options for configuring Greenplum Database statistics collection.

Statistics Target

The statistics target is the size of the most_common_vals, most_common_freqs, and histogram_bounds arrays for an individual column. By default, the target is 25. The default target can be changed by setting a server configuration parameter and the target can be set for any column using the ALTER TABLE command. Larger values increase the time needed to do ANALYZE, but may improve the quality of the legacy query optimizer (planner) estimates.

Set the system default statistics target to a different value by setting the default_statistics_target server configuration parameter. The default value is usually sufficient, and you should only raise or lower it if your tests demonstrate that query plans improve with the new target. For example, to raise the default statistics target from 25 to 50 you can use the gpconfig utility:

```
gpconfig -c default_statistics_target -v 50
```

The statististics target for individual columns can be set with the ALTER TABLE command. For example, some queries can be improved by increasing the target for certain columns, especially columns that have irregular distributions. You can set the target to zero for columns that never contribute to query otpimization. When the target is 0, ANALYZE ignores the column. For example, the following ALTER TABLE command sets the statistics target for the notes column in the emp table to zero:

```
ALTER TABLE emp ALTER COLUMN notes SET STATISTICS 0;
```

The statistics target can be set in the range 0 to 1000, or set it to -1 to revert to using the system default statistics target.

Setting the statistics target on a parent partition table affects the child partitions. If you set statistics to 0 on some columns on the parent table, the statistics for the same columns are set to 0 for all children partitions. However, if you later add or exchange another child partition, the new child partition will use either the default statistics target or, in the case of an exchange, the previous statistics target. Therefore, if you add or exchange child partitions, you should set the statistics targets on the new child table.

Automatic Statistics Collection

Greenplum Database can be set to automatically run ANALYZE on a table that either has no statistics or has changed significantly when certain operations are performed on the table. For partitioned tables, automatic statistics collection is only triggered when the operation is run directly on a leaf table, and then only the leaf table is analyzed.

Automatic statistics collection has three modes:

- none disables automatic statistics collection.
- on_no_stats triggers an analyze operation for a table with no existing statistics when any of the commands CREATE TABLE AS SELECT, INSERT, or COPY are executed on the table.
- on_change triggers an analyze operation when any of the commands CREATE TABLE AS SELECT, UPDATE, DELETE, INSERT, or COPY are executed on the table and the number of rows affected exceeds the threshold defined by the gp_autostats_on_change_threshold configuration parameter.

The automatic statistics collection mode is set separately for commands that occur within a procedural language function and commands that execute outside of a function:

- The gp_autostats_mode configuration parameter controls automatic statistics collection behavior outside of functions and is set to on_no_stats by default.
- The gp_autostats_mode_in_functions parameter controls the behavior when table operations are performed within a procedural language function and is set to none by default.

With the on_change mode, ANALYZE is triggered only if the number of rows affected exceeds the threshold defined by the gp_autostats_on_change_threshold configuration parameter. The default value for this parameter is a very high value, 2147483647, which effectively disables automatic statistics collection; you must set the threshold to a lower number to enable it. The on_change mode could trigger large, unexpected analyze operations that could disrupt the system, so it is not recommended to set it globally. It could be useful in a session, for example to automatically analyze a table following a load.

To disable automatic statistics collection outside of functions, set the gp_autostats_mode parameter to none:

```
gpconfigure -c gp_autostats_mode -v none
```

To enable automatic statistics collection in functions for tables that have no statistics, change gp_autostats_mode_in_functions to on_no_stats:

```
gpconfigure -c gp_autostats_mode_in_functions -v on_no_stats
```

Set the log_autostats system configuration parameter to on if you want to log automatic statistics collection operations.

Part II

Managing a Greenplum System

This section describes basic system administration tasks performed by a Greenplum Database system administrator.

This section contains the following topics:

- *Starting and Stopping Greenplum Database*
- *Accessing the Database*
- *Configuring the Greenplum Database System*
- *Enabling High Availability Features*
- *Backing Up and Restoring Databases*
- *Expanding a Greenplum System*
- *Migrating Data with gptransfer*
- *Defining Database Objects*
- *Routine System Maintenance Tasks*

Chapter 7

Starting and Stopping Greenplum Database

In a Greenplum Database DBMS, the database server instances (the master and all segments) are started or stopped across all of the hosts in the system in such a way that they can work together as a unified DBMS.

Because a Greenplum Database system is distributed across many machines, the process for starting and stopping a Greenplum Database system is different than the process for starting and stopping a regular PostgreSQL DBMS.

Use the `gpstart` and `gpstop` utilities to start and stop Greenplum Database, respectively. These utilities are located in the `$GPHOME/bin` directory on your Greenplum Database master host.

> **Important:**
>
> Do not issue a `KILL` command to end any Postgres process. Instead, use the database command `pg_cancel_backend()`.

For information about `gpstart` and `gpstop`, see the *Greenplum Database Utility Guide*.

Starting Greenplum Database

Start an initialized Greenplum Database system by running the `gpstart` utility on the master instance.

Use the `gpstart` utility to start a Greenplum Database system that has already been initialized by the `gpinitsystem` utility, but has been stopped by the `gpstop` utility. The `gpstart` utility starts Greenplum Database by starting all the Postgres database instances on the Greenplum Database cluster. `gpstart` orchestrates this process and performs the process in parallel.

- Run `gpstart` on the master host to start Greenplum Database:

```
$ gpstart
```

Restarting Greenplum Database

Stop the Greenplum Database system and then restart it.

The `gpstop` utility with the `-r` option can stop and then restart Greenplum Database after the shutdown completes.

- To restart Greenplum Database, enter the following command on the master host:

```
$ gpstop -r
```

Reloading Configuration File Changes Only

Reload changes to Greenplum Database configuration files without interrupting the system.

The `gpstop` utility can reload changes to the `pg_hba.conf` configuration file and to *runtime* parameters in the master `postgresql.conf` file and `pg_hba.conf` file without service interruption. Active sessions pick up changes when they reconnect to the database. Many server configuration parameters require a full system restart (`gpstop -r`) to activate. For information about server configuration parameters, see the *Greenplum Database Reference Guide*.

- Reload configuration file changes without shutting down the system using the `gpstop` utility:

```
$ gpstop -u
```

Starting the Master in Maintenance Mode

Start only the master to perform maintenance or administrative tasks without affecting data on the segments.

Maintenance mode should only be used with direction from Pivotal Technical Support. For example, you could connect to a database only on the master instance in maintenance mode and edit system catalog settings. For more information about system catalog tables, see the *Greenplum Database Reference Guide*.

1. Run `gpstart` using the -m option:

```
$ gpstart -m
```

2. Connect to the master in maintenance mode to do catalog maintenance. For example:

```
$ PGOPTIONS='-c gp_session_role=utility' psql template1
```

3. After completing your administrative tasks, stop the master in utility mode. Then, restart it in production mode.

```
$ gpstop -mr
```

Warning:

Incorrect use of maintenance mode connections can result in an inconsistent system state. Only Technical Support should perform this operation.

Stopping Greenplum Database

The `gpstop` utility stops or restarts your Greenplum Database system and always runs on the master host. When activated, `gpstop` stops all `postgres` processes in the system, including the master and all segment instances. The `gpstop` utility uses a default of up to 64 parallel worker threads to bring down the Postgres instances that make up the Greenplum Database cluster. The system waits for any active transactions to finish before shutting down. To stop Greenplum Database immediately, use fast mode.

- To stop Greenplum Database:

```
$ gpstop
```

- To stop Greenplum Database in fast mode:

```
$ gpstop -M fast
```

Chapter 8

Accessing the Database

This topic describes the various client tools you can use to connect to Greenplum Database, and how to establish a database session.

Establishing a Database Session

Users can connect to Greenplum Database using a PostgreSQL-compatible client program, such as `psql`. Users and administrators *always* connect to Greenplum Database through the *master*; the segments cannot accept client connections.

In order to establish a connection to the Greenplum Database master, you will need to know the following connection information and configure your client program accordingly.

Table 10: Connection Parameters

Connection Parameter	Description	Environment Variable
Application name	The application name that is connecting to the database. The default value, held in the `application_name` connection parameter is *psql*.	`$PGAPPNAME`
Database name	The name of the database to which you want to connect. For a newly initialized system, use the `template1` database to connect for the first time.	`$PGDATABASE`
Host name	The host name of the Greenplum Database master. The default host is the local host.	`$PGHOST`
Port	The port number that the Greenplum Database master instance is running on. The default is 5432.	`$PGPORT`
User name	The database user (role) name to connect as. This is not necessarily the same as your OS user name. Check with your Greenplum administrator if you are not sure what you database user name is. Note that every Greenplum Database system has one superuser account that is created automatically at initialization time. This account has the same name as the OS name of the user who initialized the Greenplum system (typically `gpadmin`).	`$PGUSER`

Connecting with psql provides example commands for connecting to Greenplum Database.

Supported Client Applications

Users can connect to Greenplum Database using various client applications:

- A number of *Greenplum Database Client Applications* are provided with your Greenplum installation. The `psql` client application provides an interactive command-line interface to Greenplum Database.
- *pgAdmin III for Greenplum Database* is an enhanced version of the popular management tool pgAdmin III. Since version 1.10.0, the pgAdmin III client available from PostgreSQL Tools includes support for Greenplum-specific features. Installation packages are available for download from the *pgAdmin download site* .
- Using standard *Database Application Interfaces*, such as ODBC and JDBC, users can create their own client applications that interface to Greenplum Database. Because Greenplum Database is based on PostgreSQL, it uses the standard PostgreSQL database drivers.
- Most *Third-Party Client Tools* that use standard database interfaces, such as ODBC and JDBC, can be configured to connect to Greenplum Database.

Greenplum Database Client Applications

Greenplum Database comes installed with a number of client utility applications located in the `$GPHOME/bin` directory of your Greenplum Database master host installation. The following are the most commonly used client utility applications:

Table 11: Commonly used client applications

Name	Usage
`createdb`	create a new database
`createlang`	define a new procedural language
`createuser`	define a new database role
`dropdb`	remove a database
`droplang`	remove a procedural language
`dropuser`	remove a role
`psql`	PostgreSQL interactive terminal
`reindexdb`	reindex a database
`vacuumdb`	garbage-collect and analyze a database

When using these client applications, you must connect to a database through the Greenplum master instance. You will need to know the name of your target database, the host name and port number of the master, and what database user name to connect as. This information can be provided on the command-line using the options `-d`, `-h`, `-p`, and `-U` respectively. If an argument is found that does not belong to any option, it will be interpreted as the database name first.

All of these options have default values which will be used if the option is not specified. The default host is the local host. The default port number is 5432. The default user name is your OS system user name, as is the default database name. Note that OS user names and Greenplum Database user names are not necessarily the same.

If the default values are not correct, you can set the environment variables `PGDATABASE`, `PGHOST`, `PGPORT`, and `PGUSER` to the appropriate values, or use a `psql ~/.pgpass` file to contain frequently-used passwords.

For information about Greenplum Database environment variables, see the *Greenplum Database Reference Guide*. For information about psql, see the *Greenplum Database Utility Guide*.

Connecting with psql

Depending on the default values used or the environment variables you have set, the following examples show how to access a database via `psql`:

```
$ psql -d gpdatabase -h master_host -p 5432 -U gpadmin
```

```
$ psql gpdatabase
```

```
$ psql
```

If a user-defined database has not yet been created, you can access the system by connecting to the `template1` database. For example:

```
$ psql template1
```

After connecting to a database, `psql` provides a prompt with the name of the database to which `psql` is currently connected, followed by the string `=>` (or `=#` if you are the database superuser). For example:

```
gpdatabase=>
```

At the prompt, you may type in SQL commands. A SQL command must end with a `;` (semicolon) in order to be sent to the server and executed. For example:

```
=> SELECT * FROM mytable;
```

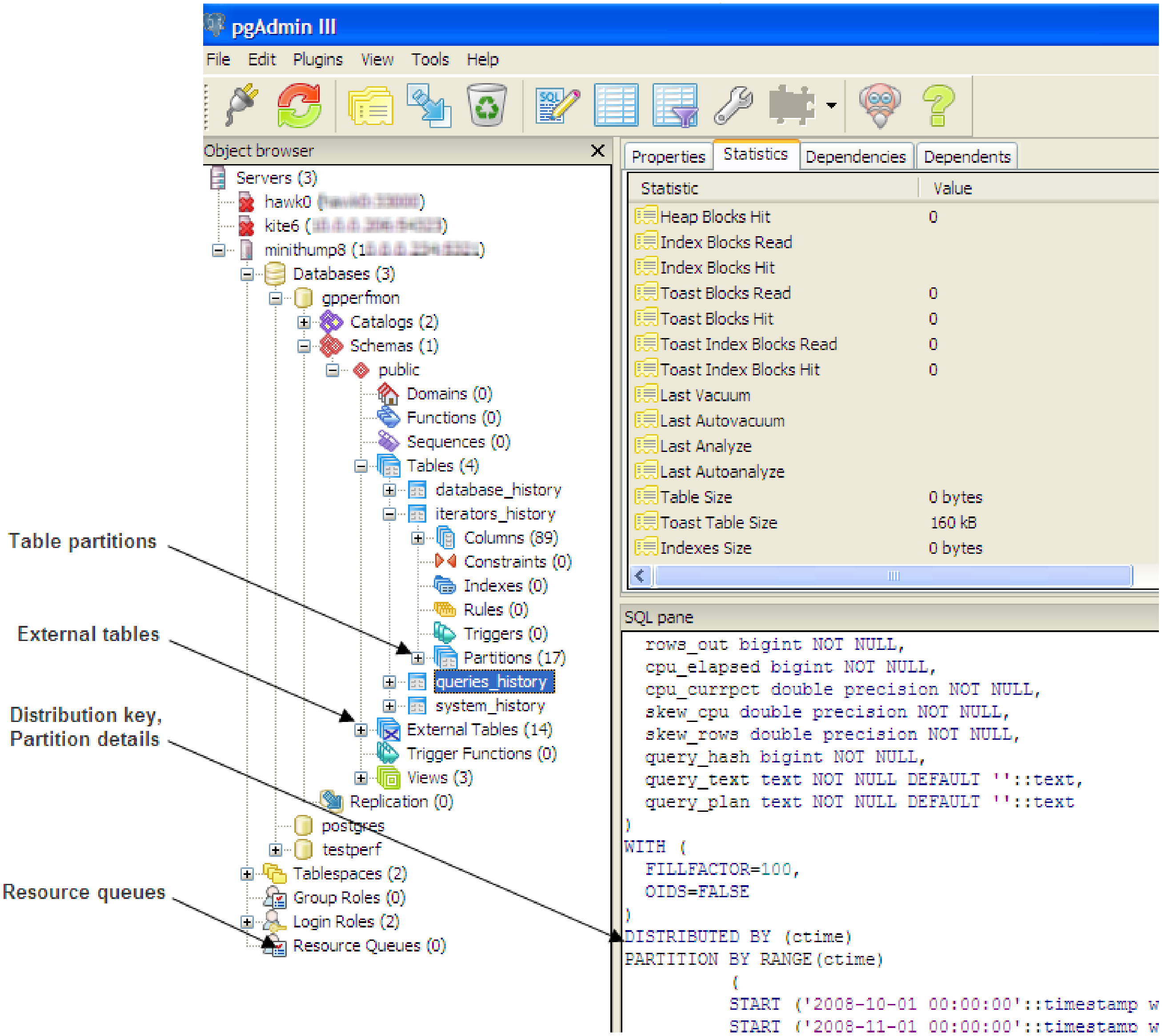
See the *Greenplum Reference Guide* for information about using the `psql` client application and SQL commands and syntax.

pgAdmin III for Greenplum Database

If you prefer a graphic interface, use pgAdmin III for Greenplum Database. This GUI client supports PostgreSQL databases with all standard pgAdmin III features, while adding support for Greenplum-specific features.

pgAdmin III for Greenplum Database supports the following Greenplum-specific features:

- External tables
- Append-optimized tables, including compressed append-optimized tables
- Table partitioning
- Resource queues
- Graphical `EXPLAIN ANALYZE`
- Greenplum server configuration parameters



Statistic	Value
Heap Blocks Hit	0
Index Blocks Read	
Index Blocks Hit	
Toast Blocks Read	0
Toast Blocks Hit	0
Toast Index Blocks Read	0
Toast Index Blocks Hit	0
Last Vacuum	
Last Autovacuum	
Last Analyze	
Last Autoanalyze	
Table Size	0 bytes
Toast Table Size	160 kB
Indexes Size	0 bytes

```
  rows_out bigint NOT NULL,
  cpu_elapsed bigint NOT NULL,
  cpu_currpct double precision NOT NULL,
  skew_cpu double precision NOT NULL,
  skew_rows double precision NOT NULL,
  query_hash bigint NOT NULL,
  query_text text NOT NULL DEFAULT ''::text,
  query_plan text NOT NULL DEFAULT ''::text
)
WITH (
  FILLFACTOR=100,
  OIDS=FALSE
)
DISTRIBUTED BY (ctime)
PARTITION BY RANGE(ctime)
          (
          START ('2008-10-01 00:00:00'::timestamp w
          START ('2008-11-01 00:00:00'::timestamp w
```

Figure 5: Greenplum Options in pgAdmin III

Installing pgAdmin III for Greenplum Database

The installation package for pgAdmin III for Greenplum Database is available for download from the official pgAdmin III download site (*http://www.pgadmin.org*). Installation instructions are included in the installation package.

Documentation for pgAdmin III for Greenplum Database

For general help on the features of the graphical interface, select **Help contents** from the **Help** menu.

For help with Greenplum-specific SQL support, select **Greenplum Database Help** from the **Help** menu. If you have an active internet connection, you will be directed to online Greenplum SQL reference documentation. Alternately, you can install the Greenplum Client Tools package. This package contains SQL reference documentation that is accessible to the help links in pgAdmin III.

Performing Administrative Tasks with pgAdmin III

This topic highlights two of the many Greenplum Database administrative tasks you can perform with pgAdmin III: editing the server configuration, and viewing a graphical representation of a query plan.

Editing the Server Configuration

The pgAdmin III interface provides two ways to update the server configuration in `postgresql.conf`: locally, through the **File** menu, and remotely on the server through the **Tools** menu. Editing the server configuration remotely may be more convenient in many cases, because it does not require you to upload or copy `postgresql.conf`.

To edit server configuration remotely

1. Connect to the server whose configuration you want to edit. If you are connected to multiple servers, make sure that the correct server is highlighted in the object browser in the left pane.
2. **Select Tools** > **Server Configuration** > **postgresql.conf**. The Backend Configuration Editor opens, displaying the list of available and enabled server configuration parameters.
3. Locate the parameter you want to edit, and double click on the entry to open the Configuration settings dialog.
4. Enter the new value for the parameter, or select/deselect **Enabled** as desired and click **OK**.
5. If the parameter can be enabled by reloading server configuration, click the green reload icon, or select **File** > **Reload** server. Many parameters require a full restart of the server.

Viewing a Graphical Query Plan

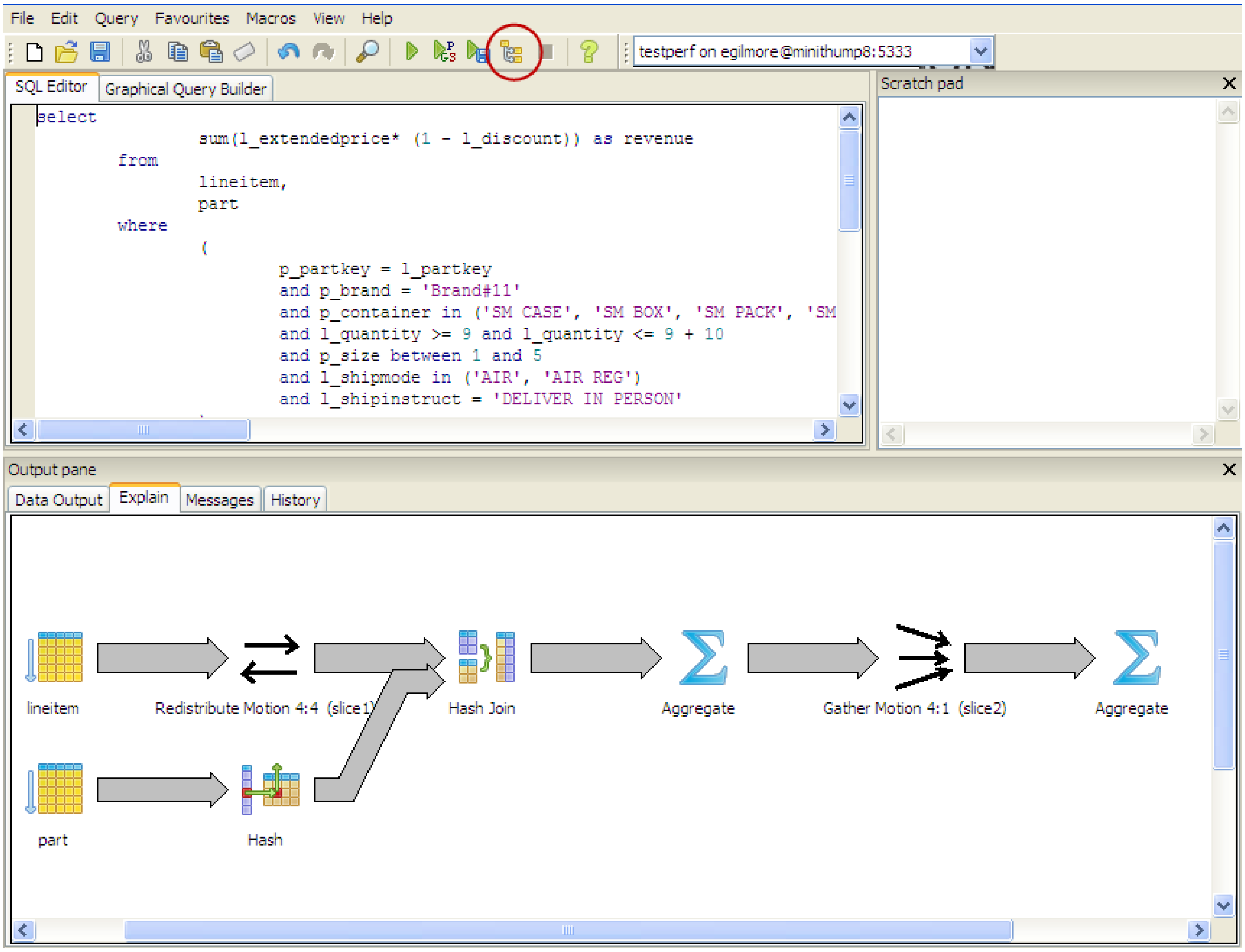
Using the pgAdmin III query tool, you can run a query with EXPLAIN to view the details of the query plan. The output includes details about operations unique to Greenplum distributed query processing such as plan slices and motions between segments. You can view a graphical depiction of the plan as well as the text-based data output.

1. With the correct database highlighted in the object browser in the left pane, select **Tools** > **Query** tool.
2. Enter the query by typing in the SQL Editor, dragging objects into the Graphical Query Builder, or opening a file.
3. Select **Query** > **Explain** options and verify the following options:

Option	Description
Verbose	this must be deselected if you want to view a graphical depiction of the query plan

Option	Description
Analyze	select this option if you want to run the query in addition to viewing the plan

4. Trigger the operation by clicking the Explain query option at the top of the pane, or by selecting **Query** > **Explain**.
   The query plan displays in the Output pane at the bottom of the screen. Select the Explain tab to view the graphical output. For example:



```
select
                sum(l_extendedprice* (1 - l_discount)) as revenue
        from
                lineitem,
                part
        where
                (
                        p_partkey = l_partkey
                        and p_brand = 'Brand#11'
                        and p_container in ('SM CASE', 'SM BOX', 'SM PACK', 'SM
                        and l_quantity >= 9 and l_quantity <= 9 + 10
                        and p_size between 1 and 5
                        and l_shipmode in ('AIR', 'AIR REG')
                        and l_shipinstruct = 'DELIVER IN PERSON'
```

Figure 6: Graphical Query Plan in pgAdmin III

Using the PgBouncer Connection Pooler

The PgBouncer utility manages connection pools for PostgreSQL and Greenplum Database connections.

The Greenplum Database installation includes the PgBouncer connection pooling software. The following topics describe how to set up and use PgBouncer with Greenplum Database. See the *PgBouncer Web site* for information about using PgBouncer with PostgreSQL.

- *Overview*
- *Configuring and Starting PgBouncer*
- *Managing PgBouncer*
- *Setting up LDAP Authentication with PgBouncer*
- *Securing PgBouncer Connections with stunnel*

Also, see reference information for PgBouncer in the *Greenplum Database Utility Guide*.

Overview

A database connection pool is a cache of database connections. Once a pool of connections is established, connection pooling eliminates the overhead of creating database connections, so clients connect much faster and the server load is reduced.

The PgBouncer connection pooler, from the PostgreSQL community, is included with Greenplum Database. PgBouncer can manage connection pools for multiple databases, and databases may be on different Greenplum Database clusters or PostgreSQL backends. PgBouncer creates a pool for each database user and database combination. A pooled connection can only be reused for another connection request for the same user and database.

The client application requires no software changes, but connects to the connection pool's host and port instead of the Greenplum Database master host and port. PgBouncer either creates a new database connection or reuses an existing connection. When the client disconnects, the connection is returned to the pool for re-use.

PgBouncer supports the standard connection interface that PostgreSQL and Greenplum Database share. A client requesting a database connection provides the host name and port where PgBouncer is running, as well as the database name, username, and password. PgBouncer looks up the requested database (which may be an alias for the actual database) in its configuration file to find the host name, port, and database name for the database connection. The configuration file entry also determines how to authenticate the user and what database role will be used for the connection—a "forced user" can override the username provided with the client's connection request.

PgBouncer requires an authentication file, a text file that contains a list of users and passwords. Passwords may be either clear text, MD5-encoded, or an LDAP/AD lookup string. You can also set up PgBouncer to query the destination database for users that are not in the authentication file.

PgBouncer shares connections in one of three pool modes:

- *Session pooling* – When a client connects, a connection is assigned to it as long as it remains connected. When the client disconnects, the connection is placed back into the pool.
- *Transaction pooling* – A connection is assigned to a client for the duration of a transaction. When PgBouncer notices the transaction is done, the connection is placed back into the pool. This mode can be used only with applications that do not use features that depend upon a session.
- *Statement pooling* – Statement pooling is like transaction pooling, but multi-statement transactions are not allowed. This mode is intended to enforce autocommit mode on the client and is targeted for PL/Proxy on PostgreSQL.

A default pool mode can be set for the PgBouncer instance and the mode can be overridden for individual databases and users.

By connecting to a virtual `pgbouncer` database, you can monitor and manage PgBouncer using SQL-like commands. Configuration parameters can be changed without having to restart PgBouncer, and the configuration file can be reloaded to pick up changes.

PgBouncer does not yet support SSL connections. If you want to encrypt traffic between clients and PgBouncer, you can use *stunnel*, a free software utility that creates TLS-encrypted tunnels using the OpenSSL cryptography library. See *Securing PgBouncer Connections with stunnel* for directions.

Configuring and Starting PgBouncer

PgBouncer can be run on the Greenplum Database master or on another server. If you install PgBouncer on a separate server, you can easily switch clients to the standby master by updating the PgBouncer configuration file and reloading the configuration using the PgBouncer Administration Console.

Follow these steps to set up PgBouncer.

1. Create a PgBouncer configuration file, for example `pgbouncer.ini`. Here is a simple configuration file:

```
[databases]
template1 = host=127.0.0.1 port=5432 dbname=template1
mydb = host=127.0.0.1 port=5432 dbname=mydb

[pgbouncer]
pool_mode = session
listen_port = 6543
listen_addr = 127.0.0.1
auth_type = md5
auth_file = users.txt
logfile = pgbouncer.log
pidfile = pgbouncer.pid
admin_users = gpadmin
```

   The file is in the `.ini` file format and has three sections: *databases*, *pgbouncer*, and *users*. The databases section lists databases with their connection details. The pgbouncer section configures the PgBouncer instance.

2. Create an authentication file. The name of the file must match the `auth_file` parameter in the `pgbouncer.ini` file, `users.txt` in this example. Each line contains a user name and password. The format of the password string matches the `auth_type` parameter in the PgBouncer configuration file. If the `auth_type` parameter is `plain`, the password string is a clear text password, for example:

```
"gpadmin" "gpadmin1234"
```

   The `auth_type` in the following example is `md5`, so the authentication field must be MD5-encoded. The format for an MD5-encoded password is:

```
"md5" + MD5(<password><username>)
```

   You can use the Linux `md5sum` command to calculate the MD5 string. For example, if the `gpadmin` password is `admin1234` the following command prints the string for the password field:

```
$ user=gpadmin; passw=admin1234; echo -n md5; echo $passwd$user | md5sum
md53ce96652dedd8226c498e09ae2d26220
```

   And here is the MD5-encoded entry for the `gpadmin` user in the PgBouncer authentication file:

```
"gpadmin" "md53ce96652dedd8226c498e09ae2d26220"
```

   To authenticate users against an LDAP or Active Directory server, the password field contains an LDAP lookup string. For example:

```
"gpdbuser1" "ldap://10.0.0.11:10389/uid=gpdbuser1,ou=users,ou=system"
```

For Active Directory, a user entry looks like this example:

```
"gpdbuser2" "ldap://10.0.0.12:389/gpdbuser2"
```

See *Setting up LDAP Authentication with PgBouncer* for the steps to configure PgBouncer to authenticate users with an LDAP server.

For details about the authentication file, see the "Authentication File Format" section of the PgBouncer reference in the *Greenplum Database Utility Guide*.

3. Launch `pgbouncer`:

   ```
   $ $GPHOME/bin/pgbouncer -d pgbouncer.ini
   ```

   The `-d` or `--daemon` option runs PgBouncer as a background process. See the PgBouncer reference in the *Greenplum Database Utility Guide* for the `pgbouncer` command syntax and its options.
4. Update client applications to connect to `pgbouncer` instead of directly to Greenplum Database server. To start `psql`, for example:

   ```
   $ psql -p 6543 -U someuser template1
   ```

Server Reset Query

When a connection is returned to the pool, it must be reset to the state of a newly created connection. PgBouncer accomplishes this by issuing a query before returning a connection to the pool. PostgreSQL 8.3 and later have the `DISCARD ALL` command for this purpose and it is the default reset query for the standard PgBouncer distribution. Greenplum Database does not support `DISCARD ALL` and if it is used an error is logged.

A reset query can be specified by setting the `server_reset_query` parameter in the PgBouncer configuration file. For Greenplum Database with PgBouncer in session pooling mode, the `server_reset_query` parameter can be set to this query:

```
RESET ALL; SET SESSION AUTHORIZATION DEFAULT
```

This is the default server reset query in the modified version of PgBouncer included with Greenplum Database 4.7.0 and later. Although the default differs from PgBouncer, it helps to ensure that the connection pool is transparent to Greenplum Database users and client applications do not have to be modified to use a connection pool.

For more information about the `server_reset_query` parameter and the PgBouncer configuration file, see the PgBouncer reference in the *Greenplum Database Utility Guide*.

Managing PgBouncer

PgBouncer has an administrative console, which is accessed by logging into the `pgbouncer` virtual database. The console accepts SQL-like commands that allow you to monitor, reconfigure, and manage PgBouncer.

Follow these steps to get started with the PgBouncer administrative console.

1. Log in to the `pgbouncer` virtual database with `psql`:

   ```
   $ psql -p 6543 -U username pgbouncer
   ```

   The username must be set in the `admin_users` parameter in the `pgbouncer.ini` configuration file. You can also log in with the current Unix username if the `pgbouncer` process is running with that user's UID.
2. To see the available commands, run the `show help` command:

   ```
   pgbouncer=# show help;
   ```

```
NOTICE:  Console usage
DETAIL:
       SHOW HELP|CONFIG|DATABASES|POOLS|CLIENTS|SERVERS|VERSION
 SHOW STATS|FDS|SOCKETS|ACTIVE_SOCKETS|LISTS|MEM
 SHOW DNS_HOSTS|DNS_ZONES
 SET key = arg
 RELOAD
 PAUSE [<db>]
 RESUME [<db>]
 DISABLE <db>
 ENABLE <db>
 KILL <db>
 SUSPEND
 SHUTDOWN
```

3. If you make changes to the `pgbouncer.ini` file, you can reload it with the `RELOAD` command:

```
pgbouncer=# RELOAD;
```

To map clients to server connections, use the `SHOW CLIENTS` and `SHOW SERVERS` views:

1. Use `ptr` and `link` to map the local client connection to the server connection.
2. Use `addr` and `port` of the client connection to identify the TCP connection from the client.
3. Use `local_addr` and `local_port` to identify the TCP connection to the server.

For complete documentation for all of the administration console commands, see the "PgBouncer Administration Console Commands" section of the PgBouncer reference in the *Greenplum Database Utility Guide*.

Upgrading PgBouncer

You can upgrade PgBouncer without dropping connections. Just launch the new PgBouncer process with the `-R` option and the same configuration file:

```
$ pgbouncer -R -d config.ini
```

The `-R` (reboot) option causes the new process to connect to the console of the old process through a Unix socket and issue the following commands:

```
SUSPEND;
SHOW FDS;
SHUTDOWN;
```

When the new process sees that the old process is gone, it resumes the work with the old connections. This is possible because the `SHOW FDS` command sends actual file descriptors to the new process. If the transition fails for any reason, kill the new process and the old process will resume.

Setting up LDAP Authentication with PgBouncer

You can authenticate Greenplum Database users against your LDAP or Active Directory service when using the PgBouncer connection pooler. When you have LDAP authentication working, you can secure client connections to PgBouncer by setting up stunnel, as described in *Securing PgBouncer Connections with stunnel*.

Before you begin, you must have an LDAP or Active Directory server and a Greenplum Database cluster.

1. Create Greenplum Database users in the LDAP/AD server, for example `gpdbuser1` and `gpdbuser2`.
2. Create the corresponding Greenplum Database users in the database:

```
createuser gpdbuser1
createuser gpdbuser2
```

3. Add the users to the Greenplum Database `pg_hba.conf` file:

```
host     all         gpdbuser1              0.0.0.0/0     trust
host     all         gpdbuser2              0.0.0.0/0     trust
```

4. Create a PgBouncer `config.ini` file with the following contents:

```
[databases]
* = host=GPDB_host_addr port=GPDB_port

[pgbouncer]
listen_port = 6432
listen_addr = 0.0.0.0
auth_type = plain
auth_file = users.txt
logfile = pgbouncer.log
pidfile = pgbouncer.pid
ignore_startup_parameters=options
```

5. Create the `users.txt` PgBouncer authentication file. The first field is the user name and the second is the LDAP or Active Directory lookup string for the user. For OpenLDAP or ApacheDS, for example, a user entry in the `users.txt` looks like this:

```
"gpdbuser1" "ldap://10.0.0.11:10389/uid=gpdbuser1,ou=users,ou=system"
```

   For Active Directory, a user entry looks like this example:

```
"gpdbuser1" "ldap://10.0.0.12:389/gpdbuser1"
```

6. Start PgBouncer with the `config.ini` file you created:

```
$ pgbouncer -d config.ini
```

Securing PgBouncer Connections with stunnel

PgBouncer does not have SSL support, so connections between database clients and PgBouncer are not encrypted. To encrypt these connections, you can use *stunnel*, a free software utility. stunnel is a proxy that uses the OpenSSL cryptographic library to add TLS encryption to network services. Newer versions of stunnel support the PostgreSQL libpq protocol, so Greenplum and PostgreSQL clients can use libpq SSL support to connect securely over the network to a stunnel instance set up as a proxy for PgBouncer. If you use a version of stunnel without libpq support, you need to set up a stunnel instance on both the client and PgBouncer host to create a secure tunnel over the network.

On most platforms, you can install stunnel using the system package manager. For Windows, you can download an installer from *https://stunnel.org*. You can also download and install stunnel from source if a packaged version is unavailable, if you want to upgrade to a newer version, or if you want to upgrade to a newer OpenSSL version. The next section provides steps to download, compile, and install OpenSSL and stunnel.

For complete stunnel documentation, visit the *stunnel documentation* web site.

Installing stunnel From Source

stunnel requires a version of the OpenSSL development package greater than 1.0. The following instructions download and build OpenSSL and stunnel for Red Hat or CentOS. Be sure to check *https://stunnel.org* and *http://openssl.org* for the latest versions.

1. Set the `PREFIX` environment variable to the directory where you want to install stunnel, for example `/usr/local/stunnel` or `/home/gpadmin/stunnel`. If you choose to install stunnel in a system directory, you may need to run the `make install` commands as root.

```
export PREFIX=/path/to/install_stunnel
```

2. The following commands download, build, and install OpenSSL version 1.0.2c:

```
wget ftp://ftp.openssl.org/source/openssl-1.0.2c.tar.gz
tar xzf openssl-1.0.2c.tar.gz
cd openssl-1.0.2c
./Configure linux-x86_64 --prefix=$PREFIX shared no-asm
make
make install_sw
cd ..
```

3. Download, build, and install stunnel with the following commands:

```
wget --no-check-certificate https://www.stunnel.org/downloads/stunnel-5.22.tar.gz
tar xzf stunnel-5.22.tar.gz
cd stunnel-5.22
LDFLAGS="-Wl,-rpath,'"'$$'"ORIGIN/../lib'"
./configure --prefix=$PREFIX --with-ssl=$PREFIX
make
make install
```

Setting up Stunnel for PgBouncer

Set up a stunnel instance as a proxy for PgBouncer. The stunnel instance accepts secure connections from database clients on its own port, decrypts the packets and forwards them to the PgBouncer instance on the same host (or on another host in a protected local network). PgBouncer results, including database results from connections managed by PgBouncer, are encrypted and returned over the network to the client.

1. On the server running PgBouncer, create a directory for stunnel configuration files, for example `/home/gpadmin/stunnel`.
2. Create a certificate and key file to use for authenticating. You can also use an existing certificate and key, or create a new pair with the following `openssl` command:

   ```
   $ openssl req -new -x509 -days 3650 -nodes -out stunnel.pem -keyout stunnel.key
   ```

   Move the `stunnel.key` and `stunnel.pem` files into your stunnel configuration directory.
3. Create a stunnel configuration file, for example, `stunnel.conf`, with the following contents:

   ```

   debug = info
   socket = l:TCP_NODELAY=1
   socket = r:TCP_NODELAY=1

   debug = 7

   cert = /PATH/TO/stunnel.pem
   key = /PATH/TO/stunnel.key

   [pg-server]
   client=no
   accept = 0.0.0.0:5433             # This is the SSL listening port
   connect = PGHOST:PGPORT   # This is the PgBouncer listen port
   protocol = pgsql
   ```

   See the *stunnel configuration file reference* for additional configuration options.
4. Run `stunnel` on the server:

   ```
   stunnel /path/to/stunnel-srv.conf
   ```

5. On the client, connect to stunnel on its host and accept port. For example:

   ```
   psql -h pgbouncer-host -p stunnel-accept-port database-name
   ```

Database Application Interfaces

You may want to develop your own client applications that interface to Greenplum Database. PostgreSQL provides a number of database drivers for the most commonly used database application programming interfaces (APIs), which can also be used with Greenplum Database. These drivers are available as a separate download. Each driver is an independent PostgreSQL development project and must be downloaded, installed and configured to connect to Greenplum Database. The following drivers are available:

Table 12: Greenplum Database Interfaces

API	PostgreSQL Driver	Download Link
ODBC	pgodbc	Available in the *Greenplum Database Connectivity* package, which can be downloaded from *https://network.pivotal.io/products*.
JDBC	pgjdbc	Available in the *Greenplum Database Connectivity* package, which can be downloaded from *https://network.pivotal.io/products*.
Perl DBI	pgperl	*http://search.cpan.org/dist/DBD-Pg/*
Python DBI	pygresql	*http://www.pygresql.org/*

General instructions for accessing a Greenplum Database with an API are:

1. Download your programming language platform and respective API from the appropriate source. For example, you can get the Java Development Kit (JDK) and JDBC API from Sun.
2. Write your client application according to the API specifications. When programming your application, be aware of the SQL support in Greenplum Database so you do not include any unsupported SQL syntax.

   See the *Greenplum Database Reference Guide* for more information.

Download the appropriate PostgreSQL driver and configure connectivity to your Greenplum Database master instance. Greenplum Database provides a client tools package that contains the supported database drivers for Greenplum Database. Download the client tools package from *Pivotal Network* and documentation from *Pivotal Documentation*.

Third-Party Client Tools

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. Pivotal has worked with the following tools on previous customer engagements 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
- IBM Cognos

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

Troubleshooting Connection Problems

A number of things can prevent a client application from successfully connecting to Greenplum Database. This topic explains some of the common causes of connection problems and how to correct them.

Table 13: Common connection problems

Problem	Solution
No `pg_hba.conf` entry for host or user	To enable Greenplum Database to accept remote client connections, you must configure your Greenplum Database master instance so that connections are allowed from the client hosts and database users that will be connecting to Greenplum Database. This is done by adding the appropriate entries to the `pg_hba.conf` configuration file (located in the master instance's data directory). For more detailed information, see *Allowing Connections to Greenplum Database*.
Greenplum Database is not running	If the Greenplum Database master instance is down, users will not be able to connect. You can verify that the Greenplum Database system is up by running the `gpstate` utility on the Greenplum master host.
Network problems Interconnect timeouts	If users connect to the Greenplum master host from a remote client, network problems can prevent a connection (for example, DNS host name resolution problems, the host system is down, and so on.). To ensure that network problems are not the cause, connect to the Greenplum master host from the remote client host. For example: `ping` *`hostname`* . If the system cannot resolve the host names and IP addresses of the hosts involved in Greenplum Database, queries and connections will fail. For some operations, connections to the Greenplum Database master use `localhost` and others use the actual host name, so you must be able to resolve both. If you encounter this error, first make sure you can connect to each host in your Greenplum Database array from the master host over the network. In the `/etc/hosts` file of the master and all segments, make sure you have the correct host names and IP addresses for all hosts involved in the Greenplum Database array. The `127.0.0.1` IP must resolve to `localhost`.
Too many clients already	By default, Greenplum Database is configured to allow a maximum of 250 concurrent user connections on the master and 750 on a segment. A connection attempt that causes that limit to be exceeded will be refused. This limit is controlled by the `max_connections` parameter in the `postgresql.conf` configuration file of the Greenplum Database master. If you change this setting for the master, you must also make appropriate changes at the segments.

Chapter 9

Configuring the Greenplum Database System

Server configuration parameters affect the behavior of Greenplum Database. They are part of the PostgreSQL "Grand Unified Configuration" system, so they are sometimes called "GUCs." Most of the Greenplum Database server configuration parameters are the same as the PostgreSQL configuration parameters, but some are Greenplum-specific.

About Greenplum Master and Local Parameters

Server configuration files contain parameters that configure server behavior. The Greenplum Database configuration file, `postgresql.conf`, resides in the data directory of the database instance.

The master and each segment instance have their own `postgresql.conf` file. Some parameters are *local*: each segment instance examines its `postgresql.conf` file to get the value of that parameter. Set local parameters on the master and on each segment instance.

Other parameters are *master* parameters that you set on the master instance. The value is passed down to (or in some cases ignored by) the segment instances at query run time.

See the *Greenplum Database Reference Guide* for information about *local* and *master* server configuration parameters.

Setting Configuration Parameters

Many configuration parameters limit who can change them and where or when they can be set. For example, to change certain parameters, you must be a Greenplum Database superuser. Other parameters can be set only at the system level in the `postgresql.conf` file or require a system restart to take effect.

Many configuration parameters are *session* parameters. You can set session parameters at the system level, the database level, the role level or the session level. Database users can change most session parameters within their session, but some require superuser permissions.

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

Setting a Local Configuration Parameter

To change a local configuration parameter across multiple segments, update the parameter in the `postgresql.conf` file of each targeted segment, both primary and mirror. Use the `gpconfig` utility to set a parameter in all Greenplum `postgresql.conf` files. For example:

```
$ gpconfig -c gp_vmem_protect_limit -v 4096
```

Restart Greenplum Database to make the configuration changes effective:

```
$ gpstop -r
```

Setting a Master Configuration Parameter

To set a master configuration parameter, set it at the Greenplum master instance. If it is also a *session* parameter, you can set the parameter for a particular database, role or session. If a parameter is set at multiple levels, the most granular level takes precedence. For example, session overrides role, role overrides database, and database overrides system.

Setting Parameters at the System Level

Master parameter settings in the master `postgresql.conf` file are the system-wide default. To set a master parameter:

1. Edit the `$MASTER_DATA_DIRECTORY/postgresql.conf` file.
2. Find the parameter to set, uncomment it (remove the preceding `#` character), and type the desired value.
3. Save and close the file.
4. For *session* parameters that do not require a server restart, upload the `postgresql.conf` changes as follows:

   ```
   $ gpstop -u
   ```

5. For parameter changes that require a server restart, restart Greenplum Database as follows:

   ```
   $ gpstop -r
   ```

For details about the server configuration parameters, see the *Greenplum Database Reference Guide*.

Setting Parameters at the Database Level

Use `ALTER DATABASE` to set parameters at the database level. For example:

```
=# ALTER DATABASE mydatabase SET search_path TO myschema;
```

When you set a session parameter at the database level, every session that connects to that database uses that parameter setting. Settings at the database level override settings at the system level.

Setting Parameters at the Role Level

Use `ALTER ROLE` to set a parameter at the role level. For example:

```
=# ALTER ROLE bob SET search_path TO bobschema;
```

When you set a session parameter at the role level, every session initiated by that role uses that parameter setting. Settings at the role level override settings at the database level.

Setting Parameters in a Session

Any session parameter can be set in an active database session using the `SET` command. For example:

```
=# SET statement_mem TO '200MB';
```

The parameter setting is valid for the rest of that session or until you issue a `RESET` command. For example:

```
=# RESET statement_mem;
```

Settings at the session level override those at the role level.

Viewing Server Configuration Parameter Settings

The SQL command `SHOW` allows you to see the current server configuration parameter settings. For example, to see the settings for all parameters:

```
$ psql -c 'SHOW ALL;'
```

`SHOW` lists the settings for the master instance only. To see the value of a particular parameter across the entire system (master and all segments), use the `gpconfig` utility. For example:

```
$ gpconfig --show max_connections
```

Configuration Parameter Categories

Configuration parameters affect categories of server behaviors, such as resource consumption, query tuning, and authentication. The following topics describe Greenplum Database configuration parameter categories.

For details about configuration parameter categories, see the *Greenplum Database Reference Guide*.

- *Configuration Parameter Categories*
- *System Resource Consumption Parameters*
- *Query Tuning Parameters*
- *Error Reporting and Logging Parameters*
- *System Monitoring Parameters*
- *Runtime Statistics Collection Parameters*
- *Automatic Statistics Collection Parameters*
- *Client Connection Default Parameters*
- *Lock Management Parameters*
- *Workload Management Parameters*
- *External Table Parameters*
- *Database Table Parameters*
- *Database and Tablespace/Filespace Parameters*
- *Past PostgreSQL Version Compatibility Parameters*
- *Greenplum Array Configuration Parameters*
- *Greenplum Master and Segment Mirroring Parameters*
- *Greenplum Database Extension Parameters*

Runtime Statistics Collection Parameters

These parameters control the server statistics collection feature. When statistics collection is enabled, you can access the statistics data using the *pg_stat* and *pg_statio* family of system catalog views.

stats_queue_level track_activities	track_counts update_process_title

Automatic Statistics Collection Parameters

When automatic statistics collection is enabled, you can run `ANALYZE` automatically in the same transaction as an `INSERT`, `UPDATE`, `DELETE`, `COPY` or `CREATE TABLE...AS SELECT` statement when a certain threshold of rows is affected (`on_change`), or when a newly generated table has no statistics (`on_no_stats`). To enable this feature, set the following server configuration parameters in your Greenplum master `postgresql.conf` file and restart Greenplum Database:

- `gp_autostats_mode`
- `gp_autostats_mode_in_functions`
- `log_autostats`
- `gp_autostats_on_change_threshold`

> **Warning:** Depending on the specific nature of your database operations, automatic statistics collection can have a negative performance impact. Carefully evaluate whether the default setting of `on_no_stats` is appropriate for your system.

Lock Management Parameters

- `deadlock_timeout`
- `max_locks_per_transaction`

Workload Management Parameters

The following configuration parameters configure the Greenplum Database workload management feature (resource queues), query prioritization, memory utilization and concurrency control.

`gp_resqueue_priority`	`max_resource_queues`
`gp_resqueue_priority_cpucores_per_ segment`	`max_resource_portals_per_ transaction`
`gp_resqueue_priority_sweeper_ interval`	`resource_cleanup_gangs_on_ wait`
`gp_vmem_idle_resource_timeout`	`resource_select_only`
`gp_vmem_protect_limit`	`runaway_detector_activation_percent`
`gp_vmem_protect_segworker_cache_ limit`	`stats_queue_level`
	`vmem_process_interrupt`

External Table Parameters

The following parameters configure the external tables feature of Greenplum Database.

See *Accessing File-Based External Tables* for more information about external tables.

`gp_external_enable_exec`	`gp_initial_bad_row_limit`
`gp_external_grant_privileges`	`gp_reject_percent_threshold`
`gp_external_max_segs`	`readable_external_table_timeout`
	`writable_external_table_bufsize`

Database Table Parameters

The following parameter configures default option settings for Greenplum Database tables.

See *Creating and Managing Tables* for more information about Greenplum Database tables.

- `gp_create_table_random_default_distribution`
- `gp_default_storage_options`
- `gp_enable_exchange_default_partition`

Append-Optimized Table Parameters

The following parameters configure the append-optimized tables feature of Greenplum Database.

See *Append-Optimized Storage* for more information about append-optimized tables.

- `gp_default_storage_options`
- `max_appendonly_tables`
- `gp_appendonly_compaction`
- `gp_appendonly_compaction_threshold`
- `validate_previous_free_tid`

Database and Tablespace/Filespace Parameters

The following parameters configure the maximum number of databases, tablespaces, and filespaces allowed in a system.

- `gp_max_tablespaces`
- `gp_max_filespaces`
- `gp_max_databases`

Past PostgreSQL Version Compatibility Parameters

The following parameters provide compatibility with older PostgreSQL versions. You do not need to change these parameters in Greenplum Database.

`add_missing_from`
`array_nulls`
`backslash_quote`
`escape_string_warning`
	`regex_flavor`
	`standard_conforming_strings`
	`transform_null_equals`

Greenplum Master and Segment Mirroring Parameters

These parameters control the configuration of the replication between Greenplum Database primary master and standby master.

- `keep_wal_segments`
- `repl_catchup_within_range`
- `replication_timeout`
- `wal_receiver_status_interval`

This parameter controls validation between Greenplum Database primary segment and standby segment during incremental resynchronization.

- `filerep_mirrorvalidation_during_resync`

Greenplum Database Extension Parameters

The parameters in this topic control the configuration of Greenplum Database extensions.

- `pgcrypto.fips`
- `pljava_classpath`
- `pljava_classpath_insecure`
- `pljava_statement_cache_size`
- `pljava_release_lingering_savepoints`
- `pljava_vmoptions`

Connection and Authentication Parameters

These parameters control how clients connect and authenticate to Greenplum Database.

See *Managing Greenplum Database Access* for information about configuring client authentication.

Connection Parameters

gp_connection_send_timeout gp_vmem_idle_resource_timeout listen_addresses max_connections max_prepared_transactions superuser_reserved_connections	tcp_keepalives_count tcp_keepalives_idle tcp_keepalives_interval unix_socket_directory unix_socket_group unix_socket_permissions

Security and Authentication Parameters

authentication_timeout db_user_namespace krb_caseins_users krb_server_keyfile	krb_srvname password_encryption password_hash_algorithm ssl ssl_ciphers

System Resource Consumption Parameters

Memory Consumption Parameters

These parameters control system memory usage. You can adjust `gp_vmem_protect_limit` to avoid running out of memory at the segment hosts during query processing.

gp_vmem_idle_resource_timeout gp_vmem_protect_limit gp_vmem_protect_segworker_cache_limit gp_workfile_limit_files_per_query gp_workfile_limit_per_query gp_workfile_limit_per_segment max_appendonly_tables max_prepared_transactions	max_stack_depth shared_buffers temp_buffers

Free Space Map Parameters

These parameters control the sizing of the *free space map, which contains* expired rows. Use `VACUUM` to reclaim the free space map disk space.

See *Vacuum and Analyze for Query Optimization* for information about vacuuming a database.

- `max_fsm_pages`
- `max_fsm_relations`

OS Resource Parameters

- `max_files_per_process`
- `shared_preload_libraries`

Cost-Based Vacuum Delay Parameters

> **Warning:** Pivotal does not recommend cost-based vacuum delay because it runs asynchronously among the segment instances. The vacuum cost limit and delay is invoked at the segment level without taking into account the state of the entire Greenplum array.

You can configure the execution cost of `VACUUM` and `ANALYZE` commands to reduce the I/O impact on concurrent database activity. When the accumulated cost of I/O operations reaches the limit, the process performing the operation sleeps for a while, Then resets the counter and continues execution

`vacuum_cost_delay`	`vacuum_cost_page_hit`
`vacuum_cost_limit`	`vacuum_cost_page_miss`
`vacuum_cost_page_dirty`	

Transaction ID Management Parameters

- `xid_stop_limit`
- `xid_warn_limit`

Query Tuning Parameters

Pivotal Query Optimizer Configuration Parameters

- `optimizer`
- `optimizer_array_expansion_threshold`
- `optimizer_analyze_root_partition`
- `optimizer_control`
- `optimizer_enable_master_only_queries`
- `optimizer_parallel_union`
- `optimizer_sort_factor`

Query Plan Operator Control Parameters

The following parameters control the types of plan operations the legacy query optimizer can use. Enable or disable plan operations to force the legacy query optimizer to choose a different plan. This is useful for testing and comparing query performance using different plan types.

`enable_bitmapscan`	`gp_enable_agg_distinct_pruning`
`enable_groupagg`	`gp_enable_direct_dispatch`
`enable_hashagg`	`gp_enable_fallback_plan`
`enable_hashjoin`	`gp_enable_fast_sri`
`enable_indexscan`	`gp_enable_groupext_distinct_ gather`
`enable_mergejoin`	`gp_enable_groupext_distinct_ pruning`
`enable_nestloop`	`gp_enable_multiphase_agg`

enable_seqscan	gp_enable_predicate_ propagation
enable_sort	gp_enable_preunique
enable_tidscan	gp_enable_sequential_window_ plans
gp_enable_adaptive_nestloop	gp_enable_sort_distinct
gp_enable_agg_distinct	gp_enable_sort_limit

Legacy Query Optimizer Costing Parameters

> **Warning:** Pivotal recommends that you do not adjust these query costing parameters. They are tuned to reflect Greenplum Database hardware configurations and typical workloads. All of these parameters are related. Changing one without changing the others can have adverse effects on performance.

cpu_index_tuple_cost	gp_motion_cost_per_row
cpu_operator_cost	gp_segments_for_planner
cpu_tuple_cost	random_page_cost
cursor_tuple_fraction	seq_page_cost
effective_cache_size	

Database Statistics Sampling Parameters

These parameters adjust the amount of data sampled by an `ANALYZE` operation. Adjusting these parameters affects statistics collection system-wide. You can configure statistics collection on particular tables and columns by using the `ALTER TABLE SET STATISTICS` clause.

- `default_statistics_target`
- `gp_analyze_relative_error`

Sort Operator Configuration Parameters

- `gp_enable_sort_distinct`
- `gp_enable_sort_limit`

Aggregate Operator Configuration Parameters

gp_enable_agg_distinct	gp_enable_groupext_distinct_ gather
gp_enable_agg_distinct_pruning	gp_enable_groupext_distinct_ pruning
gp_enable_multiphase_agg	gp_workfile_compress_algorithm
gp_enable_preunique	

Join Operator Configuration Parameters

join_collapse_limit	gp_statistics_use_fkeys
gp_adjust_selectivity_for_outerjoins	gp_workfile_compress_algorithm
gp_hashjoin_tuples_per_bucket	

Other Legacy Query Optimizer Configuration Parameters

- `from_collapse_limit`
- `gp_enable_predicate_propagation`
- `gp_max_plan_size`
- `gp_statistics_pullup_from_child_partition`

Error Reporting and Logging Parameters

Log Rotation

`log_rotation_age` `log_rotation_size`	`log_truncate_on_rotation`

When to Log

`client_min_messages`
`log_error_verbosity`
`log_min_duration_statement`
	`log_min_error_statement`
	`log_min_messages`
	`optimizer_minidump`

What to Log

`debug_pretty_print`
`debug_print_parse`
`debug_print_plan`
`debug_print_prelim_plan`
`debug_print_rewritten`
`debug_print_slice_table`
`log_autostats`
`log_connections`
`log_disconnections`
`log_dispatch_stats`
`log_duration`
	`log_executor_stats`
	`log_hostname`
	`log_parser_stats`
	`log_planner_stats`
	`log_statement`
	`log_statement_stats`
	`log_timezone`
	`gp_debug_linger`
	`gp_log_format`
	`gp_max_csv_line_length`
	`gp_reraise_signal`

System Monitoring Parameters

SNMP Alerts

The following parameters send SNMP notifications when events occur.

`gp_snmp_community` `gp_snmp_monitor_address`	`gp_snmp_use_inform_or_trap`

Email Alerts

The following parameters configure the system to send email alerts for fatal error events, such as a segment going down or a server crash and reset.

gp_email_from gp_email_smtp_password gp_email_smtp_server	gp_email_smtp_userid gp_email_to

Greenplum Command Center Agent

The following parameters configure the data collection agents for Greenplum Command Center.

gp_enable_gpperfmon gp_gpperfmon_send_interval	gpperfmon_log_alert_level gpperfmon_port

Client Connection Default Parameters

Statement Behavior Parameters

check_function_bodies default_tablespace default_transaction_isolation default_transaction_read_only	search_path statement_timeout vacuum_freeze_min_age

Locale and Formatting Parameters

client_encoding DateStyle extra_float_digits IntervalStyle lc_collate lc_ctype	lc_messages lc_monetary lc_numeric lc_time TimeZone

Other Client Default Parameters

dynamic_library_path explain_pretty_print	local_preload_libraries

Greenplum Array Configuration Parameters

The parameters in this topic control the configuration of the Greenplum Database array and its components: segments, master, distributed transaction manager, master mirror, and interconnect.

Interconnect Configuration Parameters

`gp_interconnect_fc_method`	`gp_interconnect_setup_timeout`
`gp_interconnect_hash_multiplier`	`gp_interconnect_snd_ queue_depth`
`gp_interconnect_queue_depth`	`gp_interconnect_type`
	`gp_max_packet_size`

Note: The Greenplum Database interconnect types TCP and UDP are deprecated. In the next major release, only the UDPIFC interconnect type will be supported by Greenplum Database. The server configuration parameter `gp_interconnect_type` controls the interconnect type.

Dispatch Configuration Parameters

`gp_cached_segworkers_threshold`	`gp_segment_connect_timeout`
`gp_connections_per_thread`	`gp_set_proc_affinity`
`gp_enable_direct_dispatch`	

Fault Operation Parameters

`gp_set_read_only`	`gp_fts_probe_threadcount`
`gp_fts_probe_interval`	

Distributed Transaction Management Parameters

- `gp_max_local_distributed_cache`

Read-Only Parameters

- `gp_command_count`
- `gp_content`
- `gp_dbid`
- `gp_num_contents_in_cluster`
- `gp_role`
- `gp_session_id`

Chapter 10

Enabling High Availability Features

The fault tolerance and the high-availability features of Greenplum Database can be configured.

For information about the utilities that are used to enable high availability, see the *Greenplum Database Utility Guide*.

Overview of Greenplum Database High Availability

A Greenplum Database cluster can be made highly available by providing a fault-tolerant hardware platform, by enabling Greenplum Database high-availability features, and by performing regular monitoring and maintenance procedures to ensure the health of all system components.

Hardware components will eventually fail, whether due to normal wear or an unexpected circumstance. Loss of power can lead to temporarily unavailable components. A system can be made highly available by providing redundant standbys for components that can fail so that services can continue uninterrupted when a failure does occur. In some cases, the cost of redundancy is higher than users' tolerance for interruption in service. When this is the case, the goal is to ensure that full service is able to be restored, and can be restored within an expected timeframe.

With Greenplum Database, fault tolerance and data availability is achieved with:

- *Hardware level RAID storage protection*
- *Greenplum segment mirroring*
- *Master mirroring*
- *Dual clusters*
- *Database backup and restore*

Hardware Level RAID

A best practice Greenplum Database deployment uses hardware level RAID to provide high performance redundancy for single disk failure without having to go into the database level fault tolerance. This provides a lower level of redundancy at the disk level.

Segment Mirroring

Greenplum Database stores data in multiple segments, each of which is a Greenplum Database Postgres instance. The data for each table is spread between the segments based on the distribution policy that is defined for the table in the DDL at the time the table is created. When segment mirroring is enabled, for each segment there is a *primary* and *mirror* pair. The primary and mirror perform the same IO operations and store copies of the same data.

The mirror instance for each segment is usually initialized with the `gpinitsystem` utility or the `gpexpand` utility. The mirror runs on a different host than the primary instance to protect from a single machine failure. There are different strategies for assigning mirrors to hosts. When choosing the layout of the primaries and mirrors, it is important to consider the failure scenarios to ensure that processing skew is minimized in the case of a single machine failure.

Master Mirroring

There are two masters in a highly available cluster, a *primary* and a *standby*. As with segments, the master and standby should be deployed on different hosts so that the cluster can tolerate a single host failure. Clients connect to the primary master and queries can be executed only on the primary master. The secondary master is kept up-to-date by replicating the write-ahead log (WAL) from the primary to the secondary.

If the master fails, the administrator runs the `gpactivatestandby` utility to have the standby master take over as the new primary master. You can configure a virtual IP address for the master and standby so that client programs do not have to switch to a different network address when the current master changes. If the master host fails, the virtual IP address can be swapped to he actual acting master.

Dual Clusters

An additional level of redundancy can be provided by maintaining two Greenplum Database clusters, both storing the same data.

Two methods for keeping data synchronized on dual clusters are "dual ETL" and "backup/restore."

Dual ETL provides a complete standby cluster with the same data as the primary cluster. ETL (extract, transform, and load) refers to the process of cleansing, transforming, validating, and loading incoming data into a data warehouse. With dual ETL, this process is executed twice in parallel, once on each cluster, and is validated each time. It also allows data to be queried on both clusters, doubling the query throughput. Applications can take advantage of both clusters and also ensure that the ETL is successful and validated on both clusters.

To maintain a dual cluster with the backup/restore method, create backups of the primary cluster and restore them on the secondary cluster. This method takes longer to synchronize data on the secondary cluster than the dual ETL strategy, but requires less application logic to be developed. Populating a second cluster with backups is ideal in use cases where data modifications and ETL are performed daily or less frequently.

Backup and Restore

Making regular backups of the databases is recommended except in cases where the database can be easily regenerated from the source data. Backups should be taken to protect from operational, software, and hardware errors.

Use the `gpcrondump` utility to backup Greenplum databases. `gpcrondomp` performs the backup in parallel across segments, so backup performance scales up as hardware is added to the cluster.

When designing a backup strategy, a primary concern is where to store the backup data. The data each segment manages can be backed up on the segment's local storage, but should not be stored there permanently—the backup reduces disk space available to the segment and, more importantly, a hardware failure could simultaneously destroy the segment's live data and the backup. After performing a backup, the backup files should be moved from the primary cluster to separate, safe storage. Alternatively, the backup can be made directly to separate storage.

Additional options are avaiable to backup datatabases, including the following:

Data Domain

Through native API integration backups can be streamed to an EMC Data Domain appliance.

NetBackup

Through native API integration, backups can be streamed to a Veritas NetBackup cluster.

NFS

If an NFS mount is created on each Greenplum Database host in the cluster, backups can be written directly to the NFS mount. A scale out NFS solution is recommended to ensure that backups do not bottleneck on IO throughput of the NFS device. EMC Isilon is an example that can scale out along side the Greenplum cluster.

Incremental Backups

Greenplum Database allows *incremental backup* at the partition level for append-optimized and column-oriented tables. When you perform an incremental backup, only the partitions for append-optimized and column-oriented tables that have changed since the previous backup are backed up. (Heap tables are *always* backed up.) Restoring an incremental backup requires restoring the previous full backup and subsequent incremental backups.

Incremental backup is beneficial only when the database contains large, partitioned tables where all but one or a few partitions remain unchanged between backups. An incremental backup saves just the changed partitions and the heap tables. By not backing up the unchanged partitions, the backup size and time can be significantly reduced.

If a large fact table is not partitioned, and a single row is added or changed, the entire table is backed up, and there is no savings in backup size or time. Therefore, incremental backup is only recommended for databases with large, partitioned tables and relatively small dimension tables.

If you maintain dual clusters and use incremental backup, you can populate the second cluster with the incremental backups. Use the `--noplan` option to achieve this, allowing backups from the primary site to be applied faster.

Overview of Segment Mirroring

When Greenplum Database High Availability is enabled, there are two types of segments: *primary* and *mirror*. Each primary segment has one corresponding mirror segment. A primary segment receives requests from the master to make changes to the segment's database and then replicates those changes to the corresponding mirror. If a primary segment becomes unavailable, database queries fail over to the mirror segment.

Segment mirroring employs a physical file replication scheme—data file I/O at the primary is replicated to the secondary so that the mirror's files are identical to the primary's files. Data in Greenplum Database are represented with *tuples*, which are packed into *blocks*. Database tables are stored in disk files consisting of one or more blocks. A change to a tuple changes the block it is saved in, which is then written to disk on the primary and copied over the network to the mirror. The mirror updates the corresponding block in its copy of the file.

For heap tables, blocks are saved in an in-memory cache until they are evicted to make room for newly changed blocks. This allows the system to read or update a block in memory multiple times without performing expensive disk I/O. When the block is evicted from the cache, it is written to disk and replicated to the secondary. While the block is held in cache, the primary and mirror have different images of the block. However, the databases are still consistent because the transaction log has been replicated. If a mirror takes over for a failed primary, the transactions in its log are applied to the database tables.

Other database objects — for example filespaces, which are tablespaces internally represented with directories—also use file replication to perform various file operations in a synchronous way.

Append-optimized tables do not use the in-memory caching mechanism. Changes made to append-optimized table blocks are replicated to the mirror immediately. Typically, file write operations are asynchronous, while opening, creating, and synchronizing files are "sync-replicated," which means the primary blocks until it receives the acknowledgment from the secondary.

If a primary segment fails, the file replication process stops and the mirror segment automatically starts as the active segment instance. The now active mirror's system state becomes *Change Tracking*, which means the mirror maintains a system table and change-log of all blocks updated while the primary segment is unavailable. When the failed primary segment is repaired and ready to be brought back online, an administrator initiates a recovery process and the system goes into *Resynchronization* state. The recovery process applies the logged changes to the repaired primary segment. The system state changes to *Synchronized* when the recovery process completes.

If the mirror segment fails or becomes inaccessible while the primary is active, the primary's system state changes to *Change Tracking*, and it tracks changes to be applied to the mirror when it is recovered.

Mirror segments can be placed on hosts in the cluster in different configurations, as long as the primary and mirror instance for a segment are on different hosts. Each host must have the same number of primary and mirror segments. The default mirroring configuration is *group mirroring*, where the mirror segments for each host's primary segments are placed on one other host. If a single host fails, the number of active primary segments doubles on the host that backs the failed host. *Figure 7: Group Segment Mirroring in Greenplum Database* illustrates a group mirroring configuration.

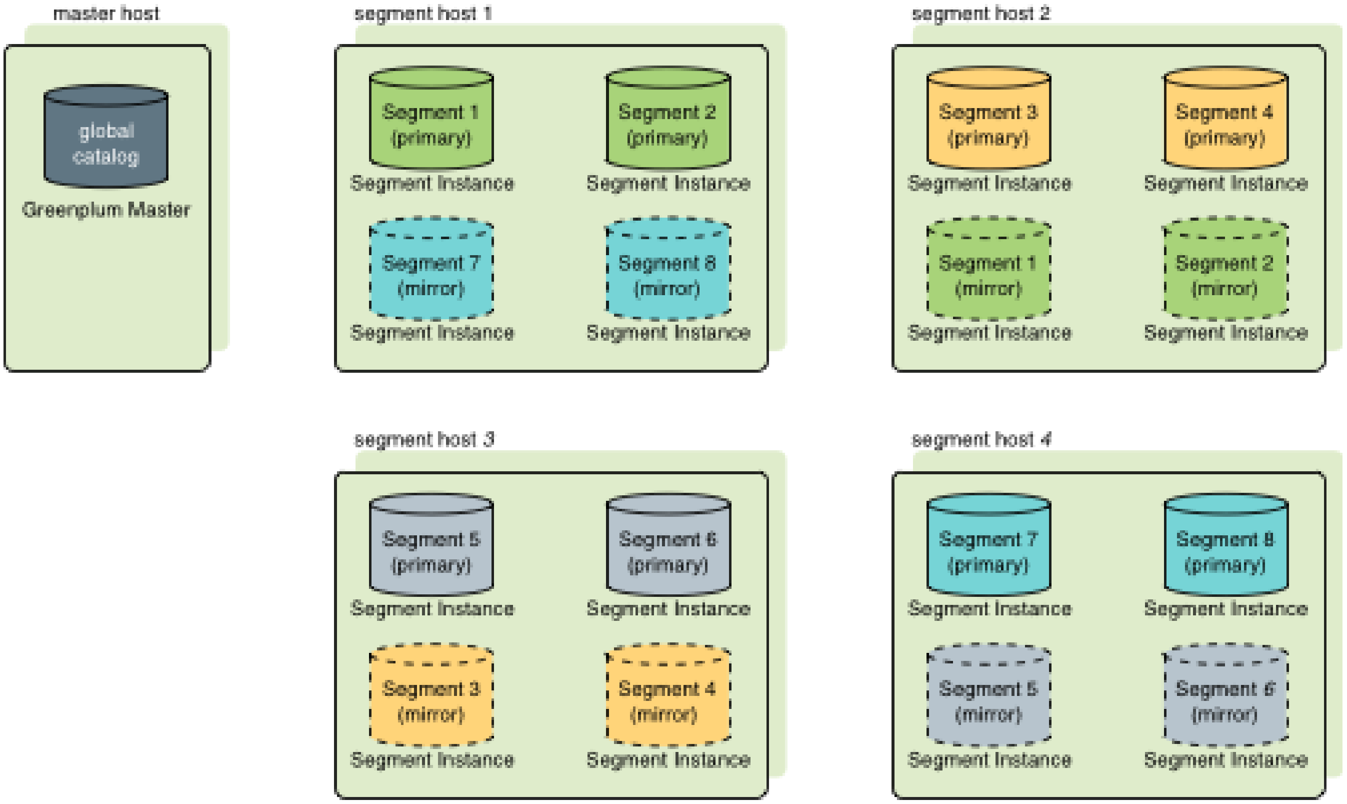


Figure 7: Group Segment Mirroring in Greenplum Database

Spread mirroring spreads each host's mirrors over multiple hosts so that if any single host fails, no other host will have more than one mirror promoted to the active primary segment. Spread mirroring is possible only if there are more hosts than segments per host. *Figure 8: Spread Segment Mirroring in Greenplum Database* illustrates the placement of mirrors in a spread segment mirroring configuration.

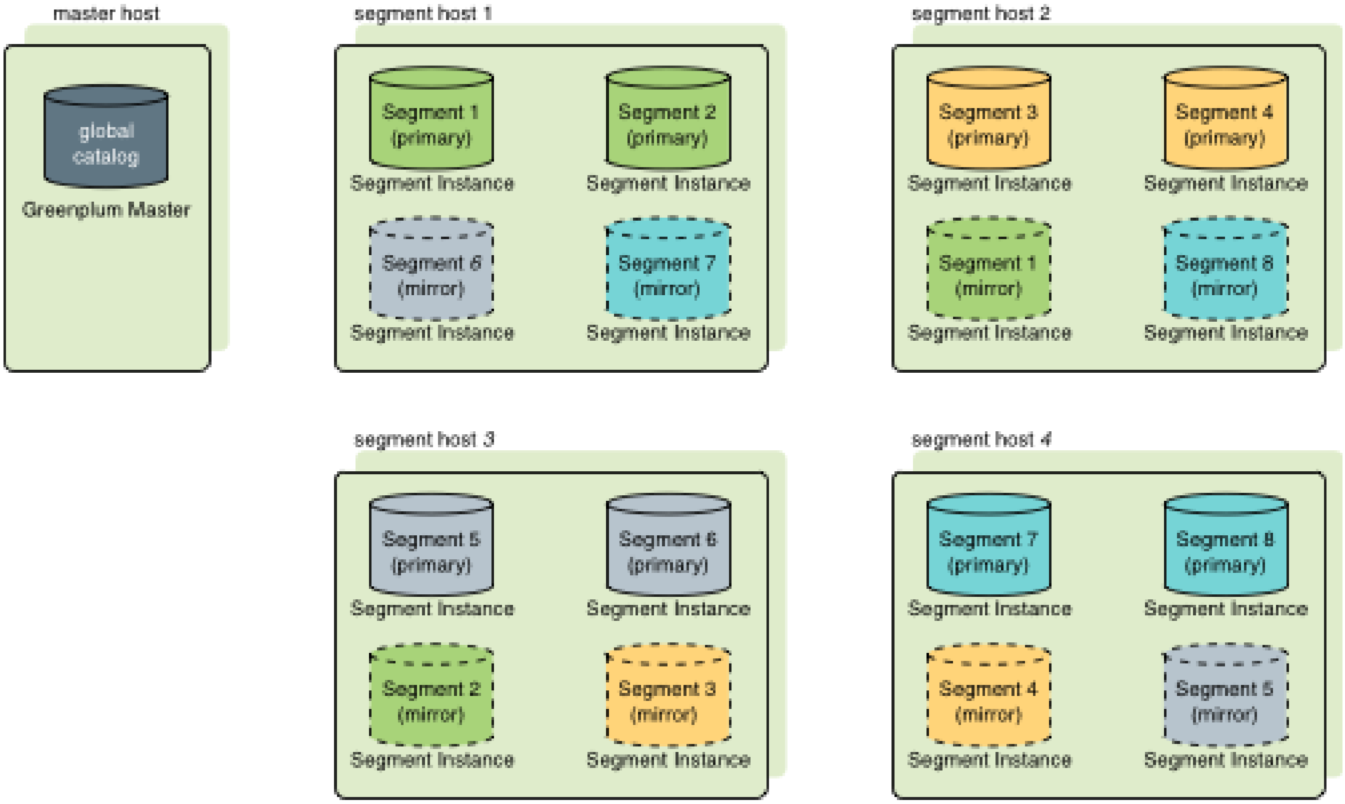


Figure 8: Spread Segment Mirroring in Greenplum Database

The Greenplum Database utilities that create mirror segments support group and spread segment configurations. Custom mirroring configurations can be described in a configuration file and passed on the command line.

Overview of Master Mirroring

You can deploy a backup or mirror of the master instance on a separate host machine or on the same host machine. A backup master or standby master serves as a warm standby if the primary master becomes nonoperational. You create a standby master from the primary master while the primary is online.

The primary master continues to provide service to users while a transactional snapshot of the primary master instance is taken. While the transactional snapshot is taken and deployed on the standby master, changes to the primary master are also recorded. After the snapshot is deployed on the standby master, the updates are deployed to synchronize the standby master with the primary master.

Once the primary master and standby master are synchronized, the standby master is kept up to date by the `walsender` and `walreceiver` replication processes. The `walreceiver` is a standby master process. The `walsender` process is a primary master process. The two processes use WAL-based streaming replication to keep the primary and standby masters synchronized.

Since the master does not house user data, only system catalog tables are synchronized between the primary and standby masters. When these tables are updated, changes are automatically copied to the standby master to keep it current with the primary.

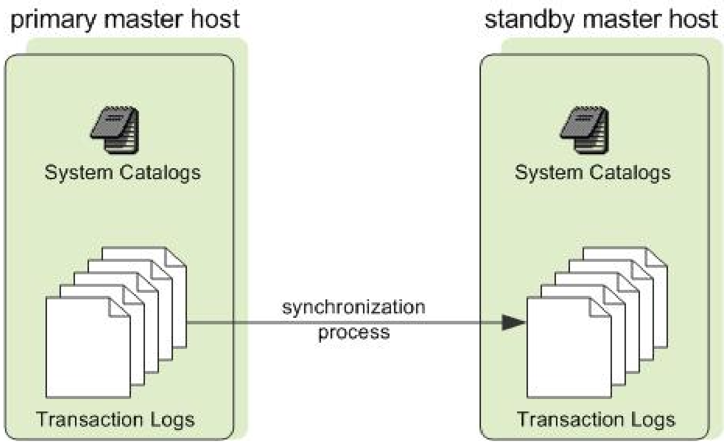


Figure 9: Master Mirroring in Greenplum Database

If the primary master fails, the replication process stops, and an administrator can activate the standby master. Upon activation of the standby master, the replicated logs reconstruct the state of the primary master at the time of the last successfully committed transaction. The activated standby then functions as the Greenplum Database master, accepting connections on the port specified when standby master was initialized.

Overview of Fault Detection and Recovery

The Greenplum Database server (`postgres`) subprocess named `ftsprobe` handles fault detection. `ftsprobe` monitors the Greenplum Database array; it connects to and scans all segments and database processes at intervals that you can configure.

If `ftsprobe` cannot connect to a segment, it marks the segment as "down" in the Greenplum Database system catalog. The segment remains nonoperational until an administrator initiates the recovery process.

With mirroring enabled, Greenplum Database automatically fails over to a mirror copy if a primary copy becomes unavailable. The system is operational if a segment instance or host fails provided all data is available on the remaining active segments.

To recover failed segments, an administrator runs the `gprecoverseg` recovery utility. This utility locates the failed segments, verifies they are valid, and compares the transactional state with the currently active segment to determine changes made while the segment was offline. `gprecoverseg` synchronizes the changed database files with the active segment and brings the segment back online. Administrators perform the recovery while Greenplum Database is up and running.

With mirroring disabled, the system automatically shuts down if a segment instance fails. Administrators manually recover all failed segments before operations resume.

See *Detecting a Failed Segment* for a more detailed description of the fault detection and recovery process and configuration options.

Enabling Mirroring in Greenplum Database

You can configure your Greenplum Database system with mirroring at setup time using `gpinitsystem` or enable mirroring later using `gpaddmirrors` and `gpinitstandby`. This topic assumes you are adding mirrors to an existing system that was initialized without mirrors.

You can enable the following types of mirroring:

- *Enabling Segment Mirroring*
- *Enabling Master Mirroring*

Enabling Segment Mirroring

Mirror segments allow database queries to fail over to a backup segment if the primary segment is unavailable. By default, mirrors are configured on the same array of hosts as the primary segments. You may choose a completely different set of hosts for your mirror segments so they do not share machines with any of your primary segments.

> **Important:** During the online data replication process, Greenplum Database should be in a quiescent state, workloads and other queries should not be running.

To add segment mirrors to an existing system (same hosts as primaries)

1. Allocate the data storage area for mirror data on all segment hosts. The data storage area must be different from your primary segments' file system location.
2. Use `gpssh-exkeys` to ensure that the segment hosts can SSH and SCP to each other without a password prompt.
3. Run the `gpaddmirrors` utility to enable mirroring in your Greenplum Database system. For example, to add 10000 to your primary segment port numbers to calculate the mirror segment port numbers:

   ```
   $ gpaddmirrors -p 10000
   ```

   Where `-p` specifies the number to add to your primary segment port numbers. Mirrors are added with the default group mirroring configuration.

To add segment mirrors to an existing system (different hosts from primaries)

1. Ensure the Greenplum Database software is installed on all hosts. See the *Greenplum Database Installation Guide* for detailed installation instructions.
2. Allocate the data storage area for mirror data on all segment hosts.
3. Use `gpssh-exkeys` to ensure the segment hosts can SSH and SCP to each other without a password prompt.
4. Create a configuration file that lists the host names, ports, and data directories on which to create mirrors. To create a sample configuration file to use as a starting point, run:

   ```
   $ gpaddmirrors -o filename
   ```

   The format of the mirror configuration file is:

   ```
   filespaceOrder=[filespace1_fsname[:filespace2_fsname:...]
   mirror[content]=content:address:port:mir_replication_port:
   pri_replication_port:fselocation[:fselocation:...]
   ```

For example, a configuration for two segment hosts and two segments per host, with no additional filespaces configured besides the default *pg_system* filespace:

```
filespaceOrder=
mirror0=0:sdw1-1:52001:53001:54001:/gpdata/mir1/gp0
mirror1=1:sdw1-2:52002:53002:54002:/gpdata/mir1/gp1
mirror2=2:sdw2-1:52001:53001:54001:/gpdata/mir1/gp2
mirror3=3:sdw2-2:52002:53002:54002:/gpdata/mir1/gp3
```

5. Run the `gpaddmirrors` utility to enable mirroring in your Greenplum Database system:

   ```
   $ gpaddmirrors -i mirror_config_file
   ```

   Where `-i` names the mirror configuration file you created.

Enabling Master Mirroring

You can configure a new Greenplum Database system with a standby master using `gpinitsystem` or enable it later using `gpinitstandby`. This topic assumes you are adding a standby master to an existing system that was initialized without one.

For information about the utilities `gpinitsystem` and `gpinitstandby`, see the *Greenplum Database Utility Guide*.

To add a standby master to an existing system

1. Ensure the standby master host is installed and configured: `gpadmin` system user created, Greenplum Database binaries installed, environment variables set, SSH keys exchanged, and data directory created. Greenplum Database
2. Run the `gpinitstandby` utility on the currently active *primary* master host to add a standby master host to your Greenplum Database system. For example:

   ```
   $ gpinitstandby -s smdw
   ```

   Where `-s` specifies the standby master host name.
3. To switch operations to a standby master, see *Recovering a Failed Master*.

To check the status of the master mirroring process (optional)

You can display the information in the Greenplum Database system view pg_stat_replication. The view lists information about the `walsender` process that is used for Greenplum Database master mirroring. For example, this command displays the process ID and state of the `walsender` process:

```
$ psql dbname -c 'SELECT procpid, state FROM pg_stat_replication;'
```

For information about the `pg_stat_replication` system view, see the *Greenplum Database Reference Guide*.

Detecting a Failed Segment

With mirroring enabled, Greenplum Database automatically fails over to a mirror segment when a primary segment goes down. Provided one segment instance is online per portion of data, users may not realize a segment is down. If a transaction is in progress when a fault occurs, the in-progress transaction rolls back and restarts automatically on the reconfigured set of segments.

If the entire Greenplum Database system becomes nonoperational due to a segment failure (for example, if mirroring is not enabled or not enough segments are online to access all user data), users will see errors when trying to connect to a database. The errors returned to the client program may indicate the failure. For example:

```
ERROR: All segment databases are unavailable
```

How Segment Failure is Detected and Managed

On the Greenplum Database master host, the Postgres `postmaster` process forks a fault probe process, `ftsprobe`. This is sometimes called the FTS (Fault Tolerance Server) process. The `postmaster` process restarts the FTS if it fails.

The FTS runs in a loop with a sleep interval between each cycle. On each loop, the FTS probes each primary segment database by making a TCP socket connection to the segment database using the hostname and port registered in the `gp_segment_configuration` table. If the connection succeeds, the segment performs a few simple checks and reports back to the FTS. The checks include executing a `stat` system call on critical segment directories and checking for internal faults in the segment instance. If no issues are detected, a positive reply is sent to the FTS and no action is taken for that segment database.

If the connection cannot be made, or if a reply is not received in the timeout period, then a retry is attempted for the segment database. If the configured maximum number of probe attempts fail, the FTS probes the segment's mirror to ensure that it is up, and then updates the `gp_segment_configuration` table, marking the primary segment "down" and setting the mirror to act as the primary. The FTS updates the `gp_configuration_history` table with the operations performed.

When there is only an active primary segment and the corresponding mirror is down, the primary goes into "Change Tracking Mode." In this mode, changes to the segment are recorded, so the mirror can be synchronized without performing a full copy of data from the primary to the mirror.

The `gprecoverseg` utility is used to bring up a mirror that is down. By default, `gprecoverseg` performs an incremental recovery, placing the mirror into resync mode, which starts to replay the recorded changes from the primary onto the mirror. If the incremental recovery cannot be completed, the recovery fails and `gprecoverseg` should be run again with the `-F` option, to perform full recovery. This causes the primary to copy all of the data to the mirror.

You can see the mode—"change tracking", "resync", or "in-sync"—for each segment, as well as the status "up" or "down", in the `gp_segment_configuration` table.

The `gp_segment_configuration` table also has columns `role` and `preferred_role`. These can have values of either `p` for primary or `m` for mirror. The `role` column shows the segment database's current role and the `preferred_role` shows the original role of the segment. In a balanced system the `role` and `preferred_role` matches for all segments. When they do not match, there may be skew resulting from the number of active primary segments on each hardware host. To rebalance the cluster and bring all the segments into their preferred role, the `gprecoverseg` command can be run with the `-r` option.

There is a set of server configuration parameters that affect FTS behavior:

gp_fts_probe_threadcount

The number of threads used for probing segments. Default: 16

gp_fts_probe_interval

How often, in seconds, to begin a new FTS loop. For example if the setting is 60 and the probe loop takes 10 seconds, the FTS process sleeps 50 seconds. If the setting is 60 and probe loop takes 75 seconds, the process sleeps 0 seconds. The default is 60, and the maximum is 3600.

gp_fts_probe_timeout

Probe timeout between master and segment, in seconds. The default is 20, and the maximum is 3600.

gp_fts_probe_retries

The number of attempts to probe a segment. For example if the setting is 5 there will be 4 retries after the first attempt fails. Default: 5

gp_log_fts

Logging level for FTS. The value may be "off", "terse", "verbose", or "debug". The "verbose" setting can be used in production to provide useful data for troubleshooting. The "debug" setting should not be used in production. Default: "terse"

gp_segment_connect_timeout

The maximum time (in seconds) allowed for a mirror to respond. Default: 180

In addition to the fault checking performed by the FTS, a primary segment that is unable to send data to its mirror can change the status of the mirror to down. The primary queues up the data and after `gp_segment_connect_timeout` seconds passes, indicates a mirror failure, causing the mirror to be marked down and the primary to go into change tracking mode.

Enabling Alerts and Notifications

To receive notifications of system events such as segment failures, enable email or SNMP alerts. See *Enabling System Alerts and Notifications*.

Checking for Failed Segments

With mirroring enabled, you may have failed segments in the system without interruption of service or any indication that a failure has occurred. You can verify the status of your system using the `gpstate` utility. `gpstate` provides the status of each individual component of a Greenplum Database system, including primary segments, mirror segments, master, and standby master.

To check for failed segments

1. On the master, run the `gpstate` utility with the `-e` option to show segments with error conditions:

   ```
   $ gpstate -e
   ```

   Segments in *Change Tracking* mode indicate the corresponding mirror segment is down. When a segment is not in its *preferred role*, the segment does not operate in the role to which it was assigned at system initialization. This means the system is in a potentially unbalanced state, as some segment hosts may have more active segments than is optimal for top system performance.

   See *Recovering From Segment Failures* for instructions to fix this situation.
2. To get detailed information about a failed segment, check the *gp_segment_configuration* catalog table. For example:

   ```
   $ psql -c "SELECT * FROM gp_segment_configuration WHERE status='d';"
   ```
3. For failed segment instances, note the host, port, preferred role, and data directory. This information will help determine the host and segment instances to troubleshoot.

4. To show information about mirror segment instances, run:

```
$ gpstate -m
```

Checking the Log Files for Failed Segments

Log files can provide information to help determine an error's cause. The master and segment instances each have their own log file in `pg_log` of the data directory. The master log file contains the most information and you should always check it first.

Use the `gplogfilter` utility to check the Greenplum Database log files for additional information. To check the segment log files, run `gplogfilter` on the segment hosts using `gpssh`.

To check the log files

1. Use `gplogfilter` to check the master log file for `WARNING`, `ERROR`, `FATAL` or `PANIC` log level messages:

```
$ gplogfilter -t
```

2. Use `gpssh` to check for `WARNING`, `ERROR`, `FATAL`, or `PANIC` log level messages on each segment instance. For example:

```
$ gpssh -f seg_hosts_file -e 'source
/usr/local/greenplum-db/greenplum_path.sh ; gplogfilter -t
/data1/primary/*/pg_log/gpdb*.log' > seglog.out
```

Recovering a Failed Segment

If the master cannot connect to a segment instance, it marks that segment as down in the Greenplum Database system catalog. The segment instance remains offline until an administrator takes steps to bring the segment back online. The process for recovering a failed segment instance or host depends on the failure cause and whether or not mirroring is enabled. A segment instance can be unavailable for many reasons:

- A segment host is unavailable; for example, due to network or hardware failures.
- A segment instance is not running; for example, there is no `postgres` database listener process.
- The data directory of the segment instance is corrupt or missing; for example, data is not accessible, the file system is corrupt, or there is a disk failure.

Figure 10: Segment Failure Troubleshooting Matrix shows the high-level steps for each of the preceding failure scenarios.

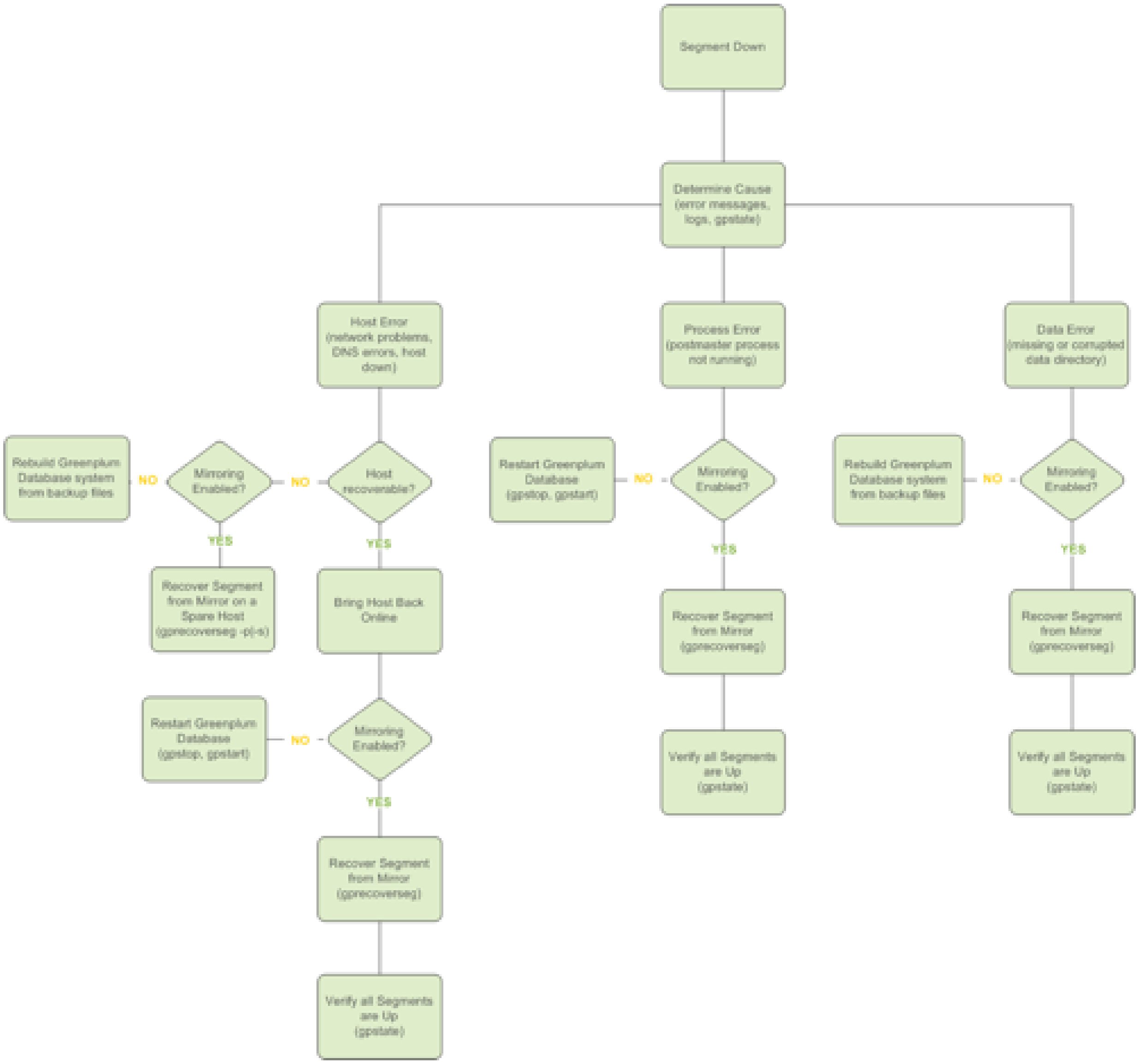

Segment Down
Determine Cause (error messages, logs, gpstate)
Host Error (network problems, DNS errors, host down)
Process Error (postmaster process not running)
Data Error (missing or corrupted data directory)
Rebuild Greenplum Database system from backup files
NO
Mirroring Enabled?
NO
Host recoverable?
Restart Greenplum Database (gpstop, gpstart)
NO
Mirroring Enabled?
Rebuild Greenplum Database system from backup files
NO
Mirroring Enabled?
YES
YES
YES
YES
Recover Segment from Mirror on a Spare Host (gprecoverseg -p|-s)
Bring Host Back Online
Recover Segment from Mirror (gprecoverseg)
Recover Segment from Mirror (gprecoverseg)
Restart Greenplum Database (gpstop, gpstart)
NO
Mirroring Enabled?
Verify all Segments are Up (gpstate)
Verify all Segments are Up (gpstate)
YES
Recover Segment from Mirror (gprecoverseg)
Verify all Segments are Up (gpstate)

Figure 10: Segment Failure Troubleshooting Matrix

Recovering From Segment Failures

Segment host failures usually cause multiple segment failures: all primary or mirror segments on the host are marked as down and nonoperational. If mirroring is not enabled and a segment goes down, the system automatically becomes nonoperational.

To recover with mirroring enabled

1. Ensure you can connect to the segment host from the master host. For example:

   ```
   $ ping failed_seg_host_address
   ```

2. Troubleshoot the problem that prevents the master host from connecting to the segment host. For example, the host machine may need to be restarted or replaced.
3. After the host is online and you can connect to it, run the `gprecoverseg` utility from the master host to reactivate the failed segment instances. For example:

   ```
   $ gprecoverseg
   ```

4. The recovery process brings up the failed segments and identifies the changed files that need to be synchronized. The process can take some time; wait for the process to complete. During this process, database write activity is suspended.
5. After `gprecoverseg` completes, the system goes into *Resynchronizing* mode and begins copying the changed files. This process runs in the background while the system is online and accepting database requests.
6. When the resynchronization process completes, the system state is *Synchronized*. Run the `gpstate` utility to verify the status of the resynchronization process:

   ```
   $ gpstate -m
   ```

To return all segments to their preferred role

When a primary segment goes down, the mirror activates and becomes the primary segment. After running `gprecoverseg`, the currently active segment remains the primary and the failed segment becomes the mirror. The segment instances are not returned to the preferred role that they were given at system initialization time. This means that the system could be in a potentially unbalanced state if segment hosts have more active segments than is optimal for top system performance. To check for unbalanced segments and rebalance the system, run:

```
$ gpstate -e
```

All segments must be online and fully synchronized to rebalance the system. Database sessions remain connected during rebalancing, but queries in progress are canceled and rolled back.

1. Run `gpstate -m` to ensure all mirrors are *Synchronized*.

   ```
   $ gpstate -m
   ```

2. If any mirrors are in *Resynchronizing* mode, wait for them to complete.
3. Run gprecoverseg with the -r option to return the segments to their preferred roles.

   ```
   $ gprecoverseg -r
   ```

4. After rebalancing, run `gpstate -e` to confirm all segments are in their preferred roles.

   ```
   $ gpstate -e
   ```

To recover from a double fault

In a double fault, both a primary segment and its mirror are down. This can occur if hardware failures on different segment hosts happen simultaneously. Greenplum Database is unavailable if a double fault occurs. To recover from a double fault:

1. Restart Greenplum Database:

   ```
   $ gpstop -r
   ```
2. After the system restarts, run `gprecoverseg`:

   ```
   $ gprecoverseg
   ```
3. After `gprecoverseg` completes, use `gpstate` to check the status of your mirrors:

   ```
   $ gpstate -m
   ```
4. If you still have segments in Change Tracking mode, run a full copy recovery:

   ```
   $ gprecoverseg -F
   ```

If a segment host is not recoverable and you have lost one or more segments, recreate your Greenplum Database system from backup files. See *Backing Up and Restoring Databases*.

To recover without mirroring enabled

1. Ensure you can connect to the segment host from the master host. For example:

   ```
   $ ping failed_seg_host_address
   ```
2. Troubleshoot the problem that is preventing the master host from connecting to the segment host. For example, the host machine may need to be restarted.
3. After the host is online, verify that you can connect to it and restart Greenplum Database. For example:

   ```
   $ gpstop -r
   ```
4. Run the `gpstate` utility to verify that all segment instances are online:

   ```
   $ gpstate
   ```

When a segment host is not recoverable

If a host is nonoperational, for example, due to hardware failure, recover the segments onto a spare set of hardware resources. If mirroring is enabled, you can recover a segment from its mirror onto an alternate host using `gprecoverseg`. For example:

```
$ gprecoverseg -i recover_config_file
```

Where the format of *`recover_config_file`* is:

```
filespaceOrder=[filespace1_name[:filespace2_name:...]failed_host_address:
port:fselocation [recovery_host_address:port:replication_port:fselocation
[:fselocation:...]]
```

For example, to recover to a different host than the failed host without additional filespaces configured (besides the default *pg_system* filespace):

```
filespaceOrder=sdw5-2:50002:/gpdata/gpseg2 sdw9-2:50002:53002:/gpdata/gpseg2
```

The *gp_segment_configuration* and *pg_filespace_entry* system catalog tables can help determine your current segment configuration so you can plan your mirror recovery configuration. For example, run the following query:

```
=# SELECT dbid, content, hostname, address, port,
   replication_port, fselocation as datadir
   FROM gp_segment_configuration, pg_filespace_entry
   WHERE dbid=fsedbid
   ORDER BY dbid;
```

The new recovery segment host must be pre-installed with the Greenplum Database software and configured exactly as the existing segment hosts.

About the Segment Recovery Process

This topic describes the process for recovering segments, initiated by the `gprecoverseg` management utility. It describes actions `gprecoverseg` performs and how the Greenplum File Replication and FTS (fault tolerance service) processes complete the recovery initiated with `gprecoverseg`.

Although `gprecoverseg` is an online operation, there are two brief periods during which all IO is paused. First, when recovery is initiated, IO is suspended while empty data files are created on the mirror. Second, after data files are synchronized, IO is suspended while system files such as transaction logs are copied from the primary to the mirror. The duration of these pauses is affected primarily by the number of file system files that must be created on the mirror and the sizes of the system flat files that must be copied. The system is online and available while database tables are replicated from the primary to the mirror, and any changes made to the database during recovery are replicated directly to the mirror.

To initiate recovery, the administrator runs the `gprecoverseg` utility. `gprecoverseg` prepares segments for recovery and initiates synchronization. When synchronization is complete and the segment status is updated in the system catalog, the segments are recovered. If the recovered segments are not running in their preferred roles, `gprecoverseg -r` can be used to bring the system back into balance.

Without the `-F` option, `gprecoverseg` recovers segments incrementally, copying only the changes since the mirror entered down status. The `-F` option fully recovers mirrors by deleting their data directories and then synchronizing all persistent data files from the primary to the mirror.

You can run `gpstate -e` to view the mirroring status of the segments before and during the recovery process. The primary and mirror segment statuses are updated as the recovery process proceeds.

Consider a single primary-mirror segment pair where the primary is active and the mirror is down. The following table shows the segment status before beginning recovery of the mirror.

	preferred_role	**role**	**mode**	**status**
Primary	`p` (primary)	`p` (primary)	`c` (change tracking)	`u` (up)
Mirror	`m` (mirror)	`m` (mirror)	`s` (synchronizing)	`d` (down)

The segments are in their preferred roles, but the mirror is down. The primary is up and is in change tracking mode because it is unable to send changes to its mirror.

Segment Recovery Preparation

The `gprecoverseg` utility prepares the segments for recovery and then exits, allowing the Greenplum file replication processes to copy data from the primary to the mirror.

During the `gprecoverseg` execution the following recovery process steps are completed.

1. The down segments are identified.
2. The mirror segment processes are initialized.
3. For full recovery (`-aF`):
    - The data directories of the down segments are deleted.
    - A new data directory structure is created.
4. The segment mode in the `gp_segment_configuration` system table is updated to 'r' (resynchronization mode).
5. The backend performs the following:
    - Suspends IO—connections to the master are allowed, but reads and writes from the segment being recovered are not allowed.
    - Scans persistent tables on the primary segment.
    - For each persistent file object (`relfilenode` in the `pg_class` system table), creates a data file on the mirror.

    The greater the number of data files, the longer IO is suspended.

    For incremental recovery, the IO is suspended for a shorter period because only file system objects added (or dropped) on the primary after the mirror was marked down need to be created (or deleted) on the mirror.
6. The `gprecoverseg` script completes.

Once `gprecoverseg` has completed, the segments are in the states shown in the following table.

	preferred_role	**role**	**mode**	**status**
Primary	`p` (primary)	`p` (primary)	`r` (resynchronizing)	`u` (up)
Mirror	`m` (mirror)	`m` (mirror)	`r` (resynchronizing)	`u` (up)

Data File Replication

Data file resynchronization is performed in the background by file replication processes. Run `gpstate -e` to check the process of resynchronization. The Greenplum system is fully available for workloads while this process completes.

Following are steps in the resynchronization process:

1. Data copy (full and incremental recovery):

    After the file system objects are created, data copy is initiated for the affected segments. The ResyncManager process scans the persistent table system catalogs to find the file objects to be synchronized. ResyncWorker processes sync the file objects from the primary to the mirror.
2. Any changes or new data created with database transactions during the data copy are mirrored directly to the mirror.
3. Once data copy has finished synchronizing persistent data files, file replication updates the shared memory state on the current primary segment to 'insync'.

Flat File Replication

During this phase, system files in the primary segment's data directory are copied to the segment data directory. IO is suspended while the following flat files are copied from the primary data directory to the segment data directory:

- `pg_xlog/*`
- `pg_clog/*`

- `pg_distributedlog/*`
- `pg_distributedxidmap/*`
- `pg_multixact/members`
- `pg_multixact/offsets`
- `pg_twophase/*`
- `global/pg_database`
- `global/pg_auth`
- `global/pg_auth_time_constraint`

IOSUSPEND ends after these files are copied.

The next time the fault tolerance server (ftsprobe) process on the master wakes, it will set the primary and mirror states to synchronized (mode=s, state=u). A distributed query will also trigger the ftsprobe process to update the state.

When all segment recovery and file replication processes are complete, the segment status in the `gp_segment_configuration` system table and `gp_state -e` output is as shown in the following table.

	`preferred_role`	**`role`**	**`mode`**	**`status`**
Primary	`p` (primary)	`p` (primary)	`s` (synchronized)	`u` (up)
Mirror	`m` (mirror)	`m` (mirror)	`s` (synchronized)	`u` (up)

Factors Affecting Duration of Segment Recovery

Following are some factors that can affect the duration of the segment recovery process.

- The number of database objects, mainly tables and indexes.
- The number of data files in segment data directories.
- The types of workloads updating data during resynchronization – both DDL and DML (insert, update, delete, and truncate).
- The size of the data.
- The size of system files, such as transaction log files, pg_database, pg_auth, and pg_auth_time_constraint.

Recovering a Failed Master

If the primary master fails, log replication stops. Use the `gpstate -f` command to check the state of standby replication. Use `gpactivatestandby` to activate the standby master. Upon activation of the standby master, Greenplum Database reconstructs the master host state at the time of the last successfully committed transaction.

To activate the standby master

1. Ensure a standby master host is configured for the system. See *Enabling Master Mirroring*.
2. Run the `gpactivatestandby` utility from the standby master host you are activating. For example:

   ```
   $ gpactivatestandby -d /data/master/gpseg-1
   ```

   Where `-d` specifies the data directory of the master host you are activating.

   After you activate the standby, it becomes the *active* or *primary* master for your Greenplum Database array.
3. After the utility finishes, run `gpstate` to check the status:

   ```
   $ gpstate -f
   ```

   The newly activated master's status should be *Active*. If you configured a new standby host, its status is *Passive*. When a standby master is not configured, the command displays `-No entries found` and the message indicates that a standby master instance is not configured.
4. After switching to the newly active master host, run `ANALYZE` on it. For example:

   ```
   $ psql dbname -c 'ANALYZE;'
   ```
5. Optional: If you did not specify a new standby host when running the `gpactivatestandby` utility, use `gpinitstandby` to configure a new standby master at a later time. Run `gpinitstandby` on your active master host. For example:

   ```
   $ gpinitstandby -s new_standby_master_hostname
   ```

Restoring Master Mirroring After a Recovery

After you activate a standby master for recovery, the standby master becomes the primary master. You can continue running that instance as the primary master if it has the same capabilities and dependability as the original master host.

You must initialize a new standby master to continue providing master mirroring unless you have already done so while activating the prior standby master. Run `gpinitstandby` on the active master host to configure a new standby master.

You may restore the primary and standby master instances on the original hosts. This process swaps the roles of the primary and standby master hosts, and it should be performed only if you strongly prefer to run the master instances on the same hosts they occupied prior to the recovery scenario.

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

To restore the master and standby instances on original hosts (optional)

1. Ensure the original master host is in dependable running condition; ensure the cause of the original failure is fixed.

2. On the original master host, move or remove the data directory, `gpseg-1`. This example moves the directory to `backup_gpseg-1`:

   ```
   $ mv /data/master/gpseg-1 /data/master/backup_gpseg-1
   ```

   You can remove the backup directory once the standby is successfully configured.
3. Initialize a standby master on the original master host. For example, run this command from the current master host, smdw:

   ```
   $ gpinitstandby -s mdw
   ```
4. After the initialization completes, check the status of standby master, mdw, run `gpstate` with the `-f` option to check the status:

   ```
   $ gpstate -f
   ```

   The status should be *In Synch*.
5. Stop Greenplum Database master instance on the standby master. For example:

   ```
   $ gpstop -m
   ```
6. Run the `gpactivatestandby` utility from the original master host, mdw, that is currently a standby master. For example:

   ```
   $ gpactivatestandby -d $MASTER_DATA_DIRECTORY
   ```

   Where the `-d` option specifies the data directory of the host you are activating.
7. After the utility completes, run `gpstate` to check the status:

   ```
   $ gpstate -f
   ```

   Verify the original primary master status is *Active*. When a standby master is not configured, the command displays `-No entries found` and the message indicates that a standby master instance is not configured.
8. On the standby master host, move or remove the data directory, `gpseg-1`. This example moves the directory:

   ```
   $ mv /data/master/gpseg-1 /data/master/backup_gpseg-1
   ```

   You can remove the backup directory once the standby is successfully configured.
9. After the original master host runs the primary Greenplum Database master, you can initialize a standby master on the original standby master host. For example:

   ```
   $ gpinitstandby -s smdw
   ```

To check the status of the master mirroring process (optional)

You can display the information in the Greenplum Database system view pg_stat_replication. The view lists information about the `walsender` process that is used for Greenplum Database master mirroring. For example, this command displays the process ID and state of the `walsender` process:

```
$ psql dbname -c 'SELECT procpid, state FROM pg_stat_replication;'
```

Chapter 11

Backing Up and Restoring Databases

This topic describes how to use Greenplum backup and restore features.

Performing backups regularly ensures that you can restore your data or rebuild your Greenplum Database system if data corruption or system failure occur. You can also use backups to migrate data from one Greenplum Database system to another.

Backup and Restore Overview

Greenplum Database supports parallel and non-parallel methods for backing up and restoring databases. Parallel operations scale regardless of the number of segments in your system, because segment hosts each write their data to local disk storage simultaneously. With non-parallel backup and restore operations, the data must be sent over the network from the segments to the master, which writes all of the data to its storage. In addition to restricting I/O to one host, non-parallel backup requires that the master have sufficient local disk storage to store the entire database.

Parallel Backup and Restore

The Greenplum Database parallel dump utility `gpcrondump` backs up the Greenplum master instance and each active segment instance at the same time.

By default, `gpcrondump` creates dump files in the `db_dumps` subdirectory of each segment instance. On the master, `gpcrondump` creates several dump files, containing database information such as DDL statements, the system catalog tables, and metadata files. On each segment, `gpcrondump` creates one dump file, which contains commands to recreate the data on that segment. Each file created for a backup begins with a 14-digit timestamp key that identifies the backup set the file belongs to.

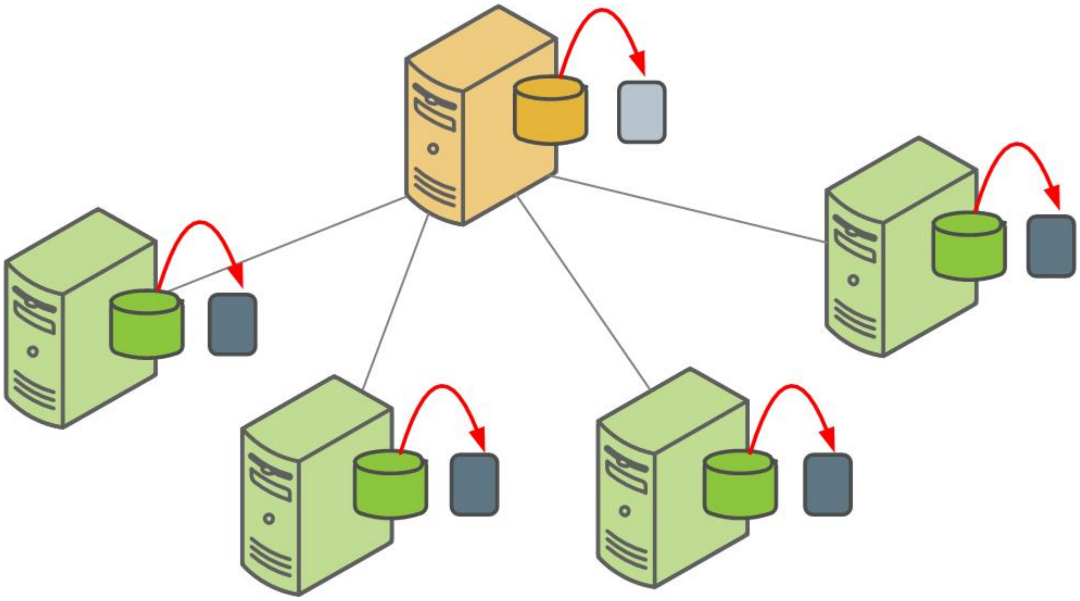


Figure 11: Parallel Backups in Greenplum Database

The `gpdbrestore` parallel restore utility takes the timestamp key generated by `gpcrondump`, validates the backup set, and restores the database objects and data into a distributed database. Parallel restore operations require a complete backup set created by `gpcrondump`, a full backup, and any required incremental backups. As the following figure illustrates, all segments restore data from local backup files simultaneously.

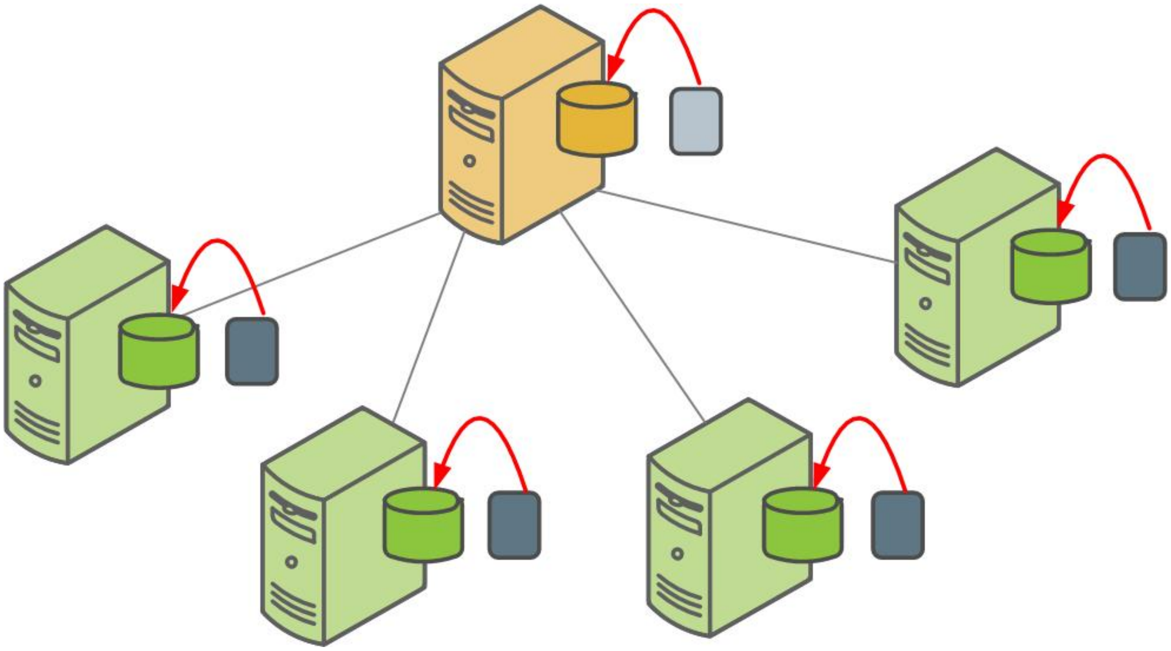


Figure 12: Parallel Restores in Greenplum Database

The `gpdbrestore` utility provides flexibility and verification options for use with the automated backup files produced by `gpcrondump` or with backup files moved from the Greenplum cluster to an alternate location. See *Restoring Greenplum Databases*. `gpdbrestore` can also be used to copy files to the alternate location.

Non-Parallel Backup and Restore

The PostgreSQL `pg_dump` and `pg_dumpall` non-parallel backup utilities can be used to create a single dump file on the master host that contains all data from all active segments.

The PostgreSQL non-parallel utilities should be used only for special cases. They are much slower than using the Greenplum backup utilities since all of the data must pass through the master. Additionally, it is often the case that the master host has insufficient disk space to save a backup of an entire distributed Greenplum database.

The `pg_restore` utility requires compressed dump files created by `pg_dump` or `pg_dumpall`. Before starting the restore, you should modify the `CREATE TABLE` statements in the dump files to include the Greenplum `DISTRIBUTED` clause. If you do not include the `DISTRIBUTED` clause, Greenplum Database assigns default values, which may not be optimal. For details, see `CREATE TABLE` in the *Greenplum Database Reference Guide*.

To perform a non-parallel restore using parallel backup files, you can copy the backup files from each segment host to the master host, and then load them through the master. See *Restoring to a Different Greenplum System Configuration*.

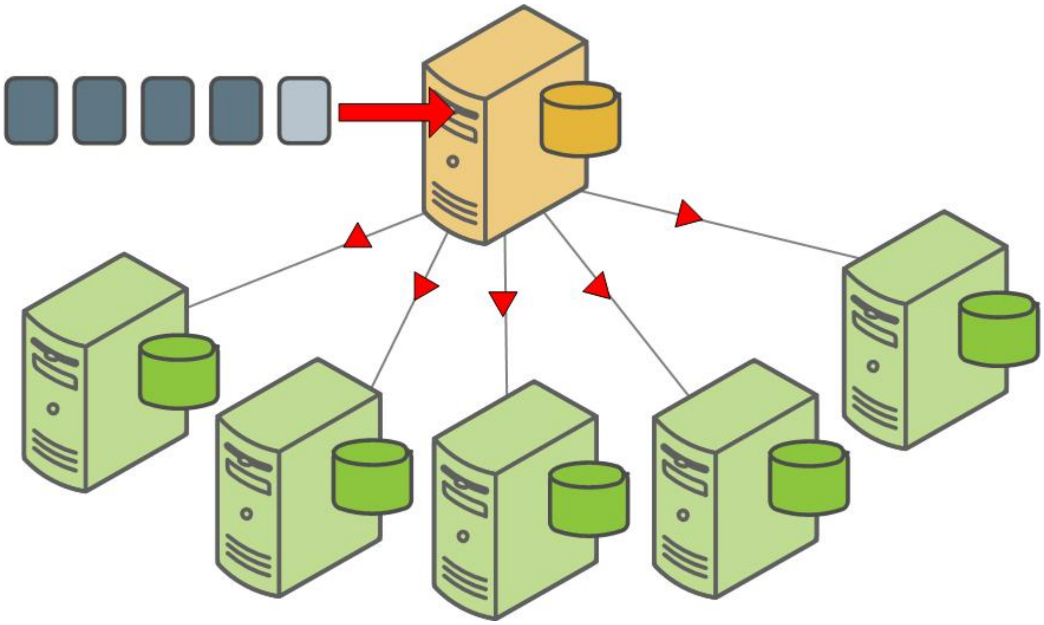


Figure 13: Non-parallel Restore Using Parallel Backup Files

Another non-parallel method for backing up Greenplum Database data is to use the `COPY TO` SQL command to copy all or a portion of a table out of the database to a delimited text file on the master host.

Backup and Restore Options

The Greenplum Database backup and restore utilities support various locations for backup files:

- With the `gpcrondump` utility, backup files may be saved in the default location, the `db_dumps` subdirectory of the master and each segment, or saved to a different directory specified with the `gpcrondump -u` option.
- Both the `gpcrondump` and `gpdbrestore` utilities have integrated support for EMC Data Domain Boost and Veritas NetBackup systems.
- Backup files can be saved through named pipes to any network accessible location.
- Backup files saved to the default location may be moved to an archive server on the network. This allows performing the backup at the highest transfer rates (when segments write the backup data to fast local disk arrays) and then freeing up disk space by moving the files to remote storage.

You can create dumps containing selected database objects:

- You can backup tables belonging to one or more schema you specify on the command line or in a text file.
- You can specify schema to exclude from the backup, as command-line options or in a list provided in a text file.
- You can backup a specified set of tables listed on the command line or in a text file. The table and schema options cannot be used together in a single backup.
- In addition to database objects, `gpcrondump` can backup the configuration files `pg_hba.conf`, `pg_ident.conf`, and `postgresql.conf`, and global database objects, such as roles and tablespaces.

You can create incremental backups:

- An incremental backup contains only append-optimized and column-oriented tables that have changed since the most recent incremental or full backup.
- For partitioned append-optimized tables, only changed append-optimized/column-oriented table partitions are backed up.
- Incremental backups include all heap tables.

- Use the `gpcrondump --incremental` flag to specify an incremental backup.
- Restoring an incremental backup requires a full backup and all subsequent incremental backups, up to the backup you are restoring.

The `gpdbrestore` utility offers many options:

- By default, `gpdbrestore` restores data to the database it was backed up from.
- The `--redirect` flag allows you to restore a backup to a different database.
- The restored database can be dropped and recreated, but the default is to restore into an existing database.
- Selected tables can be restored from a backup by listing the tables on the command line or by listing them in a text file and specifying the text file on the command line.
- You can restore a database from backup files moved to an archive server. The backup files are copied back into place on the master host and each segment host and then restored to the database.

Backing Up with gpcrondump

Use gpcrondump to backup databases, data, and objects such as database roles and server configuration files.

The gpcrondump utility dumps the contents of a Greenplum database to SQL script files on the master and each segment. The script files can then be used to restore the database.

The master backup files contain SQL commands to create the database schema. The segment data dump files contain SQL statements to load the data into the tables. The segment dump files are compressed using gzip. Optionally, the server configuration files postgresql.conf, pg_ident.conf, and pg_hba.conf and global data such as roles and tablespaces can be included in a backup.

The gpcrondump utility has one required flag, -x, which specifies the database to dump:

```
gpcrondump -x mydb
```

This performs a full backup of the specified database to the default locations.

By default, gpcrondump creates the backup files in the data directory on the master and each segment instance in the *data_directory*/db_dumps directory. You can specify a different backup location using the -u flag. For example, the following command will save backup files to the /backups directory:

```
gpcrondump mydb -u /backups
```

gpcrondump creates the db_dumps subdirectory in the specified directory. If there is more than one primary segment per host, all of the segments on the host write their backup files to the same directory. This differs from the default, where each segment writes backups to its own data directory. This can be used to consolidate backups to a single directory or mounted storage device.

In the db_dumps directory, backups are saved to a directory in the format *YYYYMMDD*, for example *data_directory*/db_dumps/20151012 for a backup created on October 12, 2015. The backup file names in the directory contain a full timestamp for the backup in the format *YYYYMMDDHHMMSS*, for example gp_dump_0_2_20151012195916.gz. The gpdbrestore command uses the most recent backup by default but you can specify an earlier backup to restore.

If you include the -g option, gpcrondump saves the configuration files with the backup. These configuration files are dumped in the master or segment data directory to db_dumps/YYYYMMDD/config_files_*timestamp*.tar. If --ddboost is specified, the backup is located on the default storage unit in the directory specified by --ddboost-backupdir when the Data Domain Boost credentials were set. The -G option backs up global objects such as roles and tablespaces to a file in the master backup directory named gp_global_1_1_*timestamp*.

If --ddboost is specified, the backup is located on the default storage unit in the directory specified by --ddboost-backupdir when the Data Domain Boost credentials were set.

There are many more gpcrondump options available to configure backups. Refer to the *Greenplum Utility Reference Guide* for information about all of the available options. See Backing Up Databases with Data Domain Boost for details of backing up with Data Domain Boost.

> **Warning:** Backing up a database with gpcrondump while simultaneously running ALTER TABLE might cause gpcrondump to fail.
>
> Backing up a database with gpcrondump while simultaneously running DDL commands might cause issues with locks. You might see either the DDL command or gpcrondump waiting to acquire locks.

Backing Up a Set of Tables

You can create a backup that includes a subset of the schema or tables in a database by using the following `gpcrondump` options:

- `-t` *`schema.tablename`* – specify a table to include in the backup. You can use the `-t` option multiple times.
- `--table-file=`*`filename`* – specify a file containing a list of tables to include in the backup.
- `-T` *`schema.tablename`* – specify a table to exclude from the backup. You can use the `-T` option multiple times.
- `--exclude-table-file=`*`filename`* – specify a file containing a list of tables to exclude from the backup.
- `-s` *`schema_name`* – include all tables qualified by a specified schema name in the backup. You can use the `-s` option multiple times.
- `--schema-file=`*`filename`* – specify a file containing a list of schemas to include in the backup.
- `-S` *`schema_name`* – exclude tables qualified by a specified schema name from the backup. You can use the `-S` option multiple times.
- `--exclude-schema-file=`*`filename`* – specify a file containing schema names to exclude from the backup.

Only a set of tables or set of schemas can be specified. For example, the `-s` option cannot be specified with the `-t` option.

Refer to *Incremental Backup with Sets* for additional information about using these `gpcrondump` options with incremental backups.

Creating Incremental Backups

The gpcrondump and gpdbrestore utilities support incremental backups and restores of append-optimized tables, including column-oriented tables. Use the gpcrondump option --incremental to create an incremental backup.

An incremental backup only backs up an append-optimized or column-oriented table if one of the following operations was performed on the table after the last full or incremental backup:

- ALTER TABLE
- DELETE
- INSERT
- TRUNCATE
- UPDATE
- DROP and then re-create the table

For partitioned append-optimized tables, only the changed partitions are backed up.

Heap tables are backed up with every full and incremental backup.

Incremental backups are efficient when the total amount of data in append-optimized table partitions or column-oriented tables that changed is small compared to the data has not changed.

Each time gpcrondump runs, it creates state files that contain row counts for each append optimized and column-oriented table and partition in the database. The state files also store metadata operations such as truncate and alter. When gpcrondump runs with the --incremental option, it compares the current state with the stored state to determine if the table or partition should be included in the incremental backup.

A unique 14-digit timestamp key identifies files that comprise an incremental backup set.

To create an incremental backup or to restore data from an incremental backup, you need the complete backup set. A complete backup set consists of a full backup and any incremental backups that were created since the last full backup. When you archive incremental backups, all incremental backups between the last full backup and the target incremental backup must be archived. You must archive all the files created on the master and all segments.

> **Important:** For incremental back up sets, a full backup and associated incremental backups, the backup set must be on a single device. For example, a backup set must all be on a Data Domain system. The backup set cannot have some backups on a Data Domain system and others on the local file system or a NetBackup system.
>
> **Note:** You can use a Data Domain server as an NFS file system (without Data Domain Boost) to perform incremental backups.

Changes to the Greenplum Database segment configuration invalidate incremental backups. After you change the segment configuration you must create a full backup before you can create an incremental backup.

Incremental Backup Example

Each backup set has a key, which is a timestamp taken when the backup is created. For example, if you create a backup on May 14, 2012, the backup set file names contain 20120514*hhmmss*. The *hhmmss* represents the time: hour, minute, and second.

For this example, assume you have created both full and incremental backups of the database *mytest*. To create the full backup, you used the following command:

```
gpcrondump -x mytest -u /backupdir
```

Later, after some changes have been made to append optimized tables, you created an increment backup with the following command:

```
gpcrondump -x mytest -u /backupdir --incremental
```

When you specify the -u option, the backups are created in the /backupdir directory on each Greenplum Database host. The file names include the following timestamp keys. The full backups have the timestamp key 20120514054532 and 20121114064330. The other backups are incremental backups.

- 20120514054532 (full backup)
- 20120714095512
- 20120914081205
- 20121114064330 (full backup)
- 20130114051246

To create a new incremental backup, you need both the most recent incremental backup 20130114051246 and the preceding full backup 20121114064330. Also, you must specify the same -u option for any incremental backups that are part of the backup set.

To restore a database with the incremental backup 20120914081205, you need the incremental backups 20120914081205 and 20120714095512, and the full backup 20120514054532.

To restore the *mytest* database with the incremental backup 20130114051246, you need only the incremental backup and the full backup 20121114064330. The restore command would be similar to this command.

```
gpdbrestore -t 20130114051246 -u /backupdir
```

Incremental Backup with Sets

To back up a set of database tables with incremental backup, identify the backup set with the --prefix option when you create the full backup with gpcrondump. For example, to create incremental backups for tables in the myschema schema, first create a full backup with a prefix, such as myschema:

```
gpcrondump -x mydb -s myschema --prefix myschema
```

The -s option specifies that tables qualified by the myschema schema are to be included in the backup. See *Backing Up a Set of Tables* for more options to specify a set of tables to back up.

Once you have a full backup you can create an incremental backup for the same set of tables by specifying the gpcrondump --incremental and --prefix options, specifying the prefix you set for the full backup. The incremental backup is automatically limited to only the tables in the full backup. For example:

```
gpcrondump -x mydb --incremental --prefix myschema
```

The following command lists the tables that were included or excluded for the full backup.

```
gpcrondump -x mydb --incremental --prefix myschema --list-filter-tables
```

Restoring From an Incremental Backup

When restoring a backup with gpdbrestore, the command-line output displays whether the restore type is incremental or a full database restore. You do not have to specify that the backup is incremental. For example, the following gpdbrestore command restores the most recent backup of the mydb database. gpdbrestore searches the db_dumps directory to locate the most recent dump and displays information about the backup it found.

```

$ gpdbrestore -s mydb
```

```
...
20151015:20:10:34:002664 gpdbrestore:mdw:gpadmin-
[INFO]:------------------------------------------
20151015:20:10:34:002664 gpdbrestore:mdw:gpadmin-[INFO]:-Greenplum database restore
 parameters
20151015:20:10:34:002664 gpdbrestore:mdw:gpadmin-
[INFO]:------------------------------------------
20151015:20:10:34:002664 gpdbrestore:mdw:gpadmin-[INFO]:-Restore type              =
 Incremental Restore
20151015:20:10:34:002664 gpdbrestore:mdw:gpadmin-[INFO]:-Database to be restored    =
 mydb
20151015:20:10:34:002664 gpdbrestore:mdw:gpadmin-[INFO]:-Drop and re-create db      =
 Off
20151015:20:10:34:002664 gpdbrestore:mdw:gpadmin-[INFO]:-Restore method             =
 Search for latest
20151015:20:10:34:002664 gpdbrestore:mdw:gpadmin-[INFO]:-Restore timestamp          =
 20151014194445
20151015:20:10:34:002664 gpdbrestore:mdw:gpadmin-[INFO]:-Restore compressed dump    =
 On
20151015:20:10:34:002664 gpdbrestore:mdw:gpadmin-[INFO]:-Restore global objects     =
 Off
20151015:20:10:34:002664 gpdbrestore:mdw:gpadmin-[INFO]:-Array fault tolerance      =
 f
20151015:20:10:34:002664 gpdbrestore:mdw:gpadmin-
[INFO]:------------------------------------------

Continue with Greenplum restore Yy|Nn (default=N):
```

`gpdbrestore` ensures that the full backup and other required incremental backups are available before restoring the backup. With the `--list-backup` option you can display the full and incremental backup sets required to perform a restore.

If the `gpdbrestore` option `-q` is specified, the backup type information is written to the log file. With the `gpdbrestore` option `--noplan`, you can restore only the data contained in an incremental backup.

Backup Process and Locks

During backups, the `gpcrondump` utility locks the *pg_class* system table and the tables that are backed up. Locking the *pg_class* table with an `EXCLUSIVE` lock ensures that no tables are added, deleted, or altered until the backup is complete. `gpcrondump` locks tables that are to be backed up with `ACCESS SHARE` locks.

The steps below describe the process `gpcrondump` follows to dump databases, including what happens before locks are placed, while locks are held, and after locks are removed.

If more than one database is specified, this process is executed for each database in sequence.

- `gpcrondump` parses the command-line arguments and verifies that options and arguments are properly specified.
- If any of the following filter options are specified, `gpcrondump` prepares filters to determine the set of tables to back up. Otherwise, all tables are included in the backup.
  - `-s` *`schema_name`* – Includes all the tables qualified by the specified schema.
  - `-S` *`schema_name`* – Excludes all tables qualified by the specified schema.
  - `--schema-file=`*`filename`* – Includes all the tables qualified by the schema listed in *filename*.
  - `--exclude-schema-file=`*`filename`* – Excludes all tables qualified by the schema listed in *filename*.
  - `-t` *`schema.table_name`* – Dumps the specified table.
  - `-T` *`schema.table_name`* – Excludes the specified table.
  - `--table-file=`*`filename`* – Dumps the tables specified in *filename*.
  - `--exclude-table-file=`*`filename`* – Dumps all tables except tables specified in *filename*.
- `gpcrondump` verifies the backup targets:
  - Verifies that the database exists.
  - Verifies that specified tables or schemas to be dumped exist.
  - Verifies that the current primary for each segment is up.
  - Verifies that the backup directory exists and is writable for the master and each segment.
  - Verifies that sufficient disk space is available to back up each segment. Note that if more than one segment is backed up to the same disk volume, this disk space check cannot ensure there is space available for all segments.
- Places an `EXCLUSIVE` lock on the `pg_class` table. The `EXCLUSIVE` lock permits only concurrent read operations. Relations such as tables, indexes, and views cannot be created or dropped in the database.

  `gpcrondump` starts a thread to watch for a lock file (*`dump_dir`*`/gp_lockfile_`*`timestamp`*) to appear, signaling the parallel backup on the segments has completed.
- `gpcrondump` locks tables that are to be backed up with an `ACCESS SHARE` lock.

  An `ACCESS SHARE` lock only conflicts with an `ACCESS EXCLUSIVE` lock. The following SQL statements acquire an `ACCESS EXCLUSIVE` lock:
  - `ALTER TABLE`
  - `CLUSTER`
  - `DROP TABLE`
  - `REINDEX`
  - `TRUNCATE`
  - `VACUUM FULL`
- Threads are created and dispatched for the master and segments to perform the database dump.
- When all threads have completed, the *`dump_dir`*`/gp_lockfile_`*`timestamp`* lock file is created, signaling `gpcrondump` to release the `EXCLUSIVE` lock on the *pg_class* catalog table, while tables are being backed up.

- gpcrondump checks the status files for each primary segment for any errors to report. If a dump fails and the -r flag was specified, the backup is rolled back.
- The ACCESS SHARE locks on the target tables are released.
- If the backup succeeded and a post-dump script was specified with the -R option, gpcrondump runs the script now.
- gpcrondump reports the backup status, sends email if configured, and saves the backup status in the public.gpcrondump_history table in the database.

Using Direct I/O

The operating system normally caches file I/O operations in memory because memory access is much faster than disk access. The application writes to a block of memory that is later flushed to the storage device, which is usually a RAID controller in a Greenplum Database system. Whenever the application accesses a block that is still resident in memory, a device access is avoided. Direct I/O allows you to bypass the cache so that the application writes directly to the storage device. This reduces CPU consumption and eliminates a data copy operation. Direct I/O is efficient for operations like backups where file blocks are only handled once.

Note: Direct I/O is supported only on Red Hat, CentOS, and SUSE.

Turning on Direct I/O

Set the `gp_backup_directIO` system configuration parameter on to enable direct I/O for backups:

```
$ gpconfig -c gp_backup_directIO -v on
```

To see if direct I/O is enabled, use this command:

```
$ gpconfig -s gp_backup_directIO
```

Decrease network data chunks sent to dump when the database is busy

The `gp_backup_directIO_read_chunk_mb` configuration parameter sets the size, in MB, of the I/O chunk when direct I/O is enabled. The default chunk size, 20MB, has been tested and found to be optimal. Decreasing it increases the backup time and increasing it results in little change to backup time.

To find the current direct I/O chunk size, enter this command:

```
$ gpconfig -s gp_backup_directIO_read_chunk_mb
```

The following example changes the default chunk size to 10MB.

```
$ gpconfig -c gp_backup_directIO_read_chunk_mb -v 10
```

Using Named Pipes

Greenplum Database allows the use of named pipes with `gpcrondump` and `gpdbrestore` to back up and restore Greenplum databases. When backing up a database with regular files, the files that contain the backup information are placed in directories on the Greenplum Database segments. If segment hosts do not have enough local disk space available to backup to files, you can use named pipes to back up to non-local storage, such as storage on another host on the network or to a backup appliance.

Backing up with named pipes is not supported if the `--ddboost` option is specified.

To back up a Greenplum database using named pipes

1. Run the `gpcrondump` command with options `-K timestamp` and `--list-backup-files`.

   This creates two text files that contain the names of backup files, one per line. The file names include the `timestamp` you specified with the `-K timestamp` option and have the suffixes `_pipes` and `_regular_files`. For example:

   ```
   gp_dump_20150519160000_pipes
   gp_dump_20150519160000_regular_files
   ```

   The file names listed in the `_pipes` file are to be created as named pipes. The file names in the `_regular_files` file should not be created as named pipes. `gpcrondump` and `gpdbrestore` use the information in these files during backup and restore operations.
2. Create named pipes on all Greenplum Database segments using the file names in the generated `_pipes` file.
3. Redirect the output of each named pipe to the destination process or file object.
4. Run `gpcrondump` to back up the database using the named pipes.

   To create a complete set of Greenplum Database backup files, the files listed in the `_regular_files` file must also be backed up.

To restore a database that used named pipes during backup

1. Direct the contents of each backup file to the input of its named pipe, for example `cat` *`filename`* `>` *`pipename`*, if the backup file is accessible as a local file object.
2. Run the `gpdbrestore` command to restore the database using the named pipes.

Example

This example shows how to back up a database over the network using named pipes and the netcat (`nc`) Linux command. The segments write backup files to the inputs of the named pipes. The outputs of the named pipes are piped through `nc` commands, which make the files available on TCP ports. Processes on other hosts can then connect to the segment hosts at the designated ports to receive the backup files. This example requires that the `nc` package is installed on all Greenplum hosts.

1. Enter the following `gpcrondump` command to generate the lists of backup files for the `testdb` database in the `/backups` directory.

   ```
   $ gpcrondump -x testdb -K 20150519160000 --list-backup-files -u /backups
   ```
2. View the files that `gpcrondump` created in the `/backup` directory:

   ```
   $ ls -lR /backups
   /backups:
   total 4
   drwxrwxr-x 3 gpadmin gpadmin 4096 May 19 21:49 db_dumps

   /backups/db_dumps:
   total 4
   ```

```
drwxrwxr-x 2 gpadmin gpadmin 4096 May 19 21:49 20150519

/backups/db_dumps/20150519:
total 8
-rw-rw-r-- 1 gpadmin gpadmin 256 May 19 21:49 gp_dump_20150519160000_pipes
-rw-rw-r-- 1 gpadmin gpadmin 391 May 19 21:49 gp_dump_20150519160000_regular_files
```

3. View the contents of the `_pipes` file.

```
$ cat /backups/db_dumps/20150519/gp_dump_20150519160000_pipes
sdw1:/backups/db_dumps/20150519/gp_dump_0_2_20150519160000.gz
sdw2:/backups/db_dumps/20150519/gp_dump_0_3_20150519160000.gz
mdw:/backups/db_dumps/20150519/gp_dump_1_1_20150519160000.gz
mdw:/backups/db_dumps/20150519/gp_dump_1_1_20150519160000_post_data.gz
```

4. Create the specified named pipes on the Greenplum Database segments. Also set up a reader for the named pipe.

```
gpssh -h sdw1
[sdw1] mkdir -p /backups/db_dumps/20150519/
[sdw1] mkfifo /backups/db_dumps/20150519/gp_dump_0_2_20150519160000.gz
[sdw1] cat /backups/db_dumps/20150519/gp_dump_0_2_20150519160000.gz | nc -l 21000
[sdw1] exit
```

   Complete these steps for each of the named pipes listed in the `_pipes` file. Be sure to choose an available TCP port for each file.

5. On the destination hosts, receive the backup files with commands like the following:

```
nc sdw1 21000 > gp_dump_0_2_20150519160000.gz
```

6. Run `gpcrondump` to begin the backup:

```
gpcrondump -x testdb -K 20150519160000 -u /backups
```

To restore a database with named pipes, reverse the direction of the backup files by sending the contents of the backup files to the inputs of the named pipes and run the `gpdbrestore` command:

```
gpdbrestore -x testdb -t 20150519160000 -u /backups
```

`gpdbrestore` reads from the named pipes' outputs.

Backing Up Databases with Data Domain Boost

EMC Data Domain Boost (DD Boost) is EMC software that can be used with the `gpcrondump` and `gpdbrestore` utilities to perform faster backups to the EMC Data Domain storage appliance. Data Domain performs deduplication on the data it stores, so after the initial backup operation, the appliance stores only pointers to data that is unchanged. This reduces the size of backups on disk. When DD Boost is used with `gpcrondump`, Greenplum Database participates in the deduplication process, reducing the volume of data sent over the network. When you restore files from the Data Domain system with Data Domain Boost, some files are copied to the master local disk and are restored from there, and others are restored directly.

With Data Domain Boost managed file replication, you can replicate Greenplum Database backup images that are stored on a Data Domain system for disaster recover purposes. The `gpmfr` utility manages the Greenplum Database backup sets that are on the primary and a remote Data Domain system. For information about `gpmfr`, see the *Greenplum Database Utility Guide*.

Managed file replication requires network configuration when a replication network is being used between two Data Domain systems:

- The Greenplum Database system requires the Data Domain login credentials to be configured using `gpcrondump`. Credentials must be created for both the local and remote Data Domain systems.
- When the non-management network interface is used for replication on the Data Domain systems, static routes must be configured on the systems to pass the replication data traffic to the correct interfaces.

Do not use Data Domain Boost with `pg_dump` or `pg_dumpall`.

Refer to Data Domain Boost documentation for detailed information.

> **Important:** For incremental back up sets, a full backup and the associated incremental backups must be on a single device. For example, a backup set must all be on a file system. The backup set cannot have some backups on the local file system and others on single storage unit of a Data Domain system. For backups on a Data Domain system, the backup set must be in a single storage unit.
>
> **Note:** You can use a Data Domain server as an NFS file system (without Data Domain Boost) to perform incremental backups.

Data Domain Boost Requirements

Using Data Domain Boost requires the following.

- Data Domain Boost in included only with Pivotal Greenplum Database.
- Purchase and install EMC Data Domain Boost and Replicator licenses on the Data Domain systems.
- Obtain sizing recommendations for Data Domain Boost. Make sure the Data Domain system supports sufficient write and read streams for the number of segment hosts in your Greenplum cluster.

Contact your EMC Data Domain account representative for assistance.

One-Time Data Domain Boost Credential Setup

There is a one-time process to set up credentials to use Data Domain Boost. Credential setup connects one Greenplum Database instance to one Data Domain instance. If you are using the `gpcrondump --replicate` option or DD Boost managed file replication capabilities for disaster recovery purposes, you must set up credentials for both the local and remote Data Domain systems.

To set up credentials, run `gpcrondump` with the following options:

```
--ddboost-host ddboost_hostname --ddboost-user ddboost_user
```

```
--ddboost-backupdir backup_directory --ddboost-storage-unit storage_unit_ID
```

The --ddboost-storage-unit is optional. If not specified, the storage unit ID is GPDB.

To remove credentials, run gpcrondump with the --ddboost-config-remove option.

To manage credentials for the remote Data Domain system that is used for backup replication, include the --ddboost-remote option with the other gpcrondump options. For example, the following options set up credentials for a Data Domain system that is used for backup replication. The system IP address is 192.0.2.230, the user ID is ddboostmyuser, and the location for the backups on the system is GPDB/gp_production:

```
--ddboost-host 192.0.2.230 --ddboost-user ddboostmyuser
--ddboost-backupdir gp_production --ddboost-remote
```

For details, see gpcrondump in the *Greenplum Database Utility Guide*.

If you use two or more network connections to connect to the Data Domain system, use gpcrondump to set up the login credentials for the Data Domain hostnames associated with the network interfaces. To perform this setup for two network connections, run gpcrondump with the following options:

```
--ddboost-host ddboost_hostname1
--ddboost-host ddboost_hostname2 --ddboost-user ddboost_user
--ddboost-backupdir backup_directory
```

About DD Boost Credential Files

The gpcrondump utility is used to schedule DD Boost backup operations. The utility is also used to set, change, or remove one-time credentials and a storage unit ID for DD Boost. The gpcrondump, gpdbrestore, and gpmfr utilities use the DD Boost credentials to access Data Domain systems. DD Boost information is stored in these files.

- DDBOOST_CONFIG is used by gpdbrestore and gpcrondump for backup and restore operations with the Data Domain system. The gpdbrestore utility creates or updates the file when you specify Data Domain information with the --ddboost-host option.
- DDBOOST_MFR_CONFIG is used by gpmfr for remote replication operations with the remote Data Domain system. The gpdbrestore utility creates or updates the file when you specify Data Domain information with the --ddboost-host option and --ddboost-remote option.

The configuration files are created in the current user (gpadmin) home directory on the Greenplum Database master and segment hosts. The path and file name cannot be changed. Information in the configuration files includes:

- Data Domain host name or IP address
- DD Boost user name
- DD Boost password
- Default Data Domain backup directory (DDBOOST_CONFIG only)
- Data Domain storage unit ID: default is GPDB (DDBOOST_CONFIG only)
- Data Domain default log level: default is WARNING
- Data Domain default log size: default is 50

Use the gpcrondump option --ddboost-show-config to display the current DD Boost configuration information from the Greenplum Database master configuration file. Specify the --remote option to display the configuration information for the remote Data Domain system.

About Data Domain Storage Units

When you use a Data Domain system to perform a backup, restore, or remote replication operation with the `gpcrondump`, `gpdbrestore`, or `gpmfr` utility, the operation uses a storage unit on a Data Domain system. You can specify the storage unit ID when you perform these operations:

- When you set the DD Boost credentials with the `gpcrondump` utility `--ddboost-host` option.

  If you specify the `--ddboost-storage-unit` option, the storage unit ID is written to the Greenplum Database DD Boost configuration file `DDBOOST_CONFIG`. If the storage unit ID is not specified, the default value is `GPDB`.

  If you specify the `--ddboost-storage-unit` option and the `--ddboost-remote` option to set DD Boost credentials for the remote Data Domain server, the storage ID information is ignored. The storage unit ID in the `DDBOOST_CONFIG` file is the default ID that is used for remote replication operations.
- When you perform a backup, restore, or remote replication operation with `gpcrondump`, `gpdbrestore`, or `gpmfr`.

  When you specify the `--ddboost-storage-unit` option, the utility uses the specified Data Domain storage unit for the operation. The value in the configuration file is not changed.

A Greenplum Database utility uses the storage unit ID based on this order of precedence from highest to lowest:

- Storage unit ID specified with `--ddboost-storage-unit`
- Storage unit ID specified in the configuration file
- Default storage unit ID `GPDB`

Greenplum Database master and segment instances use a single storage unit ID when performing a backup, restore, or remote replication operation.

> **Important:** The storage unit ID in the Greenplum Database master and segment host configuration files must be the same. The Data Domain storage unit is created by the `gpcrondump` utility from the Greenplum Database master host.
>
> The following occurs if storage unit IDs are different in the master and segment host configuration files:
>
> - If all the storage units have not been created, the operation fails.
> - If all the storage units have been created, a backup operation completes. However, the backup files are in different storage units and a restore operation fails because a full set of backup files is not in a single storage unit.

When performing a full backup operation (not an incremental backup), the storage unit is created on the Data Domain system if it does not exist.

A storage unit is not created if these `gpcrondump` options are specified: `--incremental`, `--list-backup-file`, `--list-filter-tables`, `-o`, or `--ddboost-config-remove`.

Greenplum Database replication operations use the same storage unit ID on both systems. For example, if you specify the `--ddboost-storage-unit` option for `--replicate` or `--recover` through `gpmfr` or `--replicate` from `gpcrondump`, the storage unit ID applies to both local and remote Data Domain systems.

When performing a replicate or recover operation with `gpmfr`, the storage unit on the destination Data Domain system (where the backup is being copied) is created if it does not exist.

Configuring Data Domain Boost for Greenplum Database

After you set up credentials for Data Domain Boost on the Greenplum Database, perform the following tasks in Data Domain to allow Data Domain Boost to work with Greenplum Database:

- *Configuring Distributed Segment Processing in Data Domain*

- *Configuring Advanced Load Balancing and Link Failover in Data Domain*
- *Export the Data Domain Path to the DCA Network*

Configuring Distributed Segment Processing in Data Domain

Configure the distributed segment processing option on the Data Domain system. The configuration applies to all the DCA servers and the Data Domain Boost plug-in installed on them. This option is enabled by default, but verify that it is enabled before using Data Domain Boost backups:

```
# ddboost option show
```

To enable or disable distributed segment processing:

```
# ddboost option set distributed-segment-processing {enabled | disabled}
```

Configuring Advanced Load Balancing and Link Failover in Data Domain

If you have multiple network connections on a network subnet, you can create an interface group to provide load balancing and higher network throughput on your Data Domain system. When a Data Domain system on an interface group receives data from the media server clients, the data transfer is load balanced and distributed as separate jobs on the private network. You can achieve optimal throughput with multiple 10 GbE connections.

> **Note:** To ensure that interface groups function properly, use interface groups only when using multiple network connections on the same networking subnet.

To create an interface group on the Data Domain system, create interfaces with the `net` command. If interfaces do not already exist, add the interfaces to the group, and register the Data Domain system with the backup application.

1. Add the interfaces to the group:

   ```
   # ddboost ifgroup add interface 192.0.2.1
   # ddboost ifgroup add interface 192.0.2.2
   # ddboost ifgroup add interface 192.0.2.3
   # ddboost ifgroup add interface 192.0.2.4
   ```

   > **Note:** You can create only one interface group and this group cannot be named.

2. Select one interface on the Data Domain system to register with the backup application. Create a failover aggregated interface and register that interface with the backup application.

   > **Note:** You do not have to register one of the `ifgroup` interfaces with the backup application. You can use an interface that is not part of the `ifgroup` to register with the backup application.

3. Enable `ddboost` on the Data Domain system:

   ```
   # ddboost ifgroup enable
   ```

4. Verify the Data Domain system configuration as follows:

   ```
   # ddboost ifgroup show config
   ```

   Results similar to the following are displayed.

   ```
   Interface
   -------------
   192.0.2.1
   192.0.2.2
   192.0.2.3
   192.0.2.4
   -------------
   ```

You can add or delete interfaces from the group at any time.

> **Note:** Manage Advanced Load Balancing and Link Failover (an interface group) using the `ddboost ifgroup` command or from the **Enterprise Manager Data Management** > **DD Boost** view.

Export the Data Domain Path to the DCA Network

The commands and options in this topic apply to DDOS 5.0.x and 5.1.x. See the Data Domain documentation for details.

Use the following Data Domain commands to export the `/backup/ost` directory to the DCA for Data Domain Boost backups.

```
# nfs add /backup/ost 192.0.2.0/24, 198.51.100.0/24 (insecure)
```

> **Note:** The IP addresses refer to the Greenplum system working with the Data Domain Boost system.

Create the Data Domain Login Credentials for the DCA

Create a username and password for the DCA to access the DD Boost Storage Unit (SU) at the time of backup and restore:

```
# user add user [password password] [priv {admin | security | user}]
```

Backup Options for Data Domain Boost

Specify the `gpcrondump` options to match the setup.

Data Domain Boost backs up files to a storage unit in the Data Domain system. Status and report files remain on the local disk. If needed, specify the Data Domain system storage unit with the `--ddboost-storage-unit` option. This DD Boost comand display the names of all storage units on a Data Domain system

```
ddboost storage-unit show
```

To configure Data Domain Boost to remove old backup directories before starting a backup operation, specify a `gpcrondump` backup expiration option:

- The `-c` option clears all backup directories.
- The `-o` option clears the oldest backup directory.

To remove the oldest dump directory, specify `gpcrondump --ddboost` with the `-o` option. For example, if your retention period is 30 days, use `gpcrondump --ddboost` with the `-o` option on day 31.

Use `gpcrondump --ddboost` with the `-c` option to clear out all the old dump directories in `db_dumps`. The `-c` option deletes all dump directories that are at least one day old.

Using CRON to Schedule a Data Domain Boost Backup

1. Ensure the *One-Time Data Domain Boost Credential Setup* is complete.
2. Add the option `--ddboost` to the `gpcrondump` option:

   ```
   gpcrondump -x mydatabase -z -v --ddboost
   ```

   If needed, specify the Data Domain system storage unit with the `--ddboost-storage-unit` option.

   > **Important:** Do not use compression with Data Domain Boost backups. The `-z` option turns backup compression off.

Some of the options available in `gpcrondump` have different implications when using Data Domain Boost. For details, see `gpcrondump` in the *Greenplum Database Utility Reference*.

Restoring From a Data Domain System with Data Domain Boost

1. Ensure the *One-Time Data Domain Boost Credential Setup* is complete.
2. Add the option `--ddboost` to the `gpdbrestore` command:

```
$ gpdbrestore -t backup_timestamp -v -ddboost
```

If needed, specify the Data Domain system storage unit with the `--ddboost-storage-unit` option.

Note: Some of the `gpdbrestore` options available have different implications when using Data Domain. For details, see `gpdbrestore` in the *Greenplum Database Utility Reference*.

Backing Up Databases with Veritas NetBackup

For Greenplum Database on Red Hat Enterprise Linux, you can configure Greenplum Database to perform backup and restore operations with Veritas NetBackup. You configure Greenplum Database and NetBackup and then run a Greenplum Database `gpcrondump` or `gpdbrestore` command. The following topics describe how to set up NetBackup and back up or restore Greenplum Databases.

- *About NetBackup Software*
- *System Requirements*
- *Limitations*
- *Configuring Greenplum Database Hosts for NetBackup*
- *Configuring NetBackup for Greenplum Database*
- *Performing a Back Up or Restore with NetBackup*
- *Example NetBackup Back Up and Restore Commands*

About NetBackup Software

NetBackup includes the following server and client software:

- The NetBackup master server manages NetBackup backups, archives, and restores. The master server is responsible for media and device selection for NetBackup.
- NetBackup Media servers are the NetBackup device hosts that provide additional storage by allowing NetBackup to use the storage devices that are attached to them.
- NetBackup client software that resides on the Greenplum Database hosts that contain data to be backed up.

See the *Veritas NetBackup Getting Started Guide* for information about NetBackup.

System Requirements

> **Note:** Greenplum Database uses the NetBackup API (XBSA) to communicate with NetBackup.

- Supported NetBackup Master Server software.
  - NetBackup Master Server Version 7.5 and NetBackup Media Server Version 7.5
  - NetBackup Master Server Version 7.6 and NetBackup Media Server Version 7.6
- Supported NetBackup Client version: 7.1, 7.5, or 7.6.

  NetBackup client software installed and configured on the Greenplum Database master host and all segment hosts.

  The NetBackup client software must be able to communicate with the NetBackup server software.

  For NetBackup version 7.5 or 7.6, the client version that is installed and configured on the Greenplum Database hosts must match the NetBackup Server version that stores the Greenplum Database backup.

  For NetBackup Client version 7.1, Greenplum Database supports only NetBackup Server Version 7.5.

  > **Note:** Greenplum Database support for NetBackup Client version 7.1 is deprecated. The NetBackup SDK library files for NetBackup version 7.1 will be removed from the Greenplum Database installation in a future release.

Limitations

- NetBackup is not compatible with DDBoost. Both NetBackup and DDBoost cannot be used in a single back up or restore operation.

- For incremental back up sets, a full backup and associated incremental backups, the backup set must be on a single device. For example, a backup set must all be on a NetBackup system. The backup set cannot have some backups on a NetBackup system and others on the local file system or a Data Domain system.

Configuring Greenplum Database Hosts for NetBackup

You install and configure NetBackup client software on the Greenplum Database master host and all segment hosts.

1. Install the NetBackup client software on Greenplum Database hosts. See the NetBackup installation documentation for information on installing NetBackup clients on UNIX systems.
2. Set parameters in the NetBackup configuration file `/usr/openv/netbackup/bp.conf` on the Greenplum Database master and segment hosts. Set the following parameters on each Greenplum Database host.

   Table 14: NetBackup bp.conf parameters for Greenplum Database

Parameter	Description
`SERVER`	Host name of the NetBackup Master Server
`MEDIA_SERVER`	Host name of the NetBackup Media Server
`CLIENT_NAME`	Host name of the Greenplum Database Host

   See the *Veritas NetBackup Administrator's Guide* for information about the `bp.conf` file.
3. Set the `LD_LIBRARY_PATH` environment variable for Greenplum Database hosts to use NetBackup client 7.1, 7.5 or 7.6. Greenplum Database installs NetBackup SDK library files that are used with the NetBackup 7.1, 7.5, or 7.6 client. To configure Greenplum Database to use the library files that correspond to the version of the NetBackup client that is installed on the hosts, add the following line to the file `$GPHOME/greenplum_path.sh`:

   ```
   LD_LIBRARY_PATH=$GPHOME/lib/nbuNN/lib:$LD_LIBRARY_PATH
   ```

   Replace the *NN* with, 76, 75, or 71 depending on the NetBackup client that is installed on the host.

   The `LD_LIBRARY_PATH` line should be added before this line in `$GPHOME/greenplum_path.sh`:

   ```
   export LD_LIBRARY_PATH
   ```
4. Execute this command to remove the current `LD_LIBRARY_PATH` value:

   ```
   unset LD_LIBRARY_PATH
   ```
5. Execute this command to update the environment variables for Greenplum Database:

   ```
   source $GPHOME/greenplum_path.sh
   ```

See the *Veritas NetBackup Administrator's Guide* for information about configuring NetBackup servers.

1. Ensure that the Greenplum Database hosts are listed as NetBackup clients for the NetBackup server.

   In the NetBackup Administration Console, the information for the NetBackup clients, Media Server, and Master Server is in the **NetBackup Management** node within the **Host Properties** node.
2. Configure a NetBackup storage unit. The storage unit must be configured to point to a writable disk location.

   In the NetBackup Administration Console, the information for NetBackup storage units is in **NetBackup Management** node within the **Storage** node.
3. Configure a NetBackup backup policy and schedule within the policy.

In the NetBackup Administration Console, the **Policy** node below the **Master Server** node is where you create a policy and a schedule for the policy.

- In the **Policy Attributes** tab, these values are required for Greenplum Database:

  The value in the **Policy type** field must be DataStore

  The value in the **Policy storage** field is the storage unit created in the previous step.

  The value in **Limit jobs per policy** field must be at least 3.
- In the **Policy Schedules** tab, create a NetBackup schedule for the policy.

Configuring NetBackup for Greenplum Database

See the *Veritas NetBackup Administrator's Guide* for information about configuring NetBackup servers.

1. Ensure that the Greenplum Database hosts are listed as NetBackup clients for the NetBackup server.

   In the NetBackup Administration Console, the information for the NetBackup clients, Media Server, and Master Server is in **NetBackup Management** node within the **Host Properties** node.
2. Configure a NetBackup storage unit. The storage unit must be configured to point to a writable disk location.

   In the NetBackup Administration Console, the information for NetBackup storage units is in **NetBackup Management** node within the **Storage** node.
3. Configure a NetBackup backup policy and schedule within the policy.

   In the NetBackup Administration Console, the **Policy** node below the **Master Server** node is where you create a policy and a schedule for the policy.

   - In the **Policy Attributes** tab, these values are required for Greenplum Database:

     The value in the **Policy type** field must be DataStore

     The value in the **Policy storage** field is the storage unit created in the previous step.

     The value in **Limit jobs per policy** field must be at least 3.
   - In the **Policy Schedules** tab, create a NetBackup schedule for the policy.

Performing a Back Up or Restore with NetBackup

The Greenplum Database `gpcrondump` and `gpdbrestore` utilities support options to back up or restore data to a NetBackup storage unit. When performing a back up, Greenplum Database transfers data files directly to the NetBackup storage unit. No backup data files are created on the Greenplum Database hosts. The backup metadata files are both stored on the hosts and backed up to the NetBackup storage unit.

When performing a restore, the files are retrieved from the NetBackup server, and then restored.

Following are the `gpcrondump` utility options for NetBackup:

```
--netbackup-service-host netbackup_master_server
--netbackup-policy policy_name
--netbackup-schedule schedule_name
--netbackup-block-size size (optional)
--netbackup-keyword keyword (optional)
```

The `gpdbrestore` utility provides the following options for NetBackup:

```
--netbackup-service-host netbackup_master_server
--netbackup-block-size size (optional)
```

Note: When performing a restore operation from NetBackup, you must specify the backup timestamp with the `gpdbrestore` utility `-t` option.

The policy name and schedule name are defined on the NetBackup master server. See *Configuring NetBackup for Greenplum Database* for information about policy name and schedule name. See the *Greenplum Database Utility Guide* for information about the Greenplum Database utilities.

> **Note:** You must run the `gpcrondump` or `gpdbrestore` command during a time window defined for the NetBackup schedule.

During a back up or restore operation, a separate NetBackup job is created for the following types of Greenplum Database data:

- Segment data for each segment instance
- C database data
- Metadata
- Post data for the master
- State files Global objects (`gpcrondump -G` option)
- Configuration files for master and segments (`gpcrondump -g` option)
- Report files (`gpcrondump -h` option)

In the NetBackup Administration Console, the Activity Monitor lists NetBackup jobs. For each job, the job detail displays Greenplum Database backup information.

> **Note:** When backing up or restoring a large amount of data, set the NetBackup `CLIENT_READ_TIMEOUT` option to a value that is at least twice the expected duration of the operation (in seconds). The `CLIENT_READ_TIMEOUT` default value is `300` seconds (5 minutes).
>
> For example, if a backup takes 3 hours, set the `CLIENT_READ_TIMEOUT` to `21600` (2 x 3 x 60 x 60). You can use the NetBackup `nbgetconfig` and `nbsetconfig` commands on the NetBackup server to view and change the option value.
>
> For information about `CLIENT_READ_TIMEOUT` and the `nbgetconfig`, and `nbsetconfig` commands, see the NetBackup documentation.

Example NetBackup Back Up and Restore Commands

This `gpcrondump` command backs up the database *customer* and specifies a NetBackup policy and schedule that are defined on the NetBackup master server `nbu_server1`. A block size of 1024 bytes is used to transfer data to the NetBackup server.

```
gpcrondump -x customer --netbackup-service-host=nbu_server1 \
   --netbackup-policy=gpdb_cust --netbackup-schedule=gpdb_backup \
   --netbackup-block-size=1024
```

This `gpdbrestore` command restores Greenplum Database data from the data managed by NetBackup master server `nbu_server1`. The option `-t 20130530090000` specifies the timestamp generated by `gpcrondump` when the backup was created. The `-e` option specifies that the target database is dropped before it is restored.

```
gpdbrestore -t 20130530090000 -e --netbackup-service-host=nbu_server1
```

Restoring Greenplum Databases

How you restore a database from parallel backup files depends on how you answer the following questions.

1. **Where are your backup files?** If your backup files are on the segment hosts where `gpcrondump` created them, you can restore the database with `gpdbrestore`. If you moved your backup files away from the Greenplum array, for example to an archive server with `gpcrondump`, use `gpdbrestore`.
2. **Are you recreating the Greenplum Database system, or just restoring your data?** If Greenplum Database is running and you are restoring your data, use `gpdbrestore`. If you lost your entire array and need to rebuild the entire system from backup, use `gpinitsystem`.
3. **Are you restoring to a system with the same number of segment instances as your backup set?** If you are restoring to an array with the same number of segment hosts and segment instances per host, use `gpdbrestore`. If you are migrating to a different array configuration, you must do a non-parallel restore. See *Restoring to a Different Greenplum System Configuration*.

Restoring a Database Using gpdbrestore

The `gpdbrestore` utility restores a database from backup files created by `gpcrondump`.

The `gpdbrestore` requires one of the following options to identify the backup set to restore:

- `-t` *`timestamp`* – restore the backup with the specified timestamp.
- `-b` *`YYYYYMMDD`* – restore dump files for the specified date in the `db_dumps` subdirectories on the segment data directories.
- `-s` *`database_name`* – restore the latest dump files for the specified database found in the segment data directories.
- `-R` *`hostname:path`* – restore the backup set located in the specified directory of a remote host.

To restore an incremental backup, you need a complete set of backup files—a full backup and any required incremental backups. You can use the `--list-backups` option to list the full and incremental backup sets required for an incremental backup specified by timestamp. For example:

```
$ gpdbrestore -t 20151013195916 --list-backup
```

You can restore a backup to a different database using the `--redirect` *`database`* option. The database is created if it does not exist. The following example restores the most recent backup of the `mydb` database to a new database named `mydb_snapshot`:

```
$ gpdbrestore -s grants --redirect grants_snapshot
```

You can also restore a backup to a different Greenplum Database system. See *Restoring to a Different Greenplum System Configuration* for information about this option.

To restore from an archive host using gpdbrestore

You can restore a backup saved on a host outside of the Greenplum cluster using the `-R` option. Although the Greenplum Database software does not have to be installed on the remote host, the remote host must have a `gpadmin` account configured with passwordless ssh access to all hosts in the Greenplum cluster. This is required because each segment host will use scp to copy its segment backup file from the archive host. See `gpssh-exkeys` in the *Greenplum Database Utility Guide* for help adding the remote host to the cluster.

This procedure assumes that the backup set was moved off the Greenplum array to another host in the network.

1. Ensure that the archive host is reachable from the Greenplum master host:

   ```
   $ ping archive_host
   ```
2. Ensure that you can ssh into the remote host with the `gpadmin` account and no password.

   ```
   $ ssh gpadmin@archive_host
   ```
3. Ensure that you can ping the master host from the archive host:

   ```
   $ ping mdw
   ```
4. Ensure that the restore's target database exists. For example:

   ```
   $ createdb database_name
   ```

5. From the master, run the gpdbrestore utility. The -R option specifies the host name and path to a complete backup set:

   ```
   $ gpdbrestore -R archive_host:/gpdb/backups/archive/20120714 -e dbname
   ```

   Omit the -e *dbname* option if the database has already been created.

Restoring to a Different Greenplum System Configuration

To perform a parallel restore operation using `gpdbrestore`, the system you are restoring to must have the same configuration as the system that was backed up. To restore your database objects and data into a different system configuration, for example, to expand into a system with more segments, restore your parallel backup files by loading them through the Greenplum master. To perform a non-parallel restore, you must have:

- A full backup set created by a `gpcrondump` operation. The backup file of the master contains the DDL to recreate your database objects. The backup files of the segments contain the data.
- A running Greenplum Database system.
- The database you are restoring to exists in the system.

Segment dump files contain a `COPY` command for each table followed by the data in delimited text format. Collect all of the dump files for all of the segment instances and run them through the master to restore your data and redistribute it across the new system configuration.

To restore a database to a different system configuration

1. Ensure that you have a complete backup set, including dump files of the master (`gp_dump_1_1_timestamp`, `gp_dump_1_1_timestamp_post_data`) and one for each segment instance (`gp_dump_0_2_timestamp`, `gp_dump_0_3_timestamp`, `gp_dump_0_4_timestamp`, and so on). Each dump file must have the same timestamp key. `gpcrondump` creates the dump files in each segment instance's data directory. You must collect all the dump files and move them to one location on the master host. You can copy each segment dump file to the master, load it, and then delete it after it loads successfully.
2. Ensure that the database you are restoring to is created in the system. For example:

   ```
   $ createdb database_name
   ```

3. Load the master dump file to restore the database objects. For example:

   ```
   $ psql database_name -f /gpdb/backups/gp_dump_1_1_20120714
   ```

4. Load each segment dump file to restore the data. For example:

   ```
   $ psql database_name -f /gpdb/backups/gp_dump_0_2_20120714
   $ psql database_name -f /gpdb/backups/gp_dump_0_3_20120714
   $ psql database_name -f /gpdb/backups/gp_dump_0_4_20120714
   $ psql database_name -f /gpdb/backups/gp_dump_0_5_20120714
   ...
   ```

5. Load the post data file to restore database objects such as indexes, triggers, primary key constraints, etc.

   ```
   $ psql database_name -f /gpdb/backups/gp_dump_0_5_20120714_post_data
   ```

6. Update the database sequences based on the values from the original database.

   You can use the system utilities `gunzip` and `egrep` to extract the sequence value information from the original Greenplum Database master dump file `gp_dump_1_1_timestamp.gz` into a text file. This command extracts the information into the file `schema_path_and_seq_next_val`.

   ```
   gunzip -c path_to_master_dump_directory/gp_dump_1_1_timestamp.gz | egrep "SET
    search_path|SELECT pg_catalog.setval"
      > schema_path_and_seq_next_val
   ```

This example command assumes the original Greenplum Database master dump file is in `/data/gpdb/master/gpseg-1/db_dumps/20150112`.

```
gunzip -c /data/gpdb/master/gpseg-1/db_dumps/20150112/
gp_dump_1_1_20150112140316.gz
  | egrep "SET search_path|SELECT pg_catalog.setval" >
 schema_path_and_seq_next_val
```

After extracting the information, use the Greenplum Database `psql` utility to update the sequences in the database. This example command updates the sequence information in the database *test_restore*:

```
psql test_restore -f schema_path_and_seq_next_val
```

Chapter 12

Expanding a Greenplum System

To scale up performance and storage capacity, expand your Greenplum system by adding hosts to the array.

Data warehouses typically grow over time as additional data is gathered and the retention periods increase for existing data. At times, it is necessary to increase database capacity to consolidate different data warehouses into a single database. Additional computing capacity (CPU) may also be needed to accommodate newly added analytics projects. Although it is wise to provide capacity for growth when a system is initially specified, it is not generally possible to invest in resources long before they are required. Therefore, you should expect to execute a database expansion project periodically.

Because of the Greenplum MPP architecture, when you add resources to the system, the capacity and performance are the same as if the system had been originally implemented with the added resources. Unlike data warehouse systems that require substantial downtime in order to dump and restore the data, expanding a Greenplum Database system is a phased process with minimal downtime. Regular and ad hoc workloads can continue while data is redistributed and transactional consistency is maintained. The administrator can schedule the distribution activity to fit into ongoing operations and can pause and resume as needed. Tables can be ranked so that datasets are redistributed in a prioritized sequence, either to ensure that critical workloads benefit from the expanded capacity sooner, or to free disk space needed to redistribute very large tables.

The expansion process uses standard Greenplum Database operations so it is transparent and easy for administrators to troubleshoot. Segment mirroring and any replication mechanisms in place remain active, so fault-tolerance is uncompromised and disaster recovery measures remain effective.

System Expansion Overview

Data warehouses typically grow over time, often at a continuous pace, as additional data is gathered and the retention period increases for existing data. At times, it is necessary to increase database capacity to consolidate disparate data warehouses into a single database. The data warehouse may also require additional computing capacity (CPU) to accommodate added analytics projects. It is good to provide capacity for growth when a system is initially specified, but even if you anticipate high rates of growth, it is generally unwise to invest in capacity long before it is required. Database expansion, therefore, is a project that you should expect to have to execute periodically.

When you expand your database, you should expect the following qualities:

- Scalable capacity and performance. When you add resources to a Greenplum Database, the capacity and performance are the same as if the system had been originally implemented with the added resources.
- Uninterrupted service during expansion. Regular workloads, both scheduled and ad-hoc, are not interrupted. A short, scheduled downtime period is required to initialize the new servers, similar to downtime required to restart the system. The length of downtime is unrelated to the size of the system before or after expansion.
- Transactional consistency.
- Fault tolerance. During the expansion, standard fault-tolerance mechanisms—such as segment mirroring—remain active, consistent, and effective.
- Replication and disaster recovery. Any existing replication mechanisms continue to function during expansion. Restore mechanisms needed in case of a failure or catastrophic event remain effective.
- Transparency of process. The expansion process employs standard Greenplum Database mechanisms, so administrators can diagnose and troubleshoot any problems.
- Configurable process. Expansion can be a long running process, but it can be fit into a schedule of ongoing operations. Expansion control tables allow adminstrators to prioritize the order in which tables are redistributed and the expansion activity can be paused and resumed.

The planning and physical aspects of an expansion project are a greater share of the work than expanding the database itself. It will take a multi-discipline team to plan and execute the project. Space must be acquired and prepared for the new servers. The servers must be specified, acquired, installed, cabled, configured, and tested. Consulting Greenplum Database platform engineers in the planning stages will help to ensure a successful expansion project. *Planning New Hardware Platforms* describes general considerations for deploying new hardware.

After you provision the new hardware platforms and set up their networks, configure the operating systems and run performance tests using Greenplum utilities.The Greenplum Database software distribution includes utilities that are helpful to test and burn-in the new servers before beginning the software phase of the expansion. See *Preparing and Adding Nodes* for steps to ready the new hosts for Greenplum Database.

Once the new servers are installed and tested, the software phase of the Greenplum Database expansion process begins. The software phase is designed to be minimally disruptive, transactionally consistent, reliable, and flexible.

- There is a brief period of downtime while the new segment hosts are initialized and the system is prepared for the expansion process. This downtime can be scheduled to occur during a period of low activity to avoid disrupting ongoing business operations. During the initialization process, the following tasks are performed:
    - Greenplum Database software is installed.
    - Databases and database objects are created on the new segment hosts.
    - An expansion schema is created in the master database to control the expansion process.

    - The distribution policy for each table is changed to DISTRIBUTED RANDOMLY.
- The system is restarted and applications resume.
    - New segments are immediately available and participate in new queries and data loads. The existing data, however, is skewed. It is concentrated on the original segments and must be redistributed across the new total number of primary segments.
    - Because tables now have a random distribution policy, the optimizer creates query plans that are not dependent on distribution keys. Some queries will be less efficient because more data motion operators are needed.
- Using the expansion control tables as a guide, tables and partitions are redistributed. For each table:
    - An ALTER TABLE statement is issued to change the distribution policy back to the original policy. This causes an automatic data redistribution operation, which spreads data across all of the servers, old and new, according to the original distribution policy.
    - The table's status is updated in the expansion control tables.
    - The query optimizer creates more efficient execution plans by including the distribution key in the planning.
- When all tables have been redistributed, the expansion is complete.

Redistributing data is a long-running process that creates a large volume of network and disk activity. It can take days to redistribute some very large databases. To minimize the effects of the increased activity on business operations, system administrators can pause and resume expansion activity on an ad hoc basis, or according to a predetermined schedule. Datasets can be prioritized so that critical applications benefit first from the expansion.

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

1. To *create an expansion input file*:

   ```
   gpexpand -f hosts_file
   ```
2. To *initialize segments and create the expansion schema*:

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

   gpexpand creates a data directory, copies user tables from all existing databases on the new segments, and captures metadata for each table in an expansion schema for status tracking. After this process completes, the expansion operation is committed and irrevocable.
3. *To redistribute tables*:

   ```
   gpexpand -d duration
   ```

   At initialization, gpexpand nullifies hash distribution policies on tables in all existing databases, except for parent tables of a partitioned table, and sets the distribution policy for all tables to random distribution.

   To complete system expansion, you must run gpexpand to redistribute data tables across the newly added segments. Depending on the size and scale of your system, redistribution can be accomplished in a single session during low-use hours, or you can divide the process into batches over an extended period. Each table or partition is unavailable for read or write operations during redistribution. As each table is redistributed across the new segments, database performance should incrementally improve until it exceeds pre-expansion performance levels.

   You may need to run gpexpand several times to complete the expansion in large-scale systems that require multiple redistribution sessions. gpexpand can benefit from explicit table redistribution ranking; see *Planning Table Redistribution*.

   Users can access Greenplum Database after initialization completes and the system is back online, but they may experience performance degradation on systems that rely heavily on hash distribution of tables. Normal operations such as ETL jobs, user queries, and reporting can continue, though users might experience slower response times.

When a table has a random distribution policy, Greenplum Database cannot enforce unique constraints (such as `PRIMARY KEY`). This can affect your ETL and loading processes until table redistribution completes because duplicate rows do not issue a constraint violation error.

4. To remove the expansion schema:

```
gpexpand -c
```

For information about the `gpexpand` utility and the other utilities that are used for system expansion, see the *Greenplum Database Utility Guide*.

Planning Greenplum System Expansion

Careful planning will help to ensure a successful Greenplum expansion project.

The topics in this section help to ensure that you are prepared to execute a system expansion.

- *System Expansion Checklist* is a checklist you can use to prepare for and execute the system expansion process.
- *Planning New Hardware Platforms* covers planning for acquiring and setting up the new hardware.
- *Planning New Segment Initialization* provides information about planning to initialize new segment hosts with `gpexpand`.
- *Planning Table Redistribution* provides information about planning the data redistribution after the new segment hosts have been initialized.

System Expansion Checklist

This checklist summarizes the tasks for a Greenplum Database system expansion.

Table 15: Greenplum Database System Expansion Checklist

Online Pre-Expansion Tasks * System is up and available	
□	Devise and execute a plan for ordering, building, and networking new hardware platforms.
□	Devise a database expansion plan. Map the number of segments per host, schedule the offline period for testing performance and creating the expansion schema, and schedule the intervals for table redistribution.
□	Perform a complete schema dump.
□	Install Greenplum Database binaries on new hosts.
□	Copy SSH keys to the new hosts (`gpssh-exkeys`).
□	Validate the operating system environment of the new hardware (`gpcheck`).
□	Validate disk I/O and memory bandwidth of the new hardware (`gpcheckperf`).
□	Validate that the master data directory has no extremely large files in the `pg_log` or `gpperfmon/data` directories.
□	Validate that there are no catalog issues (`gpcheckcat`).

□	Prepare an expansion input file (`gpexpand`).
Offline Expansion Tasks * The system is locked and unavailable to all user activity during this process.	
□	Validate the operating system environment of the combined existing and new hardware (`gpcheck`).
□	Validate disk I/O and memory bandwidth of the combined existing and new hardware (`gpcheckperf`).
□	Initialize new segments into the array and create an expansion schema (`gpexpand-i` *input_file*).
Online Expansion and Table Redistribution * System is up and available	
□	Before you start table redistribution, stop any automated snapshot processes or other processes that consume disk space.
□	Redistribute tables through the expanded system (`gpexpand`).
□	Remove expansion schema (`gpexpand -c`).
□	Run `analyze` to update distribution statistics. During the expansion, use `gpexpand -a`, and post-expansion, use `analyze`.

Planning New Hardware Platforms

A deliberate, thorough approach to deploying compatible hardware greatly minimizes risk to the expansion process.

Hardware resources and configurations for new segment hosts should match those of the existing hosts. Work with *Greenplum Platform Engineering* before making a hardware purchase to expand Greenplum Database.

The steps to plan and set up new hardware platforms vary for each deployment. Some considerations include how to:

- Prepare the physical space for the new hardware; consider cooling, power supply, and other physical factors.
- Determine the physical networking and cabling required to connect the new and existing hardware.
- Map the existing IP address spaces and developing a networking plan for the expanded system.
- Capture the system configuration (users, profiles, NICs, and so on) from existing hardware to use as a detailed list for ordering new hardware.
- Create a custom build plan for deploying hardware with the desired configuration in the particular site and environment.

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

Planning New Segment Initialization

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

The time required depends on the number of schema objects in the Greenplum system and other factors related to hardware performance. In most environments, the initialization of new segments requires less than thirty minutes offline.

> **Note:** After you begin initializing new segments, you can no longer restore the system using backup files created for the pre-expansion system. When initialization successfully completes, the expansion is committed and cannot be rolled back.

Planning Mirror Segments

If your existing array has mirror segments, the new segments must have mirroring configured. If there are no mirrors configured for existing segments, you cannot add mirrors to new hosts with the `gpexpand` utility.

For Greenplum Database arrays with mirror segments, ensure you add enough new host machines to accommodate new mirror segments. The number of new hosts required depends on your mirroring strategy:

- **Spread Mirroring** — Add at least one more host to the array than the number of segments per host. The number of separate hosts must be greater than the number of segment instances per host to ensure even spreading.
- **Grouped Mirroring** — Add at least two new hosts so the mirrors for the first host can reside on the second host, and the mirrors for the second host can reside on the first. For more information, see *About Segment Mirroring*.

Increasing Segments Per Host

By default, new hosts are initialized with as many primary segments as existing hosts have. You can increase the segments per host or add new segments to existing hosts.

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

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

About the Expansion Schema

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

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

- *gpexpand.status*
- *gpexpand.status_detail*
- *gpexpand.expansion_progress*

Control expansion process aspects by modifying *gpexpand.status_detail*. For example, removing a record from this table prevents the system from expanding the table across new segments. Control the order in which tables are processed for redistribution by updating the `rank` value for a record. For more information, see *Ranking Tables for Redistribution*.

Planning Table Redistribution

Table redistribution is performed while the system is online. For many Greenplum systems, table redistribution completes in a single `gpexpand` session scheduled during a low-use period. Larger systems may require multiple sessions and setting the order of table redistribution to minimize performance impact. Pivotal recommends completing the table redistribution in one session if possible.

> **Important:** To perform table redistribution, your segment hosts must have enough disk space to temporarily hold a copy of your largest table. All tables are unavailable for read and write operations during redistribution.

The performance impact of table redistribution depends on the size, storage type, and partitioning design of a table. Per table, redistributing a table with `gpexpand` takes as much time as a `CREATE TABLE AS SELECT` operation does. When redistributing a terabyte-scale fact table, the expansion utility can use much of the available system resources, with resulting impact on query performance or other database workloads.

Managing Redistribution in Large-Scale Greenplum Systems

You can manage the order in which tables are redistributed by adjusting their ranking. See *Ranking Tables for Redistribution*. Manipulating the redistribution order can help adjust for limited disk space and restore optimal query performance.

When planning the redistribution phase, consider the impact of the exclusive lock taken on each table during redistribution. User activity on a table can delay its redistribution. Tables are unavailable during redistribution.

Systems with Abundant Free Disk Space

In systems with abundant free disk space (required to store a copy of the largest table), you can focus on restoring optimum query performance as soon as possible by first redistributing important tables that queries use heavily. Assign high ranking to these tables, and schedule redistribution operations for times of low system usage. Run one redistribution process at a time until large or critical tables have been redistributed.

Systems with Limited Free Disk Space

If your existing hosts have limited disk space, you may prefer to first redistribute smaller tables (such as dimension tables) to clear space to store a copy of the largest table. Available disk space on the original segments increases as each table is redistributed across the expanded array. When enough free space exists on all segments to store a copy of the largest table, you can redistribute large or critical tables. Redistribution of large tables requires exclusive locks; schedule this procedure for off-peak hours.

Also consider the following:

- Run multiple parallel redistribution processes during off-peak hours to maximize available system resources.
- When running multiple processes, operate within the connection limits for your Greenplum system. For information about limiting concurrent connections, see *Limiting Concurrent Connections*.

Redistributing Append-Optimized and Compressed Tables

`gpexpand` redistributes append-optimized and compressed append-optimized tables at different rates than heap tables. The CPU capacity required to compress and decompress data tends to increase the impact on system performance. For similar-sized tables with similar data, you may find overall performance differences like the following:

- Uncompressed append-optimized tables expand 10% faster than heap tables.

- `zlib`-compressed append-optimized tables expand at a significantly slower rate than uncompressed append-optimized tables, potentially up to 80% slower.
- Systems with data compression such as ZFS/LZJB take longer to redistribute.

> **Important:** If your system nodes use data compression, use identical compression on new nodes to avoid disk space shortage.

Redistributing Tables with Primary Key Constraints

There is a time period during which primary key constraints cannot be enforced between the initialization of new segments and successful table redistribution. Duplicate data inserted into tables during this period prevents the expansion utility from redistributing the affected tables.

After a table is redistributed, the primary key constraint is properly enforced again. If an expansion process violates constraints, the expansion utility logs errors and displays warnings when it completes. To fix constraint violations, perform one of the following remedies:

- Clean up duplicate data in the primary key columns, and re-run `gpexpand`.
- Drop the primary key constraints, and re-run `gpexpand`.

Redistributing Tables with User-Defined Data Types

You cannot perform redistribution with the expansion utility on tables with dropped columns of user-defined data types. To redistribute tables with dropped columns of user-defined types, first re-create the table using `CREATE TABLE AS SELECT`. After this process removes the dropped columns, redistribute the table with `gpexpand`.

Redistributing Partitioned Tables

Because the expansion utility can process each individual partition on a large table, an efficient partition design reduces the performance impact of table redistribution. Only the child tables of a partitioned table are set to a random distribution policy. The read/write lock for redistribution applies to only one child table at a time.

Redistributing Indexed Tables

Because the `gpexpand` utility must re-index each indexed table after redistribution, a high level of indexing has a large performance impact. Systems with intensive indexing have significantly slower rates of table redistribution.

Preparing and Adding Nodes

Verify your new nodes are ready for integration into the existing Greenplum system.

To prepare new system nodes for expansion, install the Greenplum Database software binaries, exchange the required SSH keys, and run performance tests.

Pivotal recommends running performance tests first on the new nodes and then all nodes. Run the tests on all nodes with the system offline so user activity does not distort results.

Generally, Pivotal recommends running performance tests when an administrator modifies node networking or other special conditions in the system. For example, if you will run the expanded system on two network clusters, run tests on each cluster.

Adding New Nodes to the Trusted Host Environment

New nodes must exchange SSH keys with the existing nodes to enable Greenplum administrative utilities to connect to all segments without a password prompt. Pivotal recommends performing the key exchange process twice.

First perform the process as `root`, for administration convenience, and then as the user `gpadmin`, for management utilities. Perform the following tasks in order:

1. *To exchange SSH keys as root*
2. *To create the gpadmin user*
3. *To exchange SSH keys as the gpadmin user*

> **Note:** The Greenplum Database segment host naming convention is `sdwN` where `sdw` is a prefix and `N` is an integer. For example, on a Greenplum Database DCA system, segment host names would be `sdw1`, `sdw2` and so on. For hosts with multiple interfaces, the convention is to append a dash (`-`) and number to the host name. For example, `sdw1-1` and `sdw1-2` are the two interface names for host `sdw1`.

To exchange SSH keys as root

1. Create a host file with the existing host names in your array and a separate host file with the new expansion host names. For existing hosts, you can use the same host file used to set up SSH keys in the system. In the files, list all hosts (master, backup master, and segment hosts) with one name per line and no extra lines or spaces. Exchange SSH keys using the configured host names for a given host if you use a multi-NIC configuration. In this example, `mdw` is configured with a single NIC, and `sdw1`, `sdw2`, and `sdw3` are configured with 4 NICs:

```
mdw
sdw1-1
sdw1-2
sdw1-3
sdw1-4
sdw2-1
sdw2-2
sdw2-3
sdw2-4
sdw3-1
sdw3-2
sdw3-3
sdw3-4
```

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

```
$ su -
```

```
# source /usr/local/greenplum-db/greenplum_path.sh
```

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

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

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

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

To create the `gpadmin` user

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

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

2. Set a password for the new `gpadmin` user. On Linux, you can do this on all segment hosts simultaneously using `gpssh`. For example:

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

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

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

To exchange SSH keys as the `gpadmin` user

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

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

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

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

Verifying OS Settings

Use the `gpcheck` utility to verify all new hosts in your array have the correct OS settings to run Greenplum Database software.

To run gpcheck

1. Log in on the master host as the user who will run your Greenplum Database system (for example, `gpadmin`).

   ```
   $ su - gpadmin
   ```

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

   ```
   $ gpcheck -f new_hosts_file
   ```

Validating Disk I/O and Memory Bandwidth

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

To run gpcheckperf

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

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

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

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

Integrating New Hardware into the System

Before initializing the system with the new segments, shut down the system with gpstop to prevent user activity from skewing performance test results. Then, repeat the performance tests using host files that include *all* nodes, existing and new:

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

Initializing New Segments

Use the gpexpand utility to initialize the new segments, create the expansion schema, and set a system-wide random distribution policy for the database.

The first time you run gpexpand with a valid input file it creates the expansion schema and sets the distribution policy for all tables to DISTRIBUTED RANDOMLY. After these steps are completed, running gpexpand detects if the expansion schema has been created and, if so, performs table redistribution.

- *Creating an Input File for System Expansion*
- *Running gpexpand to Initialize New Segments*
- *Rolling Back a Failed Expansion Setup*

Creating an Input File for System Expansion

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

If you create the input file using the interactive interview, you may specify a file with a list of expansion hosts in the interview prompt. If your platform or command shell limits the length of the host list, specifying the hosts with -f may be mandatory.

Creating an input file in Interactive Mode

Before you run gpexpand to create an input file in interactive mode, ensure you know:

- The number of new hosts (or a hosts file)
- The new hostnames (or a hosts file)
- The mirroring strategy used in existing hosts, if any
- The number of segments to add per host, if any

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

To create an input file in interactive mode

1. Log in on the master host as the user who will run your Greenplum Database system; for example, gpadmin.
2. Run gpexpand. The utility displays messages about how to prepare for an expansion operation, and it prompts you to quit or continue.

   Optionally, specify a hosts file using -f. For example:

   ```
   $ gpexpand -f /home/gpadmin/new_hosts_file
   ```
3. At the prompt, select Y to continue.
4. Unless you specified a hosts file using -f, you are prompted to enter hostnames. Enter a comma separated list of the hostnames of the new expansion hosts. Do not include interface hostnames. For example:

   ```
   > sdw4, sdw5, sdw6, sdw7
   ```

   To add segments to existing hosts only, enter a blank line at this prompt. Do not specify localhost or any existing host name.
5. Enter the mirroring strategy used in your system, if any. Options are spread|grouped|none. The default setting is grouped.

Ensure you have enough hosts for the selected grouping strategy. For more information about mirroring, see *Planning Mirror Segments*.

6. Enter the number of new primary segments to add, if any. By default, new hosts are initialized with the same number of primary segments as existing hosts. Increase segments per host by entering a number greater than zero. The number you enter will be the number of additional segments initialized on all hosts. For example, if existing hosts currently have two segments each, entering a value of `2` initializes two more segments on existing hosts, and four segments on new hosts.
7. If you are adding new primary segments, enter the new primary data directory root for the new segments. Do not specify the actual data directory name, which is created automatically by `gpexpand` based on the existing data directory names.

   For example, if your existing data directories are as follows:

   ```
   /gpdata/primary/gp0
   /gpdata/primary/gp1
   ```

   then enter the following (one at each prompt) to specify the data directories for two new primary segments:

   ```
   /gpdata/primary
   /gpdata/primary
   ```

   When the initialization runs, the utility creates the new directories `gp2` and `gp3` under `/gpdata/primary`.
8. If you are adding new mirror segments, enter the new mirror data directory root for the new segments. Do not specify the data directory name; it is created automatically by `gpexpand` based on the existing data directory names.

   For example, if your existing data directories are as follows:

   ```
   /gpdata/mirror/gp0
   /gpdata/mirror/gp1
   ```

   enter the following (one at each prompt) to specify the data directories for two new mirror segments:

   ```
   /gpdata/mirror
   /gpdata/mirror
   ```

   When the initialization runs, the utility will create the new directories `gp2` and `gp3` under `/gpdata/mirror`.

   These primary and mirror root directories for new segments must exist on the hosts, and the user running `gpexpand` must have permissions to create directories in them.

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

   ```
   gpexpand_inputfile_yyyymmdd_145134
   ```

Expansion Input File Format

Pivotal recommends using the interactive interview process to create your own input file unless your expansion scenario has atypical needs.

The format for expansion input files is:

```
hostname:address:port:fselocation:dbid:content:preferred_role:replication_port
```

For example:

```
sdw5:sdw5-1:50011:/gpdata/primary/gp9:11:9:p:53011
```

```
sdw5:sdw5-2:50012:/gpdata/primary/gp10:12:10:p:53011
sdw5:sdw5-2:60011:/gpdata/mirror/gp9:13:9:m:63011
sdw5:sdw5-1:60012:/gpdata/mirror/gp10:14:10:m:63011
```

For each new segment, this format of expansion input file requires the following:

Table 16: Data for the expansion configuration file

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

Running gpexpand to Initialize New Segments

After you have created an input file, run `gpexpand` to initialize new segments. The utility automatically stops Greenplum Database segment initialization and restarts the system when the process finishes.

To run gpexpand with an input file

1. Log in on the master host as the user who will run your Greenplum Database system; for example, `gpadmin`.

2. Run the `gpexpand` utility, specifying the input file with `-i`. Optionally, use `-D` to specify the database in which to create the expansion schema. For example:

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

   The utility detects if an expansion schema exists for the Greenplum Database system. If a schema exists, remove it with `gpexpand -c` before you start a new expansion operation. See *Removing the Expansion Schema*.

   When the new segments are initialized and the expansion schema is created, the utility prints a success message and exits.

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

Rolling Back a Failed Expansion Setup

You can roll back an expansion setup operation only if the operation fails.

If the expansion fails during the initialization step, while the database is down, you must first restart the database in master-only mode by running the `gpstart -m` command.

Roll back the failed expansion with the following command, specifying the database that contains the expansion schema:

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

Redistributing Tables

Redistribute tables to balance existing data over the newly expanded cluster.

After creating an expansion schema, you can bring Greenplum Database back online and redistribute tables across the entire array with `gpexpand`. Target low-use hours when the utility's CPU usage and table locks have minimal impact on operations. Rank tables to redistribute the largest or most critical tables in preferential order.

> **Note:** When redistributing data, Greenplum Database must be running in production mode. Greenplum Database cannot be restricted mode or in master mode. The `gpstart` options `-R` or `-m` cannot be specified to start Greenplum Database.

While table redistribution is underway any new tables or partitions created are distributed across all segments exactly as they would be under normal operating conditions. Queries can access all segments, even before the relevant data is redistributed to tables on the new segments. The table or partition being redistributed is locked and unavailable for read or write operations. When its redistribution completes, normal operations resume.

- *Ranking Tables for Redistribution*
- *Redistributing Tables Using gpexpand*
- *Monitoring Table Redistribution*

Ranking Tables for Redistribution

For large systems, Pivotal recommends controlling table redistribution order. Adjust tables' `rank` values in the expansion schema to prioritize heavily-used tables and minimize performance impact. Available free disk space can affect table ranking; see *Managing Redistribution in Large-Scale Greenplum Systems*.

To rank tables for redistribution by updating `rank` values in *gpexpand.status_detail*, connect to Greenplum Database using `psql` or another supported client. Update *gpexpand.status_detail* with commands such as:

```
=> UPDATE gpexpand.status_detail SET rank=10;

=> UPDATE gpexpand.status_detail SET rank=1 WHERE fq_name = 'public.lineitem';
=> UPDATE gpexpand.status_detail SET rank=2 WHERE fq_name = 'public.orders';
```

These commands lower the priority of all tables to `10` and then assign a rank of `1` to `lineitem` and a rank of `2` to `orders`. When table redistribution begins, `lineitem` is redistributed first, followed by `orders` and all other tables in *gpexpand.status_detail*. To exclude a table from redistribution, remove the table from *gpexpand.status_detail*.

Redistributing Tables Using gpexpand

To redistribute tables with gpexpand

1. Log in on the master host as the user who will run your Greenplum Database system, for example, `gpadmin`.
2. Run the `gpexpand` utility. You can use the `-d` or `-e` option to define the expansion session time period. For example, to run the utility for up to 60 consecutive hours:

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

   The utility redistributes tables until the last table in the schema completes or it reaches the specified duration or end time. `gpexpand` updates the status and time in *gpexpand.status* when a session starts and finishes.

Monitoring Table Redistribution

You can query the expansion schema during the table redistribution process. The view *gpexpand.expansion_progress* provides a current progress summary, including the estimated rate of table redistribution and estimated time to completion. You can query the table *gpexpand.status_detail* for per-table status information.

Viewing Expansion Status

After the first table completes redistribution, *gpexpand.expansion_progress* calculates its estimates and refreshes them based on all tables' redistribution rates. Calculations restart each time you start a table redistribution session with `gpexpand`. To monitor progress, connect to Greenplum Database using `psql` or another supported client; query *gpexpand.expansion_progress* with a command like the following:

```
=# SELECT * FROM gpexpand.expansion_progress;
             name             |         value
------------------------------+-----------------------
 Bytes Left                   | 5534842880
 Bytes Done                   | 142475264
 Estimated Expansion Rate     | 680.75667095996092 MB/s
 Estimated Time to Completion | 00:01:01.008047
 Tables Expanded              | 4
 Tables Left                  | 4
(6 rows)
```

Viewing Table Status

The table *gpexpand.status_detail* stores status, time of last update, and more facts about each table in the schema. To see a table's status, connect to Greenplum Database using `psql` or another supported client and query *gpexpand.status_detail*:

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

Removing the Expansion Schema

To clean up after expanding the Greenplum cluster, remove the expansion schema.

You can safely remove the expansion schema after the expansion operation is complete and verified. To run another expansion operation on a Greenplum system, first remove the existing expansion schema.

To remove the expansion schema

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

```
$ gpexpand -c
$
```

Note: Some systems require you to press Enter twice.

Chapter 13

Migrating Data with gptransfer

This topic describes how to use the gptransfer utility to transfer data between databases.

The gptransfer migration utility transfers Greenplum Database metadata and data from one Greenplum database to another Greenplum database, allowing you to migrate the entire contents of a database, or just selected tables, to another database. The source and destination databases may be in the same or a different cluster. Data is transferred in parallel across all the segments, using the gpfdist data loading utility to attain the highest transfer rates.

gptransfer handles the setup and execution of the data transfer. Participating clusters must already exist, have network access between all hosts in both clusters, and have certificate-authenticated ssh access between all hosts in both clusters.

The interface includes options to transfer one or more full databases, or one or more database tables. A full database transfer includes the database schema, table data, indexes, views, roles, user-defined functions, and resource queues. Configuration files, including postgres.conf and pg_hba.conf, must be transferred manually by an administrator. Extensions installed in the database with gppkg, such as MADlib and programming language extensions, must be installed in the destination database by an administrator.

See the *Greenplum Database Utility Guide* for complete syntax and usage information for the gptransfer utility.

Prerequisites

- The gptransfer utility can only be used with Greenplum Database, including the EMC DCA appliance. Pivotal HAWQ is not supported as a source or destination.
- The source and destination Greenplum clusters must both be version 4.2 or higher.
- At least one Greenplum instance must include the gptransfer utility in its distribution. The utility is included with Greenplum Database version 4.2.8.1 and higher and 4.3.2.0 and higher. If neither the source or destination includes gptransfer, you must upgrade one of the clusters to use gptransfer.
- The gptransfer utility can be run from the cluster with the source or destination database.
- The number of *segments* in the destination cluster must be greater than or equal to the number of *hosts* in the source cluster. The number of segments in the destination may be smaller than the number of segments in the source, but the data will transfer at a slower rate.
- The segment hosts in both clusters must have network connectivity with each other.
- Every host in both clusters must be able to connect to every other host with certificate-authenticated SSH. You can use the gpssh_exkeys utility to exchange public keys between the hosts of both clusters.

What gptransfer Does

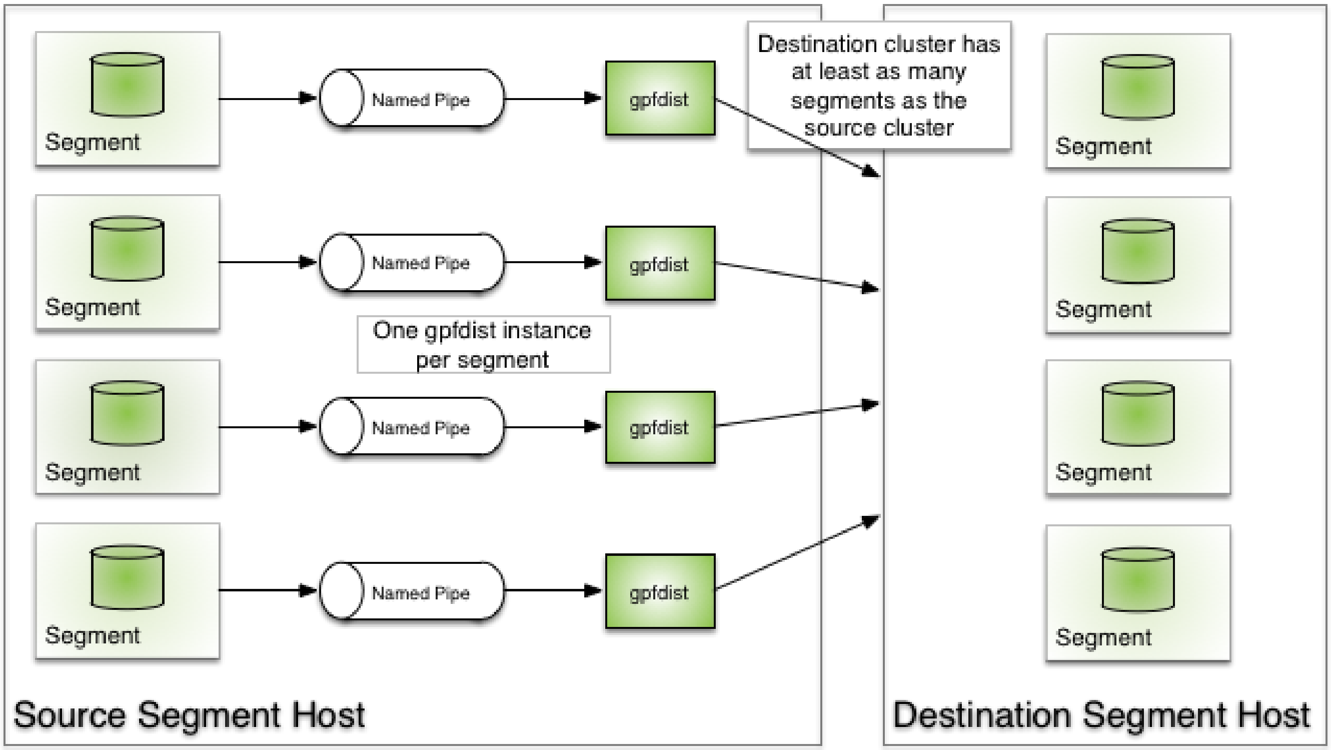
gptransfer uses writeable and readable external tables, the Greenplum gpfdist parallel data-loading utility, and named pipes to transfer data from the source database to the destination database. Segments on the source cluster select from the source database table and insert into a writeable external table. Segments in the destination cluster select from a readable external table and insert into the destination database table. The writeable and readable external tables are backed by named pipes on the source cluster's segment hosts, and each named pipe has a gpfdist process serving the pipe's output to the readable external table on the destination segments.

gptransfer orchestrates the process by processing the database objects to be transferred in batches. For each table to be transferred, it performs the following tasks:

- creates a writeable external table in the source database
- creates a readable external table in the destination database
- creates named pipes and `gpfdist` processes on segment hosts in the source cluster
- executes a `SELECT INTO` statement in the source database to insert the source data into the writeable external table
- executes a `SELECT INTO` statement in the destination database to insert the data from the readable external table into the destination table
- optionally validates the data by comparing row counts or MD5 hashes of the rows in the source and destination
- cleans up the external tables, named pipes, and `gpfdist` processes

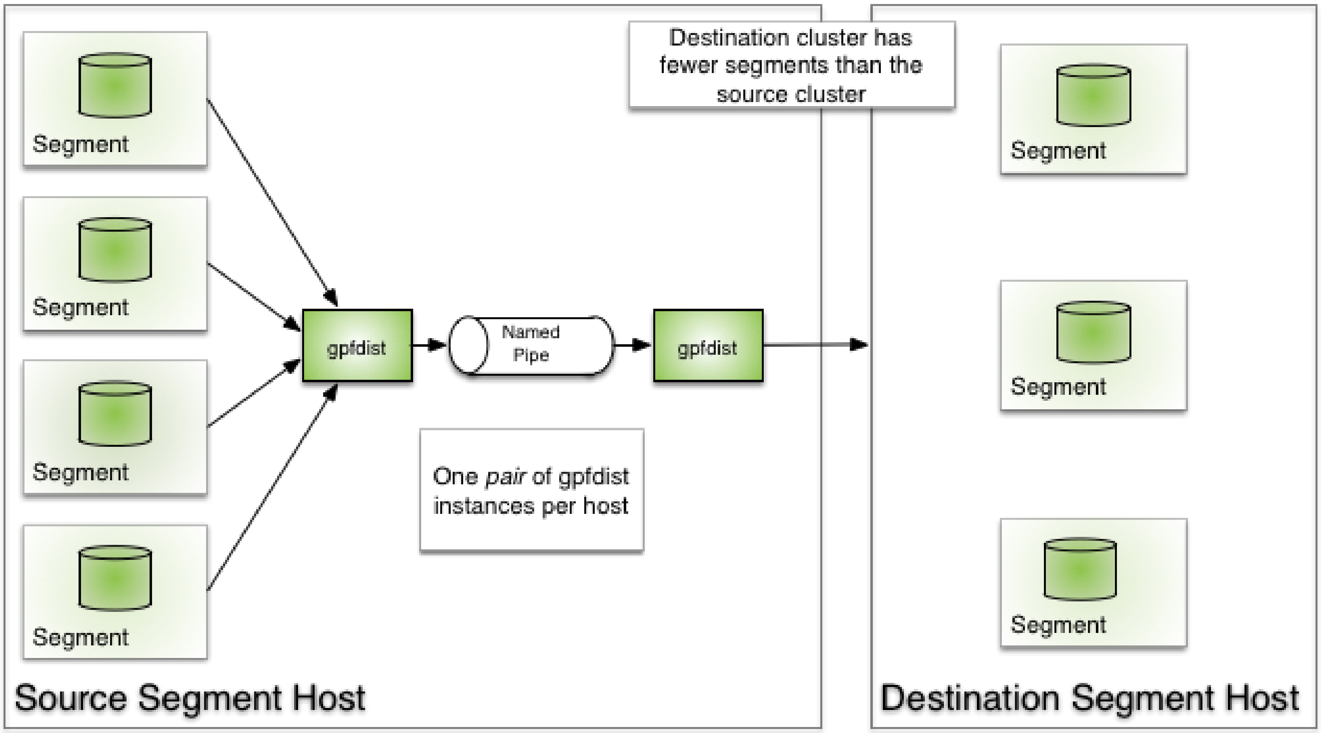
Fast Mode and Slow Mode

`gptransfer` sets up data transfer using the `gpfdist` parallel file serving utility, which serves the data evenly to the destination segments. Running more `gpfdist` processes increases the parallelism and the data transfer rate. When the destination cluster has the same or a greater number of segments than the source cluster, `gptransfer` sets up one named pipe and one `gpfdist` process for each source segment. This is the configuration for optimal data transfer rates and is called *fast mode*. The following figure illustrates a setup on a segment host when the destination cluster has at least as many segments as the source cluster.



The configuration of the input end of the named pipes differs when there are fewer segments in the destination cluster than in the source cluster. `gptransfer` handles this alternative setup automatically. The difference in configuration means that transferring data into a destination cluster with fewer segments than the source cluster is not as fast as transferring into a destination cluster of the same or greater size. It is called *slow mode* because there are fewer `gpfdist` processes serving the data to the destination cluster, although the transfer is still quite fast with one `gpfdist` per segment host.

When the destination cluster is smaller than the source cluster, there is one named pipe per segment host and all segments on the host send their data through it. The segments on the source host write their data to a writeable web external table connected to a `gpfdist` process on the input end of the named pipe. This consolidates the table data into a single named pipe. A `gpfdist` process on the output of the named pipe serves the consolidated data to the destination cluster. The following figure illustrates this configuration.



On the destination side, `gptransfer` defines a readable external table with the `gpfdist` server on the source host as input and selects from the readable external table into the destination table. The data is distributed evenly to all the segments in the destination cluster.

Batch Size and Sub-batch Size

The degree of parallelism of a `gptransfer` execution is determined by two command-line options: `--batch-size` and `--sub-batch-size`. The `--batch-size` option specifies the number of tables to transfer in a batch. The default batch size is 2, which means that two table transfers are in process at any time. The minimum batch size is 1 and the maximum is 10. The `--sub-batch-size` parameter specifies the maximum number of parallel sub-processes to start to do the work of transferring a table. The default is 25 and the maximum is 50. The product of the batch size and sub-batch size is the amount of parallelism. If set to the defaults, for example, `gptransfer` can perform 50 concurrent tasks. Each thread is a Python process and consumes memory, so setting these values too high can cause a Python Out of Memory error. For this reason, the batch sizes should be tuned for your environment.

Preparing Hosts for gptransfer

When you install a Greenplum Database cluster, you set up all the master and segment hosts so that the Greenplum Database administrative user (`gpadmin`) can connect with SSH from every host in the cluster to any other host in the cluster without providing a password. The `gptransfer` utility requires this capability between every host in the source and destination clusters. First, ensure that the clusters have network connectivity with each other. Then, prepare a hosts file containing a list of all the hosts in both clusters, and use the `gpssh-exkeys` utility to exchange keys. See the reference for `gpssh-exkeys` in the *Greenplum Database Utility Guide*.

The host map file is a text file that lists the segment hosts in the source cluster. It is used to enable communication between the hosts in Greenplum clusters. The file is specified on the `gptransfer` command line with the `--source-map-file=`*`host_map_file`* command option. It is a required option when using `gptransfer` to copy data between two separate Greenplum clusters.

The file contains a list in the following format:

```
host1_name,host1_ip_addr
host2_name,host2_ipaddr
...
```

The file uses IP addresses instead of host names to avoid any problems with name resolution between the clusters.

Limitations

`gptransfer` transfers data from user databases only; the `postgres`, `template0`, and `template1` databases cannot be transferred. Administrators must transfer configuration files manually and install extensions into the destination database with `gppkg`.

The destination cluster must have at least as many segments as the source cluster has segment hosts. Transferring data to a smaller cluster is not as fast as transferring data to a larger cluster.

Transferring small or empty tables can be unexpectedly slow. There is significant fixed overhead in setting up external tables and communications processes for parallel data loading between segments that occurs whether or not there is actual data to transfer. It can be more efficient to transfer the schema and smaller tables to the destination database using other methods, then use `gptransfer` with the `-t` option to transfer large tables.

Full Mode and Table Mode

When run with the `--full` option, `gptransfer` copies all user-created databases, tables, views, indexes, roles, user-defined functions, and resource queues in the source cluster to the destination cluster. The destination system cannot contain any user-defined databases, only the default databases postgres, template0, and template1. If `gptransfer` finds a database on the destination it fails with a message like the following:

```
[ERROR]:- gptransfer: error: --full option specified but tables exist on destination
 system
```

> **Note:** The `--full` option cannot be specified with the `-t`, `-d`, `-f`, or `--partition-transfer` options.

To copy tables individually, specify the tables using either the `-t` command-line option (one option per table) or by using the `-f` command-line option to specify a file containing a list of tables to transfer. Tables are specified in the fully-qualified format *`database`*`.`*`schema`*`.`*`table`*. The table definition, indexes, and table data are copied. The database must already exist on the destination cluster.

By default, `gptransfer` fails if you attempt to transfer a table that already exists in the destination database:

```
[INFO]:-Validating transfer table set...
[CRITICAL]:- gptransfer failed. (Reason='Table database.schema.table exists in
 database database .') exiting...
```

Override this behavior with the `--skip-existing`, `--truncate`, or `--drop` options.

The following table shows the objects that are copied in full mode and table mode.

Object	Full Mode	Table Mode
Data	Yes	Yes
Indexes	Yes	Yes
Roles	Yes	No

Object	Full Mode	Table Mode
Functions	Yes	No
Resource Queues	Yes	No
`postgres.conf`	No	No
`pg_hba.conf`	No	No
`gppkg`	No	No

The `--full` option and the `--schema-only` option can be used together if you want to copy databases in phases, for example, during scheduled periods of downtime or low activity. Run `gptransfer --full --schema-only ...` to create the databases on the destination cluster, but with no data. Then you can transfer the tables in stages during scheduled down times or periods of low activity. Be sure to include the `--truncate` or `--drop` option when you later transfer tables to prevent the transfer from failing because the table already exists at the destination.

Locking

The `-x` option enables table locking. An exclusive lock is placed on the source table until the copy and validation, if requested, are complete.

Validation

By default, `gptransfer` does not validate the data transferred. You can request validation using the `--validate=`*`type`* option. The validation *type* can be one of the following:

- `count` – Compares the row counts for the tables in the source and destination databases.
- `md5` – Sorts tables on both source and destination, and then performs a row-by-row comparison of the MD5 hashes of the sorted rows.

If the database is accessible during the transfer, be sure to add the `-x` option to lock the table. Otherwise, the table could be modified during the transfer, causing validation to fail.

Failed Transfers

A failure on a table does not end the `gptransfer` job. When a transfer fails, `gptransfer` displays an error message and adds the table name to a failed transfers file. At the end of the `gptransfer` session, `gptransfer` writes a message telling you there were failures, and providing the name of the failed transfer file. For example:

```
[WARNING]:-Some tables failed to transfer. A list of these tables
[WARNING]:-has been written to the file failed_transfer_tables_20140808_101813.txt
[WARNING]:-This file can be used with the -f option to continue
```

The failed transfers file is in the format required by the `-f` option, so you can use it to start a new `gptransfer` session to retry the failed transfers.

Best Practices

Be careful not to exceed host memory by specifying too much parallelism with the `--batch-size` and `--sub-batch-size` command line options. Too many sub-processes can exhaust memory, causing a Python Out of Memory error. Start with a smaller batch size and sub-batch size, and increase based on your experiences.

Transfer a database in stages. First, run `gptransfer` with the `--schema-only` and `-d` *`database`* options, then transfer the tables in phases. After running `gptransfer` with the `--schema-only` option, be sure to add the `--truncate` or `--drop` option to prevent a failure because a table already exists.

Be careful choosing `gpfdist` and external table parameters such as the delimiter for external table data and the maximum line length. For example, don't choose a delimiter that can appear within table data.

If you have many empty tables to transfer, consider a DDL script instead of `gptransfer`. The `gptransfer` overhead to set up each table for transfer is significant and not an efficient way to transfer empty tables.

`gptransfer` creates table indexes before transferring the data. This slows the data transfer since indexes are updated at the same time the data is inserted in the table. For large tables especially, consider dropping indexes before running `gptransfer` and recreating the indexes when the transfer is complete.

Chapter 14

Monitoring a Greenplum System

You can monitor a Greenplum Database system using a variety of tools included with the system or available as add-ons. SNMP support allows Greenplum to be integrated with popular system management frameworks.

Observing the Greenplum Database system day-to-day performance helps administrators understand the system behavior, plan workflow, and troubleshoot problems. This chapter discusses tools for monitoring database performance and activity.

Also, be sure to review *Recommended Monitoring and Maintenance Tasks* for monitoring activities you can script to quickly detect problems in the system.

Monitoring Database Activity and Performance

Pivotal provides an optional system monitoring and management tool, Greenplum Command Center, that administrators can enable within Greenplum Database 4.3.

Enabling Greenplum Command Center is a two-part process. First, enable the Greenplum Database server to collect and store system metrics. Next, install and configure the Greenplum Command Center Console, an online application used to view the system metrics collected and store them in the Command Center's dedicated Greenplum database.

The Greenplum Command Center Console ships separately from your Greenplum Database 4.3 installation. Download the Greenplum Command Center Console package from *Pivotal Network* and documentation from *Pivotal Documentation*. See the *Greenplum Database Command Center Administrator Guide* for more information on installing and using the Greenplum Command Center Console.

Monitoring System State

As a Greenplum Database administrator, you must monitor the system for problem events such as a segment going down or running out of disk space on a segment host. The following topics describe how to monitor the health of a Greenplum Database system and examine certain state information for a Greenplum Database system.

- *Enabling System Alerts and Notifications*
- *Checking System State*
- *Checking Disk Space Usage*
- *Checking for Data Distribution Skew*
- *Viewing Metadata Information about Database Objects*
- *Viewing Session Memory Usage Information*
- *Viewing Query Workfile Usage Information*

Enabling System Alerts and Notifications

You can configure a Greenplum Database system to trigger SNMP (Simple Network Management Protocol) alerts or send email notifications to system administrators if certain database events occur. These events include:

- All `PANIC`-level error conditions
- All `FATAL`-level error conditions

- `ERROR`-level conditions that are "internal errors" (for example, SIGSEGV errors)
- Database system shutdown and restart
- Segment failure and recovery
- Standby master out-of-sync conditions
- Master host manual shutdown or other software problem (in certain failure scenarios, Greenplum Database cannot send an alert or notification)

This topic includes the following sub-topics:

- *Using SNMP with a Greenplum Database System*
- *Enabling Email Notifications*

Note that SNMP alerts and email notifications report the same event information. There is no difference in the event information that either tool reports. For information about the SNMP event information, see *Greenplum Database SNMP OIDs and Error Codes*.

Using SNMP with a Greenplum Database System

Greenplum Database supports SNMP to monitor the state of a Greenplum Database system using MIBs (Management Information Bases). MIBs are collections of objects that describe an SNMP-manageable entity — in this case, a Greenplum Database system.

The Greenplum Database SNMP support allows a Network Management System to obtain information about the hardware, operating system, and Greenplum Database from the same port (161) and IP address. It also enables the auto-discovery of Greenplum Database instances.

Prerequisites

Before setting up SNMP support on Greenplum Database, ensure SNMP is installed on the master host. If the `snmpd` file is not present in the `/usr/sbin` directory, then SNMP is not installed on the system. Depending on the platform on which you are running Greenplum Database, install the following:

Table 17: SNMP Prerequisites

Operating System	Packages[1]
Red Hat Enterprise	net-snmp net-snmp-libs net-snmp-utils
CentOS	net-snmp
SUSE	N/A

[1] SNMP is installed by default on SUSE platforms.

The `snmp.conf` configuration file is located in `/etc/snmp/`.

Pre-installation Tasks

After you establish that SNMP is on the master host, log in as `root`, open a text editor, and edit the *path_to*`/snmp/snmpd.conf` file. To use SNMP with Greenplum Database, the minimum configuration change required to the `snmpd.conf` file is specifying a community name. For example:

```
rocommunity public
```

> **Note:** Replace `public` with the name of your SNMP community. You should also configure `syslocation` and `syscontact`. Configure other SNMP settings as required for your environment and save the file.

For more information about the `snmpd.conf` file, enter:

```
man snmpd.conf
```

> **Note:** On SUSE Linux platforms, make sure to review and configure security settings in the `snmp.conf` file so `snmpd` accepts connections from sub-agents and returns all available Object IDs (OIDs).

After you finish configuring the `snmpd.conf` file, start the system `snmpd` daemon:

```
# /sbin/chkconfig snmpd on
```

Then, verify the system `snmpd` daemon is running. Enter:

```
# snmpwalk -v 1 -c community_name localhost .1.3.6.1.2.1.1.1.0
```

For example:

```
# snmpwalk -v 1 -c public localhost .1.3.6.1.2.1.1.1.0
```

If this command returns "`Timeout: No Response from localhost`", then the system `snmpd` daemon is not running. If the daemon is running, output similar to the following displays:

```
SNMPv2-MIB::sysDescr.0 = STRING: Linux hostname
2.6.18-92.el5 #1 SMP Tue Jun 10 18:51:06 EDT 2010 x86_64
```

Setting up SNMP Notifications

1. To configure a Greenplum Database system to send SNMP notifications when alerts occur, set the following parameters on the Greenplum Database master host with the `gpconfig` utility:

   - `gp_snmp_community`: Set this parameter to the community name you specified for your environment.
   - `gp_snmp_monitor_address`: Enter the `hostname:port` of your network monitor application. Typically, the port number is `162`. If there are multiple monitor addresses, separate them with a comma.
   - `gp_snmp_use_inform_or_trap`: Enter either `trap` or `inform`. Trap notifications are SNMP messages sent from one application to another (for example, between Greenplum Database and a network monitoring application). These messages are unacknowledged by the monitoring application, but generate less network overhead.

     Inform notifications are the same as trap messages, except the application sends an acknowledgement to the application that generated the alert. In this case, the monitoring application sends acknowledgement messages to Greenplum Database-generated trap notifications. While inform messages create more overhead, they inform Greenplum Database the monitoring application has received the traps.

     The following example commands set the server configuration parameters with the Greenplum Database gpconfig utility:

     ```
     $ gpconfig -c gp_snmp_community -v public --masteronly
     $ gpconfig -c gp_snmp_monitor_address -v mdw:162 --masteronly
     $ gpconfig -c gp_snmp_use_inform_or_trap -v trap --masteronly
     ```

2. To test SNMP notifications, you can use the `snmptrapd` trap receiver. As root, enter:

   ```
   # /usr/sbin/snmptrapd -m ALL -Lf ~/filename.log
   ```

   `-Lf` indicates that traps are logged to a file. `-Le` indicates that traps are logged to `stderr` instead. `-m ALL` loads all available MIBs (you can also specify individual MIBs if required).

Enabling Email Notifications

Complete the following steps to enable Greenplum Database to send email notifications to system administrators whenever certain database events occur.

1. Open `$MASTER_DATA_DIRECTORY/postgresql.conf` in a text editor.
2. In the `EMAIL ALERTS` section, uncomment the following parameters and enter the appropriate values for your email server and domain. For example:

   ```
   gp_email_smtp_server='smtp.company.com:25'
   gp_email_smtp_userid='gpadmin@example.com'
   gp_email_smtp_password='mypassword'
   gp_email_from='Greenplum Database <gpadmin@example.com>'
   gp_email_to='dba@example.com;John Smith <jsmith@example.com>'
   ```

   You may create specific email accounts or groups in your email system that send and receive email alerts from the Greenplum Database system. For example:

   ```
   gp_email_from='GPDB Production Instance <gpdb@example.com>'
   gp_email_to='gpdb_dba_group@example.com'
   ```

   You can also specify multiple email addresses for both `gp_email` parameters. Use a semi-colon (;) to separate each email address. For example:

   ```
   gp_email_to='gpdb_dba_group@example.com;admin@example.com'
   ```
3. Save and close the `postgresql.conf` file.
4. Reload the Greenplum Database `postgresql.conf` file:

   ```
   $ gpstop -u
   ```

Testing Email Notifications

The Greenplum Database master host must be able to connect to the SMTP email server you specify for the gp_email_smtp_server parameter. To test connectivity, use the `ping` command:

```
$ ping my_email_server
```

If the master host can contact the SMTP server, log in to `psql` and test email notifications with the following command:

```
$ psql template1
=# SELECT gp_elog('Test GPDB Email',true); gp_elog
```

The address you specified for the gp_email_to parameter should receive an email with Test GPDB Email in the subject line.

You can also test email notifications by using a public SMTP server, such as Google's Gmail SMTP server, and an external email address. For example:

```
gp_email_smtp_server='smtp.gmail.com:25'
#gp_email_smtp_userid=''
#gp_email_smtp_password=''
gp_email_from='gpadmin@example.com'
gp_email_to='test_account@example.com'
```

Note: If you have difficulty sending and receiving email notifications, verify the security settings for you organization's email server and firewall.

Checking System State

A Greenplum Database system is comprised of multiple PostgreSQL instances (the master and segments) spanning multiple machines. To monitor a Greenplum Database system, you need to know information about the system as a whole, as well as status information of the individual instances. The `gpstate` utility provides status information about a Greenplum Database system.

Viewing Master and Segment Status and Configuration

The default `gpstate` action is to check segment instances and show a brief status of the valid and failed segments. For example, to see a quick status of your Greenplum Database system:

```
$ gpstate
```

To see more detailed information about your Greenplum Database array configuration, use `gpstate` with the `-s` option:

```
$ gpstate -s
```

Viewing Your Mirroring Configuration and Status

If you are using mirroring for data redundancy, you may want to see the list of mirror segment instances in the system, their current synchronization status, and the mirror to primary mapping. For example, to see the mirror segments in the system and their status:

```
$ gpstate -m
```

To see the primary to mirror segment mappings:

```
$ gpstate -c
```

To see the status of the standby master mirror:

```
$ gpstate -f
```

Checking Disk Space Usage

A database administrator's most important monitoring task is to make sure the file systems where the master and segment data directories reside do not grow to more than 70 percent full. A filled data disk will not result in data corruption, but it may prevent normal database activity from continuing. If the disk grows too full, it can cause the database server to shut down.

You can use the `gp_disk_free` external table in the `gp_toolkit` administrative schema to check for remaining free space (in kilobytes) on the segment host file systems. For example:

```
=# SELECT * FROM gp_toolkit.gp_disk_free
   ORDER BY dfsegment;
```

Checking Sizing of Distributed Databases and Tables

The `gp_toolkit` administrative schema contains several views that you can use to determine the disk space usage for a distributed Greenplum Database database, schema, table, or index.

For a list of the available sizing views for checking database object sizes and disk space, see the *Greenplum Database Reference Guide*.

Viewing Disk Space Usage for a Database

To see the total size of a database (in bytes), use the *gp_size_of_database* view in the *gp_toolkit* administrative schema. For example:

```
=> SELECT * FROM gp_toolkit.gp_size_of_database
   ORDER BY soddatname;
```

Viewing Disk Space Usage for a Table

The *gp_toolkit* administrative schema contains several views for checking the size of a table. The table sizing views list the table by object ID (not by name). To check the size of a table by name, you must look up the relation name (`relname`) in the *pg_class* table. For example:

```
=> SELECT relname AS name, sotdsize AS size, sotdtoastsize
   AS toast, sotdadditionalsize AS other
   FROM gp_size_of_table_disk as sotd, pg_class
   WHERE sotd.sotdoid=pg_class.oid ORDER BY relname;
```

For a list of the available table sizing views, see the *Greenplum Database Reference Guide*.

Viewing Disk Space Usage for Indexes

The *gp_toolkit* administrative schema contains a number of views for checking index sizes. To see the total size of all index(es) on a table, use the *gp_size_of_all_table_indexes* view. To see the size of a particular index, use the *gp_size_of_index* view. The index sizing views list tables and indexes by object ID (not by name). To check the size of an index by name, you must look up the relation name (`relname`) in the *pg_class* table. For example:

```
=> SELECT soisize, relname as indexname
   FROM pg_class, gp_size_of_index
   WHERE pg_class.oid=gp_size_of_index.soioid
   AND pg_class.relkind='i';
```

Checking for Data Distribution Skew

All tables in Greenplum Database are distributed, meaning their data is divided evenly across all of the segments in the system. Unevenly distributed data may diminish query processing performance. A table's distribution policy is determined at table creation time. For information about choosing the table distribution policy, see the following topics:

- *Viewing a Table's Distribution Key*
- *Viewing Data Distribution*
- *Checking for Query Processing Skew*

The *gp_toolkit* administrative schema also contains a number of views for checking data distribution skew on a table. For information about how to check for uneven data distribution, see the *Greenplum Database Reference Guide*.

Viewing a Table's Distribution Key

To see the columns used as the data distribution key for a table, you can use the `\d+` meta-command in `psql` to examine the definition of a table. For example:

```
=# \d+ sales
                Table "retail.sales"
 Column      |     Type     | Modifiers | Description
------------+--------------+-----------+-------------
 sale_id     | integer      |           |
 amt         | float        |           |
 date         | date          |            |
Has OIDs: no
Distributed by: (sale_id)
```

Viewing Data Distribution

To see the data distribution of a table's rows (the number of rows on each segment), you can run a query such as:

```
=# SELECT gp_segment_id, count(*)
   FROM table_name GROUP BY gp_segment_id;
```

A table is considered to have a balanced distribution if all segments have roughly the same number of rows.

Checking for Query Processing Skew

When a query is being processed, all segments should have equal workloads to ensure the best possible performance. If you identify a poorly-performing query, you may need to investigate further using the `EXPLAIN` command. For information about using the `EXPLAIN` command and query profiling, see *Query Profiling*.

Query processing workload can be skewed if the table's data distribution policy and the query predicates are not well matched. To check for processing skew, you can run a query such as:

```
=# SELECT gp_segment_id, count(*) FROM table_name
   WHERE column='value' GROUP BY gp_segment_id;
```

This will show the number of rows returned by segment for the given `WHERE` predicate.

Avoiding an Extreme Skew Warning

You may receive the following warning message while executing a query that performs a hash join operation:

```
Extreme skew in the innerside of Hashjoin
```

This occurs when the input to a hash join operator is skewed. It does not prevent the query from completing successfully. You can follow these steps to avoid skew in the plan:

1. Ensure that all fact tables are analyzed.
2. Verify that any populated temporary table used by the query is analyzed.
3. View the `EXPLAIN ANALYZE` plan for the query and look for the following:
    - If there are scans with multi-column filters that are producing more rows than estimated, then set the `gp_selectivity_damping_factor` server configuration parameter to 2 or higher and retest the query.
    - If the skew occurs while joining a single fact table that is relatively small (less than 5000 rows), set the `gp_segments_for_planner` server configuration parameter to 1 and retest the query.
4. Check whether the filters applied in the query match distribution keys of the base tables. If the filters and distribution keys are the same, consider redistributing some of the base tables with different distribution keys.
5. Check the cardinality of the join keys. If they have low cardinality, try to rewrite the query with different joining columns or or additional filters on the tables to reduce the number of rows. These changes could change the query semantics.

Viewing Metadata Information about Database Objects

Greenplum Database tracks various metadata information in its system catalogs about the objects stored in a database, such as tables, views, indexes and so on, as well as global objects such as roles and tablespaces.

Viewing the Last Operation Performed

You can use the system views *pg_stat_operations* and *pg_stat_partition_operations* to look up actions performed on an object, such as a table. For example, to see the actions performed on a table, such as when it was created and when it was last vacuumed and analyzed:

```
=> SELECT schemaname as schema, objname as table,
   usename as role, actionname as action,
   subtype as type, statime as time
   FROM pg_stat_operations
   WHERE objname='cust';
 schema | table | role | action  | type  | time
--------+-------+------+---------+-------+-------------------------
  sales | cust  | main | CREATE  | TABLE | 2010-02-09 18:10:07.867977-08
  sales | cust  | main | VACUUM  |       | 2010-02-10 13:32:39.068219-08
  sales | cust  | main | ANALYZE |       | 2010-02-25 16:07:01.157168-08
(3 rows)
```

Viewing the Definition of an Object

To see the definition of an object, such as a table or view, you can use the `\d+` meta-command when working in `psql`. For example, to see the definition of a table:

```
=> \d+ mytable
```

Viewing Session Memory Usage Information

You can create and use the *session_level_memory_consumption* view that provides information about the current memory utilization for sessions that are running queries on Greenplum Database. The view contains session information and information such as the database that the session is connected to, the query that the session is currently running, and memory consumed by the session processes.

Creating the session_level_memory_consumption View

To create the *session_level_memory_consumption* view in a Greenplum Database, run the script `$GPHOME/share/postgresql/contrib/gp_session_state.sql` once for each database. For example, to install the view in the database `testdb`, use this command:

```
$ psql -d testdb -f $GPHOME/share/postgresql/contrib/gp_session_state.sql
```

The session_level_memory_consumption View

The *session_level_memory_consumption* view provides information about memory consumption for sessions that are running SQL queries.

In the view, the column `is_runaway` indicates whether Greenplum Database considers the session a runaway session based on the vmem memory consumption of the session's queries. When the queries consume an excessive amount of memory, Greenplum Database considers the session a runaway. The Greenplum Database server configuration parameter `runaway_detector_activation_percent` controlling when Greenplum Database considers a session a runaway session.

For information about the parameter, see "Server Configuration Parameters" in the *Greenplum Database Reference Guide*.

Table 18: session_level_memory_consumption

column	type	references	description
`datname`	name		Name of the database that the session is connected to.
`sess_id`	integer		Session ID.
`usename`	name		Name of the session user.
`current_query`	text		Current SQL query that the session is running.
`segid`	integer		Segment ID.
`vmem_mb`	integer		Total vmem memory usage for the session in MB.
`is_runaway`	boolean		Session is marked as runaway on the segment.
`qe_count`	integer		Number of query processes for the session.
`active_qe_count`	integer		Number of active query processes for the session.
`dirty_qe_count`	integer		Number of query processes that have not yet released their memory. The value is `-1` for sessions that are not running.
`runaway_vmem_mb`	integer		Amount of vmem memory that the session was consuming when it was marked as a runaway session.
`runaway_command_cnt`	integer		Command count for the session when it was marked as a runaway session.

Viewing Query Workfile Usage Information

The Greenplum Database administrative schema *gp_toolkit* contains views that display information about Greenplum Database workfiles. Greenplum Database creates workfiles on disk if it does not have sufficient memory to execute the query in memory. This information can be used for troubleshooting and tuning queries. The information in the views can also be used to specify the values for the Greenplum Database configuration parameters `gp_workfile_limit_per_query` and `gp_workfile_limit_per_segment`.

These are the views in the schema *gp_toolkit*:

- The gp_workfile_entries view contains one row for each operator using disk space for workfiles on a segment at the current time.
- The gp_workfile_usage_per_query view contains one row for each query using disk space for workfiles on a segment at the current time.
- The gp_workfile_usage_per_segment view contains one row for each segment. Each row displays the total amount of disk space used for workfiles on the segment at the current time.

For information about using *gp_toolkit*, see *Using gp_toolkit*.

Viewing the Database Server Log Files

Every database instance in Greenplum Database (master and segments) runs a PostgreSQL database server with its own server log file. Daily log files are created in the `pg_log` directory of the master and each segment data directory.

Log File Format

The server log files are written in comma-separated values (CSV) format. Some log entries will not have values for all log fields. For example, only log entries associated with a query worker process will have the `slice_id` populated. You can identify related log entries of a particular query by the query's session identifier (`gp_session_id`) and command identifier (`gp_command_count`).

The following fields are written to the log:

Table 19: Greenplum Database Server Log Format

#	Field Name	Data Type	Description
1	event_time	timestamp with time zone	Time that the log entry was written to the log
2	user_name	varchar(100)	The database user name
3	database_name	varchar(100)	The database name
4	process_id	varchar(10)	The system process ID (prefixed with "p")
5	thread_id	varchar(50)	The thread count (prefixed with "th")
6	remote_host	varchar(100)	On the master, the hostname/address of the client machine. On the segment, the hostname/address of the master.
7	remote_port	varchar(10)	The segment or master port number
8	session_start_time	timestamp with time zone	Time session connection was opened
9	transaction_id	int	Top-level transaction ID on the master. This ID is the parent of any subtransactions.
10	gp_session_id	text	Session identifier number (prefixed with "con")
11	gp_command_ count	text	The command number within a session (prefixed with "cmd")
12	gp_segment	text	The segment content identifier (prefixed with "seg" for primaries or "mir" for mirrors). The master always has a content ID of -1.

#	Field Name	Data Type	Description
13	slice_id	text	The slice ID (portion of the query plan being executed)
14	distr_tranx_id	text	Distributed transaction ID
15	local_tranx_id	text	Local transaction ID
16	sub_tranx_id	text	Subtransaction ID
17	event_severity	varchar(10)	Values include: LOG, ERROR, FATAL, PANIC, DEBUG1, DEBUG2
18	sql_state_code	varchar(10)	SQL state code associated with the log message
19	event_message	text	Log or error message text
20	event_detail	text	Detail message text associated with an error or warning message
21	event_hint	text	Hint message text associated with an error or warning message
22	internal_query	text	The internally-generated query text
23	internal_query_pos	int	The cursor index into the internally-generated query text
24	event_context	text	The context in which this message gets generated
25	debug_query_string	text	User-supplied query string with full detail for debugging. This string can be modified for internal use.
26	error_cursor_pos	int	The cursor index into the query string
27	func_name	text	The function in which this message is generated
28	file_name	text	The internal code file where the message originated
29	file_line	int	The line of the code file where the message originated
30	stack_trace	text	Stack trace text associated with this message

Searching the Greenplum Server Log Files

Greenplum Database provides a utility called gplogfilter can search through a Greenplum Database log file for entries matching the specified criteria. By default, this utility searches through the Greenplum Database master log file in the default logging location. For example, to display the last three lines of the master log file:

```
$ gplogfilter -n 3
```

To search through all segment log files simultaneously, run gplogfilter through the gpssh utility. For example, to display the last three lines of each segment log file:

```
$ gpssh -f seg_host_file
```

```
=> source /usr/local/greenplum-db/greenplum_path.sh
=> gplogfilter -n 3 /gpdata/gp*/pg_log/gpdb*.log
```

Using gp_toolkit

Use the Greenplum Database administrative schema *gp_toolkit* to query the system catalogs, log files, and operating environment for system status information. The `gp_toolkit` schema contains several views you can access using SQL commands. The *gp_toolkit* schema is accessible to all database users. Some objects require superuser permissions. Use a command similar to the following to add the *gp_toolkit* schema to your schema search path:

```
=> ALTER ROLE myrole SET search_path TO myschema,gp_toolkit;
```

For a description of the available administrative schema views and their usages, see the *Greenplum Database Reference Guide*.

Greenplum Database SNMP OIDs and Error Codes

When a Greenplum Database system is configured to trigger SNMP alerts or send email notifications to system administrators if certain database events occur, the alerts and notifications contain Object IDs (OIDs) and SQL error codes.

- *Greenplum Database SNMP OIDs*
- *SQL Standard Error Codes*

For information about enabling Greenplum Database to use SNMP, see *Enabling System Alerts and Notifications*

Greenplum Database SNMP OIDs

This is the Greenplum Database OID hierarchy structure:

```
iso(1)
identified-organization(3)
dod(6)
internet(1)
private(4)
enterprises(1)
gpdbMIB(31327)
gpdbObjects(1)
gpdbAlertMsg(1)
```

gpdbAlertMsg

```
1.3.6.1.4.1.31327.1.1: STRING: alert message text
```

gpdbAlertSeverity

```
1.3.6.1.4.1.31327.1.2: INTEGER: severity level
```

gpdbAlertSeverity can have one of the following values:

```
gpdbSevUnknown(0)
gpdbSevOk(1)
gpdbSevWarning(2)
gpdbSevError(3)
gpdbSevFatal(4)
gpdbSevPanic(5)
gpdbSevSystemDegraded(6)
gpdbSevSystemDown(7)
```

gpdbAlertSqlstate

```
1.3.6.1.4.1.31327.1.3: STRING: SQL standard error codes
```

For a list of codes, see SQL Standard Error Codes.

gpdbAlertDetail

```
1.3.6.1.4.1.31327.1.4: STRING: detailed alert message text
```

gpdbAlertSqlStmt

```
1.3.6.1.4.1.31327.1.5: STRING: SQL statement generating this alert if
 applicable
```

gpdbAlertSystemName

```
1.3.6.1.4.1.31327.1.6: STRING: hostname
```

SQL Standard Error Codes

The following table lists all the defined error codes. Some are not used, but are defined by the SQL standard. The error classes are also shown. For each error class there is a standard error code having the last three characters 000. This code is used only for error conditions that fall within the class but do not have any more-specific code assigned.

The PL/pgSQL condition name for each error code is the same as the phrase shown in the table, with underscores substituted for spaces. For example, code 22012, DIVISION BY ZERO, has condition name DIVISION_BY_ZERO. Condition names can be written in either upper or lower case.

> **Note:** PL/pgSQL does not recognize warning, as opposed to error, condition names; those are classes 00, 01, and 02.

Table 20: SQL Codes

Error Code	Meaning	Constant
Class 00 — Successful Completion		
00000	SUCCESSFUL COMPLETION	successful_completion
Class 01 — Warning		
01000	WARNING	warning
0100C	DYNAMIC RESULT SETS RETURNED	dynamic_result_sets_returned
01008	IMPLICIT ZERO BIT PADDING	implicit_zero_bit_padding
01003	NULL VALUE ELIMINATED IN SET FUNCTION	null_value_eliminated_in_set_function
01007	PRIVILEGE NOT GRANTED	privilege_not_granted
01006	PRIVILEGE NOT REVOKED	privilege_not_revoked
01004	STRING DATA RIGHT TRUNCATION	string_data_right_truncation
01P01	DEPRECATED FEATURE	deprecated_feature
Class 02 — No Data (this is also a warning class per the SQL standard)		
02000	NO DATA	no_data

Error Code	Meaning	Constant
02001	NO ADDITIONAL DYNAMIC RESULT SETS RETURNED	no_additional_dynamic_result_sets_returned
Class 03 — SQL Statement Not Yet Complete		
03000	SQL STATEMENT NOT YET COMPLETE	sql_statement_not_yet_complete
Class 08 — Connection Exception		
08000	CONNECTION EXCEPTION	connection_exception
08003	CONNECTION DOES NOT EXIST	connection_does_not_exist
08006	CONNECTION FAILURE	connection_failure
08001	SQLCLIENT UNABLE TO ESTABLISH SQLCONNECTION	sqlclient_unable_to_establish_sqlconnection
08004	SQLSERVER REJECTED ESTABLISHMENT OF SQLCONNECTION	sqlserver_rejected_establishment_of_sqlconnection
08007	TRANSACTION RESOLUTION UNKNOWN	transaction_resolution_unknown
08P01	PROTOCOL VIOLATION	protocol_violation
Class 09 — Triggered Action Exception		
09000	TRIGGERED ACTION EXCEPTION	triggered_action_exception
Class 0A — Feature Not Supported		
0A000	FEATURE NOT SUPPORTED	feature_not_supported
Class 0B — Invalid Transaction Initiation		
0B000	INVALID TRANSACTION INITIATION	invalid_transaction_initiation
Class 0F — Locator Exception		
0F000	LOCATOR EXCEPTION	locator_exception
0F001	INVALID LOCATOR SPECIFICATION	invalid_locator_specification
Class 0L — Invalid Grantor		
0L000	INVALID GRANTOR	invalid_grantor
0LP01	INVALID GRANT OPERATION	invalid_grant_operation
Class 0P — Invalid Role Specification		
0P000	INVALID ROLE SPECIFICATION	invalid_role_specification
Class 21 — Cardinality Violation		
21000	CARDINALITY VIOLATION	cardinality_violation
Class 22 — Data Exception		
22000	DATA EXCEPTION	data_exception
2202E	ARRAY SUBSCRIPT ERROR	array_subscript_error

Error Code	Meaning	Constant
22021	CHARACTER NOT IN REPERTOIRE	character_not_in_repertoire
22008	DATETIME FIELD OVERFLOW	datetime_field_overflow
22012	DIVISION BY ZERO	division_by_zero
22005	ERROR IN ASSIGNMENT	error_in_assignment
2200B	ESCAPE CHARACTER CONFLICT	escape_character_conflict
22022	INDICATOR OVERFLOW	indicator_overflow
22015	INTERVAL FIELD OVERFLOW	interval_field_overflow
2201E	INVALID ARGUMENT FOR LOGARITHM	invalid_argument_for_logarithm
2201F	INVALID ARGUMENT FOR POWER FUNCTION	invalid_argument_for_power_function
2201G	INVALID ARGUMENT FOR WIDTH BUCKET FUNCTION	invalid_argument_for_width_bucket_function
22018	INVALID CHARACTER VALUE FOR CAST	invalid_character_value_for_cast
22007	INVALID DATETIME FORMAT	invalid_datetime_format
22019	INVALID ESCAPE CHARACTER	invalid_escape_character
2200D	INVALID ESCAPE OCTET	invalid_escape_octet
22025	INVALID ESCAPE SEQUENCE	invalid_escape_sequence
22P06	NONSTANDARD USE OF ESCAPE CHARACTER	nonstandard_use_of_escape_character
22010	INVALID INDICATOR PARAMETER VALUE	invalid_indicator_parameter_value
22020	INVALID LIMIT VALUE	invalid_limit_value
22023	INVALID PARAMETER VALUE	invalid_parameter_value
2201B	INVALID REGULAR EXPRESSION	invalid_regular_expression
22009	INVALID TIME ZONE DISPLACEMENT VALUE	invalid_time_zone_displacement_value
2200C	INVALID USE OF ESCAPE CHARACTER	invalid_use_of_escape_character
2200G	MOST SPECIFIC TYPE MISMATCH	most_specific_type_mismatch
22004	NULL VALUE NOT ALLOWED	null_value_not_allowed
22002	NULL VALUE NO INDICATOR PARAMETER	null_value_no_indicator_parameter
22003	NUMERIC VALUE OUT OF RANGE	numeric_value_out_of_range
22026	STRING DATA LENGTH MISMATCH	string_data_length_mismatch
22001	STRING DATA RIGHT TRUNCATION	string_data_right_truncation
22011	SUBSTRING ERROR	substring_error

Error Code	Meaning	Constant
22027	TRIM ERROR	trim_error
22024	UNTERMINATED C STRING	unterminated_c_string
2200F	ZERO LENGTH CHARACTER STRING	zero_length_character_string
22P01	FLOATING POINT EXCEPTION	floating_point_exception
22P02	INVALID TEXT REPRESENTATION	invalid_text_representation
22P03	INVALID BINARY REPRESENTATION	invalid_binary_representation
22P04	BAD COPY FILE FORMAT	bad_copy_file_format
22P05	UNTRANSLATABLE CHARACTER	untranslatable_character
Class 23 — Integrity Constraint Violation		
23000	INTEGRITY CONSTRAINT VIOLATION	integrity_constraint_violation
23001	RESTRICT VIOLATION	restrict_violation
23502	NOT NULL VIOLATION	not_null_violation
23503	FOREIGN KEY VIOLATION	foreign_key_violation
23505	UNIQUE VIOLATION	unique_violation
23514	CHECK VIOLATION	check_violation
Class 24 — Invalid Cursor State		
24000	INVALID CURSOR STATE	invalid_cursor_state
Class 25 — Invalid Transaction State		
25000	INVALID TRANSACTION STATE	invalid_transaction_state
25001	ACTIVE SQL TRANSACTION	active_sql_transaction
25002	BRANCH TRANSACTION ALREADY ACTIVE	branch_transaction_already_active
25008	HELD CURSOR REQUIRES SAME ISOLATION LEVEL	held_cursor_requires_same_isolation_level
25003	INAPPROPRIATE ACCESS MODE FOR BRANCH TRANSACTION	inappropriate_access_mode_for_branch_transaction
25004	INAPPROPRIATE ISOLATION LEVEL FOR BRANCH TRANSACTION	inappropriate_isolation_level_for_branch_transaction
25005	NO ACTIVE SQL TRANSACTION FOR BRANCH TRANSACTION	no_active_sql_transaction_for_branch_transaction
25006	READ ONLY SQL TRANSACTION	read_only_sql_transaction
25007	SCHEMA AND DATA STATEMENT MIXING NOT SUPPORTED	schema_and_data_statement_mixing_not_supported
25P01	NO ACTIVE SQL TRANSACTION	no_active_sql_transaction
25P02	IN FAILED SQL TRANSACTION	in_failed_sql_transaction
Class 26 — Invalid SQL Statement Name		

Error Code	Meaning	Constant
26000	INVALID SQL STATEMENT NAME	invalid_sql_statement_name
Class 27 — Triggered Data Change Violation		
27000	TRIGGERED DATA CHANGE VIOLATION	triggered_data_change_violation
Class 28 — Invalid Authorization Specification		
28000	INVALID AUTHORIZATION SPECIFICATION	invalid_authorization_specification
Class 2B — Dependent Privilege Descriptors Still Exist		
2B000	DEPENDENT PRIVILEGE DESCRIPTORS STILL EXIST	dependent_privilege_descriptors_still_exist
2BP01	DEPENDENT OBJECTS STILL EXIST	dependent_objects_still_exist
Class 2D — Invalid Transaction Termination		
2D000	INVALID TRANSACTION TERMINATION	invalid_transaction_termination
Class 2F — SQL Routine Exception		
2F000	SQL ROUTINE EXCEPTION	sql_routine_exception
2F005	FUNCTION EXECUTED NO RETURN STATEMENT	function_executed_no_return_statement
2F002	MODIFYING SQL DATA NOT PERMITTED	modifying_sql_data_not_permitted
2F003	PROHIBITED SQL STATEMENT ATTEMPTED	prohibited_sql_statement_attempted
2F004	READING SQL DATA NOT PERMITTED	reading_sql_data_not_permitted
Class 34 — Invalid Cursor Name		
34000	INVALID CURSOR NAME	invalid_cursor_name
Class 38 — External Routine Exception		
38000	EXTERNAL ROUTINE EXCEPTION	external_routine_exception
38001	CONTAINING SQL NOT PERMITTED	containing_sql_not_permitted
38002	MODIFYING SQL DATA NOT PERMITTED	modifying_sql_data_not_permitted
38003	PROHIBITED SQL STATEMENT ATTEMPTED	prohibited_sql_statement_attempted
38004	READING SQL DATA NOT PERMITTED	reading_sql_data_not_permitted
Class 39 — External Routine Invocation Exception		
39000	EXTERNAL ROUTINE INVOCATION EXCEPTION	external_routine_invocation_exception
39001	INVALID SQLSTATE RETURNED	invalid_sqlstate_returned

Error Code	Meaning	Constant
39004	NULL VALUE NOT ALLOWED	null_value_not_allowed
39P01	TRIGGER PROTOCOL VIOLATED	trigger_protocol_violated
39P02	SRF PROTOCOL VIOLATED	srf_protocol_violated
Class 3B — Savepoint Exception		
3B000	SAVEPOINT EXCEPTION	savepoint_exception
3B001	INVALID SAVEPOINT SPECIFICATION	invalid_savepoint_specification
Class 3D — Invalid Catalog Name		
3D000	INVALID CATALOG NAME	invalid_catalog_name
Class 3F — Invalid Schema Name		
3F000	INVALID SCHEMA NAME	invalid_schema_name
Class 40 — Transaction Rollback		
40000	TRANSACTION ROLLBACK	transaction_rollback
40002	TRANSACTION INTEGRITY CONSTRAINT VIOLATION	transaction_integrity_constraint_violation
40001	SERIALIZATION FAILURE	serialization_failure
40003	STATEMENT COMPLETION UNKNOWN	statement_completion_unknown
40P01	DEADLOCK DETECTED	deadlock_detected
Class 42 — Syntax Error or Access Rule Violation		
42000	SYNTAX ERROR OR ACCESS RULE VIOLATION	syntax_error_or_access_rule_violation
42601	SYNTAX ERROR	syntax_error
42501	INSUFFICIENT PRIVILEGE	insufficient_privilege
42846	CANNOT COERCE	cannot_coerce
42803	GROUPING ERROR	grouping_error
42830	INVALID FOREIGN KEY	invalid_foreign_key
42602	INVALID NAME	invalid_name
42622	NAME TOO LONG	name_too_long
42939	RESERVED NAME	reserved_name
42804	DATATYPE MISMATCH	datatype_mismatch
42P18	INDETERMINATE DATATYPE	indeterminate_datatype
42809	WRONG OBJECT TYPE	wrong_object_type
42703	UNDEFINED COLUMN	undefined_column
42883	UNDEFINED FUNCTION	undefined_function
42P01	UNDEFINED TABLE	undefined_table

Error Code	Meaning	Constant
42P02	UNDEFINED PARAMETER	undefined_parameter
42704	UNDEFINED OBJECT	undefined_object
42701	DUPLICATE COLUMN	duplicate_column
42P03	DUPLICATE CURSOR	duplicate_cursor
42P04	DUPLICATE DATABASE	duplicate_database
42723	DUPLICATE FUNCTION	duplicate_function
42P05	DUPLICATE PREPARED STATEMENT	duplicate_prepared_statement
42P06	DUPLICATE SCHEMA	duplicate_schema
42P07	DUPLICATE TABLE	duplicate_table
42712	DUPLICATE ALIAS	duplicate_alias
42710	DUPLICATE OBJECT	duplicate_object
42702	AMBIGUOUS COLUMN	ambiguous_column
42725	AMBIGUOUS FUNCTION	ambiguous_function
42P08	AMBIGUOUS PARAMETER	ambiguous_parameter
42P09	AMBIGUOUS ALIAS	ambiguous_alias
42P10	INVALID COLUMN REFERENCE	invalid_column_reference
42611	INVALID COLUMN DEFINITION	invalid_column_definition
42P11	INVALID CURSOR DEFINITION	invalid_cursor_definition
42P12	INVALID DATABASE DEFINITION	invalid_database_definition
42P13	INVALID FUNCTION DEFINITION	invalid_function_definition
42P14	INVALID PREPARED STATEMENT DEFINITION	invalid_prepared_statement_definition
42P15	INVALID SCHEMA DEFINITION	invalid_schema_definition
42P16	INVALID TABLE DEFINITION	invalid_table_definition
42P17	INVALID OBJECT DEFINITION	invalid_object_definition
Class 44 — WITH CHECK OPTION Violation		
44000	WITH CHECK OPTION VIOLATION	with_check_option_violation
Class 53 — Insufficient Resources		
53000	INSUFFICIENT RESOURCES	insufficient_resources
53100	DISK FULL	disk_full
53200	OUT OF MEMORY	out_of_memory
53300	TOO MANY CONNECTIONS	too_many_connections
Class 54 — Program Limit Exceeded		
54000	PROGRAM LIMIT EXCEEDED	program_limit_exceeded
54001	STATEMENT TOO COMPLEX	statement_too_complex

Error Code	Meaning	Constant
54011	TOO MANY COLUMNS	too_many_columns
54023	TOO MANY ARGUMENTS	too_many_arguments
Class 55 — Object Not In Prerequisite State		
55000	OBJECT NOT IN PREREQUISITE STATE	object_not_in_prerequisite_state
55006	OBJECT IN USE	object_in_use
55P02	CANT CHANGE RUNTIME PARAM	cant_change_runtime_param
55P03	LOCK NOT AVAILABLE	lock_not_available
Class 57 — Operator Intervention		
57000	OPERATOR INTERVENTION	operator_intervention
57014	QUERY CANCELED	query_canceled
57P01	ADMIN SHUTDOWN	admin_shutdown
57P02	CRASH SHUTDOWN	crash_shutdown
57P03	CANNOT CONNECT NOW	cannot_connect_now
Class 58 — System Error (errors external to Greenplum Database)		
58030	IO ERROR	io_error
58P01	UNDEFINED FILE	undefined_file
58P02	DUPLICATE FILE	duplicate_file
Class F0 — Configuration File Error		
F0000	CONFIG FILE ERROR	config_file_error
F0001	LOCK FILE EXISTS	lock_file_exists
Class P0 — PL/pgSQL Error		
P0000	PLPGSQL ERROR	plpgsql_error
P0001	RAISE EXCEPTION	raise_exception
P0002	NO DATA FOUND	no_data_found
P0003	TOO MANY ROWS	too_many_rows
Class XX — Internal Error		
XX000	INTERNAL ERROR	internal_error
XX001	DATA CORRUPTED	data_corrupted
XX002	INDEX CORRUPTED	index_corrupted

Chapter 15

Routine System Maintenance Tasks

To keep a Greenplum Database system running efficiently, the database must be regularly cleared of expired data and the table statistics must be updated so that the query optimizer has accurate information.

Greenplum Database requires that certain tasks be performed regularly to achieve optimal performance. The tasks discussed here are required, but database administrators can automate them using standard UNIX tools such as `cron` scripts. An administrator sets up the appropriate scripts and checks that they execute successfully. See *Recommended Monitoring and Maintenance Tasks* for additional suggested maintenance activities you can implement to keep your Greenplum system running optimally.

Routine Vacuum and Analyze

The design of the MVCC transaction concurrency model used in Greenplum Database means that deleted or updated data rows still occupy physical space on disk even though they are not visible to new transactions. If your database has many updates and deletes, many expired rows exist and the space they use must be reclaimed with the `VACUUM` command. The `VACUUM` command also collects table-level statistics, such as numbers of rows and pages, so it is also necessary to vacuum append-optimized tables, even when there is no space to reclaim from updated or deleted rows.

Vacuuming an append-optimized table follows a different process than vacuuming heap tables. On each segment, a new segment file is created and visible rows are copied into it from the current segment. When the segment file has been copied, the original is scheduled to be dropped and the new segment file is made available. This requires sufficient available disk space for a copy of the visible rows until the original segment file is dropped.

If the ratio of hidden rows to total rows in a segment file is less than a threshold value (10, by default), the segment file is not compacted. The threshold value can be configured with the `gp_appendonly_compaction_threshold` server configuration parameter. `VACUUM FULL` ignores the value of `gp_appendonly_compaction_threshold` and rewrites the segment file regardless of the ratio.

You can use the `__gp_aovisimap_compaction_info()` function in the the *gp_toolkit* schema to investigate the effectiveness of a `VACUUM` operation on append-optimized tables.

For information about the `__gp_aovisimap_compaction_info()` function see, "Checking Append-Optimized Tables" in the *Greenplum Database Reference Guide*.

`VACUUM` can be disabled for append-optimized tables using the `gp_appendonly_compaction` server configuration parameter.

For details about vacuuming a database, see *Vacuuming the Database*.

For information about the `gp_appendonly_compaction_threshold` server configuration parameter and the `VACUUM` command, see the *Greenplum Database Reference Guide*.

Transaction ID Management

Greenplum's MVCC transaction semantics depend on comparing transaction ID (XID) numbers to determine visibility to other transactions. Transaction ID numbers are compared using modulo 2^{32} arithmetic, so a Greenplum system that runs more than about two billion transactions can experience transaction ID wraparound, where past transactions appear to be in the future. This means past transactions' outputs become invisible. Therefore, it is necessary to `VACUUM` every table in every database at least once per two billion transactions.

> **Important:** Greenplum Database monitors transaction IDs. If you do not vacuum the database regularly, Greenplum Database will generate a warning and error.

Greenplum Database issues the following warning when a significant portion of the transaction IDs are no longer available and before transaction ID wraparound occurs:

```
WARNING: database "database_name" must be vacuumed within number_of_transactions
transactions
```

When the warning is issued, a `VACUUM` operation is required. If a `VACUUM` operation is not performed, Greenplum Database stops creating transactions when it reaches a limit prior to when transaction ID wraparound occurs. Greenplum Database issues this error when it stops creating transactions to avoid possible data loss:

```
FATAL: database is not accepting commands to avoid
wraparound data loss in database "database_name"
```

The Greenplum Database configuration parameter `xid_warn_limit` controls when the warning is displayed. The parameter `xid_stop_limit` controls when Greenplum Database stops creating transactions.

Recovering from a Transaction ID Limit Error

When Greenplum Database reaches the `xid_stop_limit` transaction ID limit due to infrequent `VACUUM` maintenance, it becomes unresponsive. To recover from this situation, perform the following steps as database administrator:

1. Shut down Greenplum Database.
2. Temporarily lower the `xid_stop_limit` by 10,000,000.
3. Start Greenplum Database.
4. Run `VACUUM FREEZE` on all affected databases.
5. Reset the `xid_stop_limit` to its original value.
6. Restart Greenplum Database.

For information about the configuration parameters, see the *Greenplum Database Reference Guide*.

For information about transaction ID wraparound see the *PostgreSQL documentation*.

System Catalog Maintenance

Numerous database updates with `CREATE` and `DROP` commands increase the system catalog size and affect system performance. For example, running many `DROP TABLE` statements degrades the overall system performance due to excessive data scanning during metadata operations on catalog tables. The performance loss occurs between thousands to tens of thousands of `DROP TABLE` statements, depending on the system.

Pivotal recommends you regularly run a system catalog maintenance procedure to reclaim the space occupied by deleted objects. If a regular procedure has not been run for a long time, you may need to run a more intensive procedure to clear the system catalog. This topic describes both procedures.

Regular System Catalog Maintenance

It is recommended that you periodically run `VACUUM` and `REINDEX` on the system catalog to clear the space that deleted objects occupy in the system tables and indexes. If regular database operations include numerous `DROP` statements, it is safe and appropriate to run a system catalog maintenance procedure with `VACUUM` daily at off-peak hours. You can do this while the system is available.

The following example script performs a `VACUUM`, `REINDEX`, and `ANALYZE` of the Greenplum Database system catalog:

```
#!/bin/bash
```

```
DBNAME="<database-name>"
SYSTABLES="' pg_catalog.' || relname || ';' from pg_class a, pg_namespace b
where a.relnamespace=b.oid and b.nspname='pg_catalog' and a.relkind='r'"
psql -tc "SELECT 'VACUUM' || $SYSTABLES" $DBNAME | psql -a $DBNAME
reindexdb --system -d $DBNAME
analyzedb -s pg_catalog -d $DBNAME
```

Intensive System Catalog Maintenance

If a system catalog maintenance procedure has not been performed in a long time, the catalog can become bloated with dead space; this causes excessively long wait times for simple metadata operations. A wait of more than two seconds to list user tables, such as with the `\d` metacommand from within `psql`, is an indication of catalog bloat.

If you see indications of system catalog bloat, you must perform an intensive system catalog maintenance procedure with `VACUUM FULL` during a scheduled downtime period. During this period, stop all catalog activity on the system; the `FULL` system catalog maintenance procedure takes exclusive locks against the system catalog.

Running regular system catalog maintenance procedures can prevent the need for this more costly procedure.

Vacuum and Analyze for Query Optimization

Greenplum Database uses a cost-based query optimizer that relies on database statistics. Accurate statistics allow the query optimizer to better estimate selectivity and the number of rows that a query operation retrieves. These estimates help it choose the most efficient query plan. The `ANALYZE` command collects column-level statistics for the query optimizer.

You can run both `VACUUM` and `ANALYZE` operations in the same command. For example:

```
=# VACUUM ANALYZE mytable;
```

Running the `VACUUM ANALYZE` command might produce incorrect statistics when the command is run on a table with a significant amount of bloat (a significant amount of table disk space is occupied by deleted or obsolete rows). For large tables, the `ANALYZE` command calculates statistics from a random sample of rows. It estimates the number rows in the table by multiplying the average number of rows per page in the sample by the number of actual pages in the table. If the sample contains many empty pages, the estimated row count can be inaccurate.

For a table, you can view information about the amount of unused disk space (space that is occupied by deleted or obsolete rows) in the *gp_toolkit* view *gp_bloat_diag*. If the `bdidiag` column for a table contains the value `significant amount of bloat suspected`, a significant amount of table disk space consists of unused space. Entries are added to the *gp_bloat_diag* view after a table has been vacuumed.

To remove unused disk space from the table, you can run the command `VACUUM FULL` on the table. Due to table lock requirements, `VACUUM FULL` might not be possible until a maintenance period.

As a temporary workaround, run `ANALYZE` to compute column statistics and then run `VACUUM` on the table to generate an accurate row count. This example runs `ANALYZE` and then `VACUUM` on the *cust_info* table.

```
ANALYZE cust_info;
VACUUM cust_info;
```

> **Important:** If you intend to execute queries on partitioned tables with the Pivotal Query Optimizer enabled, you must collect statistics on the partitioned table root partition with the `ANALYZE ROOTPARTITION` command. For information about the Pivotal Query Optimizer, see *Overview of the Pivotal Query Optimizer*.
>
> **Note:** You can use the Greenplum Database utility `analyzedb` to update table statistics. Tables can be analyzed concurrently. For append optimized tables, `analyzedb` updates statistics only if the

statistics are not current. For information about the `analyzedb` utility, see the *Greenplum Database Utility Guide*.

Routine Reindexing

For B-tree indexes, a freshly-constructed index is slightly faster to access than one that has been updated many times because logically adjacent pages are usually also physically adjacent in a newly built index. Reindexing older indexes periodically can improve access speed. If all but a few index keys on a page have been deleted, there will be wasted space on the index page. A reindex will reclaim that wasted space. In Greenplum Database it is often faster to drop an index (`DROP INDEX`) and then recreate it (`CREATE INDEX`) than it is to use the `REINDEX` command.

For table columns with indexes, some operations such as bulk updates or inserts to the table might perform more slowly because of the updates to the indexes. To enhance performance of bulk operations on tables with indexes, you can drop the indexes, perform the bulk operation, and then re-create the index.

Managing Greenplum Database Log Files

- *Database Server Log Files*
- *Management Utility Log Files*

Database Server Log Files

Greenplum Database log output tends to be voluminous, especially at higher debug levels, and you do not need to save it indefinitely. Administrators rotate the log files periodically so new log files are started and old ones are removed.

Greenplum Database has log file rotation enabled on the master and all segment instances. Daily log files are created in the `pg_log` subdirectory of the master and each segment data directory using the following naming convention: `gpdb-`*`YYYY-MM-DD_hhmmss`*`.csv`. Although log files are rolled over daily, they are not automatically truncated or deleted. Administrators need to implement scripts or programs to periodically clean up old log files in the `pg_log` directory of the master and each segment instance.

For information about viewing the database server log files, see *Viewing the Database Server Log Files*.

Management Utility Log Files

Log files for the Greenplum Database management utilities are written to `~/gpAdminLogs` by default. The naming convention for management log files is:

```
script_name_date.log
```

The log entry format is:

```
timestamp:utility:host:user:[INFO|WARN|FATAL]:message
```

The log file for a particular utility execution is appended to its daily log file each time that utility is run.

Chapter 16

Recommended Monitoring and Maintenance Tasks

This section lists monitoring and maintenance activities recommended to ensure high availability and consistent performance of your Greenplum Database cluster.

The tables in the following sections suggest activities that a Greenplum System Administrator can perform periodically to ensure that all components of the system are operating optimally. Monitoring activities help you to detect and diagnose problems early. Maintenance activities help you to keep the system up-to-date and avoid deteriorating performance, for example, from bloated system tables or diminishing free disk space.

It is not necessary to implement all of these suggestions in every cluster; use the frequency and severity recommendations as a guide to implement measures according to your service requirements.

Database State Monitoring Activities

Table 21: Database State Monitoring Activities

Activity	Procedure	Corrective Actions

List segments that are currently down. If any rows are returned, this should generate a warning or alert.
Recommended frequency: run every 5 to 10 minutes
Severity: IMPORTANT
	Run the following query in the `postgres` database:
	`SELECT * FROM gp_segment_configuration`
	`WHERE status <> 'u';`
		If the query returns any rows, follow these steps to correct the problem:
		1. Verify that the hosts with down segments are responsive.
		2. If hosts are OK, check the `pg_log` files for the primaries and mirrors of the down segments to discover the root cause of the segments going down.
		3. If no unexpected errors are found, run the `gprecoverseg` utility to bring the segments back online.

Check for segments that are currently in change tracking mode. If any rows are returned, this should generate a warning or alert.
Recommended frequency: run every 5 to 10 minutes
Severity: IMPORTANT
	Execute the following query in the `postgres` database:
	`SELECT * FROM gp_segment_configuration`
	`WHERE mode = 'c';`
		If the query returns any rows, follow these steps to correct the problem:
		1. Verify that hosts with down segments are responsive.
		2. If hosts are OK, check the `pg_log` files for the primaries and mirrors of the down segments to determine the root cause of the segments going down.
		3. If no unexpected errors are found, run the `gprecoverseg` utility to bring the segments back online.

Activity	Procedure	Corrective Actions

Check for segments that are currently re-syncing. If rows are returned, this should generate a warning or alert.
Recommended frequency: run every 5 to 10 minutes
Severity: IMPORTANT
	Execute the following query in the `postgres` database:
	`SELECT * FROM gp_segment_configuration`
	`WHERE mode = 'r';`
		When this query returns rows, it implies that the segments are in the process of being re-synched. If the state does not change from 'r' to 's', then check the `pg_log` files from the primaries and mirrors of the affected segments for errors.

Check for segments that are not operating in their optimal role. If any segments are found, the cluster may not be balanced. If any rows are returned this should generate a warning or alert.
Recommended frequency: run every 5 to 10 minutes
Severity: IMPORTANT
	Execute the following query in the `postgres` database:
	`SELECT * FROM gp_segment_configuration`
	`WHERE preferred_role <> role;`
		When the segments are not running in their preferred role, hosts have uneven numbers of primary segments on each host, implying that processing is skewed. Wait for a potential window and restart the database to bring the segments into their preferred roles.

Run a distributed query to test that it runs on all segments. One row should be returned for each primary segment.
Recommended frequency: run every 5 to 10 minutes
Severity: CRITICAL
	Execute the following query in the `postgres` database:
	`SELECT gp_segment_id, count(*)`
	`FROM gp_dist_random('pg_class')`
	`GROUP BY 1;`
		If this query fails, there is an issue dispatching to some segments in the cluster. This is a rare event. Check the hosts that are not able to be dispatched to ensure there is no hardware or networking issue.

Test the state of master mirroring on a Greenplum Database 4.2 or earlier cluster. If the value is "Not Synchronized", raise an alert or warning.
Recommended frequency: run every 5 to 10 minutes
Severity: IMPORTANT
	Execute the following query in the `postgres` database:
	`SELECT summary_state`
	`FROM gp_master_mirroring;`
		Check the `pg_log` from the master and standby master for errors. If there are no unexpected errors and the machines are up, run the `gpinitstandby` utility to bring the standby online. This requires a database restart on GPDB 4.2 and earlier.

Test the state of master mirroring on Greenplum Database 4.3 and later. If the value is not "STREAMING", raise an alert or warning. Recommended frequency: run every 5 to 10 minutes Severity: IMPORTANT	Run the following `psql` command: `psql` *`dbname`* `-c 'SELECT procpid, state FROM pg_stat_replication;'`	Check the `pg_log` file from the master and standby master for errors. If there are no unexpected errors and the machines are up, run the `gpinitstandby` utility to bring the standby online.

Activity	Procedure	Corrective Actions
Perform a basic check to see if the master is up and functioning. Recommended frequency: run every 5 to 10 minutes Severity: CRITICAL	Run the following query in the `postgres` database: `SELECT count(*) FROM gp_segment_configuration;`	If this query fails the active master may be down. Try again several times and then inspect the active master manually. If the active master is down, reboot or power cycle the active master to ensure no processes remain on the active master and then trigger the activation of the standby master.

Database Alert Log Monitoring

Table 22: Database Alert Log Monitoring Activities

Activity	Procedure	Corrective Actions

Check for FATAL and ERROR log messages from the system.
Recommended frequency: run every 15 minutes
Severity: WARNING
This activity and the next are two methods for monitoring messages in the log_alert_history table. It is only necessary to set up one or the other.
	Run the following query in the `gpperfmon` database:
	`SELECT * FROM log_alert_history`
	`WHERE logseverity in ('FATAL', 'ERROR')`
	`AND logtime > (now() - interval '15 minutes');`
		Send an alert to the DBA to analyze the alert. You may want to add additional filters to the query to ignore certain messages of low interest.

Set up server configuration parameters to send SNMP or email alerts. Recommended frequency: N/A. Alerts are generated by the system. Severity: WARNING *This activity and the previous are two methods for monitoring messages in the log_alert_history table. It is only necessary to set up one or the other.*	Enable server configuration parameters to send alerts via SNMP or email: • `gp_email_smtp_server` • `gp_email_smtp_userid` • `gp_email_smtp_password` or `gp_snmp_monitor_address` • `gp_snmp_community` • `gp_snmp_use_inform_or_trap`	DBA takes action based on the nature of the alert.

Hardware and Operating System Monitoring

Table 23: Hardware and Operating System Monitoring Activities

Activity	Procedure	Corrective Actions
Underlying platform check for maintenance required or system down of the hardware. Recommended frequency: real-time, if possible, or every 15 minutes Severity: CRITICAL	Set up SNMP or other system check for hardware and OS errors.	If required, remove a machine from the Greenplum cluster to resolve hardware and OS issues, then, after add it back to the cluster and run `gprecoverseg`.
Check disk space usage on volumes used for Greenplum Database data storage and the OS. Recommended frequency: every 5 to 30 minutes Severity: CRITICAL	Set up a disk space check. • Set a threshold to raise an alert when a disk reaches a percentage of capacity. The recommended threshold is 75% full. • It is not recommended to run the system with capacities approaching 100%.	Free space on the system by removing some data or files.
Check for errors or dropped packets on the network interfaces. Recommended frequency: hourly Severity: IMPORTANT	Set up a network interface checks.	Work with network and OS teams to resolve errors.
Check for RAID errors or degraded RAID performance. Recommended frequency: every 5 minutes Severity: CRITICAL	Set up a RAID check.	• Replace failed disks as soon as possible. • Work with system administration team to resolve other RAID or controller errors as soon as possible.
Run the Greenplum `gpcheck` utility to test that the configuration of the cluster complies with Pivotal recommendations. Recommended frequency: when creating a cluster or adding new machines to the cluster Severity: IMPORTANT	Run `gpcheck`.	Work with system administration team to update configuration according to the recommendations made by the `gpcheck` utility.

Activity	Procedure	Corrective Actions
Check for adequate I/O bandwidth and I/O skew. Recommended frequency: when create a cluster or when hardware issues are suspected.	Run the Greenplum `gpcheckperf` utility.	The cluster may be under-specified if data transfer rates are not similar to the following: • 2GB per second disk read • 1 GB per second disk write • 10 Gigabit per second network read and write If transfer rates are lower than expected, consult with your data architect regarding performance expectations. If the machines on the cluster display an uneven performance profile, work with the system administration team to fix faulty machines.

Catalog Monitoring

Table 24: Catalog Monitoring Activities

Activity	Procedure	Corrective Actions
Run catalog consistency checks to ensure the catalog on each host in the cluster is consistent and in a good state. Recommended frequency: weekly Severity: IMPORTANT	Run the Greenplum `gpcheckcat` utility in each database: `gpcheckcat -O`	Run repair scripts for any issues detected.
Run a persistent table catalog check. Recommended frequency: monthly Severity: CRITICAL	During a downtime, with no users on the system, run the Greenplum `gpcheckcat` utility in each database: `gpcheckcat -R persistent`	If any problems are identified, contact Pivotal support for assistance.
Check for `pg_class` entries that have no corresponding pg_`attribute` entry. Recommended frequency: monthly Severity: IMPORTANT	During a downtime, with no users on the system, run the Greenplum `gpcheckcat` utility in each database: `gpcheckcat -R pgclass`	Run the repair scripts for any issues identified.

Activity	Procedure	Corrective Actions
Check for leaked temporary schema and missing schema definition. Recommended frequency: monthly Severity: IMPORTANT	During a downtime, with no users on the system, run the Greenplum `gpcheckcat` utility in each database: `gpcheckcat -R namespace`	Run the repair scripts for any issues identified.
Check constraints on randomly distributed tables. Recommended frequency: monthly Severity: IMPORTANT	During a downtime, with no users on the system, run the Greenplum `gpcheckcat` utility in each database: `gpcheckcat -R distribution_policy`	Run the repair scripts for any issues identified.
Check for dependencies on non-existent objects. Recommended frequency: monthly Severity: IMPORTANT	During a downtime, with no users on the system, run the Greenplum `gpcheckcat` utility in each database: `gpcheckcat -R dependency`	Run the repair scripts for any issues identified.

Data Maintenance

Table 25: Data Maintenance Activities

Activity	Procedure	Corrective Actions
Check for missing statistics on tables.	Check the `gp_stats_missing` view in each database: `SELECT * FROM gp_toolkit.gp_stats_missing;`	Run `ANALYZE` on tables that are missing statistics.
Check for tables that have bloat (dead space) in data files that cannot be recovered by a regular `VACUUM` command. Recommended frequency: weekly or monthly Severity: WARNING	Check the `gp_bloat_diag` view in each database: `SELECT * FROM gp_toolkit.gp_bloat_diag;`	Execute a `VACUUM FULL` statement at a time when users are not accessing the table to remove bloat and compact the data.

Database Maintenance

Table 26: Database Maintenance Activities

Activity	Procedure	Corrective Actions
Mark deleted rows in heap tables so that the space they occupy can be reused. Recommended frequency: daily Severity: CRITICAL	Vacuum user tables: `VACUUM <table>;`	Vacuum updated tables regularly to prevent bloating.
Update table statistics. Recommended frequency: after loading data and before executing queries Severity: CRITICAL	Analyze user tables. You can use the `analyzedb` management utility: `analyzedb -d <database> -a`	Analyze updated tables regularly so that the optimizer can produce efficient query execution plans.
Backup the database data. Recommended frequency: daily, or as required by your backup plan Severity: CRITICAL	Run the `gpcrondump` utility to create a backup of the master and segment databases in parallel.	Best practice is to have a current backup ready in case the database must be restored.

Vacuum, reindex, and analyze system catalogs to maintain an efficient catalog.
Recommended frequency: weekly, or more often if database objects are created and dropped frequently
	1. `VACUUM` the system tables in each database.
	2. Run `REINDEX SYSTEM` in each database, or use the `reindexdb` command-line utility with the `-s` option:
	`reindexdb -s <database>`
	3. `ANALYZE` each of the system tables:
	`analyzedb -s pg_catalog -d <database>`
		The optimizer retrieves information from the system tables to create query plans. If system tables and indexes are allowed to become bloated over time, scanning the system tables increases query execution time. It is important to run `ANALYZE` after reindexing, because `REINDEX` leaves indexes with no statistics.

Patching and Upgrading

Table 27: Patch and Upgrade Activities

Activity	Procedure	Corrective Actions
Ensure any bug fixes or enhancements are applied to the kernel. Recommended frequency: at least every 6 months Severity: IMPORTANT	Follow the vendor's instructions to update the Linux kernel.	Keep the kernel current to include bug fixes and security fixes, and to avoid difficult future upgrades.
Install Greenplum Database minor releases, for example 4.3.*x.y*. Recommended frequency: quarterly Severity: IMPORTANT	Follow upgrade instructions in the Greenplum Database *Release Notes*. Always upgrade to the latest in the series.	Keep the Greenplum Database software current to incorporate bug fixes, performance enhancements, and feature enhancements into your Greenplum cluster.

Part III

Managing Greenplum Database Access

Securing Greenplum Database includes protecting access to the database through network configuration, database user authentication, and encryption.

This section contains the following topics:

- *Configuring Client Authentication*
  - *Using LDAP Authentication with TLS/SSL*
  - *Using Kerberos Authentication*
- *Managing Roles and Privileges*
- *Configuring IPsec for Greenplum Database*

Chapter 17

Configuring Client Authentication

This topic explains how to configure client connections and authentication for Greenplum Database.

When a Greenplum Database system is first initialized, the system contains one predefined *superuser* role. This role will have the same name as the operating system user who initialized the Greenplum Database system. This role is referred to as `gpadmin`. By default, the system is configured to only allow local connections to the database from the `gpadmin` role. To allow any other roles to connect, or to allow connections from remote hosts, you configure Greenplum Database to allow such connections.

> **Note:** The PgBouncer connection pooler is bundled with Greenplum Database. PgBouncer can be configured to support LDAP or Active Directory authentication for users connecting to Greenplum Database through the connection pooler client connections to Greenplum Database. See *Using the PgBouncer Connection Pooler* .

Allowing Connections to Greenplum Database

Client access and authentication is controlled by the standard PostgreSQL host-based authentication file, `pg_hba.conf`. In Greenplum Database, the `pg_hba.conf` file of the master instance controls client access and authentication to your Greenplum Database system. Greenplum Database segments have `pg_hba.conf` files that are configured to allow only client connections from the master host and never accept client connections. Do not alter the `pg_hba.conf` file on your segments.

See *The pg_hba.conf File* in the PostgreSQL documentation for more information.

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

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

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

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

The following table describes meaning of each field.

Table 28: pg_hba.conf Fields

Field	Description
local	Matches connection attempts using UNIX-domain sockets. Without a record of this type, UNIX-domain socket connections are disallowed.
host	Matches connection attempts made using TCP/IP. Remote TCP/IP connections will not be possible unless the server is started with an appropriate value for the listen_addresses server configuration parameter.
hostssl	Matches connection attempts made using TCP/IP, but only when the connection is made with SSL encryption. SSL must be enabled at server start time by setting the ssl configuration parameter
hostnossl	Matches connection attempts made over TCP/IP that do not use SSL.

Field	Description
database	Specifies which database names this record matches. The value `all` specifies that it matches all databases. Multiple database names can be supplied by separating them with commas. A separate file containing database names can be specified by preceding the file name with @.
role	Specifies which database role names this record matches. The value `all` specifies that it matches all roles. If the specified role is a group and you want all members of that group to be included, precede the role name with a +. Multiple role names can be supplied by separating them with commas. A separate file containing role names can be specified by preceding the file name with @.
CIDR-address	Specifies the client machine IP address range that this record matches. It contains an IP address in standard dotted decimal notation and a CIDR mask length. IP addresses can only be specified numerically, not as domain or host names. The mask length indicates the number of high-order bits of the client IP address that must match. Bits to the right of this must be zero in the given IP address. There must not be any white space between the IP address, the /, and the CIDR mask length. Typical examples of a CIDR-address are 192.0.2.89/32 for a single host, or 192.0.2.0/24 for a small network, or 10.6.0.0/16 for a larger one. To specify a single host, use a CIDR mask of 32 for IPv4 or 128 for IPv6. In a network address, do not omit trailing zeroes.
IP-address IP-mask	These fields can be used as an alternative to the CIDR-address notation. Instead of specifying the mask length, the actual mask is specified in a separate column. For example, 255.255.255.255 represents a CIDR mask length of 32. These fields only apply to host, hostssl, and hostnossl records.
authentication-method	Specifies the authentication method to use when connecting. Greenplum supports the *authentication methods* supported by PostgreSQL 9.0.

Editing the pg_hba.conf File

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

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

Editing pg_hba.conf

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

```
# allow the gpadmin user local access to all databases
# using ident authentication
local   all   gpadmin   ident         sameuser
host    all   gpadmin   127.0.0.1/32  ident
host    all   gpadmin   ::1/128       ident
# allow the 'dba' role access to any database from any
# host with IP address 192.168.x.x and use md5 encrypted
# passwords to authenticate the user
# Note that to use SHA-256 encryption, replace md5 with
# password in the line below
host    all   dba   192.168.0.0/32  md5
# allow all roles access to any database from any
# host and use ldap to authenticate the user. Greenplum role
# names must match the LDAP common name.
host    all   all   192.168.0.0/32  ldap ldapserver=usldap1
ldapport=1389 ldapprefix="cn="
ldapsuffix=",ou=People,dc=company,dc=com"
```

3. Save and close the file.
4. Reload the `pg_hba.conf` configuration file for your changes to take effect:

```
$ gpstop -u
```

> **Note:** Note that you can also control database access by setting object privileges as described in *Managing Object Privileges*. The `pg_hba.conf` file just controls who can initiate a database session and how those connections are authenticated.

Limiting Concurrent Connections

Greenplum Database allocates some resources on a per-connection basis, so setting the maximum number of connections allowed is recommended.

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

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

For example:

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

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

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

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

The following steps set the parameter values with the Greenplum Database utility `gpconfig`.

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

To change the number of allowed connections

1. Log into the Greenplum Database master host as the Greenplum Database administrator and source the file `$GPHOME/greenplum_path.sh`.
2. Set the value of the `max_connections` parameter. This `gpconfig` command sets the value on the segments to 1000 and the value on the master to 200.

```
$ gpconfig -c max_connections -v 1000 -m 200
```

The value on the segments must be greater than the value on the master. Pivotal recommends that the value of `max_connections` on segments be 5-10 times the value on the master.

3. Set the value of the `max_prepared_transactions` parameter. This `gpconfig` command sets the value to 200 on the master and all segments.

   ```
   $ gpconfig -c max_prepared_transactions -v 200
   ```

   The value of `max_prepared_transactions` must be greater than or equal to `max_connections` on the master.
4. Stop and restart your Greenplum Database system.

   ```
   $ gpstop -r
   ```
5. You can check the value of parameters on the master and segments with the `gpconfig -s` option. This `gpconfig` command displays the values of the `max_connections` parameter.

   ```
   $ gpconfig -s max_connections
   ```

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

Encrypting Client/Server Connections

Enable SSL for client connections to Greenplum Database to encrypt the data passed over the network between the client and the database.

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

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

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

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

Creating a Self-signed Certificate without a Passphrase for Testing Only

To create a quick self-signed certificate for the server for testing, use the following OpenSSL command:

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

Enter the information requested by the prompts. Be sure to enter the local host name as *Common Name*. The challenge password can be left blank.

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

To use this certificate with Greenplum Database, remove the passphrase with the following commands:

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

```
# rm privkey.pem
```

Enter the old passphrase when prompted to unlock the existing key.

Then, enter the following command to turn the certificate into a self-signed certificate and to copy the key and certificate to a location where the server will look for them.

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

Finally, change the permissions on the key with the following command. The server will reject the file if the permissions are less restrictive than these.

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

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

Using LDAP Authentication with TLS/SSL

You can control access to Greenplum Database with an LDAP server and, optionally, secure the connection with encryption by adding parameters to `pg_hba.conf` file entries.

Greenplum Database supports LDAP authentication with the TLS/SSL protocol to encrypt communication with an LDAP server:

- LDAP authentication with STARTTLS and TLS protocol – STARTTLS starts with a clear text connection (no encryption) and upgrades it to a secure connection (with encryption).
- LDAP authentication with a secure connection and TLS/SSL (LDAPS) – Greenplum Database uses the TLS or SSL protocol based on the protocol that is used by the LDAP server.

If no protocol is specified, Greenplum Database communicates with the LDAP server with a clear text connection.

To use LDAP authentication, the Greenplum Database master host must be configured as an LDAP client. See your LDAP documentation for information about configuring LDAP clients.

> **Note:** The PgBouncer connection pooler bundled with Greenplum Database is modified to support LDAP or Active Directory authentication for users connecting to Greenplum Database through the connection pooler. See *Setting up LDAP Authentication with PgBouncer* for instructions.

Enabing LDAP Authentication with STARTTLS and TLS

To enable STARTTLS with the TLS protocol, in the pg_hba.conf file, add an `ldap` line and specify the `ldaptls` parameter with the value 1. The default port is 389. In this example, the authentication method parameters include the `ldaptls` parameter.

```
ldap ldapserver=myldap.com ldaptls=1 ldapprefix="uid="
 ldapsuffix=",ou=People,dc=pivotal,dc=com"
```

Specify a non-default port with the `ldapport` parameter. In this example, the authentication method includes the `ldaptls` parameter and the `ldapport` parameter to specify the port 550.

```
ldap ldapserver=myldap.com ldaptls=1 ldapport=500 ldapprefix="uid="
 ldapsuffix=",ou=People,dc=pivotal,dc=com"
```

Enabing LDAP Authentication with a Secure Connection and TLS/SSL

To enable a secure connection with TLS/SSL, add `ldaps://` as the prefix to the LDAP server name specified in the `ldapserver` parameter. The default port is 636.

This example `ldapserver` parameter specifies a secure connection and the TLS/SSL protocol for the LDAP server `myldap.com`.

```
ldapserver=ldaps://myldap.com
```

To specify a non-default port, add a colon (:) and the port number after the LDAP server name. This example `ldapserver` parameter includes the `ldaps://` prefix and the non-default port 550.

```
ldapserver=ldaps://myldap.com:550
```

Configuring Authentication with a System-wide OpenLDAP System

If you have a system-wide OpenLDAP system and logins are configured to use LDAP with TLS or SSL in the pg_hba.conf file, logins may fail with the following message:

```
could not start LDAP TLS session: error code '-11'
```

To use an existing OpenLDAP system for authentication, Greenplum Database must be set up to use the LDAP server's CA certificate to validate user certificates. Follow these steps on both the master and standby hosts to configure Greenplum Database:

1. Copy the base64-encoded root CA chain file from the Active Directory or LDAP server to the Greenplum Database master and standby master hosts. This example uses the directory `/etc/pki/tls/certs`.
2. Change to the directory where you copied the CA certificate file and, as the root user, generate the hash for OpenLDAP:

   ```
   # cd /etc/pki/tls/certs
   # openssl x509 -noout -hash -in <ca-certificate-file>
   # ln -s <ca-certificate-file> <ca-certificate-file>.0
   ```
3. Configure an OpenLDAP configuration file for Greenplum Database with the CA certificate directory and certificate file specified.

   As the root user, edit the OpenLDAP configuration file `/etc/openldap/ldap.conf`:

   ```
   SASL_NOCANON on
    URI ldaps://ldapA.pivotal.priv ldaps://ldapB.pivotal.priv ldaps://
   ldapC.pivotal.priv
    BASE dc=pivotal,dc=priv
    TLS_CACERTDIR /etc/pki/tls/certs
    TLS_CACERT /etc/pki/tls/certs/<ca-certificate-file>
   ```

   Note: For certificate validation to succeed, the hostname in the certificate must match a hostname in the URI property. Otherwise, you must also add `TLS_REQCERT allow` to the file.
4. As the gpadmin user, edit `/usr/local/greenplum-db/greenplum_path.sh` and add the following line.

   ```
   export LDAPCONF=/etc/openldap/ldap.conf
   ```

Notes

Greenplum Database logs an error if the following are specified in an `pg_hba.conf` file entry:

- If both the `ldaps://` prefix and the `ldaptls=1` parameter are specified.
- If both the `ldaps://` prefix and the `ldapport` parameter are specified.

Enabling encrypted communication for LDAP authentication only encrypts the communication between Greenplum Database and the LDAP server.

See *Encrypting Client/Server Connections* for information about encrypting client connections.

Examples

These are example entries from an `pg_hba.conf` file.

This example specifies LDAP authentication with no encryption between Greenplum Database and the LDAP server.

```
host all plainuser 0.0.0.0/0 ldap ldapserver=myldap.com ldapprefix="uid="
 ldapsuffix=",ou=People,dc=pivotal,dc=com"
```

This example specifies LDAP authentication with the STARTTLS and TLS protocol between Greenplum Database and the LDAP server.

```
host all tlsuser 0.0.0.0/0 ldap ldapserver=myldap.com ldaptls=1 ldapprefix="uid="
 ldapsuffix=",ou=People,dc=pivotal,dc=com"
```

This example specifies LDAP authentication with a secure connection and TLS/SSL protocol between Greenplum Database and the LDAP server.

```
host all ldapsuser 0.0.0.0/0 ldap ldapserver=ldaps://myldap.com ldapprefix="uid="
 ldapsuffix=",ou=People,dc=pivotal,dc=com"
```

Using Kerberos Authentication

You can control access to Greenplum Database with a Kerberos authentication server.

Greenplum Database supports the Generic Security Service Application Program Interface (GSSAPI) with Kerberos authentication. GSSAPI provides automatic authentication (single sign-on) for systems that support it. You specify the Greenplum Database users (roles) that require Kerberos authentication in the Greenplum Database configuration file `pg_hba.conf`. The login fails if Kerberos authentication is not available when a role attempts to log in to Greenplum Database.

Kerberos provides a secure, encrypted authentication service. It does not encrypt data exchanged between the client and database and provides no authorization services. To encrypt data exchanged over the network, you must use an SSL connection. To manage authorization for access to Greenplum databases and objects such as schemas and tables, you use settings in the `pg_hba.conf` file and privileges given to Greenplum Database users and roles within the database. For information about managing authorization privileges, see *Managing Roles and Privileges*.

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

Requirements for Using Kerberos with Greenplum Database

The following items are required for using Kerberos with Greenplum Database:

- Kerberos Key Distribution Center (KDC) server using the `krb5-server` library
- Kerberos version 5 `krb5-libs` and `krb5-workstation` packages installed on the Greenplum Database master host
- Greenplum Database version with support for Kerberos
- System time on the Kerberos server and Greenplum Database master host must be synchronized. (Install Linux `ntp` package on both servers.)
- Network connectivity between the Kerberos server and the Greenplum Database master
- Java 1.7.0_17 or later is required to use Kerberos-authenticated JDBC on Red Hat Enterprise Linux 6.x
- Java 1.6.0_21 or later is required to use Kerberos-authenticated JDBC on Red Hat Enterprise Linux 4.x or 5.x

Enabling Kerberos Authentication for Greenplum Database

Complete the following tasks to set up Kerberos authentication with Greenplum Database:

1. Verify your system satisfies the prequisites for using Kerberos with Greenplum Database. See *Requirements for Using Kerberos with Greenplum Database*.
2. Set up, or identify, a Kerberos Key Distribution Center (KDC) server to use for authentication. See *Install and Configure a Kerberos KDC Server*.
3. In a Kerberos database on the KDC server, set up a Kerberos realm and principals on the server. For Greenplum Database, a principal is a Greenplum Database role that uses Kerberos authentication. In the Kerberos database, a realm groups together Kerberos principals that are Greenplum Database roles.
4. Create Kerberos `keytab` files for Greenplum Database. To access Greenplum Database, you create a service key known only by Kerberos and Greenplum Database. On the Kerberos server, the service key is stored in the Kerberos database.

   On the Greenplum Database master, the service key is stored in key tables, which are files known as keytabs. The service keys are usually stored in the keytab file `/etc/krb5.keytab`. This service key is the equivalent of the service's password, and must be kept secure. Data that is meant to be read-only by the service is encrypted using this key.
5. Install the Kerberos client packages and the keytab file on Greenplum Database master.

6. Create a Kerberos ticket for `gpadmin` on the Greenplum Database master node using the keytab file. The ticket contains the Kerberos authentication credentials that grant access to the Greenplum Database.

With Kerberos authentication configured on the Greenplum Database, you can use Kerberos for PSQL and JDBC.

Set up Greenplum Database with Kerberos for PSQL

Set up Greenplum Database with Kerberos for JDBC

Install and Configure a Kerberos KDC Server

Steps to set up a Kerberos Key Distribution Center (KDC) server on a Red Hat Enterprise Linux host for use with Greenplum Database.

Follow these steps to install and configure a Kerberos Key Distribution Center (KDC) server on a Red Hat Enterprise Linux host.

1. Install the Kerberos server packages:

```
sudo yum install krb5-libs krb5-server krb5-workstation
```

2. Edit the `/etc/krb5.conf` configuration file. The following example shows a Kerberos server with a default `KRB.GREENPLUM.COM` realm.

```
[logging]
 default = FILE:/var/log/krb5libs.log
 kdc = FILE:/var/log/krb5kdc.log
 admin_server = FILE:/var/log/kadmind.log

[libdefaults]
 default_realm = KRB.GREENPLUM.COM
 dns_lookup_realm = false
 dns_lookup_kdc = false
 ticket_lifetime = 24h
 renew_lifetime = 7d
 forwardable = true
 default_tgs_enctypes = aes128-cts des3-hmac-sha1 des-cbc-crc des-cbc-md5
 default_tkt_enctypes = aes128-cts des3-hmac-sha1 des-cbc-crc des-cbc-md5
 permitted_enctypes = aes128-cts des3-hmac-sha1 des-cbc-crc des-cbc-md5

[realms]
 KRB.GREENPLUM.COM = {
  kdc = kerberos-gpdb:88
  admin_server = kerberos-gpdb:749
  default_domain = kerberos-gpdb
 }

[domain_realm]
 .kerberos-gpdb = KRB.GREENPLUM.COM
 kerberos-gpdb = KRB.GREENPLUM.COM

[appdefaults]
 pam = {
    debug = false
    ticket_lifetime = 36000
    renew_lifetime = 36000
    forwardable = true
    krb4_convert = false
 }
```

The `kdc` and `admin_server` keys in the `[realms]` section specify the host (`kerberos-gpdb`) and port where the Kerberos server is running. IP numbers can be used in place of host names.

If your Kerberos server manages authentication for other realms, you would instead add the KRB.GREENPLUM.COM realm in the [realms] and [domain_realm] section of the kdc.conf file. See the *Kerberos documentation* for information about the kdc.conf file.

3. To create a Kerberos KDC database, run the kdb5_util.

```
kdb5_util create -s
```

The kdb5_util create option creates the database to store keys for the Kerberos realms that are managed by this KDC server. The -s option creates a stash file. Without the stash file, every time the KDC server starts it requests a password.

4. Add an administrative user to the KDC database with the kadmin.local utility. Because it does not itself depend on Kerberos authentication, the kadmin.local utility allows you to add an initial administrative user to the local Kerberos server. To add the user gpadmin as an administrative user to the KDC database, run the following command:

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

Most users do not need administrative access to the Kerberos server. They can use kadmin to manage their own principals (for example, to change their own password). For information about kadmin, see the *Kerberos documentation*.

5. If needed, edit the /var/kerberos/krb5kdc/kadm5.acl file to grant the appropriate permissions to gpadmin.
6. Start the Kerberos daemons:

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

7. To start Kerberos automatically upon restart:

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

Create Greenplum Database Roles in the KDC Database

Add principals to the Kerberos realm for Greenplum Database.

Start kadmin.local in interactive mode, then add two principals to the Greenplum Database Realm.

1. Start kadmin.local in interactive mode:

```
kadmin.local
```

2. Add principals:

```
kadmin.local: addprinc gpadmin/kerberos-gpdb@KRB.EXAMPLE.COM
kadmin.local: addprinc postgres/master.test.com@KRB.EXAMPLE.COM
```

The adprinc commands prompt for passwords for each principal. The first addprinc creates a Greenplum Database user as a principal, gpadmin/kerberos-gpdb. The second addprinc command creates the postgres process on the Greenplum Database master host as a principal in the Kerberos KDC. This principal is required when using Kerberos authentication with Greenplum Database.

3. Create a Kerberos keytab file with kadmin.local. The following example creates a keytab file gpdb-kerberos.keytab in the current directory with authentication information for the two principals.

```
kadmin.local: xst -k gpdb-kerberos.keytab
    gpadmin/kerberos-gpdb@KRB.EXAMPLE.COM
    postgres/master.test.com@KRB.EXAMPLE.COM
```

You will copy this file to the Greenplum Database master host.

4. Exit kadmin.local interactive mode with the quit command:

```
kadmin.local: quit
```

Install and Configure the Kerberos Client

Steps to install the Kerberos client on the Greenplum Database master host.

Install the Kerberos client libraries on the Greenplum Database master and configure the Kerberos client.

1. Install the Kerberos packages on the Greenplum Database master.

   ```
   sudo yum install krb5-libs krb5-workstation
   ```

2. Ensure that the `/etc/krb5.conf` file is the same as the one that is on the Kerberos server.
3. Copy the `gpdb-kerberos.keytab` file that was generated on the Kerberos server to the Greenplum Database master host.
4. Remove any existing tickets with the Kerberos utility `kdestroy`. Run the utility as root.

   ```
   sudo kdestroy
   ```

5. Use the Kerberos utility `kinit` to request a ticket using the keytab file on the Greenplum Database master for `gpadmin/kerberos-gpdb@KRB.EXAMPLE.COM`. The -t option specifies the keytab file on the Greenplum Database master.

   ```
   # kinit -k -t gpdb-kerberos.keytab gpadmin/kerberos-gpdb@KRB.EXAMPLE.COM
   ```

6. Use the Kerberos utility `klist` to display the contents of the Kerberos ticket cache on the Greenplum Database master. The following is an example:

   ```
   # klist
   Ticket cache: FILE:/tmp/krb5cc_108061
   Default principal: gpadmin/kerberos-gpdb@KRB.EXAMPLE.COM
   Valid starting     Expires            Service principal
   03/28/13 14:50:26  03/29/13 14:50:26  krbtgt/KRB.GREENPLUM.COM
        @KRB.EXAMPLE.COM
       renew until 03/28/13 14:50:26
   ```

Set up Greenplum Database with Kerberos for PSQL

Configure a Greenplum Database to use Kerberos.

After you have set up Kerberos on the Greenplum Database master, you can configure Greenplum Database to use Kerberos. For information on setting up the Greenplum Database master, see *Install and Configure the Kerberos Client*.

1. Create a Greenplum Database administrator role in the database `template1` for the Kerberos principal that is used as the database administrator. The following example uses `gpamin/kerberos-gpdb`.

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

   The role you create in the database `template1` will be available in any new Greenplum Database that you create.
2. Modify `postgresql.conf` to specify the location of the keytab file. For example, adding this line to the `postgresql.conf` specifies the folder `/home/gpadmin` as the location of the keytab file `gpdb-kerberos.keytab`.

   ```
   krb_server_keyfile = '/home/gpadmin/gpdb-kerberos.keytab'
   ```

3. Modify the Greenplum Database file `pg_hba.conf` to enable Kerberos support. Then restart Greenplum Database (`gpstop -ar`). For example, adding the following line to `pg_hba.conf` adds GSSAPI and

Kerberos support. The value for `krb_realm` is the Kerberos realm that is used for authentication to Greenplum Database.

```
host all all 0.0.0.0/0 gss include_realm=0 krb_realm=KRB.GREENPLUM.COM
```

For information about the `pg_hba.conf` file, see *The pg_hba.conf file* in the Postgres documentation.

4. Create a ticket using `kinit` and show the tickets in the Kerberos ticket cache with `klist`.
5. As a test, log in to the database as the `gpadmin` role with the Kerberos credentials `gpadmin/kerberos-gpdb`:

```
psql -U "gpadmin/kerberos-gpdb" -h master.test template1
```

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

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

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

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

The following username map is defined in the Greenplum Database file `$MASTER_DATA_DIRECTORY/pg_ident.conf`:

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

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

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

For more information about specifying username maps see *Username maps* in the Postgres documentation.

6. If a Kerberos principal is not a Greenplum Database user, a message similar to the following is displayed from the `psql` command line when the user attempts to log in to the database:

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

The principal must be added as a Greenplum Database user.

Set up Greenplum Database with Kerberos for JDBC

Enable Kerberos-authenticated JDBC access to Greenplum Database.

You can configure Greenplum Database to use Kerberos to run user-defined Java functions.

1. Ensure that Kerberos is installed and configured on the Greenplum Database master. See *Install and Configure the Kerberos Client*.
2. Create the file `.java.login.config` in the folder /home/gpadmin and add the following text to the file:

```
pgjdbc {
  com.sun.security.auth.module.Krb5LoginModule required
  doNotPrompt=true
  useTicketCache=true
  debug=true
  client=true;
};
```

3. Create a Java application that connects to Greenplum Database using Kerberos authentication. The following example database connection URL uses a PostgreSQL JDBC driver and specifies parameters for Kerberos authentication:

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

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

Chapter 18

Managing Roles and Privileges

The Greenplum Database authorization mechanism stores roles and permissions to access database objects in the database and is administered using SQL statements or command-line utilities.

Greenplum Database manages database access permissions using *roles*. The concept of roles subsumes the concepts of *users* and *groups*. A role can be a database user, a group, or both. Roles can own database objects (for example, tables) and can assign privileges on those objects to other roles to control access to the objects. Roles can be members of other roles, thus a member role can inherit the object privileges of its parent role.

Every Greenplum Database system contains a set of database roles (users and groups). Those roles are separate from the users and groups managed by the operating system on which the server runs. However, for convenience you may want to maintain a relationship between operating system user names and Greenplum Database role names, since many of the client applications use the current operating system user name as the default.

In Greenplum Database, users log in and connect through the master instance, which then verifies their role and access privileges. The master then issues commands to the segment instances behind the scenes as the currently logged in role.

Roles are defined at the system level, meaning they are valid for all databases in the system.

In order to bootstrap the Greenplum Database system, a freshly initialized system always contains one predefined *superuser* role (also referred to as the system user). This role will have the same name as the operating system user that initialized the Greenplum Database system. Customarily, this role is named `gpadmin`. In order to create more roles you first have to connect as this initial role.

Security Best Practices for Roles and Privileges

- **Secure the gpadmin system user.** Greenplum requires a UNIX user id to install and initialize the Greenplum Database system. This system user is referred to as `gpadmin` in the Greenplum documentation. This `gpadmin` user is the default database superuser in Greenplum Database, as well as the file system owner of the Greenplum installation and its underlying data files. This default administrator account is fundamental to the design of Greenplum Database. The system cannot run without it, and there is no way to limit the access of this gpadmin user id. Use roles to manage who has access to the database for specific purposes. You should only use the `gpadmin` account for system maintenance tasks such as expansion and upgrade. Anyone who logs on to a Greenplum host as this user id can read, alter or delete any data; including system catalog data and database access rights. Therefore, it is very important to secure the gpadmin user id and only provide access to essential system administrators. Administrators should only log in to Greenplum as `gpadmin` when performing certain system maintenance tasks (such as upgrade or expansion). Database users should never log on as `gpadmin`, and ETL or production workloads should never run as `gpadmin`.
- **Assign a distinct role to each user that logs in.** For logging and auditing purposes, each user that is allowed to log in to Greenplum Database should be given their own database role. For applications or web services, consider creating a distinct role for each application or service. See *Creating New Roles (Users)*.
- **Use groups to manage access privileges.** See *Role Membership*.
- **Limit users who have the SUPERUSER role attribute.** Roles that are superusers bypass all access privilege checks in Greenplum Database, as well as resource queuing. Only system administrators should be given superuser rights. See *Altering Role Attributes*.

Creating New Roles (Users)

A user-level role is considered to be a database role that can log in to the database and initiate a database session. Therefore, when you create a new user-level role using the CREATE ROLE command, you must specify the LOGIN privilege. For example:

```
=# CREATE ROLE jsmith WITH LOGIN;
```

A database role may have a number of attributes that define what sort of tasks that role can perform in the database. You can set these attributes when you create the role, or later using the ALTER ROLE command. See *Table 29: Role Attributes* for a description of the role attributes you can set.

Altering Role Attributes

A database role may have a number of attributes that define what sort of tasks that role can perform in the database.

Table 29: Role Attributes

Attributes	Description
SUPERUSER | NOSUPERUSER	Determines if the role is a superuser. You must yourself be a superuser to create a new superuser. NOSUPERUSER is the default.
CREATEDB | NOCREATEDB	Determines if the role is allowed to create databases. NOCREATEDB is the default.
CREATEROLE | NOCREATEROLE	Determines if the role is allowed to create and manage other roles. NOCREATEROLE is the default.
INHERIT | NOINHERIT	Determines whether a role inherits the privileges of roles it is a member of. A role with the INHERIT attribute can automatically use whatever database privileges have been granted to all roles it is directly or indirectly a member of. INHERIT is the default.
LOGIN | NOLOGIN	Determines whether a role is allowed to log in. A role having the LOGIN attribute can be thought of as a user. Roles without this attribute are useful for managing database privileges (groups). NOLOGIN is the default.
CONNECTION LIMIT *connlimit*	If role can log in, this specifies how many concurrent connections the role can make. -1 (the default) means no limit.
CREATEEXTTABLE | NOCREATEEXTTABLE	Determines whether a role is allowed to create external tables. NOCREATEEXTTABLE is the default. For a role with the CREATEEXTTABLE attribute, the default external table type is readable and the default protocol is gpfdist. Note that external tables that use the file or execute protocols can only be created by superusers.
PASSWORD '*password*'	Sets the role's password. If you do not plan to use password authentication you can omit this option. If no password is specified, the password will be set to null and password authentication will always fail for that user. A null password can optionally be written explicitly as PASSWORD NULL.

Attributes	Description
ENCRYPTED | UNENCRYPTED	Controls whether a new password is stored as a hash string in the pg_authid system catalog. If neither ENCRYPTED nor UNENCRYPTED is specified, the default behavior is determined by the password_encryption configuration parameter, which is on by default. If the supplied *password* string is already in hashed format, it is stored as-is, regardless of whether ENCRYPTED or UNENCRYPTED is specified. See *Protecting Passwords in Greenplum Database* for additional information about protecting login passwords.
VALID UNTIL '*timestamp*'	Sets a date and time after which the role's password is no longer valid. If omitted the password will be valid for all time.
RESOURCE QUEUE *queue_name*	Assigns the role to the named resource queue for workload management. Any statement that role issues is then subject to the resource queue's limits. Note that the RESOURCE QUEUE attribute is not inherited; it must be set on each user-level (LOGIN) role.
DENY {deny_interval | deny_point}	Restricts access during an interval, specified by day or day and time. For more information see *Time-based Authentication*.

You can set these attributes when you create the role, or later using the ALTER ROLE command. For example:

```
=# ALTER ROLE jsmith WITH PASSWORD 'passwd123';
=# ALTER ROLE admin VALID UNTIL 'infinity';
=# ALTER ROLE jsmith LOGIN;
=# ALTER ROLE jsmith RESOURCE QUEUE adhoc;
=# ALTER ROLE jsmith DENY DAY 'Sunday';
```

A role can also have role-specific defaults for many of the server configuration settings. For example, to set the default schema search path for a role:

```
=# ALTER ROLE admin SET search_path TO myschema, public;
```

Role Membership

It is frequently convenient to group users together to ease management of object privileges: that way, privileges can be granted to, or revoked from, a group as a whole. In Greenplum Database this is done by creating a role that represents the group, and then granting membership in the group role to individual user roles.

Use the CREATE ROLE SQL command to create a new group role. For example:

```
=# CREATE ROLE admin CREATEROLE CREATEDB;
```

Once the group role exists, you can add and remove members (user roles) using the GRANT and REVOKE commands. For example:

```
=# GRANT admin TO john, sally;
=# REVOKE admin FROM bob;
```

For managing object privileges, you would then grant the appropriate permissions to the group-level role only (see *Table 30: Object Privileges*). The member user roles then inherit the object privileges of the group role. For example:

```
=# GRANT ALL ON TABLE mytable TO admin;
=# GRANT ALL ON SCHEMA myschema TO admin;
=# GRANT ALL ON DATABASE mydb TO admin;
```

The role attributes `LOGIN`, `SUPERUSER`, `CREATEDB`, `CREATEROLE`, `CREATEEXTTABLE`, and`RESOURCE QUEUE` are never inherited as ordinary privileges on database objects are. User members must actually `SET ROLE` to a specific role having one of these attributes in order to make use of the attribute. In the above example, we gave `CREATEDB` and `CREATEROLE` to the `admin` role. If `sally` is a member of `admin`, she could issue the following command to assume the role attributes of the parent role:

```
=> SET ROLE admin;
```

Managing Object Privileges

When an object (table, view, sequence, database, function, language, schema, or tablespace) is created, it is assigned an owner. The owner is normally the role that executed the creation statement. For most kinds of objects, the initial state is that only the owner (or a superuser) can do anything with the object. To allow other roles to use it, privileges must be granted. Greenplum Database supports the following privileges for each object type:

Table 30: Object Privileges

Object Type	Privileges

Tables, Views, Sequences	`SELECT`
	`INSERT`
	`UPDATE`
	`DELETE`
	`RULE`
	`ALL`

External Tables	`SELECT`
	`RULE`
	`ALL`

Databases	`CONNECT`
	`CREATE`
	`TEMPORARY | TEMP`
	`ALL`

Functions	`EXECUTE`
Procedural Languages	`USAGE`

Schemas	`CREATE`
	`USAGE`
	`ALL`

Object Type	Privileges

Custom Protocol	SELECT
	INSERT
	UPDATE
	DELETE
	RULE
	ALL

> **Note:** Privileges must be granted for each object individually. For example, granting ALL on a database does not grant full access to the objects within that database. It only grants all of the database-level privileges (CONNECT, CREATE, TEMPORARY) to the database itself.

Use the GRANT SQL command to give a specified role privileges on an object. For example:

```
=# GRANT INSERT ON mytable TO jsmith;
```

To revoke privileges, use the REVOKE command. For example:

```
=# REVOKE ALL PRIVILEGES ON mytable FROM jsmith;
```

You can also use the DROP OWNED and REASSIGN OWNED commands for managing objects owned by deprecated roles (Note: only an object's owner or a superuser can drop an object or reassign ownership). For example:

```
=# REASSIGN OWNED BY sally TO bob;
=# DROP OWNED BY visitor;
```

Simulating Row and Column Level Access Control

Row-level or column-level access is not supported, nor is labeled security. Row-level and column-level access can be simulated using views to restrict the columns and/or rows that are selected. Row-level labels can be simulated by adding an extra column to the table to store sensitivity information, and then using views to control row-level access based on this column. Roles can then be granted access to the views rather than the base table.

Encrypting Data

PostgreSQL provides an optional package of encryption/decryption functions called pgcrypto, which can also be installed and used in Greenplum Database. The pgcrypto package is not installed by default with Greenplum Database, however you can download a pgcrypto package from *Pivotal Network*, and use the Greenplum Package Manager (gppkg) to install pgcrypto across your entire cluster.

The pgcrypto functions allow database administrators to store certain columns of data in encrypted form. This adds an extra layer of protection for sensitive data, as data stored in Greenplum Database in encrypted form cannot be read by anyone who does not have the encryption key, nor can it be read directly from the disks.

With Greenplum Database 4.3.4 and later, you can enable pgcrypto support for *Federal Information Processing Standard* (FIPS) 140-2. The Greenplum Database server configuration parameter pgcrypto.fips controls the pgcrypto support for FIPS 140-2. For information about the parameter, see "Server Configuration Parameters" in the *Greenplum Database Reference Guide*.

> **Note:** The pgcrypto functions run inside the database server, which 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.

Protecting Passwords in Greenplum Database

In its default configuration, Greenplum Database saves MD5 hashes of login users' passwords in the `pg_authid` system catalog rather than saving clear text passwords. Anyone who is able to view the `pg_authid` table can see hash strings, but no passwords. This also ensures that passwords are obscured when the database is dumped to backup files.

The hash function executes when the password is set by using any of the following commands:

- `CREATE USER` *`name`* `WITH ENCRYPTED PASSWORD '`*`password`*`'`
- `CREATE ROLE` *`name`* `WITH LOGIN ENCRYPTED PASSWORD '`*`password`*`'`
- `ALTER USER` *`name`* `WITH ENCRYPTED PASSWORD '`*`password`*`'`
- `ALTER ROLE` *`name`* `WITH ENCRYPTED PASSWORD '`*`password`*`'`

The `ENCRYPTED` keyword may be omitted when the `password_encryption` system configuration parameter is `on`, which is the default value. The `password_encryption` configuration parameter determines whether clear text or hashed passwords are saved when the `ENCRYPTED` or `UNENCRYPTED` keyword is not present in the command.

> **Note:** The SQL command syntax and `password_encryption` configuration variable include the term *encrypt*, but the passwords are not technically encrypted. They are *hashed* and therefore cannot be decrypted.

The hash is calculated on the concatenated clear text password and role name. The MD5 hash produces a 32-byte hexadecimal string prefixed with the characters `md5`. The hashed password is saved in the `rolpassword` column of the `pg_authid` system table.

Although it is not recommended, passwords may be saved in clear text in the database by including the `UNENCRYPTED` keyword in the command or by setting the `password_encryption` configuration variable to `off`. Note that changing the configuration value has no effect on existing passwords, only newly created or updated passwords.

To set `password_encryption` globally, execute these commands in a shell as the `gpadmin` user:

```
$ gpconfig -c password_encryption -v 'off'
$ gpstop -u
```

To set `password_encryption` in a session, use the SQL `SET` command:

```
=# SET password_encryption = 'on';
```

Passwords may be hashed using the SHA-256 (or SHA-256-FIPS) hash algorithm instead of the default MD5 hash algorithm. The algorithm produces a 64-byte hexadecimal string prefixed with the characters `sha256`. The SHA-256-FIPS hash algorithm supports *Federal Information Processing Standard* (FIPS) 140-2, which is generally the reason to use SHA-256 instead of MD5. If SHA-256-FIPS is specified and Greenplum Database is not linked with the RSA BSAFE library, an error is raised. When SHA-256 is specified, no error is raised; the hash is produced by the RSA BSAFE library or the OpenSSL library linked with Greenplum Database.

> **Note:**
>
> Although SHA-256 uses a stronger cryptographic algorithm and produces a longer hash string, it cannot be used with the MD5 authentication method. To use SHA-256 password hashing the authentication method must be set to `password` in the `pg_hba.conf` configuration file so that clear text passwords are sent to Greenplum Database. Because clear text passwords are sent over the network, it is very important to use SSL for client connections when you use SHA-256. The default `md5` authentication method, on the other hand, hashes the password twice before sending it to Greenplum Database, once on the password and role name and then again with a salt value shared between the client and server, so the clear text password is never sent on the network.

To enable SHA-256 hashing, change the `password_hash_algorithm` configuration parameter from its default value, `md5`, to `sha-256` or `sha-256-fips`. The parameter can be set either globally or at the session level. To set `password_hash_algorithm` globally, execute these commands in a shell as the `gpadmin` user:

```
$ gpconfig -c password_hash_algorithm -v 'sha-256'
$ gpstop -u
```

To set `password_hash_algorithm` in a session, use the SQL `SET` command:

```
=# SET password_hash_algorithm = 'sha-256';
```

Time-based Authentication

Greenplum Database enables the administrator to restrict access to certain times by role. Use the `CREATE ROLE` or `ALTER ROLE` commands to specify time-based constraints.

For details, refer to the *Greenplum Database Security Configuration Guide*.

Chapter 19

Configuring IPsec for Greenplum Database

This topic describes how to set up and configure Internet Protocol Security (IPsec) for a Greenplum Database cluster.

- *IPsec Overview*
- *Installing Openswan*
- *Configuring Openswan Connections*

IPsec Overview

Internet Protocol Security (IPsec) is a suite of protocols that authenticate and encrypt communications at the IP network level (OSI layer 3). Using IPsec can help to prevent network attacks, such as packet sniffing, altering network packets, identity spoofing, and man-in-the-middle attacks in the Greenplum Database system.

When IPsec is enabled for Greenplum Database, a virtual private network (VPN), or tunnel, is established between every pair of hosts in the cluster and every packet exchanged between them is encrypted and sent through the tunnel. If you have *n* hosts in the cluster, *n*(*n*-1)/2 VPNs are needed to connect each host to every other host. You may choose to add other hosts on the network, for example ETL servers, to the IPsec configuration.

Encrypting IP traffic has a cost in network performance. To ensure suitable network bandwidth is available after IPsec configuration, use the Greenplum Database `gpcheckperf` utility. See the Greenplum Database Utility Guide for help with `gpcheckperf`. If network bandwidth is insufficient for performance and database workloads you may need to tune the configuration or use a higher bandwidth network medium.

This section describes how to set up and configure IPsec for a Greenplum cluster on Red Hat or CentOS hosts using Openswan, a popular IPsec implementation for Linux. Openswan provides user tools to enable IPsec on Linux. It uses the Internet Key Exchange (IKE) protocol and X.509 certificates to handshake and exchange session keys, and uses the netlink API to interface with the IPsec support built into the Linux kernel.

The IKE protocol allows two peers to negotiate the authentication and encryption algorithms the tunnels will use. The negotiation occurs in two phases. During phase 1, the peers perform a Diffie-Hellman key exchange to establish a secure, encrypted channel. Phase 1 must successfully complete before phase 2 begins.

During phase 2, the peers negotiate the authentication and encryption algorithms to be used with the IPsec tunnels. The result of the phase 2 negotiation is a *security association* (SA). It contains the source, the destination, and an instruction. The Linux kernel uses the SA to set up the connection.

The peers can authenticate each other using one of the following methods:

- RSA public key encryption. Each host has a public and private key. Public keys are distributed to all hosts in the cluster so that any host can authenticate any other host.
- Pre-shared keys. This method is easiest to configure, but is not as secure as using RSA public key encryption.
- X.509 certificates. A certificate is issued for each host by a certificate authority (CA). The host is authenticated based on trust conferred by the CA. This is most often used when many hosts connect to a central gateway.

RSA public key encryption is the preferred method.

There are two connection modes for a connection: tunnel or transport. In tunnel mode, the entire IP packet is encrypted, including the IP headers. In transport mode, the headers are exposed. The tunnel mode is preferred for greater security.

The following resources are recommended for additional information about IPsec and Openswan:

- *Internet Key Exchange* (IKE) - Wikipedia article that describes the IKE protocol used to set up a security association.
- *Security Association* - Wikipedia article that describes the attributes and purpose of a security association.
- *AES instruction set* - Wikipedia article that provides an overview of the Intel Advanced Encryption Standard (AES) instruction set and lists the CPU families that support it.
- *ipsec.conf(5)* - man page for the `ipsec.conf` configuration file.
- *setkey(8)* - man page for the `setkey` utility used to manage the Security Association Database (SAD) and Security Policy Database (SPD) in the Linux kernel.
- *Openswan* - Red Hat Openswan package overview; applies to CentOS also.
- *Host-to-Host VPN Using Openswan* - Red Hat guide for creating a host-to-host VPN using Openswan; can also be used with CentOS.

Installing Openswan

Openswan may be installed using the package manager on your system, by downloading an installable package from the Openswan Web site, or by downloading and compiling source code.

Pivotal recommends that you use Openswan version 2.6.43 or later. If your package manager has an earlier version, you can download an RPM of the latest Openswan release from the Openswan Web site. You can also download Openswan source code from the Openswan Web site.

The following instructions assume you are installing Openswan on hosts running 64-bit Red Hat 6.x or CentOS 6.x.

First, determine if Openswan is already installed and if so, which version:

```
$ sudo yum info installed openswan
```

If the recommended version is already installed, continue with *Configuring and Verifying the Openswan Installation*.

If an older version is installed, uninstall it before continuing:

```
$ sudo yum remove openswan
```

Installing Openswan with an RPM

Enter the following command to see which version of Openswan is available in the package repository:

```
$ sudo yum list available openswan
```

If the recommended version is available, install it on each host in the cluster:

```
$ sudo yum install -y openswan
```

If the recommended version is not in the repository, you can download it from the Openswan Web site at *https://download.openswan.org*. Browse to the `/rhel6/x86_64` directory to find the RPM.

Install the downloaded RPM with a command like the following:

```
$ sudo rpm -i openswanX-version.x86_64.rpm
```

Installing Openswan from Source

If you cannot install Openswan with an RPM you can download the source, compile it, and install it.

1. Download the Openswan source from the Openswan Web site at *Openswan Web site*.
2. Extract the archive and review the `README` file to ensure that the prerequisite packages are installed on your build machine. For example:

   ```
   sudo yum install gmp gmp-devel gawk flex bison \
       iproute2 iptables sed awk bash cut python
   ```

3. Build the Openswan tools by following the instructions in the `README` file. For example:

   ```
   $ make programs
   $ sudo make install
   ```

Configuring and Verifying the Openswan Installation

Follow the steps in this section to configure each host and verify the Openswan installation.

1. Edit `/etc/sysctl.conf` and modify or add the following variables:

   ```
   net.ipv4.conf.default.accept_redirects = 0
   net.ipv4.conf.all.accept_redirects = 0
   net.ipv4.conf.default.send_redirects = 0
   net.ipv4.conf.all.send_redirects = 0
   net.ipv4.ip_forward = 1
   ```

   Execute `sysctl -p` to reload the file.
2. Restore the default SELinux security contexts for the IPsec directory by running the following command:

   ```
   # restorecon -Rv /etc/ipsec.d
   ```

3. If a firewall is enabled, modify it to allow IPsec packets:

   - UDP port 500 for the Internet Key Exchange (IKE) protocol
   - UDP port 4500 for IKE NAT-Traversal
   - Protocol 50 for Encapsulated Security Payload (ESP) IPsec packets
   - *(not recommended)* Protocol 51 for Authenticated Header (AH) IPsec packets

   Here is an example of IPsec rules for iptables:

   ```
   iptables -A INPUT -p udp --sport 500 --dport 500 -j ACCEPT
   iptables -A OUTPUT -p udp --sport 500 --dport 500 -j ACCEPT
   iptables -A INPUT -p udp --sport 4500 --dport 4500 -j ACCEPT
   iptables -A OUTPUT -p udp --sport 4500 --dport 4500 -j ACCEPT
   iptables -A INPUT -p 50 -j ACCEPT
   iptables -A OUTPUT -p 50 -j ACCEPT
   ```

4. Edit the `/etc/ipsec.conf` file and make the following changes:

   - Change `protostack` from `auto` to `netkey`:

     ```
     protostack=netkey
     ```

   - Uncomment or add the following line:

     ```
     include /etc/ipsec.d/*.conf
     ```

     This allows you to create and install a separate configuration file for each host-to-host tunnel.
5. Start IPsec with the `service` command.

   ```
   # service start ipsec
   ```

6. Run `ipsec verify` to check the IPsec installation. Python must be installed to run this command.

```
$ sudo ipsec verify
```

The output looks like the following:

```
Checking if IPsec got installed and started correctly:

Version check and ipsec on-path                          [OK]
Openswan U2.6.43/K2.6.32-504.16.2.el6.x86_64 (netkey)
See `ipsec --copyright' for copyright information.
Checking for IPsec support in kernel                     [OK]
 NETKEY: Testing XFRM related proc values
         ICMP default/send_redirects                     [OK]
         ICMP default/accept_redirects                   [OK]
         XFRM larval drop                                [OK]
Hardware random device check                            [N/A]
Two or more interfaces found, checking IP forwarding     [OK]
Checking rp_filter                                      [ENABLED]
 /proc/sys/net/ipv4/conf/all/rp_filter                   [ENABLED]
 /proc/sys/net/ipv4/conf/lo/rp_filter                    [ENABLED]
 /proc/sys/net/ipv4/conf/eth0/rp_filter                  [ENABLED]
 /proc/sys/net/ipv4/conf/pan0/rp_filter                  [ENABLED]
Checking that pluto is running                           [OK]
 Pluto listening for IKE on udp 500                      [OK]
 Pluto listening for IKE on tcp 500                      [NOT IMPLEMENTED]
 Pluto listening for IKE/NAT-T on udp 4500               [OK]
 Pluto listening for IKE/NAT-T on tcp 4500               [NOT IMPLEMENTED]
 Pluto listening for IKE on tcp 10000 (cisco)            [NOT IMPLEMENTED]
Checking NAT and MASQUERADEing                           [TEST INCOMPLETE]
Checking 'ip' command                                    [OK]
Checking 'iptables' command                              [OK]
```

Note: The result for `Checking 'ip' command` may be `[IP XFRM BROKEN]`, depending on the version of iproute on your system. This can be a misdiagnosis caused by a change in IP XFRM message output between iproute versions.

7. Enable starting IPsec on boot with the following command:

```
# chkconfig ipsec on
```

Configuring Openswan Connections

Set up an IPsec tunnel between each pair of hosts on the cluster by creating a connection. On each connection, one host is designated the "left" host and the other the "right" host. For example, if you have a master (mdw), a standby master (smdw), and three segment hosts (sdw1, sdw2, sdw3), you will need ten connections, as shown in the following table.

Table 31: IPsec connections for a five-host cluster

Connection Number	Left host	Right host
1	mdw	smdw
2	mdw	sdw1
3	mdw	sdw2
4	mdw	sdw3
5	smdw	sdw1
6	smdw	sdw2
7	smdw	sdw3

Connection Number	Left host	Right host
8	sdw1	sdw2
9	sdw1	sdw3
10	sdw2	sdw3

Complete these tasks to configure connections:

- *Create Host Keys*
- *Create a Connection Configuration File*
- *Test the IPsec Connection*

Create Host Keys

To enable RSA public key authentication for the tunnels, each host must have an RSA key pair. As the root user on each host, enter the following command to generate an authentication key.

```
# ipsec newhostkey --output /etc/ipsec.d/ipsec.secrets --bits 4096
```

The key is saved in the `/etc/ipsec.d/ipsec.secrets` file and its attributes are set to allow access to the root user only.

To view a host's public key, use the `ipsec showhostkey` command with the `--left` or `--right` option. The command outputs the public key in a format suitable for pasting into the connection configuration file. In this example, the keys are shortened for readability:

```
# ipsec showhostkey --left
 # rsakey AQOW+RwpL
 leftrsasigkey=0sAQOW+RwpLg7CGoyywCnv+vnasGJI7...
# ipsec showhostkey --right
 # rsakey AQOW+RwpL
 rightrsasigkey=0sAQOW+RwpLg7CGoyywCnv+vnasGJI7...
```

You will need to use this command as you configure the tunnel between each host pair.

Create a Connection Configuration File

IPsec tunnels are configured by creating a `conn` section in the `/etc/ipsec.conf` file. Because we added the `include /etc/ipsec.d/*.conf` directive to `/etc/ipsec.conf`, we can create a `.conf` file for each connection.

Follow these steps to configure a connection for each pair of hosts.

1. Log in to the host that will be on the "left" side of the tunnel.
2. Create a new configuration file in the `/etc/ipsec.d` directory. Choose a name that includes both host names and has a `.conf` extension. The following configuration file, named `mdw-sdw1.conf`, configures the connection between the hosts mdw and sdw1:

```
conn mdw-sdw1
    leftid=mdw
    left=192.0.2.214
    leftrsasigkey=0sAQOW+RwpLg7CGoyywCnv+vnasGJI7... # shortened for readability
    rightid=sdw1
    right=192.0.2.215
    rightrsasigkey=0sAQNfdDCoDte5bGaGLGkHTKa5GMRl... # shortened for readability
    type=tunnel
    authby=rsasig
    ike=aes192-sha2;dh20
    phase2alg=aes_gcm_c-160-null
    auto=start
```

See the *ipsec.conf* man page for the complete list of available parameters and their default values.

The connection name in the example is `mdw-sdw1`.

For the `leftrsasigkey` use the output from running `ipsec showhostkey --left` on the "left" host. For `rightrsasigkey` use the output from running `ipsec showhostkey --right` on the "right" host.

Following are recommendations for configuring parameters for Greenplum Database IPsec connections to obtain the best security and performance:

type

Set to `tunnel`, the default mode.

authby

Set to `rsasig`. This is more secure than using pre-shared keys (`psk`).

auto

Set to `start` so that the tunnel is brought up when IPsec starts up.

ike

The `ike` parameter is used during phase 1 to authenticate the peers and negotiate secure session keys for phase2. The parameter value is an entry in the format:

```
cipher-hash;modpgroup, cipher-hash;modpgroup, ....
```

- *cipher* is an encryption algorithm. AES is more secure than 3DES, which is more secure than DES. AES has length of 128, 192, or 256 bits. The more bits, the stronger the encryption, but more time is required for computation.
- *hash* is the hash algorithm. SHA2 is stronger than SHA1, which is stronger than MD5. SHA2 is recommended, but if SHA2 is not supported on the device, use SHA1. SHA2 is a family of hash functions—SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256—not all of which are supported by Openswan. To find out which algorithms Openswan supports, run the following command after starting the `ipsec` service:

```
# ipsec auto -status
000 algorithm IKE hash: id=2, name=OAKLEY_SHA1, hashsize=20
000 algorithm IKE hash: id=4, name=OAKLEY_SHA2_256, hashsize=32
000 algorithm IKE hash: id=6, name=OAKLEY_SHA2_512, hashsize=64
000 algorithm ESP encrypt: id=2, name=ESP_DES, ivlen=8,
 keysizemin=64, keysizemax=64
000 algorithm ESP encrypt: id=3, name=ESP_3DES, ivlen=8,
 keysizemin=192, keysizemax=192
000 algorithm ESP encrypt: id=18, name=ESP_AES_GCM_A, ivlen=8,
 keysizemin=160,
      keysizemax=288000 algorithm ESP encrypt: id=19,
 name=ESP_AES_GCM_B, ivlen=12, keysizemin=160,
      keysizemax=288
000 algorithm ESP encrypt: id=20, name=ESP_AES_GCM_C, ivlen=16,
 keysizemin=160,
      keysizemax=288
```

  See http://en.wikipedia.org/wiki/SHA-2 for information about SHA2.
- *modpgroup* is the Diffie-Hellman group. The peers negotiate a shared secret using the Diffie-Hellman protocol. The Diffie-Hellman group is a set of standardized parameters the peers agree to use as the basis for their calculations. The groups are numbered, and higher numbered groups are more secure (and more compute-intensive) than lower numbered groups. Avoid the lowest numbered groups: 1 (`modp768`), 3 (`modp1024`), and 5 (`modp1576`), which are not considered secure. Choose a higher level group, such as `dh14`, `dh15`, `dh19`, `dh20`, `dh21`, or `dh24`.

phase2

Set to `esp`, the default, to encrypt data. The `ah` setting creates a connection that authenticates, but does not encrypt IP packets.

phase2alg

The `phase2alg` parameter specifies algorithms to use for encrypting and authenticating data. The format and defaults are the same as for the `ike` parameter.

The AES cipher and SHA hash algorithm are more secure. For effective use of emerging 10-gigabit and 40-gigabit network devices, and to enable high speed communication channels, the AES_GCM algorithm is currently the recommended best option. To use AES_GCM, verify that the CPU supports the AES_NI instruction set. See *AES instruction set* for a list of CPUs that support AES_NI.

To see if the CPU supports AES-NI, see if the `aes` flag is set in `/proc/cpuinfo`:

```
grep aes /proc/cpuinfo
```

To see if AES-N1 has been *enabled*, search `/proc/crypto` for the module:

```
grep module /proc/crypto | sort -u
```

To see if the `aesni_intel` kernel module is loaded:

```
/sbin/modinfo aesni_intel
```

To specify the AES_GCM algorithm, use the following syntax:

```
phase2alg=aes_gcm_c-160-null
```

Openswan requires adding the salt size (32 bits) to the key size (128, 192, or 256 bits). In the example above, "160" is calculated by adding a 128-bit key size to the 32 bit salt size. The other valid values are 224 and 288.

3. Use `scp` to copy the configuration file to the "right" host. For example:

   ```
   # scp /etc/ipsec.d/mdw-sdw1.conf sdw1:/etc/ipsec.d/
   ```

4. Ensure that IPsec is started by executing the following command on both the "left" and "right" hosts:

   ```
   # ipsec service start
   ```

5. Load the tunnel on both left and right hosts with the following command:

   ```
   # ipsec auto --add mdw-sdw
   ```

6. Bring up the tunnel on both left and right hosts with the following command:

   ```
   # ipsec auto --up mdw-sdw
   ```

Test the IPsec Connection

To verify IPsec packets are flowing through a network interface, run the following `tcdump` command on one host and then ping that host from another host.

```
tcdump -n -i interface_name host hostname
```

For example, run the `tcpdump` command on sdw1 and then, on mdw, `ping sdw2`:

```
# tcpdump -n -i eth0 host mdw
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
08:22:10.186743 IP 192.0.2.214 > 192.0.2.215: ESP(spi=0xe56f19ea,seq=0x1), length 132
```

```
08:22:10.186808 IP 192.0.2.214 > 192.0.2.215: ICMP echo request, id 30987, seq 1,
 length 64
08:22:10.186863 IP 192.0.2.215 > 192.0.2.214: ESP(spi=0x4e55824c,seq=0x1), length 132
08:22:11.189663 IP 192.0.2.214 > 192.0.2.215: ESP(spi=0xe56f19ea,seq=0x2), length 132
08:22:11.189707 IP 192.0.2.214 > 192.0.2.215: ICMP echo request, id 30987, seq 2,
 length 64
```

The ESP packets verify that the IP packets are encrypted and encapsulated.

When you have connections set up between all of the hosts in the cluster and Greenplum Database is running, you can run the `tcpdump` command on segment hosts to observe database activity in the IPsec tunnels.

Part IV

Working with Databases

The topics in this section describe how to create and manage database objects and manipulate data in the database. Additional topics describe parallel data loading and writing SQL queries in the Greenplum MPP environment.

This section contains the following topics:

- *Defining Database Objects*
- *Managing Data*
- *Loading and Unloading Data*
- *Querying Data*

Chapter 20

Defining Database Objects

This section covers data definition language (DDL) in Greenplum Database and how to create and manage database objects.

Creating objects in a Greenplum Database includes making up-front choices about data distribution, storage options, data loading, and other Greenplum features that will affect the ongoing performance of your database system. Understanding the options that are available and how the database will be used will help you make the right decisions.

Most of the advanced Greenplum features are enabled with extensions to the SQL `CREATE` DDL statements.

Creating and Managing Databases

A Greenplum Database system is a single instance of Greenplum Database. There can be several separate Greenplum Database systems installed, but usually just one is selected by environment variable settings. See your Greenplum administrator for details.

There can be multiple databases in a Greenplum Database system. This is different from some database management systems (such as Oracle) where the database instance *is* the database. Although you can create many databases in a Greenplum system, client programs can connect to and access only one database at a time — you cannot cross-query between databases.

About Template Databases

Each new database you create is based on a *template*. Greenplum provides a default database, *template1*. Use *template1* to connect to Greenplum Database for the first time. Greenplum Database uses *template1* to create databases unless you specify another template. Do not create any objects in *template1* unless you want those objects to be in every database you create.

Greenplum Database uses two other database templates, *template0* and *postgres*, internally. Do not drop or modify *template0* or *postgres*. You can use *template0* to create a completely clean database containing only the standard objects predefined by Greenplum Database at initialization, especially if you modified *template1*.

Creating a Database

The `CREATE DATABASE` command creates a new database. For example:

```
=> CREATE DATABASE new_dbname;
```

To create a database, you must have privileges to create a database or be a Greenplum Database superuser. If you do not have the correct privileges, you cannot create a database. Contact your Greenplum Database administrator to either give you the necessary privilege or to create a database for you.

You can also use the client program `createdb` to create a database. For example, running the following command in a command line terminal connects to Greenplum Database using the provided host name and port and creates a database named *mydatabase*:

```
$ createdb -h masterhost -p 5432 mydatabase
```

The host name and port must match the host name and port of the installed Greenplum Database system.

Some objects, such as roles, are shared by all the databases in a Greenplum Database system. Other objects, such as tables that you create, are known only in the database in which you create them.

Cloning a Database

By default, a new database is created by cloning the standard system database template, *template1*. Any database can be used as a template when creating a new database, thereby providing the capability to 'clone' or copy an existing database and all objects and data within that database. For example:

```
=> CREATE DATABASE new_dbname TEMPLATE old_dbname;
```

Viewing the List of Databases

If you are working in the psql client program, you can use the \l meta-command to show the list of databases and templates in your Greenplum Database system. If using another client program and you are a superuser, you can query the list of databases from the pg_database system catalog table. For example:

```
=> SELECT datname from pg_database;
```

Altering a Database

The ALTER DATABASE command changes database attributes such as owner, name, or default configuration attributes. For example, the following command alters a database by setting its default schema search path (the search_path configuration parameter):

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

To alter a database, you must be the owner of the database or a superuser.

Dropping a Database

The DROP DATABASE command drops (or deletes) a database. It removes the system catalog entries for the database and deletes the database directory on disk that contains the data. You must be the database owner or a superuser to drop a database, and you cannot drop a database while you or anyone else is connected to it. Connect to template1 (or another database) before dropping a database. For example:

```
=> \c template1
=> DROP DATABASE mydatabase;
```

You can also use the client program dropdb to drop a database. For example, the following command connects to Greenplum Database using the provided host name and port and drops the database *mydatabase*:

```
$ dropdb -h masterhost -p 5432 mydatabase
```

Warning: Dropping a database cannot be undone.

Creating and Managing Tablespaces

Tablespaces allow database administrators to have multiple file systems per machine and decide how to best use physical storage to store database objects. They are named locations within a filespace in which you can create objects. Tablespaces allow you to assign different storage for frequently and infrequently used database objects or to control the I/O performance on certain database objects. For example, place frequently-used tables on file systems that use high performance solid-state drives (SSD), and place other tables on standard hard drives.

A tablespace requires a file system location to store its database files. In Greenplum Database, the master and each segment (primary and mirror) require a distinct storage location. The collection of file system locations for all components in a Greenplum system is a *filespace*. Filespaces can be used by one or more tablespaces.

Creating a Filespace

A filespace sets aside storage for your Greenplum system. A filespace is a symbolic storage identifier that maps onto a set of locations in your Greenplum hosts' file systems. To create a filespace, prepare the logical file systems on all of your Greenplum hosts, then use the `gpfilespace` utility to define the filespace. You must be a database superuser to create a filespace.

> **Note:** Greenplum Database is not directly aware of the file system boundaries on your underlying systems. It stores files in the directories that you tell it to use. You cannot control the location on disk of individual files within a logical file system.

To create a filespace using gpfilespace

1. Log in to the Greenplum Database master as the `gpadmin` user.

```
$ su - gpadmin
```

2. Create a filespace configuration file:

```
$ gpfilespace -o gpfilespace_config
```

3. At the prompt, enter a name for the filespace, the primary segment file system locations, the mirror segment file system locations, and a master file system location. Primary and mirror locations refer to directories on segment hosts; the master location refers to a directory on the master host and standby master, if configured. For example, if your configuration has 2 primary and 2 mirror segments per host:

```
Enter a name for this filespace> fastdisk
primary location 1> /gpfs1/seg1
primary location 2> /gpfs1/seg2
mirror location 1> /gpfs2/mir1
mirror location 2> /gpfs2/mir2
master location> /gpfs1/master
```

4. gpfilespace creates a configuration file. Examine the file to verify that the gpfilespace configuration is correct.
5. Run gpfilespace again to create the filespace based on the configuration file:

```
$ gpfilespace -c gpfilespace_config
```

Moving the Location of Temporary or Transaction Files

You can move temporary or transaction files to a specific filespace to improve database performance when running queries, creating backups, and to store data more sequentially.

The dedicated filespace for temporary and transaction files is tracked in two separate flat files called gp_temporary_files_filespace and gp_transaction_files_filespace. These are located in the pg_system directory on each primary and mirror segment, and on master and standby. You must be a superuser to move temporary or transaction files. Only the `gpfilespace` utility can write to this file.

About Temporary and Transaction Files

Unless otherwise specified, temporary and transaction files are stored together with all user data. The default location of temporary files, *<filespace_directory>*/*<tablespace_oid>*/*<database_oid>*/pgsql_tmp is changed when you use `gpfilespace --movetempfiles` for the first time.

Also note the following information about temporary or transaction files:

- You can dedicate only one filespace for temporary or transaction files, although you can use the same filespace to store other types of files.
- You cannot drop a filespace if it used by temporary files.
- You must create the filespace in advance. See *Creating a Filespace*.

To move temporary files using gpfilespace

1. Check that the filespace exists and is different from the filespace used to store all other user data.
2. Issue smart shutdown to bring the Greenplum Database offline.

   If any connections are still in progess,the gpfilespace --movetempfiles utility will fail.
3. Bring Greenplum Database online with no active session and run the following command:

   ```
   gpfilespace --movetempfilespace filespace_name
   ```

   The location of the temporary files is stored in the segment configuration shared memory (PMModuleState) and used whenever temporary files are created, opened, or dropped.

To move transaction files using gpfilespace

1. Check that the filespace exists and is different from the filespace used to store all other user data.
2. Issue smart shutdown to bring the Greenplum Database offline.

   If any connections are still in progess,the `gpfilespace --movetransfiles` utility will fail.
3. Bring Greenplum Database online with no active session and run the following command:

   ```
   gpfilespace --movetransfilespace filespace_name
   ```

   The location of the transaction files is stored in the segment configuration shared memory (PMModuleState) and used whenever transaction files are created, opened, or dropped.

Creating a Tablespace

After you create a filespace, use the `CREATE TABLESPACE` command to define a tablespace that uses that filespace. For example:

```
=# CREATE TABLESPACE fastspace FILESPACE fastdisk;
```

Database superusers define tablespaces and grant access to database users with the `GRANTCREATE` command. For example:

```
=# GRANT CREATE ON TABLESPACE fastspace TO admin;
```

Using a Tablespace to Store Database Objects

Users with the `CREATE` privilege on a tablespace can create database objects in that tablespace, such as tables, indexes, and databases. The command is:

```
CREATE TABLE tablename(options) TABLESPACE spacename
```

For example, the following command creates a table in the tablespace *space1*:

```
CREATE TABLE foo(i int) TABLESPACE space1;
```

You can also use the `default_tablespace` parameter to specify the default tablespace for `CREATE TABLE` and `CREATE INDEX` commands that do not specify a tablespace:

```
SET default_tablespace = space1;
CREATE TABLE foo(i int);
```

The tablespace associated with a database stores that database's system catalogs, temporary files created by server processes using that database, and is the default tablespace selected for tables and indexes created within the database, if no `TABLESPACE` is specified when the objects are created. If you do not specify a tablespace when you create a database, the database uses the same tablespace used by its template database.

You can use a tablespace from any database if you have appropriate privileges.

Viewing Existing Tablespaces and Filespaces

Every Greenplum Database system has the following default tablespaces.

- `pg_global` for shared system catalogs.
- `pg_default`, the default tablespace. Used by the *template1* and *template0* databases.

These tablespaces use the system default filespace, `pg_system`, the data directory location created at system initialization.

To see filespace information, look in the *pg_filespace* and *pg_filespace_entry* catalog tables. You can join these tables with *pg_tablespace* to see the full definition of a tablespace. For example:

```
=# SELECT spcname as tblspc, fsname as filespc,
          fsedbid as seg_dbid, fselocation as datadir
   FROM   pg_tablespace pgts, pg_filespace pgfs,
          pg_filespace_entry pgfse
   WHERE  pgts.spcfsoid=pgfse.fsefsoid
          AND pgfse.fsefsoid=pgfs.oid
   ORDER BY tblspc, seg_dbid;
```

Dropping Tablespaces and Filespaces

To drop a tablespace, you must be the tablespace owner or a superuser. You cannot drop a tablespace until all objects in all databases using the tablespace are removed.

Only a superuser can drop a filespace. A filespace cannot be dropped until all tablespaces using that filespace are removed.

The `DROP TABLESPACE` command removes an empty tablespace.

The `DROP FILESPACE` command removes an empty filespace.

Note: You cannot drop a filespace if it stores temporary or transaction files.

Creating and Managing Schemas

Schemas logically organize objects and data in a database. Schemas allow you to have more than one object (such as tables) with the same name in the database without conflict if the objects are in different schemas.

The Default "Public" Schema

Every database has a default schema named *public*. If you do not create any schemas, objects are created in the *public* schema. All database roles (users) have `CREATE` and `USAGE` privileges in the *public* schema. When you create a schema, you grant privileges to your users to allow access to the schema.

Creating a Schema

Use the `CREATE SCHEMA` command to create a new schema. For example:

```
=> CREATE SCHEMA myschema;
```

To create or access objects in a schema, write a qualified name consisting of the schema name and table name separated by a period. For example:

```
myschema.table
```

See *Schema Search Paths* for information about accessing a schema.

You can create a schema owned by someone else, for example, to restrict the activities of your users to well-defined namespaces. The syntax is:

```
=> CREATE SCHEMA schemaname AUTHORIZATION username;
```

Schema Search Paths

To specify an object's location in a database, use the schema-qualified name. For example:

```
=> SELECT * FROM myschema.mytable;
```

You can set the `search_path` configuration parameter to specify the order in which to search the available schemas for objects. The schema listed first in the search path becomes the *default* schema. If a schema is not specified, objects are created in the default schema.

Setting the Schema Search Path

The `search_path` configuration parameter sets the schema search order. The `ALTER DATABASE` command sets the search path. For example:

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

You can also set `search_path` for a particular role (user) using the `ALTER ROLE` command. For example:

```
=> ALTER ROLE sally SET search_path TO myschema, public,
pg_catalog;
```

Viewing the Current Schema

Use the `current_schema()` function to view the current schema. For example:

```
=> SELECT current_schema();
```

Use the `SHOW` command to view the current search path. For example:

```
=> SHOW search_path;
```

Dropping a Schema

Use the `DROP SCHEMA` command to drop (delete) a schema. For example:

```
=> DROP SCHEMA myschema;
```

By default, the schema must be empty before you can drop it. To drop a schema and all of its objects (tables, data, functions, and so on) use:

```
=> DROP SCHEMA myschema CASCADE;
```

System Schemas

The following system-level schemas exist in every database:

- `pg_catalog` contains the system catalog tables, built-in data types, functions, and operators. It is always part of the schema search path, even if it is not explicitly named in the search path.
- `information_schema` consists of a standardized set of views that contain information about the objects in the database. These views get system information from the system catalog tables in a standardized way.
- `pg_toast` stores large objects such as records that exceed the page size. This schema is used internally by the Greenplum Database system.
- `pg_bitmapindex` stores bitmap index objects such as lists of values. This schema is used internally by the Greenplum Database system.
- `pg_aoseg` stores append-optimized table objects. This schema is used internally by the Greenplum Database system.
- `gp_toolkit` is an administrative schema that contains external tables, views, and functions that you can access with SQL commands. All database users can access `gp_toolkit` to view and query the system log files and other system metrics.

Creating and Managing Tables

Greenplum Database tables are similar to tables in any relational database, except that table rows are distributed across the different segments in the system. When you create a table, you specify the table's distribution policy.

Creating a Table

The `CREATE TABLE` command creates a table and defines its structure. When you create a table, you define:

- The columns of the table and their associated data types. See *Choosing Column Data Types*.
- Any table or column constraints to limit the data that a column or table can contain. See *Setting Table and Column Constraints*.
- The distribution policy of the table, which determines how Greenplum Database divides data is across the segments. See *Choosing the Table Distribution Policy*.
- The way the table is stored on disk. See *Choosing the Table Storage Model*.
- The table partitioning strategy for large tables. See *Creating and Managing Databases*.

Choosing Column Data Types

The data type of a column determines the types of data values the column can contain. Choose the data type that uses the least possible space but can still accommodate your data and that best constrains the data. For example, use character data types for strings, date or timestamp data types for dates, and numeric data types for numbers.

For table columns that contain textual data, Pivotal recommends specifying the data type `VARCHAR` or `TEXT`. Specifying the data type `CHAR` is not recommended. In Greenplum Database, the data types `VARCHAR` or `TEXT` handles padding added to the data (space characters added after the last non-space character) as significant characters, the data type `CHAR` does not. For information on the character data types, see the `CREATE TABLE` command in the *Greenplum Database Reference Guide*.

Use the smallest numeric data type that will accommodate your numeric data and allow for future expansion. For example, using `BIGINT` for data that fits in `INT` or `SMALLINT` wastes storage space. If you expect that your data values will expand over time, consider that changing from a smaller datatype to a larger datatype after loading large amounts of data is costly. For example, if your current data values fit in a `SMALLINT` but it is likely that the values will expand, `INT` is the better long-term choice.

Use the same data types for columns that you plan to use in cross-table joins. Cross-table joins usually use the primary key in one table and a foreign key in the other table. When the data types are different, the database must convert one of them so that the data values can be compared correctly, which adds unnecessary overhead.

Greenplum Database has a rich set of native data types available to users. See the *Greenplum Database Reference Guide* for information about the built-in data types.

Setting Table and Column Constraints

You can define constraints on columns and tables to restrict the data in your tables. Greenplum Database support for constraints is the same as PostgreSQL with some limitations, including:

- `CHECK` constraints can refer only to the table on which they are defined.
- `UNIQUE` and `PRIMARY KEY` constraints must be compatible with their table’s distribution key and partitioning key, if any.

Note: UNIQUE and PRIMARY KEY constraints are not allowed on append-optimized tables because the UNIQUE indexes that are created by the constraints are not allowed on append-optimized tables.

- FOREIGN KEY constraints are allowed, but not enforced.
- Constraints that you define on partitioned tables apply to the partitioned table as a whole. You cannot define constraints on the individual parts of the table.

Check Constraints

Check constraints allow you to specify that the value in a certain column must satisfy a Boolean (truth-value) expression. For example, to require positive product prices:

```
=> CREATE TABLE products
            ( product_no integer,
              name text,
              price numeric CHECK (price > 0) );
```

Not-Null Constraints

Not-null constraints specify that a column must not assume the null value. A not-null constraint is always written as a column constraint. For example:

```
=> CREATE TABLE products
       ( product_no integer NOT NULL,
         name text NOT NULL,
         price numeric );
```

Unique Constraints

Unique constraints ensure that the data contained in a column or a group of columns is unique with respect to all the rows in the table. The table must be hash-distributed (not DISTRIBUTED RANDOMLY), and the constraint columns must be the same as (or a superset of) the table's distribution key columns. For example:

```
=> CREATE TABLE products
       ( product_no integer UNIQUE,
         name text,
         price numeric)
      DISTRIBUTED BY (product_no);
```

Primary Keys

A primary key constraint is a combination of a UNIQUE constraint and a NOT NULL constraint. The table must be hash-distributed (not DISTRIBUTED RANDOMLY), and the primary key columns must be the same as (or a superset of) the table's distribution key columns. If a table has a primary key, this column (or group of columns) is chosen as the distribution key for the table by default. For example:

```
=> CREATE TABLE products
       ( product_no integer PRIMARY KEY,
         name text,
         price numeric)
      DISTRIBUTED BY (product_no);
```

Foreign Keys

Foreign keys are not supported. You can declare them, but referential integrity is not enforced.

Foreign key constraints specify that the values in a column or a group of columns must match the values appearing in some row of another table to maintain referential integrity between two related tables.

Referential integrity checks cannot be enforced between the distributed table segments of a Greenplum database.

Choosing the Table Distribution Policy

All Greenplum Database tables are distributed. When you create or alter a table, you optionally specify `DISTRIBUTED BY` (hash distribution) or `DISTRIBUTED RANDOMLY` (round-robin distribution) to determine the table row distribution.

> **Note:** The Greenplum Database server configuration parameter `gp_create_table_random_default_distribution` controls the table distribution policy if the `DISTRIBUTED BY` clause is not specified when you create a table.
>
> For information about the parameter, see "Server Configuration Parameters" of the *Greenplum Database Reference Guide*.

Consider the following points when deciding on a table distribution policy.

- **Even Data Distribution** — For the best possible performance, all segments should contain equal portions of data. If the data is unbalanced or skewed, the segments with more data must work harder to perform their portion of the query processing. Choose a distribution key that is unique for each record, such as the primary key.
- **Local and Distributed Operations** — Local operations are faster than distributed operations. Query processing is fastest if the work associated with join, sort, or aggregation operations is done locally, at the segment level. Work done at the system level requires distributing tuples across the segments, which is less efficient. When tables share a common distribution key, the work of joining or sorting on their shared distribution key columns is done locally. With a random distribution policy, local join operations are not an option.
- **Even Query Processing** — For best performance, all segments should handle an equal share of the query workload. Query workload can be skewed if a table's data distribution policy and the query predicates are not well matched. For example, suppose that a sales transactions table is distributed on the customer ID column (the distribution key). If a predicate in a query references a single customer ID, the query processing work is concentrated on just one segment.

Declaring Distribution Keys

`CREATE TABLE`'s optional clauses `DISTRIBUTED BY` and `DISTRIBUTED RANDOMLY` specify the distribution policy for a table. The default is a hash distribution policy that uses either the `PRIMARY KEY` (if the table has one) or the first column of the table as the distribution key. Columns with geometric or user-defined data types are not eligible as Greenplum distribution key columns. If a table does not have an eligible column, Greenplum distributes the rows randomly or in round-robin fashion.

To ensure even distribution of data, choose a distribution key that is unique for each record. If that is not possible, choose `DISTRIBUTED RANDOMLY`. For example:

```
=> CREATE TABLE products
                        (name varchar(40),
                         prod_id integer,
                         supplier_id integer)
             DISTRIBUTED BY (prod_id);
```

```
=> CREATE TABLE random_stuff
                        (things text,
                         doodads text,
                         etc text)
             DISTRIBUTED RANDOMLY;
```

Choosing the Table Storage Model

Greenplum Database supports several storage models and a mix of storage models. When you create a table, you choose how to store its data. This topic explains the options for table storage and how to choose the best storage model for your workload.

- *Heap Storage*
- *Append-Optimized Storage*
- *Choosing Row or Column-Oriented Storage*
- *Using Compression (Append-Optimized Tables Only)*
- *Checking the Compression and Distribution of an Append-Optimized Table*
- *Altering a Table*
- *Dropping a Table*

> **Note:** To simplify the creation of database tables, you can specify the default values for some table storage options with the Greenplum Database server configuration parameter `gp_default_storage_options`.
>
> For information about the parameter, see "Server Configuration Parameters" in the *Greenplum Database Reference Guide*.

Heap Storage

By default, Greenplum Database uses the same heap storage model as PostgreSQL. Heap table storage works best with OLTP-type workloads where the data is often modified after it is initially loaded. `UPDATE` and `DELETE` operations require storing row-level versioning information to ensure reliable database transaction processing. Heap tables are best suited for smaller tables, such as dimension tables, that are often updated after they are initially loaded.

Append-Optimized Storage

Append-optimized table storage works best with denormalized fact tables in a data warehouse environment. Denormalized fact tables are typically the largest tables in the system. Fact tables are usually loaded in batches and accessed by read-only queries. Moving large fact tables to an append-optimized storage model eliminates the storage overhead of the per-row update visibility information, saving about 20 bytes per row. This allows for a leaner and easier-to-optimize page structure. The storage model of append-optimized tables is optimized for bulk data loading. Single row `INSERT` statements are not recommended.

To create a heap table

Row-oriented heap tables are the default storage type.

```
=> CREATE TABLE foo (a int, b text) DISTRIBUTED BY (a);
```

To create an append-optimized table

Use the `WITH` clause of the `CREATE TABLE` command to declare the table storage options. The default is to create the table as a regular row-oriented heap-storage table. For example, to create an append-optimized table with no compression:

```
=> CREATE TABLE bar (a int, b text)
    WITH (appendonly=true)
    DISTRIBUTED BY (a);
```

UPDATE and DELETE are not allowed on append-optimized tables in a serializable transaction and will cause the transaction to abort. CLUSTER, DECLARE...FOR UPDATE, and triggers are not supported with append-optimized tables.

Choosing Row or Column-Oriented Storage

Greenplum provides a choice of storage orientation models: row, column, or a combination of both. This topic provides general guidelines for choosing the optimum storage orientation for a table. Evaluate performance using your own data and query workloads.

- Row-oriented storage: good for OLTP types of workloads with many iterative transactions and many columns of a single row needed all at once, so retrieving is efficient.
- Column-oriented storage: good for data warehouse workloads with aggregations of data computed over a small number of columns, or for single columns that require regular updates without modifying other column data.

For most general purpose or mixed workloads, row-oriented storage offers the best combination of flexibility and performance. However, there are use cases where a column-oriented storage model provides more efficient I/O and storage. Consider the following requirements when deciding on the storage orientation model for a table:

- **Updates of table data.** If you load and update the table data frequently, choose a row-oriented heap table. Column-oriented table storage is only available on append-optimized tables.

  See *Heap Storage* for more information.
- **Frequent INSERTs.** If rows are frequently inserted into the table, consider a row-oriented model. Column-oriented tables are not optimized for write operations, as column values for a row must be written to different places on disk.
- **Number of columns requested in queries.** If you typically request all or the majority of columns in the SELECT list or WHERE clause of your queries, consider a row-oriented model. Column-oriented tables are best suited to queries that aggregate many values of a single column where the WHERE or HAVING predicate is also on the aggregate column. For example:

  ```
  SELECT SUM(salary)...
  ```

  ```
  SELECT AVG(salary)... WHERE salary > 10000
  ```

  Or where the WHERE predicate is on a single column and returns a relatively small number of rows. For example:

  ```
  SELECT salary, dept ... WHERE state='CA'
  ```
- **Number of columns in the table.** Row-oriented storage is more efficient when many columns are required at the same time, or when the row-size of a table is relatively small. Column-oriented tables can offer better query performance on tables with many columns where you access a small subset of columns in your queries.
- **Compression.** Column data has the same data type, so storage size optimizations are available in column-oriented data that are not available in row-oriented data. For example, many compression schemes use the similarity of adjacent data to compress. However, the greater adjacent compression achieved, the more difficult random access can become, as data must be uncompressed to be read.

To create a column-oriented table

The WITH clause of the CREATE TABLE command specifies the table's storage options. The default is a row-oriented heap table. Tables that use column-oriented storage must be append-optimized tables. For example, to create a column-oriented table:

```
=> CREATE TABLE bar (a int, b text)
```

```
    WITH (appendonly=true, orientation=column)
    DISTRIBUTED BY (a);
```

Using Compression (Append-Optimized Tables Only)

There are two types of in-database compression available in the Greenplum Database for append-optimized tables:

- Table-level compression is applied to an entire table.
- Column-level compression is applied to a specific column. You can apply different column-level compression algorithms to different columns.

The following table summarizes the available compression algorithms.

Table 32: Compression Algorithms for Append-Optimized Tables

Table Orientation	Available Compression Types	Supported Algorithms
Row	Table	`ZLIB` and `QUICKLZ`
Column	Column and Table	`RLE_TYPE`, `ZLIB`, and `QUICKLZ`

When choosing a compression type and level for append-optimized tables, consider these factors:

- CPU usage. Your segment systems must have the available CPU power to compress and uncompress the data.
- Compression ratio/disk size. Minimizing disk size is one factor, but also consider the time and CPU capacity required to compress and scan data. Find the optimal settings for efficiently compressing data without causing excessively long compression times or slow scan rates.
- Speed of compression. QuickLZ compression generally uses less CPU capacity and compresses data faster at a lower compression ratio than zlib. zlib provides higher compression ratios at lower speeds.

  For example, at compression level 1 (`compresslevel=1`), QuickLZ and zlib have comparable compression ratios, though at different speeds. Using zlib with `compresslevel=6` can significantly increase the compression ratio compared to QuickLZ, though with lower compression speed.
- Speed of decompression/scan rate. Performance with compressed append-optimized tables depends on hardware, query tuning settings, and other factors. Perform comparison testing to determine the actual performance in your environment.

> **Note:** Do not create compressed append-optimized tables on file systems that use compression. If the file system on which your segment data directory resides is a compressed file system, your append-optimized table must not use compression.

Performance with compressed append-optimized tables depends on hardware, query tuning settings, and other factors. Greenplum recommends performing comparison testing to determine the actual performance in your environment.

> **Note:** QuickLZ compression level can only be set to level 1; no other options are available. Compression level with zlib can be set at values from 1 - 9. Compression level with RLE can be set at values from 1 - 4.
>
> An `ENCODING` clause specifies compression type and level for individual columns. When an `ENCODING` clause conflicts with a `WITH` clause, the `ENCODING` clause has higher precedence than the `WITH` clause.

To create a compressed table

The WITH clause of the CREATE TABLE command declares the table storage options. Tables that use compression must be append-optimized tables. For example, to create an append-optimized table with zlib compression at a compression level of 5:

```
=> CREATE TABLE foo (a int, b text)
   WITH (appendonly=true, compresstype=zlib, compresslevel=5);
```

Checking the Compression and Distribution of an Append-Optimized Table

Greenplum provides built-in functions to check the compression ratio and the distribution of an append-optimized table. The functions take either the object ID or a table name. You can qualify the table name with a schema name.

Table 33: Functions for compressed append-optimized table metadata

Function	Return Type	Description

get_ao_distribution(name)
get_ao_distribution(oid)
	Set of (dbid, tuplecount) rows	Shows the distribution of an append-optimized table's rows across the array. Returns a set of rows, each of which includes a segment *dbid* and the number of tuples stored on the segment.

get_ao_compression_ratio(name)
get_ao_compression_ratio(oid)
	float8	Calculates the compression ratio for a compressed append-optimized table. If information is not available, this function returns a value of -1.

The compression ratio is returned as a common ratio. For example, a returned value of 3.19, or 3.19:1, means that the uncompressed table is slightly larger than three times the size of the compressed table.

The distribution of the table is returned as a set of rows that indicate how many tuples are stored on each segment. For example, in a system with four primary segments with *dbid* values ranging from 0 - 3, the function returns four rows similar to the following:

```
=# SELECT get_ao_distribution('lineitem_comp');
 get_ao_distribution
---------------------
(0,7500721)
(1,7501365)
(2,7499978)
(3,7497731)
(4 rows)
```

Support for Run-length Encoding

Greenplum Database supports Run-length Encoding (RLE) for column-level compression. RLE data compression stores repeated data as a single data value and a count. For example, in a table with two columns, a date and a description, that contains 200,000 entries containing the value date1 and 400,000 entries containing the value date2, RLE compression for the date field is similar to date1 200000 date2 400000. RLE is not useful with files that do not have large sets of repeated data as it can greatly increase the file size.

There are four levels of RLE compression available. The levels progressively increase the compression ratio, but decrease the compression speed.

Greenplum Database versions 4.2.1 and later support column-oriented RLE compression. To backup a table with RLE compression that you intend to restore to an earlier version of Greenplum Database, alter the table to have no compression or a compression type supported in the earlier version (`ZLIB` or `QUICKLZ`) before you start the backup operation.

In Greenplum Database 4.3.3 and later, Greenplum Database combines delta compression with RLE compression for data in columns of type `BIGINT`, `INTEGER`, `DATE`, `TIME`, or `TIMESTAMP`. The delta compression algorithm is based on the change between consecutive column values and is designed to improve compression when data is loaded in sorted order or when the compression is applied to data in sorted order.

When Greenplum Database is upgraded to 4.3.3, these rules apply for data in columns that are compressed with RLE:

- Existing column data are compressed with only RLE compression.
- New data are compressed with delta compression combined with RLE compression in the columns of type that support it.

If switching the Greenplum Database binary from 4.3.3 to 4.3.2 is required, the following steps are recommended.

1. Alter append-optimized, column oriented tables with RLE compression columns to use either no compression or a compression type `ZLIB` or `QUICKLZ`.
2. Back up the database.

> **Note:** If you backup a table that uses RLE column compression from a Greenplum Database 4.3.3, you can restore the table in Greenplum Database 4.3.2. However, the compression in the Greenplum Database 4.3.2 is RLE compression, not RLE compression combined with delta compression.

Adding Column-level Compression

You can add the following storage directives to a column for append-optimized tables with column orientation:

- Compression type
- Compression level
- Block size for a column

Add storage directives using the `CREATE TABLE`, `ALTER TABLE`, and `CREATE TYPE` commands.

The following table details the types of storage directives and possible values for each.

Table 34: Storage Directives for Column-level Compression

Name	Definition	Values	Comment
`COMPRESSTYPE`	Type of compression.	`zlib:` deflate algorithm `quicklz`: fast compression `RLE_TYPE`: run-length encoding `none`: no compression	Values are not case-sensitive.

Name	Definition	Values	Comment
COMPRESSLEVEL	Compression level.	zlib compression: 1-9	1 is the fastest method with the least compression. 1 is the default. 9 is the slowest method with the most compression.
		QuickLZ compression: 1 – use compression	1 is the default.
		RLE_TYPE compression: 1 – 4 1 - apply RLE only 2 - apply RLE then apply zlib compression level 1 3 - apply RLE then apply zlib compression level 5 4 - apply RLE then apply zlib compression level 9	1 is the fastest method with the least compression. 4 is the slowest method with the most compression. 1 is the default.
BLOCKSIZE	The size in bytes for each block in the table	8192 – 2097152	The value must be a multiple of 8192.

The following is the format for adding storage directives.

```
[ ENCODING ( storage_directive [,…] ) ]
```

where the word ENCODING is required and the storage directive has three parts:

- The name of the directive
- An equals sign
- The specification

Separate multiple storage directives with a comma. Apply a storage directive to a single column or designate it as the default for all columns, as shown in the following CREATE TABLE clauses.

General Usage:

```
column_name data_type ENCODING ( storage_directive [, … ] ), …
```

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

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

Example:

```
C1 char ENCODING (compresstype=quicklz, blocksize=65536)
```

```
COLUMN C1 ENCODING (compresstype=zlib, compresslevel=6, blocksize=65536)
```

```
DEFAULT COLUMN ENCODING (compresstype=quicklz)
```

Default Compression Values

If the compression type, compression level and block size are not defined, the default is no compression, and the block size is set to the Server Configuration Parameter `block_size`.

Precedence of Compression Settings

Column compression settings are inherited from the table level to the partition level to the subpartition level. The lowest-level settings have priority.

- Column compression settings specified at the table level override any compression settings for the entire table.
- Column compression settings specified for partitions override any compression settings at the column or table levels.
- Column compression settings specified for subpartitions override any compression settings at the partition, column or table levels.
- When an `ENCODING` clause conflicts with a `WITH` clause, the `ENCODING` clause has higher precedence than the `WITH` clause.

> **Note:** The `INHERITS` clause is not allowed in a table that contains a storage directive or a column reference storage directive.
>
> Tables created using the `LIKE` clause ignore storage directive and column reference storage directives.

Optimal Location for Column Compression Settings

The best practice is to set the column compression settings at the level where the data resides. See *Example 5*, which shows a table with a partition depth of 2. `RLE_TYPE` compression is added to a column at the subpartition level.

Storage Directives Examples

The following examples show the use of storage directives in `CREATE TABLE` statements.

Example 1

In this example, column `c1` is compressed using `zlib` and uses the block size defined by the system. Column `c2` is compressed with `quicklz`, and uses a block size of `65536`. Column `c3` is not compressed and uses the block size defined by the system.

```
CREATE TABLE T1 (c1 int ENCODING (compresstype=zlib),
                  c2 char ENCODING (compresstype=quicklz, blocksize=65536),
                  c3 char     WITH (appendonly=true, orientation=column);
```

Example 2

In this example, column `c1` is compressed using `zlib` and uses the block size defined by the system. Column `c2` is compressed with `quicklz`, and uses a block size of `65536`. Column `c3` is compressed using `RLE_TYPE` and uses the block size defined by the system.

```
CREATE TABLE T2 (c1 int ENCODING (compresstype=zlib),
                  c2 char ENCODING (compresstype=quicklz, blocksize=65536),
                  c3 char,
                  COLUMN c3 ENCODING (compresstype=RLE_TYPE)
                  )
    WITH (appendonly=true, orientation=column)
```

Example 3

In this example, column `c1` is compressed using `zlib` and uses the block size defined by the system. Column `c2` is compressed with `quicklz`, and uses a block size of `65536`. Column `c3` is compressed using `zlib` and uses the block size defined by the system. Note that column `c3` uses `zlib` (not `RLE_TYPE`) in the partitions, because the column storage in the partition clause has precedence over the storage directive in the column definition for the table.

```
CREATE TABLE T3 (c1 int ENCODING (compresstype=zlib),
                  c2 char ENCODING (compresstype=quicklz, blocksize=65536),
                  c3 char, COLUMN c3 ENCODING (compresstype=RLE_TYPE) )
    WITH (appendonly=true, orientation=column)
    PARTITION BY RANGE (c3) (START ('1900-01-01'::DATE)
                             END ('2100-12-31'::DATE),
                             COLUMN c3 ENCODING (zlib));
```

Example 4

In this example, `CREATE TABLE` assigns the zlib compresstype storage directive to `c1`. Column `c2` has no storage directive and inherits the compression type (`quicklz`) and block size (`65536`) from the `DEFAULT COLUMN ENCODING` clause.

Column `c3`'s `ENCODING` clause defines its compression type, `RLE_TYPE`. The `DEFAULT COLUMN ENCODING` clause defines `c3`'s block size, `65536`.

The `ENCODING` clause defined for a specific column overrides the `DEFAULT ENCODING` clause, so column `c4` has a compress type of `none` and the default block size.

```
CREATE TABLE T4 (c1 int ENCODING (compresstype=zlib),
                  c2 char,
                  c4 smallint ENCODING (compresstype=none),
                  DEFAULT COLUMN ENCODING (compresstype=quicklz,
                                           blocksize=65536),
                  COLUMN c3 ENCODING (compresstype=RLE_TYPE)
                  )
   WITH (appendonly=true, orientation=column);
```

Example 5

This example creates an append-optimized, column-oriented table, T5. T5 has two partitions, `p1` and `p2`, each of which has subpartitions. Each subpartition has `ENCODING` clauses:

- The `ENCODING` clause for partition `p1`'s subpartition `sp1` defines column `i`'s compression type as `zlib` and block size as 65536.
- The `ENCODING` clauses for partition `p2`'s subpartition `sp1` defines column `i`'s compression type as `rle_type` and block size is the default value. Column `k` uses the default compression and its block size is 8192.

```
CREATE TABLE T5(i int, j int, k int, l int)
    WITH (appendonly=true, orientation=column)
    PARTITION BY range(i) SUBPARTITION BY range(j)
    (
       p1 start(1) end(2)
       ( subpartition sp1 start(1) end(2)
         column i encoding(compresstype=zlib, blocksize=65536)
       ),
       partition p2 start(2) end(3)
       ( subpartition sp1 start(1) end(2)
           column i encoding(compresstype=rle_type)
           column k encoding(blocksize=8192)
       )
    );
```

For an example showing how to add a compressed column to an existing table with the ALTER TABLE command, see *Adding a Compressed Column to Table* .

Adding Compression in a TYPE Command

You can define a compression type to simplify column compression statements. For example, the following CREATE TYPE command defines a compression type, comptype, that specifies quicklz compression.

where comptype is defined as:

```
CREATE TYPE comptype (
   internallength = 4,
   input = comptype_in,
   output = comptype_out,
   alignment = int4,
   default = 123,
   passedbyvalue,
   compresstype="quicklz",
   blocksize=65536,
   compresslevel=1
   );
```

You can then use comptype in a CREATE TABLE command to specify quicklz compression for a column:

```
CREATE TABLE t2 (c1 comptype)
    WITH (APPENDONLY=true, ORIENTATION=column);
```

For information about creating and adding compression parameters to a type, see CREATE TYPE. For information about changing compression specifications in a type, see ALTER TYPE.

Choosing Block Size

The blocksize is the size, in bytes, for each block in a table. Block sizes must be between 8192 and 2097152 bytes, and be a multiple of 8192. The default is 32768.

Specifying large block sizes can consume large amounts of memory. Block size determines buffering in the storage layer. Greenplum maintains a buffer per partition, and per column in column-oriented tables. Tables with many partitions or columns consume large amounts of memory.

Altering a Table

The ALTER TABLE command changes the definition of a table. Use ALTER TABLE to change table attributes such as column definitions, distribution policy, storage model, and partition structure (see also *Maintaining Partitioned Tables*). For example, to add a not-null constraint to a table column:

```
=> ALTER TABLE address ALTER COLUMN street SET NOT NULL;
```

Altering Table Distribution

ALTER TABLE provides options to change a table's distribution policy . When the table distribution options change, the table data is redistributed on disk, which can be resource intensive. You can also redistribute table data using the existing distribution policy.

Changing the Distribution Policy

For partitioned tables, changes to the distribution policy apply recursively to the child partitions. This operation preserves the ownership and all other attributes of the table. For example, the following command redistributes the table sales across all segments using the customer_id column as the distribution key:

```
ALTER TABLE sales SET DISTRIBUTED BY (customer_id);
```

When you change the hash distribution of a table, table data is automatically redistributed. Changing the distribution policy to a random distribution does not cause the data to be redistributed. For example, the following `ALTER TABLE` command has no immediate effect:

```
ALTER TABLE sales SET DISTRIBUTED RANDOMLY;
```

Redistributing Table Data

To redistribute table data for tables with a random distribution policy (or when the hash distribution policy has not changed) use `REORGANIZE=TRUE`. Reorganizing data may be necessary to correct a data skew problem, or when segment resources are added to the system. For example, the following command redistributes table data across all segments using the current distribution policy, including random distribution.

```
ALTER TABLE sales SET WITH (REORGANIZE=TRUE);
```

Altering the Table Storage Model

Table storage, compression, and orientation can be declared only at creation. To change the storage model, you must create a table with the correct storage options, load the original table data into the new table, drop the original table, and rename the new table with the original table's name. You must also re-grant any table permissions. For example:

```
CREATE TABLE sales2 (LIKE sales)
WITH (appendonly=true, compresstype=quicklz,
      compresslevel=1, orientation=column);
INSERT INTO sales2 SELECT * FROM sales;
DROP TABLE sales;
ALTER TABLE sales2 RENAME TO sales;
GRANT ALL PRIVILEGES ON sales TO admin;
GRANT SELECT ON sales TO guest;
```

See *Splitting a Partition* to learn how to change the storage model of a partitioned table.

Adding a Compressed Column to Table

Use `ALTER TABLE` command to add a compressed column to a table. All of the options and constraints for compressed columns described in *Adding Column-level Compression* apply to columns added with the `ALTER TABLE` command.

The following example shows how to add a column with `zlib` compression to a table, `T1`.

```
ALTER TABLE T1
      ADD COLUMN c4 int DEFAULT 0
      ENCODING (COMPRESSTYPE=zlib);
```

Inheritance of Compression Settings

A partition that is added to a table that has subpartitions with compression settings inherits the compression settings from the subpartition.The following example shows how to create a table with subpartition encodings, then alter it to add a partition.

```
CREATE TABLE ccddl (i int, j int, k int, l int)
  WITH
    (APPENDONLY = TRUE, ORIENTATION=COLUMN)
  PARTITION BY range(j)
  SUBPARTITION BY list (k)
  SUBPARTITION template(
    SUBPARTITION sp1 values(1, 2, 3, 4, 5),
    COLUMN i ENCODING(COMPRESSTYPE=ZLIB),
    COLUMN j ENCODING(COMPRESSTYPE=QUICKLZ),
    COLUMN k ENCODING(COMPRESSTYPE=ZLIB),
    COLUMN l ENCODING(COMPRESSTYPE=ZLIB))
  (PARTITION p1 START(1) END(10),
   PARTITION p2 START(10) END(20))
;

ALTER TABLE ccddl
  ADD PARTITION p3 START(20) END(30)
;
```

Running the `ALTER TABLE` command creates partitions of table `ccddl` named `ccddl_1_prt_p3` and `ccddl_1_prt_p3_2_prt_sp1`. Partition `ccddl_1_prt_p3` inherits the different compression encodings of subpartition `sp1`.

Dropping a Table

The `DROP TABLE` command removes tables from the database. For example:

```
DROP TABLE mytable;
```

To empty a table of rows without removing the table definition, use `DELETE` or `TRUNCATE`. For example:

```
DELETE FROM mytable;

TRUNCATE mytable;
```

`DROP TABLE` always removes any indexes, rules, triggers, and constraints that exist for the target table. Specify `CASCADE` to drop a table that is referenced by a view. `CASCADE` removes dependent views.

Partitioning Large Tables

Table partitioning enables supporting very large tables, such as fact tables, by logically dividing them into smaller, more manageable pieces. Partitioned tables can improve query performance by allowing the Greenplum Database query optimizer to scan only the data needed to satisfy a given query instead of scanning all the contents of a large table.

- *About Table Partitioning*
- *Deciding on a Table Partitioning Strategy*
- *Creating Partitioned Tables*
- *Loading Partitioned Tables*
- *Verifying Your Partition Strategy*
- *Viewing Your Partition Design*
- *Maintaining Partitioned Tables*

About Table Partitioning

Partitioning does not change the physical distribution of table data across the segments. Table distribution is physical: Greenplum Database physically divides partitioned tables and non-partitioned tables across segments to enable parallel query processing. Table *partitioning* is logical: Greenplum Database logically divides big tables to improve query performance and facilitate data warehouse maintenance tasks, such as rolling old data out of the data warehouse.

Greenplum Database supports:

- *range partitioning*: division of data based on a numerical range, such as date or price.
- *list partitioning*: division of data based on a list of values, such as sales territory or product line.
- A combination of both types.

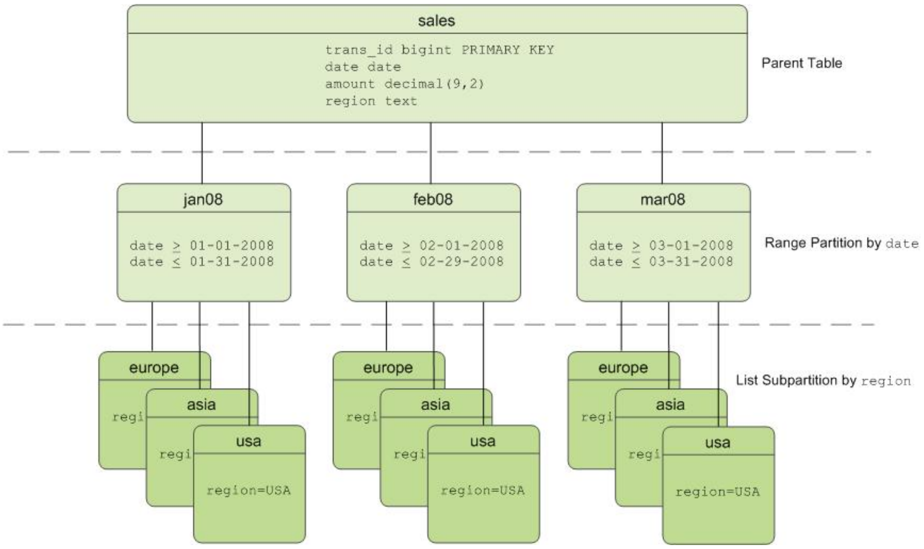


Figure 14: Example Multi-level Partition Design

Table Partitioning in Greenplum Database

Greenplum Database divides tables into parts (also known as partitions) to enable massively parallel processing. Tables are partitioned during `CREATE TABLE` using the `PARTITION BY` (and optionally the `SUBPARTITION BY`) clause. Partitioning creates a top-level (or parent) table with one or more levels of sub-tables (or child tables). Internally, Greenplum Database creates an inheritance relationship between the top-level table and its underlying partitions, similar to the functionality of the `INHERITS` clause of PostgreSQL.

Greenplum uses the partition criteria defined during table creation to create each partition with a distinct `CHECK` constraint, which limits the data that table can contain. The query optimizer uses `CHECK` constraints to determine which table partitions to scan to satisfy a given query predicate.

The Greenplum system catalog stores partition hierarchy information so that rows inserted into the top-level parent table propagate correctly to the child table partitions. To change the partition design or table structure, alter the parent table using `ALTER TABLE` with the `PARTITION` clause.

To insert data into a partitioned table, you specify the root partitioned table, the table created with the `CREATE TABLE` command. You also can specify a leaf child table of the partitioned table in an `INSERT` command. An error is returned if the data is not valid for the specified leaf child table. Specifying a non-leaf or a non-root partition table in the DML command is not supported.

Deciding on a Table Partitioning Strategy

Not all tables are good candidates for partitioning. If the answer is *yes* to all or most of the following questions, table partitioning is a viable database design strategy for improving query performance. If the answer is *no* to most of the following questions, table partitioning is not the right solution for that table. Test your design strategy to ensure that query performance improves as expected.

- **Is the table large enough?** Large fact tables are good candidates for table partitioning. If you have millions or billions of records in a table, you may see performance benefits from logically breaking that data up into smaller chunks. For smaller tables with only a few thousand rows or less, the administrative overhead of maintaining the partitions will outweigh any performance benefits you might see.
- **Are you experiencing unsatisfactory performance?** As with any performance tuning initiative, a table should be partitioned only if queries against that table are producing slower response times than desired.
- **Do your query predicates have identifiable access patterns?** Examine the `WHERE` clauses of your query workload and look for table columns that are consistently used to access data. For example, if most of your queries tend to look up records by date, then a monthly or weekly date-partitioning design might be beneficial. Or if you tend to access records by region, consider a list-partitioning design to divide the table by region.
- **Does your data warehouse maintain a window of historical data?** Another consideration for partition design is your organization's business requirements for maintaining historical data. For example, your data warehouse may require that you keep data for the past twelve months. If the data is partitioned by month, you can easily drop the oldest monthly partition from the warehouse and load current data into the most recent monthly partition.
- **Can the data be divided into somewhat equal parts based on some defining criteria?** Choose partitioning criteria that will divide your data as evenly as possible. If the partitions contain a relatively equal number of records, query performance improves based on the number of partitions created. For example, by dividing a large table into 10 partitions, a query will execute 10 times faster than it would against the unpartitioned table, provided that the partitions are designed to support the query's criteria.

Do not create more partitions than are needed. Creating too many partitions can slow down management and maintenance jobs, such as vacuuming, recovering segments, expanding the cluster, checking disk usage, and others.

Partitioning does not improve query performance unless the query optimizer can eliminate partitions based on the query predicates. Queries that scan every partition run slower than if the table were not partitioned, so avoid partitioning if few of your queries achieve partition elimination. Check the explain plan for queries to make sure that partitions are eliminated. See *Query Profiling* for more about partition elimination.

Be very careful with multi-level partitioning because the number of partition files can grow very quickly. For example, if a table is partitioned by both day and city, and there are 1,000 days of data and 1,000 cities, the total number of partitions is one million. Column-oriented tables store each column in a physical table, so if this table has 100 columns, the system would be required to manage 100 million files for the table.

Before settling on a multi-level partitioning strategy, consider a single level partition with bitmap indexes. Indexes slow down data loads, so performance testing with your data and schema is recommended to decide on the best strategy.

Creating Partitioned Tables

You partition tables when you create them with `CREATE TABLE`. This topic provides examples of SQL syntax for creating a table with various partition designs.

To partition a table:

1. Decide on the partition design: date range, numeric range, or list of values.
2. Choose the column(s) on which to partition the table.
3. Decide how many levels of partitions you want. For example, you can create a date range partition table by month and then subpartition the monthly partitions by sales region.

- *Defining Date Range Table Partitions*
- *Defining Numeric Range Table Partitions*
- *Defining List Table Partitions*
- *Defining Multi-level Partitions*
- *Partitioning an Existing Table*

Defining Date Range Table Partitions

A date range partitioned table uses a single `date` or `timestamp` column as the partition key column. You can use the same partition key column to create subpartitions if necessary, for example, to partition by month and then subpartition by day. Consider partitioning by the most granular level. For example, for a table partitioned by date, you can partition by day and have 365 daily partitions, rather than partition by year then subpartition by month then subpartition by day. A multi-level design can reduce query planning time, but a flat partition design runs faster.

You can have Greenplum Database automatically generate partitions by giving a `START` value, an `END` value, and an `EVERY` clause that defines the partition increment value. By default, `START` values are always inclusive and `END` values are always exclusive. For example:

```
CREATE TABLE sales (id int, date date, amt decimal(10,2))
DISTRIBUTED BY (id)
PARTITION BY RANGE (date)
( START (date '2008-01-01') INCLUSIVE
   END (date '2009-01-01') EXCLUSIVE
   EVERY (INTERVAL '1 day') );
```

You can also declare and name each partition individually. For example:

```
CREATE TABLE sales (id int, date date, amt decimal(10,2))
DISTRIBUTED BY (id)
PARTITION BY RANGE (date)
( PARTITION Jan08 START (date '2008-01-01') INCLUSIVE ,
  PARTITION Feb08 START (date '2008-02-01') INCLUSIVE ,
  PARTITION Mar08 START (date '2008-03-01') INCLUSIVE ,
  PARTITION Apr08 START (date '2008-04-01') INCLUSIVE ,
  PARTITION May08 START (date '2008-05-01') INCLUSIVE ,
  PARTITION Jun08 START (date '2008-06-01') INCLUSIVE ,
  PARTITION Jul08 START (date '2008-07-01') INCLUSIVE ,
  PARTITION Aug08 START (date '2008-08-01') INCLUSIVE ,
  PARTITION Sep08 START (date '2008-09-01') INCLUSIVE ,
  PARTITION Oct08 START (date '2008-10-01') INCLUSIVE ,
  PARTITION Nov08 START (date '2008-11-01') INCLUSIVE ,
  PARTITION Dec08 START (date '2008-12-01') INCLUSIVE
```

```
                  END (date '2009-01-01') EXCLUSIVE );
```

You do not have to declare an `END` value for each partition, only the last one. In this example, `Jan08` ends where `Feb08` starts.

Defining Numeric Range Table Partitions

A numeric range partitioned table uses a single numeric data type column as the partition key column. For example:

```
CREATE TABLE rank (id int, rank int, year int, gender
char(1), count int)
DISTRIBUTED BY (id)
PARTITION BY RANGE (year)
( START (2001) END (2008) EVERY (1),
  DEFAULT PARTITION extra );
```

For more information about default partitions, see *Adding a Default Partition*.

Defining List Table Partitions

A list partitioned table can use any data type column that allows equality comparisons as its partition key column. A list partition can also have a multi-column (composite) partition key, whereas a range partition only allows a single column as the partition key. For list partitions, you must declare a partition specification for every partition (list value) you want to create. For example:

```
CREATE TABLE rank (id int, rank int, year int, gender
char(1), count int )
DISTRIBUTED BY (id)
PARTITION BY LIST (gender)
( PARTITION girls VALUES ('F'),
  PARTITION boys VALUES ('M'),
  DEFAULT PARTITION other );
```

> **Note:** The current Greenplum Database legacy optimizer allows list partitions with multi-column (composite) partition keys. A range partition only allows a single column as the partition key. The Pivotal Query Optimizer does not support composite keys, so Pivotal does not recommend using composite partition keys.

For more information about default partitions, see *Adding a Default Partition*.

Defining Multi-level Partitions

You can create a multi-level partition design with subpartitions of partitions. Using a *subpartition template* ensures that every partition has the same subpartition design, including partitions that you add later. For example, the following SQL creates the two-level partition design shown in *Figure 14: Example Multi-level Partition Design*:

```
CREATE TABLE sales (trans_id int, date date, amount
decimal(9,2), region text)
DISTRIBUTED BY (trans_id)
PARTITION BY RANGE (date)
SUBPARTITION BY LIST (region)
SUBPARTITION TEMPLATE
( SUBPARTITION usa VALUES ('usa'),
  SUBPARTITION asia VALUES ('asia'),
  SUBPARTITION europe VALUES ('europe'),
  DEFAULT SUBPARTITION other_regions)
  (START (date '2011-01-01') INCLUSIVE
   END (date '2012-01-01') EXCLUSIVE
   EVERY (INTERVAL '1 month'),
   DEFAULT PARTITION outlying_dates );
```

The following example shows a three-level partition design where the `sales` table is partitioned by `year`, then `month`, then `region`. The `SUBPARTITION TEMPLATE` clauses ensure that each yearly partition has the same subpartition structure. The example declares a `DEFAULT` partition at each level of the hierarchy.

```
CREATE TABLE p3_sales (id int, year int, month int, day int,
region text)
DISTRIBUTED BY (id)
PARTITION BY RANGE (year)
    SUBPARTITION BY RANGE (month)
       SUBPARTITION TEMPLATE (
        START (1) END (13) EVERY (1),
        DEFAULT SUBPARTITION other_months )
           SUBPARTITION BY LIST (region)
             SUBPARTITION TEMPLATE (
               SUBPARTITION usa VALUES ('usa'),
               SUBPARTITION europe VALUES ('europe'),
               SUBPARTITION asia VALUES ('asia'),
               DEFAULT SUBPARTITION other_regions )
( START (2002) END (2012) EVERY (1),
  DEFAULT PARTITION outlying_years );
```

Caution: When you create multi-level partitions on ranges, it is easy to create a large number of subpartitions, some containing little or no data. This can add many entries to the system tables, which increases the time and memory required to optimize and execute queries. Increase the range interval or choose a different partitioning strategy to reduce the number of subpartitions created.

Partitioning an Existing Table

Tables can be partitioned only at creation. If you have a table that you want to partition, you must create a partitioned table, load the data from the original table into the new table, drop the original table, and rename the partitioned table with the original table's name. You must also re-grant any table permissions. For example:

```
CREATE TABLE sales2 (LIKE sales)
PARTITION BY RANGE (date)
( START (date '2008-01-01') INCLUSIVE
   END (date '2009-01-01') EXCLUSIVE
   EVERY (INTERVAL '1 month') );
INSERT INTO sales2 SELECT * FROM sales;
DROP TABLE sales;
ALTER TABLE sales2 RENAME TO sales;
GRANT ALL PRIVILEGES ON sales TO admin;
GRANT SELECT ON sales TO guest;
```

Limitations of Partitioned Tables

For each partition level, a partitioned table can have a maximum of 32,767 partitions.

A primary key or unique constraint on a partitioned table must contain all the partitioning columns. A unique index can omit the partitioning columns; however, it is enforced only on the parts of the partitioned table, not on the partitioned table as a whole.

The Pivotal Query Optimizer supports uniform multi-level partitioned tables. If Pivotal Query Optimizer is enabled and the multi-level partitioned table is not uniform, Greenplum Database executes queries against the table with the legacy query optimizer. For information about uniform multi-level partitioned tables, see *About Uniform Multi-level Partitioned Tables*.

Exchanging a leaf child partition with an external table is not supported if the partitioned table is created with the `SUBPARITION` clause or if a partition has a subpartition. For information about exchanging a leaf child partition with an external table, see *Exchanging a Leaf Child Partition with an External Table*.

These are limitations for partitioned tables when a leaf child partition of the table is an external table:

- Queries that run against partitioned tables that contain external table partitions are executed with the legacy query optimizer.
- The external table partition is a read only external table. Commands that attempt to access or modify data in the external table partition return an error. For example:
    - `INSERT`, `DELETE`, and `UPDATE` commands that attempt to change data in the external table partition return an error.
    - `TRUNCATE` commands return an error.
    - `COPY` commands cannot copy data to a partitioned table that updates an external table partition.
    - `COPY` commands that attempt to copy from an external table partition return an error unless you specify the `IGNORE EXTERNAL PARTITIONS` clause with `COPY` command. If you specify the clause, data is not copied from external table partitions.

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

      ```
      COPY (SELECT * from my_sales ) TO stdout
      ```
    - `VACUUM` commands skip external table partitions.
- The following operations are supported if no data is changed on the external table partition. Otherwise, an error is returned.
    - Adding or dropping a column.
    - Changing the data type of column.
- These `ALTER PARTITION` operations are not supported if the partitioned table contains an external table partition:
    - Setting a subpartition template.
    - Altering the partition properties.
    - Creating a default partition.
    - Setting a distribution policy.
    - Setting or dropping a `NOT NULL` constraint of column.
    - Adding or dropping constraints.
    - Splitting an external partition.
- The Greenplum Database utility `gpcrondump` does not back up data from a leaf child partition of a partitioned table if the leaf child partition is a readable external table.

Loading Partitioned Tables

After you create the partitioned table structure, top-level parent tables are empty. Data is routed to the bottom-level child table partitions. In a multi-level partition design, only the subpartitions at the bottom of the hierarchy can contain data.

Rows that cannot be mapped to a child table partition are rejected and the load fails. To avoid unmapped rows being rejected at load time, define your partition hierarchy with a `DEFAULT` partition. Any rows that do not match a partition's `CHECK` constraints load into the `DEFAULT` partition. See *Adding a Default Partition*.

At runtime, the query optimizer scans the entire table inheritance hierarchy and uses the `CHECK` table constraints to determine which of the child table partitions to scan to satisfy the query's conditions. The `DEFAULT` partition (if your hierarchy has one) is always scanned. `DEFAULT` partitions that contain data slow down the overall scan time.

When you use `COPY` or `INSERT` to load data into a parent table, the data is automatically rerouted to the correct partition, just like a regular table.

Best practice for loading data into partitioned tables is to create an intermediate staging table, load it, and then exchange it into your partition design. See *Exchanging a Partition*.

Verifying Your Partition Strategy

When a table is partitioned based on the query predicate, you can use `EXPLAIN` to verify that the query optimizer scans only the relevant data to examine the query plan.

For example, suppose a *sales* table is date-range partitioned by month and subpartitioned by region as shown in *Figure 14: Example Multi-level Partition Design*. For the following query:

```
EXPLAIN SELECT * FROM sales WHERE date='01-07-12' AND
region='usa';
```

The query plan for this query should show a table scan of only the following tables:

- the default partition returning 0-1 rows (if your partition design has one)
- the January 2012 partition (*sales_1_prt_1*) returning 0-1 rows
- the USA region subpartition (*sales_1_2_prt_usa*) returning *some number* of rows.

The following example shows the relevant portion of the query plan.

```
->  Seq Scan onsales_1_prt_1 sales (cost=0.00..0.00 rows=0
     width=0)
Filter: "date"=01-07-08::date AND region='USA'::text
->  Seq Scan onsales_1_2_prt_usa sales (cost=0.00..9.87
rows=20
      width=40)
```

Ensure that the query optimizer does not scan unnecessary partitions or subpartitions (for example, scans of months or regions not specified in the query predicate), and that scans of the top-level tables return 0-1 rows.

Troubleshooting Selective Partition Scanning

The following limitations can result in a query plan that shows a non-selective scan of your partition hierarchy.

- The query optimizer can selectively scan partitioned tables only when the query contains a direct and simple restriction of the table using immutable operators such as:

  =, < , <= , >, >= , and <>
- Selective scanning recognizes `STABLE` and `IMMUTABLE` functions, but does not recognize `VOLATILE` functions within a query. For example, `WHERE` clauses such as `date > CURRENT_DATE` cause the query optimizer to selectively scan partitioned tables, but `time > TIMEOFDAY` does not.

Viewing Your Partition Design

You can look up information about your partition design using the *pg_partitions* view. For example, to see the partition design of the *sales* table:

```
SELECT partitionboundary, partitiontablename, partitionname,
partitionlevel, partitionrank
FROM pg_partitions
WHERE tablename='sales';
```

The following table and views show information about partitioned tables.

- *pg_partition* - Tracks partitioned tables and their inheritance level relationships.
- *pg_partition_templates* - Shows the subpartitions created using a subpartition template.
- *pg_partition_columns* - Shows the partition key columns used in a partition design.

For information about Greenplum Database system catalog tables and views, see the *Greenplum Database Reference Guide*.

Maintaining Partitioned Tables

To maintain a partitioned table, use the `ALTER TABLE` command against the top-level parent table. The most common scenario is to drop old partitions and add new ones to maintain a rolling window of data in a range partition design. You can convert (*exchange*) older partitions to the append-optimized compressed storage format to save space. If you have a default partition in your partition design, you add a partition by *splitting* the default partition.

- *Adding a Partition*
- *Renaming a Partition*
- *Adding a Default Partition*
- *Dropping a Partition*
- *Truncating a Partition*
- *Exchanging a Partition*
- *Splitting a Partition*
- *Modifying a Subpartition Template*
- *Exchanging a Leaf Child Partition with an External Table*

> **Important:** When defining and altering partition designs, use the given partition name, not the table object name. Although you can query and load any table (including partitioned tables) directly using SQL commands, you can only modify the structure of a partitioned table using the `ALTER TABLE...PARTITION` clauses.
>
> Partitions are not required to have names. If a partition does not have a name, use one of the following expressions to specify a part: `PARTITION FOR (value)` or)`PARTITION FOR(RANK(number)`.

Adding a Partition

You can add a partition to a partition design with the `ALTER TABLE` command. If the original partition design included subpartitions defined by a *subpartition template*, the newly added partition is subpartitioned according to that template. For example:

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

If you did not use a subpartition template when you created the table, you define subpartitions when adding a partition:

```
ALTER TABLE sales ADD PARTITION
            START (date '2009-02-01') INCLUSIVE
            END (date '2009-03-01') EXCLUSIVE
      ( SUBPARTITION usa VALUES ('usa'),
        SUBPARTITION asia VALUES ('asia'),
        SUBPARTITION europe VALUES ('europe') );
```

When you add a subpartition to an existing partition, you can specify the partition to alter. For example:

```
ALTER TABLE sales ALTER PARTITION FOR (RANK(12))
      ADD PARTITION africa VALUES ('africa');
```

> **Note:** You cannot add a partition to a partition design that has a default partition. You must split the default partition to add a partition. See *Splitting a Partition*.

Renaming a Partition

Partitioned tables use the following naming convention. Partitioned subtable names are subject to uniqueness requirements and length limitations.

```
<parentname>_<level>_prt_<partition_name>
```

For example:

```
sales_1_prt_jan08
```

For auto-generated range partitions, where a number is assigned when no name is given):

```
sales_1_prt_1
```

To rename a partitioned child table, rename the top-level parent table. The <*parentname*> changes in the table names of all associated child table partitions. For example, the following command:

```
ALTER TABLE sales RENAME TO globalsales;
```

Changes the associated table names:

```
globalsales_1_prt_1
```

You can change the name of a partition to make it easier to identify. For example:

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

Changes the associated table name as follows:

```
sales_1_prt_jan08
```

When altering partitioned tables with the `ALTER TABLE` command, always refer to the tables by their partition name (*jan08*) and not their full table name (*sales_1_prt_jan08*).

> **Note:** The table name cannot be a partition name in an `ALTER TABLE` statement. For example, `ALTER TABLE sales...` is correct, `ALTER TABLE sales_1_part_jan08...` is not allowed.

Adding a Default Partition

You can add a default partition to a partition design with the `ALTER TABLE` command.

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

If your partition design is multi-level, each level in the hierarchy must have a default partition. For example:

```
ALTER TABLE sales ALTER PARTITION FOR (RANK(1)) ADD DEFAULT
PARTITION other;

ALTER TABLE sales ALTER PARTITION FOR (RANK(2)) ADD DEFAULT
PARTITION other;

ALTER TABLE sales ALTER PARTITION FOR (RANK(3)) ADD DEFAULT
PARTITION other;
```

If incoming data does not match a partition's `CHECK` constraint and there is no default partition, the data is rejected. Default partitions ensure that incoming data that does not match a partition is inserted into the default partition.

Dropping a Partition

You can drop a partition from your partition design using the `ALTER TABLE` command. When you drop a partition that has subpartitions, the subpartitions (and all data in them) are automatically dropped as well. For range partitions, it is common to drop the older partitions from the range as old data is rolled out of the data warehouse. For example:

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

Truncating a Partition

You can truncate a partition using the `ALTER TABLE` command. When you truncate a partition that has subpartitions, the subpartitions are automatically truncated as well.

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

Exchanging a Partition

You can exchange a partition using the `ALTER TABLE` command. Exchanging a partition swaps one table in place of an existing partition. You can exchange partitions only at the lowest level of your partition hierarchy (only partitions that contain data can be exchanged).

Partition exchange can be useful for data loading. For example, load a staging table and swap the loaded table into your partition design. You can use partition exchange to change the storage type of older partitions to append-optimized tables. For example:

```
CREATE TABLE jan12 (LIKE sales) WITH (appendonly=true);
INSERT INTO jan12 SELECT * FROM sales_1_prt_1 ;
ALTER TABLE sales EXCHANGE PARTITION FOR (DATE '2012-01-01')
WITH TABLE jan12;
```

> **Note:** This example refers to the single-level definition of the table `sales`, before partitions were added and altered in the previous examples.
>
> **Warning:** If you specify the `WITHOUT VALIDATION` clause, you must ensure that the data in table that you are exchanging for an existing partition is valid against the constraints on the partition. Otherwise, queries against the partitioned table might return incorrect results.

The Greenplum Database server configuration parameter `gp_enable_exchange_default_partition` controls availability of the `EXCHANGE DEFAULT PARTITION` clause. The default value for the parameter is `off`, the clause is not available and Greenplum Database returns an error if the clause is specified in an `ALTER TABLE` command.

For information about the parameter, see "Server Configuration Parameters" in the *Greenplum Database Reference Guide*.

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

Splitting a Partition

Splitting a partition divides a partition into two partitions. You can split a partition using the `ALTER TABLE` command. You can split partitions only at the lowest level of your partition hierarchy: only partitions that contain data can be split. The split value you specify goes into the *latter* partition.

For example, to split a monthly partition into two with the first partition containing dates January 1-15 and the second partition containing dates January 16-31:

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

If your partition design has a default partition, you must split the default partition to add a partition.

When using the `INTO` clause, specify the current default partition as the second partition name. For example, to split a default range partition to add a new monthly partition for January 2009:

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

Modifying a Subpartition Template

Use `ALTER TABLE` SET SUBPARTITION TEMPLATE to modify the subpartition template of a partitioned table. Partitions added after you set a new subpartition template have the new partition design. Existing partitions are not modified.

The following example alters the subpartition template of this partitioned table:

```
CREATE TABLE sales (trans_id int, date date, amount decimal(9,2), region text)
  DISTRIBUTED BY (trans_id)
  PARTITION BY RANGE (date)
  SUBPARTITION BY LIST (region)
  SUBPARTITION TEMPLATE
    ( SUBPARTITION usa VALUES ('usa'),
      SUBPARTITION asia VALUES ('asia'),
      SUBPARTITION europe VALUES ('europe'),
      DEFAULT SUBPARTITION other_regions )
  ( START (date '2014-01-01') INCLUSIVE
    END (date '2014-04-01') EXCLUSIVE
    EVERY (INTERVAL '1 month') );
```

This `ALTER TABLE` command, modifies the subpartition template.

```
ALTER TABLE sales SET SUBPARTITION TEMPLATE
( SUBPARTITION usa VALUES ('usa'),
  SUBPARTITION asia VALUES ('asia'),
  SUBPARTITION europe VALUES ('europe'),
  SUBPARTITION africa VALUES ('africa'),
  DEFAULT SUBPARTITION regions );
```

When you add a date-range partition of the table sales, it includes the new regional list subpartition for Africa. For example, the following command creates the subpartitions `usa`, `asia`, `europe`, `africa`, and a default partition named `other`:

```
ALTER TABLE sales ADD PARTITION "4"
  START ('2014-04-01') INCLUSIVE
  END ('2014-05-01') EXCLUSIVE ;
```

To view the tables created for the partitioned table `sales`, you can use the command `\dt sales*` from the psql command line.

To remove a subpartition template, use `SET SUBPARTITION TEMPLATE` with empty parentheses. For example, to clear the sales table subpartition template:

```
ALTER TABLE sales SET SUBPARTITION TEMPLATE ();
```

Exchanging a Leaf Child Partition with an External Table

You can exchange a leaf child partition of a partitioned table with a readable external table.The external table data can reside on a host file system, an NFS mount, or a Hadoop file system (HDFS).

For example, if you have a partitioned table that is created with monthly partitions and most of the queries against the table only access the newer data, you can copy the older, less accessed data to external tables and exchange older partitions with the external tables. For queries that only access the newer data, you could create queries that use partition elimination to prevent scanning the older, unneeded partitions.

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

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

For information about exchanging and altering a leaf child partition, see the `ALTER TABLE` command in the *Greenplum Database Command Reference*.

For information about limitations of partitioned tables that contain a external table partition, see *Limitations of Partitioned Tables*.

Example Exchanging a Partition with an External Table

This is a simple example that exchanges a leaf child partition of this partitioned table for an external table. The partitioned table contains data for the years 2000 through 2003.

```
CREATE TABLE sales (id int, year int, qtr int, day int, region text)
  DISTRIBUTED BY (id)
  PARTITION BY RANGE (year)
  ( PARTITION yr START (2000) END (2004) EVERY (1) ) ;
```

There are four leaf child partitions for the partitioned table. Each leaf child partition contains the data for a single year. The leaf child partition table `sales_1_prt_yr_1` contains the data for the year 2000. These steps exchange the table `sales_1_prt_yr_1` with an external table the uses the `gpfdist` protocol:

1. Ensure that the external table protocol is enabled for the Greenplum Database system.

   This example uses the `gpfdist` protocol. This command starts the `gpfdist` protocol.

   ```
   $ gpfdist
   ```

2. Create a writable external table.

   This `CREATE WRITABLE EXTENAL TABLE` command creates a writeable external table with the same columns as the partitioned table.

   ```
   CREATE WRITABLE EXTERNAL TABLE my_sales_ext ( LIKE sales_1_prt_yr_1 )
     LOCATION ( 'gpfdist://gpdb_test/sales_2000' )
     FORMAT 'csv'
     DISTRIBUTED BY (id) ;
   ```

3. Create a readable external table that reads the data from that destination of the writable external table created in the previous step.

   This `CREATE EXTENAL TABLE` create a readable external that uses the same external data as the writeable external data.

   ```
   CREATE EXTERNAL TABLE sales_2000_ext ( LIKE sales_1_prt_yr_1)
     LOCATION ( 'gpfdist://gpdb_test/sales_2000' )
     FORMAT 'csv' ;
   ```

4. Copy the data from the leaf child partition into the writable external table.

This INSERT command copies the data from the child leaf partition table of the partitioned table into the external table.

```
INSERT INTO my_sales_ext SELECT * FROM sales_1_prt_yr_1 ;
```

5. Exchange the existing leaf child partition with the external table.

   This ALTER TABLE command specifies the EXCHANGE PARTITION clause to switch the readable external table and the leaf child partition.

   ```
   ALTER TABLE sales ALTER PARTITION yr_1
      EXCHANGE PARTITION yr_1
      WITH TABLE sales_2000_ext WITHOUT VALIDATION;
   ```

   The external table becomes the leaf child partition with the table name sales_1_prt_yr_1 and the old leaf child partition becomes the table sales_2000_ext.

   > **Warning:** In order to ensure queries against the partitioned table return the correct results, the external table data must be valid against the CHECK constraints on the leaf child partition. In this case, the data was taken from the child leaf partition table on which the CHECK constraints were defined.

6. Drop the table that was rolled out of the partitioned table.

   ```
   DROP TABLE sales_2000_ext ;
   ```

You can rename the name of the leaf child partition to indicate that sales_1_prt_yr_1 is an external table.

This example command changes the partitionname to yr_1_ext and the name of the child leaf partition table to sales_1_prt_yr_1_ext.

```
ALTER TABLE sales RENAME PARTITION yr_1 TO  yr_1_ext ;
```

Creating and Using Sequences

You can use sequences to auto-increment unique ID columns of a table whenever a record is added. Sequences are often used to assign unique identification numbers to rows added to a table. You can declare an identifier column of type `SERIAL` to implicitly create a sequence for use with a column.

Creating a Sequence

The `CREATE SEQUENCE` command creates and initializes a special single-row sequence generator table with the given sequence name. The sequence name must be distinct from the name of any other sequence, table, index, or view in the same schema. For example:

```
CREATE SEQUENCE myserial START 101;
```

Using a Sequence

After you create a sequence generator table using `CREATE SEQUENCE`, you can use the `nextval` function to operate on the sequence. For example, to insert a row into a table that gets the next value of a sequence:

```
INSERT INTO vendors VALUES (nextval('myserial'), 'acme');
```

You can also use the `setval` function to reset a sequence's counter value. For example:

```
SELECT setval('myserial', 201);
```

A `nextval` operation is never rolled back. Afetched value is considered used, even if the transaction that performed the `nextval` fails. This means that failed transactions can leave unused holes in the sequence of assigned values. `setval` operations are never rolled back.

Note that the `nextval` function is not allowed in `UPDATE` or `DELETE` statements if mirroring is enabled, and the `currval` and `lastval` functions are not supported in Greenplum Database.

To examine the current settings of a sequence, query the sequence table:

```
SELECT * FROM myserial;
```

Altering a Sequence

The `ALTER SEQUENCE` command changes the parameters of an existing sequence generator. For example:

```
ALTER SEQUENCE myserial RESTART WITH 105;
```

Any parameters not set in the `ALTER SEQUENCE` command retain their prior settings.

Dropping a Sequence

The `DROP SEQUENCE` command removes a sequence generator table. For example:

```
DROP SEQUENCE myserial;
```

Using Indexes in Greenplum Database

In most traditional databases, indexes can greatly improve data access times. However, in a distributed database such as Greenplum, indexes should be used more sparingly. Greenplum Database performs very fast sequential scans; indexes use a random seek pattern to locate records on disk. Greenplum data is distributed across the segments, so each segment scans a smaller portion of the overall data to get the result. With table partitioning, the total data to scan may be even smaller. Because business intelligence (BI) query workloads generally return very large data sets, using indexes is not efficient.

Greenplum recommends trying your query workload without adding indexes. Indexes are more likely to improve performance for OLTP workloads, where the query is returning a single record or a small subset of data. Indexes can also improve performance on compressed append-optimized tables for queries that return a targeted set of rows, as the optimizer can use an index access method rather than a full table scan when appropriate. For compressed data, an index access method means only the necessary rows are uncompressed.

Greenplum Database automatically creates `PRIMARY KEY` constraints for tables with primary keys. To create an index on a partitioned table, create an index on the partitioned table that you created. The index is propagated to all the child tables created by Greenplum Database. Creating an index on a table that is created by Greenplum Database for use by a partitioned table is not supported.

Note that a `UNIQUE CONSTRAINT` (such as a `PRIMARY KEY CONSTRAINT`) implicitly creates a `UNIQUE INDEX` that must include all the columns of the distribution key and any partitioning key. The `UNIQUE CONSTRAINT` is enforced across the entire table, including all table partitions (if any).

Indexes add some database overhead — they use storage space and must be maintained when the table is updated. Ensure that the query workload uses the indexes that you create, and check that the indexes you add improve query performance (as compared to a sequential scan of the table). To determine whether indexes are being used, examine the query `EXPLAIN` plans. See *Query Profiling*.

Consider the following points when you create indexes.

- **Your Query Workload.** Indexes improve performance for workloads where queries return a single record or a very small data set, such as OLTP workloads.
- Compressed Tables. Indexes can improve performance on compressed append-optimized tables for queries that return a targeted set of rows. For compressed data, an index access method means only the necessary rows are uncompressed.
- **Avoid indexes on frequently updated columns.** Creating an index on a column that is frequently updated increases the number of writes required when the column is updated.
- **Create selective B-tree indexes.** Index selectivity is a ratio of the number of distinct values a column has divided by the number of rows in a table. For example, if a table has 1000 rows and a column has 800 distinct values, the selectivity of the index is 0.8, which is considered good. Unique indexes always have a selectivity ratio of 1.0, which is the best possible. Greenplum Database allows unique indexes only on distribution key columns.
- **Use Bitmap indexes for low selectivity columns.** The Greenplum Database Bitmap index type is not available in regular PostgreSQL. See *About Bitmap Indexes*.
- **Index columns used in joins.** An index on a column used for frequent joins (such as a foreign key column) can improve join performance by enabling more join methods for the query optimizer to use.
- **Index columns frequently used in predicates.** Columns that are frequently referenced in `WHERE` clauses are good candidates for indexes.
- **Avoid overlapping indexes.** Indexes that have the same leading column are redundant.
- **Drop indexes for bulk loads.** For mass loads of data into a table, consider dropping the indexes and re-creating them after the load completes. This is often faster than updating the indexes.
- **Consider a clustered index.** Clustering an index means that the records are physically ordered on disk according to the index. If the records you need are distributed randomly on disk, the database

has to seek across the disk to fetch the records requested. If the records are stored close together, the fetching operation is more efficient. For example, a clustered index on a date column where the data is ordered sequentially by date. A query against a specific date range results in an ordered fetch from the disk, which leverages fast sequential access.

To cluster an index in Greenplum Database

Using the `CLUSTER` command to physically reorder a table based on an index can take a long time with very large tables. To achieve the same results much faster, you can manually reorder the data on disk by creating an intermediate table and loading the data in the desired order. For example:

```
CREATE TABLE new_table (LIKE old_table)
       AS SELECT * FROM old_table ORDER BY myixcolumn;
DROP old_table;
ALTER TABLE new_table RENAME TO old_table;
CREATE INDEX myixcolumn_ix ON old_table;
VACUUM ANALYZE old_table;
```

Index Types

Greenplum Database supports the Postgres index types B-tree and GiST. Hash and GIN indexes are not supported. Each index type uses a different algorithm that is best suited to different types of queries. B-tree indexes fit the most common situations and are the default index type. See *Index Types* in the PostgreSQL documentation for a description of these types.

Note: Greenplum Database allows unique indexes only if the columns of the index key are the same as (or a superset of) the Greenplum distribution key. Unique indexes are not supported on append-optimized tables. On partitioned tables, a unique index cannot be enforced across all child table partitions of a partitioned table. A unique index is supported only within a partition.

About Bitmap Indexes

Greenplum Database provides the Bitmap index type. Bitmap indexes are best suited to data warehousing applications and decision support systems with large amounts of data, many ad hoc queries, and few data modification (DML) transactions.

An index provides pointers to the rows in a table that contain a given key value. A regular index stores a list of tuple IDs for each key corresponding to the rows with that key value. Bitmap indexes store a bitmap for each key value. Regular indexes can be several times larger than the data in the table, but bitmap indexes provide the same functionality as a regular index and use a fraction of the size of the indexed data.

Each bit in the bitmap corresponds to a possible tuple ID. If the bit is set, the row with the corresponding tuple ID contains the key value. A mapping function converts the bit position to a tuple ID. Bitmaps are compressed for storage. If the number of distinct key values is small, bitmap indexes are much smaller, compress better, and save considerable space compared with a regular index. The size of a bitmap index is proportional to the number of rows in the table times the number of distinct values in the indexed column.

Bitmap indexes are most effective for queries that contain multiple conditions in the `WHERE` clause. Rows that satisfy some, but not all, conditions are filtered out before the table is accessed. This improves response time, often dramatically.

When to Use Bitmap Indexes

Bitmap indexes are best suited to data warehousing applications where users query the data rather than update it. Bitmap indexes perform best for columns that have between 100 and 100,000 distinct values and when the indexed column is often queried in conjunction with other indexed columns. Columns with fewer than 100 distinct values, such as a gender column with two distinct values (male and female), usually do not benefit much from any type of index. On a column with more than 100,000 distinct values, the performance and space efficiency of a bitmap index decline.

Bitmap indexes can improve query performance for ad hoc queries. `AND` and `OR` conditions in the `WHERE` clause of a query can be resolved quickly by performing the corresponding Boolean operations directly on the bitmaps before converting the resulting bitmap to tuple ids. If the resulting number of rows is small, the query can be answered quickly without resorting to a full table scan.

When Not to Use Bitmap Indexes

Do not use bitmap indexes for unique columns or columns with high cardinality data, such as customer names or phone numbers. The performance gains and disk space advantages of bitmap indexes start to diminish on columns with 100,000 or more unique values, regardless of the number of rows in the table.

Bitmap indexes are not suitable for OLTP applications with large numbers of concurrent transactions modifying the data.

Use bitmap indexes sparingly. Test and compare query performance with and without an index. Add an index only if query performance improves with indexed columns.

Creating an Index

The `CREATE INDEX` command defines an index on a table. A B-tree index is the default index type. For example, to create a B-tree index on the column *gender* in the table *employee*:

```
CREATE INDEX gender_idx ON employee (gender);
```

To create a bitmap index on the column *title* in the table *films*:

```
CREATE INDEX title_bmp_idx ON films USING bitmap (title);
```

Examining Index Usage

Greenplum Database indexes do not require maintenance and tuning. You can check which indexes are used by the real-life query workload. Use the `EXPLAIN` command to examine index usage for a query.

The query plan shows the steps or *plan nodes* that the database will take to answer a query and time estimates for each plan node. To examine the use of indexes, look for the following query plan node types in your `EXPLAIN` output:

- **Index Scan** - A scan of an index.
- **Bitmap Heap Scan** - Retrieves all
- from the bitmap generated by BitmapAnd, BitmapOr, or BitmapIndexScan and accesses the heap to retrieve the relevant rows.
- **Bitmap Index Scan** - Compute a bitmap by OR-ing all bitmaps that satisfy the query predicates from the underlying index.
- **BitmapAnd** or **BitmapOr** - Takes the bitmaps generated from multiple BitmapIndexScan nodes, ANDs or ORs them together, and generates a new bitmap as its output.

You have to experiment to determine the indexes to create. Consider the following points.

- Run `ANALYZE` after you create or update an index. `ANALYZE` collects table statistics. The query optimizer uses table statistics to estimate the number of rows returned by a query and to assign realistic costs to each possible query plan.
- Use real data for experimentation. Using test data for setting up indexes tells you what indexes you need for the test data, but that is all.
- Do not use very small test data sets as the results can be unrealistic or skewed.
- Be careful when developing test data. Values that are similar, completely random, or inserted in sorted order will skew the statistics away from the distribution that real data would have.
- You can force the use of indexes for testing purposes by using run-time parameters to turn off specific plan types. For example, turn off sequential scans (`enable_seqscan`) and nested-loop joins

(`enable_nestloop`), the most basic plans, to force the system to use a different plan. Time your query with and without indexes and use the `EXPLAIN ANALYZE` command to compare the results.

Managing Indexes

Use the `REINDEX` command to rebuild a poorly-performing index. `REINDEX` rebuilds an index using the data stored in the index's table, replacing the old copy of the index.

To rebuild all indexes on a table

```
REINDEX my_table;
```

To rebuild a particular index

```
REINDEX my_index;
```

Dropping an Index

The `DROP INDEX` command removes an index. For example:

```
DROP INDEX title_idx;
```

When loading data, it can be faster to drop all indexes, load, then recreate the indexes.

Creating and Managing Views

Views enable you to save frequently used or complex queries, then access them in a SELECT statement as if they were a table. A view is not physically materialized on disk: the query runs as a subquery when you access the view.

If a subquery is associated with a single query, consider using the WITH clause of the SELECT command instead of creating a seldom-used view.

Creating Views

The CREATE VIEW command defines a view of a query. For example:

```
CREATE VIEW comedies AS SELECT * FROM films WHERE kind = 'comedy';
```

Views ignore ORDER BY and SORT operations stored in the view.

Dropping Views

The DROP VIEW command removes a view. For example:

```
DROP VIEW topten;
```

Chapter 21

Managing Data

This section provides information about manipulating data and concurrent access in Greenplum Database.

This topic includes the following subtopics:

- *About Concurrency Control in Greenplum Database*
- *Inserting Rows*
- *Updating Existing Rows*
- *Deleting Rows*
- *Working With Transactions*
- *Vacuuming the Database*

About Concurrency Control in Greenplum Database

Greenplum Database and PostgreSQL do not use locks for concurrency control. They maintain data consistency using a multiversion model, Multiversion Concurrency Control (MVCC). MVCC achieves transaction isolation for each database session, and each query transaction sees a snapshot of data. This ensures the transaction sees consistent data that is not affected by other concurrent transactions.

Because MVCC does not use explicit locks for concurrency control, lock contention is minimized and Greenplum Database maintains reasonable performance in multiuser environments. Locks acquired for querying (reading) data do not conflict with locks acquired for writing data.

Greenplum Database provides multiple lock modes to control concurrent access to data in tables. Most Greenplum Database SQL commands automatically acquire the appropriate locks to ensure that referenced tables are not dropped or modified in incompatible ways while a command executes. For applications that cannot adapt easily to MVCC behavior, you can use the `LOCK` command to acquire explicit locks. However, proper use of MVCC generally provides better performance.

Table 35: Lock Modes in Greenplum Database

Lock Mode	Associated SQL Commands	Conflicts With
ACCESS SHARE	`SELECT`	ACCESS EXCLUSIVE
ROW SHARE	`SELECT FOR SHARE`	EXCLUSIVE, ACCESS EXCLUSIVE
ROW EXCLUSIVE	`INSERT`, `COPY`	SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
SHARE UPDATE EXCLUSIVE	`VACUUM` (without `FULL`), `ANALYZE`	SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
SHARE	`CREATE INDEX`	ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE

Lock Mode	Associated SQL Commands	Conflicts With
SHARE ROW EXCLUSIVE		ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
EXCLUSIVE	DELETE, UPDATE, SELECT FOR UPDATE, See *Note*	ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE
ACCESS EXCLUSIVE	ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL	ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE

Note: In Greenplum Database, UPDATE, DELETE, and SELECT FOR UPDATE acquire the more restrictive lock EXCLUSIVE rather than ROW EXCLUSIVE.

Inserting Rows

Use the INSERT command to create rows in a table. This command requires the table name and a value for each column in the table; you may optionally specify the column names in any order. If you do not specify column names, list the data values in the order of the columns in the table, separated by commas.

For example, to specify the column names and the values to insert:

```
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

To specify only the values to insert:

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

Usually, the data values are literals (constants), but you can also use scalar expressions. For example:

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod <
'2004-05-07';
```

You can insert multiple rows in a single command. For example:

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

To insert data into a partitioned table, you specify the root partitioned table, the table created with the CREATE TABLE command. You also can specify a leaf child table of the partitioned table in an INSERT command. An error is returned if the data is not valid for the specified leaf child table. Specifying a child table that is not a leaf child table in the INSERT command is not supported.

To insert large amounts of data, use external tables or the COPY command. These load mechanisms are more efficient than INSERT for inserting large quantities of rows. See *Loading and Unloading Data* for more information about bulk data loading.

The storage model of append-optimized tables is optimized for bulk data loading. Greenplum does not recommend single row INSERT statements for append-optimized tables. For append-optimized tables, Greenplum Database supports a maximum of 127 concurrent INSERT transactions into a single append-optimized table.

Updating Existing Rows

The UPDATE command updates rows in a table. You can update all rows, a subset of all rows, or individual rows in a table. You can update each column separately without affecting other columns.

To perform an update, you need:

- The name of the table and columns to update
- The new values of the columns
- One or more conditions specifying the row or rows to be updated.

For example, the following command updates all products that have a price of *5* to have a price of *10*:

```
UPDATE products SET price = 10 WHERE price = 5;
```

Using UPDATE in Greenplum Database has the following restrictions:

- The Greenplum distribution key columns may not be updated.
- If mirrors are enabled, you cannot use STABLE or VOLATILE functions in an UPDATE statement.
- Greenplum Database does not support the RETURNING clause.
- Greenplum Database partitioning columns cannot be updated.

Deleting Rows

The DELETE command deletes rows from a table. Specify a WHERE clause to delete rows that match certain criteria. If you do not specify a WHERE clause, all rows in the table are deleted. The result is a valid, but empty, table. For example, to remove all rows from the products table that have a price of *10*:

```
DELETE FROM products WHERE price = 10;
```

To delete all rows from a table:

```
DELETE FROM products;
```

Using DELETE in Greenplum Database has similar restrictions to using UPDATE:

- If mirrors are enabled, you cannot use STABLE or VOLATILE functions in an UPDATE statement.
- The RETURNING clause is not supported in Greenplum Database.

Truncating a Table

Use the TRUNCATE command to quickly remove all rows in a table. For example:

```
TRUNCATE mytable;
```

This command empties a table of all rows in one operation. Note that TRUNCATE does not scan the table, therefore it does not process inherited child tables or ON DELETE rewrite rules. The command truncates only rows in the named table.

Working With Transactions

Transactions allow you to bundle multiple SQL statements in one all-or-nothing operation.

The following are the Greenplum Database SQL transaction commands:

- BEGIN or START TRANSACTION starts a transaction block.
- END or COMMIT commits the results of a transaction.
- ROLLBACK abandons a transaction without making any changes.

- `SAVEPOINT` marks a place in a transaction and enables partial rollback. You can roll back commands executed after a savepoint while maintaining commands executed before the savepoint.
- `ROLLBACK TO SAVEPOINT` rolls back a transaction to a savepoint.
- `RELEASE SAVEPOINT` destroys a savepoint within a transaction.

Transaction Isolation Levels

Greenplum Database accepts the standard SQL transaction levels as follows:

- *read uncommitted* and *read committed* behave like the standard *read committed*
- *repeatable read* is disallowed. If the behavior of *repeatable read* is required, use *serializable*.
- *serializable* behaves in a manner similar to SQL standard *serializable*

The following information describes the behavior of the Greenplum transaction levels:

- **read committed/read uncommitted** — Provides fast, simple, partial transaction isolation. With read committed and read uncommitted transaction isolation, `SELECT`, `UPDATE`, and `DELETE` transactions operate on a snapshot of the database taken when the query started.

A `SELECT` query:

- Sees data committed before the query starts.
- Sees updates executed within the transaction.
- Does not see uncommitted data outside the transaction.
- Can possibly see changes that concurrent transactions made if the concurrent transaction is committed after the initial read in its own transaction.

Successive `SELECT` queries in the same transaction can see different data if other concurrent transactions commit changes before the queries start. `UPDATE` and `DELETE` commands find only rows committed before the commands started.

Read committed or read uncommitted transaction isolation allows concurrent transactions to modify or lock a row before `UPDATE` or `DELETE` finds the row. Read committed or read uncommitted transaction isolation may be inadequate for applications that perform complex queries and updates and require a consistent view of the database.

- **serializable** — Provides strict transaction isolation in which transactions execute as if they run one after another rather than concurrently. Applications on the serializable level must be designed to retry transactions in case of serialization failures. In Greenplum Database, `SERIALIZABLE` prevents dirty reads, non-repeatable reads, and phantom reads without expensive locking, but there are other interactions that can occur between some `SERIALIZABLE` transactions in Greenplum Database that prevent them from being truly serializable. Transactions that run concurrently should be examined to identify interactions that are not prevented by disallowing concurrent updates of the same data. Problems identified can be prevented by using explicit table locks or by requiring the conflicting transactions to update a dummy row introduced to represent the conflict.

A `SELECT` query:

- Sees a snapshot of the data as of the start of the transaction (not as of the start of the current query within the transaction).
- Sees only data committed before the query starts.
- Sees updates executed within the transaction.
- Does not see uncommitted data outside the transaction.
- Does not see changes that concurrent transactions made.

  Successive `SELECT` commands within a single transaction always see the same data.

  `UPDATE`, `DELETE, SELECT FOR UPDATE,` and `SELECT FOR SHARE` commands find only rows committed before the command started. If a concurrent transaction has already updated, deleted, or locked a target row when the row is found, the serializable or repeatable read transaction waits for the concurrent transaction to update the row, delete the row, or roll back.

If the concurrent transaction updates or deletes the row, the serializable or repeatable read transaction rolls back. If the concurrent transaction rolls back, then the serializable or repeatable read transaction updates or deletes the row.

The default transaction isolation level in Greenplum Database is *read committed*. To change the isolation level for a transaction, declare the isolation level when you `BEGIN` the transaction or use the `SET TRANSACTION` command after the transaction starts.

Vacuuming the Database

Deleted or updated data rows occupy physical space on disk even though new transactions cannot see them. Periodically running the `VACUUM` command removes these expired rows. For example:

```
VACUUM mytable;
```

The `VACUUM` command collects table-level statistics such as the number of rows and pages. Vacuum all tables after loading data, including append-optimized tables. For information about recommended routine vacuum operations, see *Routine Vacuum and Analyze*.

> **Important:** The `VACUUM`, `VACUUM FULL`, and `VACUUM ANALYZE` commands should be used to maintain the data in a Greenplum database especially if updates and deletes are frequently performed on your database data. See the `VACUUM` command in the *Greenplum Database Reference Guide* for information about using the command.

Configuring the Free Space Map

Expired rows are held in the *free space map*. The free space map must be sized large enough to hold all expired rows in your database. If not, a regular `VACUUM` command cannot reclaim space occupied by expired rows that overflow the free space map.

`VACUUM FULL` reclaims all expired row space, but it is an expensive operation and can take an unacceptably long time to finish on large, distributed Greenplum Database tables. If the free space map overflows, you can recreate the table with a `CREATE TABLE AS` statement and drop the old table. Pivotal recommends not using `VACUUM FULL`.

Size the free space map with the following server configuration parameters:

- `max_fsm_pages`
- `max_fsm_relations`

Chapter 22

Loading and Unloading Data

The topics in this section describe methods for loading and writing data into and out of a Greenplum Database, and how to format data files.

Greenplum Database supports high-performance parallel data loading and unloading, and for smaller amounts of data, single file, non-parallel data import and export.

Greenplum Database can read from and write to several types of external data sources, including text files, Hadoop file systems, and web servers.

- The `COPY` SQL command transfers data between an external text file on the master host and a Greenplum database table.
- External tables allow you to query data outside of the database directly and in parallel using SQL commands such as `SELECT`, `JOIN`, or `SORT EXTERNAL TABLE DATA`, and you can create views for external tables. External tables are often used to load external data into a regular database table using a command such as `CREATE TABLE table AS SELECT * FROM ext_table`.
- External web tables provide access to dynamic data. They can be backed with data from URLs accessed using the HTTP protocol or by the output of an OS script running on one or more segments.
- The `gpfdist` utility is the Greenplum parallel file distribution program. It is an HTTP server that is used with external tables to allow Greenplum segments to load external data in parallel, from multiple file systems. You can run multiple instances of `gpfdist` on different hosts and network interfaces and access them in parallel.
- The `gpload` utility automates the steps of a load task using a YAML-formatted control file.

The method you choose to load data depends on the characteristics of the source data—its location, size, format, and any transformations required.

In the simplest case, the `COPY` SQL command loads data into a table from a text file that is accessible to the Greenplum master instance. This requires no setup and provides good performance for smaller amounts of data. With the `COPY` command, the data copied into or out of the database passes between a single file on the master host and the database. This limits the total size of the dataset to the capacity of the file system where the external file resides and limits the data transfer to a single file write stream.

More efficient data loading options for large datasets take advantage of the Greenplum Database MPP architecture, using the Greenplum segments to load data in parallel. These methods allow data to load simultaneously from multiple file systems, through multiple NICs, on multiple hosts, achieving very high data transfer rates. External tables allow you to access external files from within the database as if they are regular database tables. When used with `gpfdist`, the Greenplum parallel file distribution program, external tables provide full parallelism by using the resources of all Greenplum segments to load or unload data.

Greenplum Database leverages the parallel architecture of the Hadoop Distributed File System to access files on that system.

Working with File-Based External Tables

External tables provide access to data stored in data sources outside of Greenplum Database as if the data were stored in regular database tables. Data can be read from or written to external tables.

An external table is a Greenplum database table backed with data that resides outside of the database. An external table is either readable or writable. It can be used like a regular database table in SQL commands such as `SELECT` and `INSERT` and joined with other tables. External tables are most often used to load and unload database data.

Web-based external tables provide access to data served by an HTTP server or an operating system process. See *Creating and Using Web External Tables* for more about web-based tables.

Accessing File-Based External Tables

External tables enable accessing external files as if they are regular database tables. They are often used to move data into and out of a Greenplum database.

To create an external table definition, you specify the format of your input files and the location of your external data sources. For information input file formats, see *Formatting Data Files*.

Use one of the following protocols to access external table data sources. You cannot mix protocols in `CREATE EXTERNAL TABLE` statements:

- `file://` accesses external data files on segment host that the Greenplum Database superuser (`gpadmin`) can access. See *file:// Protocol*.
- `gpfdist://` points to a directory on the file host and serves external data files to all Greenplum Database segments in parallel. See *gpfdist:// Protocol*.
- `gpfdists://` is the secure version of `gpfdist`. See *gpfdists:// Protocol*.
- `gphdfs://` accesses files on a Hadoop Distributed File System (HDFS). See *gphdfs:// Protocol*.

  The files can be stored on an Amazon EMR instance HDFS. See *Using Amazon EMR with Greenplum Database installed on AWS* .
- `s3://` accesses files in an Amazon S3 bucket. See *s3:// Protocol*.

External tables access external files from within the database as if they are regular database tables. External tables defined with the `gpfdist`/`gpfdists`, `gphdfs`, and `s3` protocols utilize Greenplum parallelism by using the resources of all Greenplum Database segments to load or unload data. The `gphdfs` protocol leverages the parallel architecture of the Hadoop Distributed File System to access files on that system. The `s3` protocol utilizes the Amazon Web Services (AWS) capabilities.

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.

The steps for using external tables are:

1. Define the external table.

   To use the `s3` protocol, you must also configure Greenplum Database and enable the protocol. See *s3:// Protocol*.
2. Do one of the following:
   - Start the Greenplum Database file server(s) when using the `gpfdist` or `gpdists` protocols.
   - Verify that you have already set up the required one-time configuration for the `gphdfs` protocol.
   - Verify the Greenplum Database configuration for the `s3` protocol.
3. Place the data files in the correct locations.
4. Query the external table with SQL commands.

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 in 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.

file:// Protocol

The `file://` protocol is used in a URI that specifies the location of an operating system file. The URI includes the host name, port, and path to the file. Each file must reside on a segment host in a location accessible by the Greenplum superuser (`gpadmin`). The host name used in the URI must match a segment host name registered in the `gp_segment_configuration` system catalog table.

The `LOCATION` clause can have multiple URIs, as shown in this example:

```
CREATE EXTERNAL TABLE ext_expenses (
   name text, date date, amount float4, category text, desc1 text )
LOCATION ('file://host1:5432/data/expense/*.csv',
          'file://host2:5432/data/expense/*.csv',
          'file://host3:5432/data/expense/*.csv')
FORMAT 'CSV' (HEADER);
```

The number of URIs you specify in the `LOCATION` clause is the number of segment instances that will work in parallel to access the external table. For each URI, Greenplum assigns a primary segment on the specified host to the file. For maximum parallelism when loading data, divide the data into as many equally sized files as you have primary segments. This ensures that all segments participate in the load. 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 four primary segment instances per segment host, you can place four external files on each segment host. 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;
```

gpfdist:// Protocol

The `gpfdist://` protocol is used in a URI to reference a running `gpfdist` instance. The `gpfdist` utility serves external data files from a directory on a file host to all Greenplum Database segments in parallel.

gpfdist is located in the $GPHOME/bin directory on your Greenplum Database master host and on each segment host.

Run gpfdist on the host where the external data files reside. gpfdist uncompresses gzip (.gz) and bzip2 (.bz2) files automatically. You can use the wildcard character (*) or other C-style pattern matching to denote multiple files to read. The files specified are assumed to be relative to the directory that you specified when you started the gpfdist instance.

All primary segments access the external file(s) in parallel, subject to the number of segments set in the gp_external_max_segments server configuration parameter. Use multiple gpfdist data sources in a CREATE EXTERNAL TABLE statement to scale the external table's scan performance. For more information about configuring gpfdist, see *Using the Greenplum Parallel File Server (gpfdist)*.

See the gpfdist reference documentation for more information about using gpfdist with external tables.

gpfdists:// Protocol

The gpfdists:// protocol is a secure version of the gpfdist:// protocol. To use it, you run the gpfdist utility with the --ssl option. When specified in a URI, the gpfdists:// protocol 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.

gpfdists implements SSL security in a client/server scheme with the following attributes and limitations:

- 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.
- The client certificate file, client.crt
- The client private key file, client.key

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 gpload YAML control file with the SSL option set to true. Running gpload starts the gpfdist server with the --ssl option, then uses the gpfdists protocol.

Using gpfdists 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 security, see *Example 3—Multiple gpfdists instances* .

gphdfs:// Protocol

The `gphdfs://` protocol specifies an external file path on a Hadoop Distributed File System (HDFS). The protocol allows specifying external files in Hadoop clusters configured with or without Hadoop HA (high availability) and in MapR clusters. File names may contain wildcard characters and the files can be in `CSV`, `TEXT`, or custom formats.

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 it contains. The following figure illustrates an external table located on a HDFS file system.

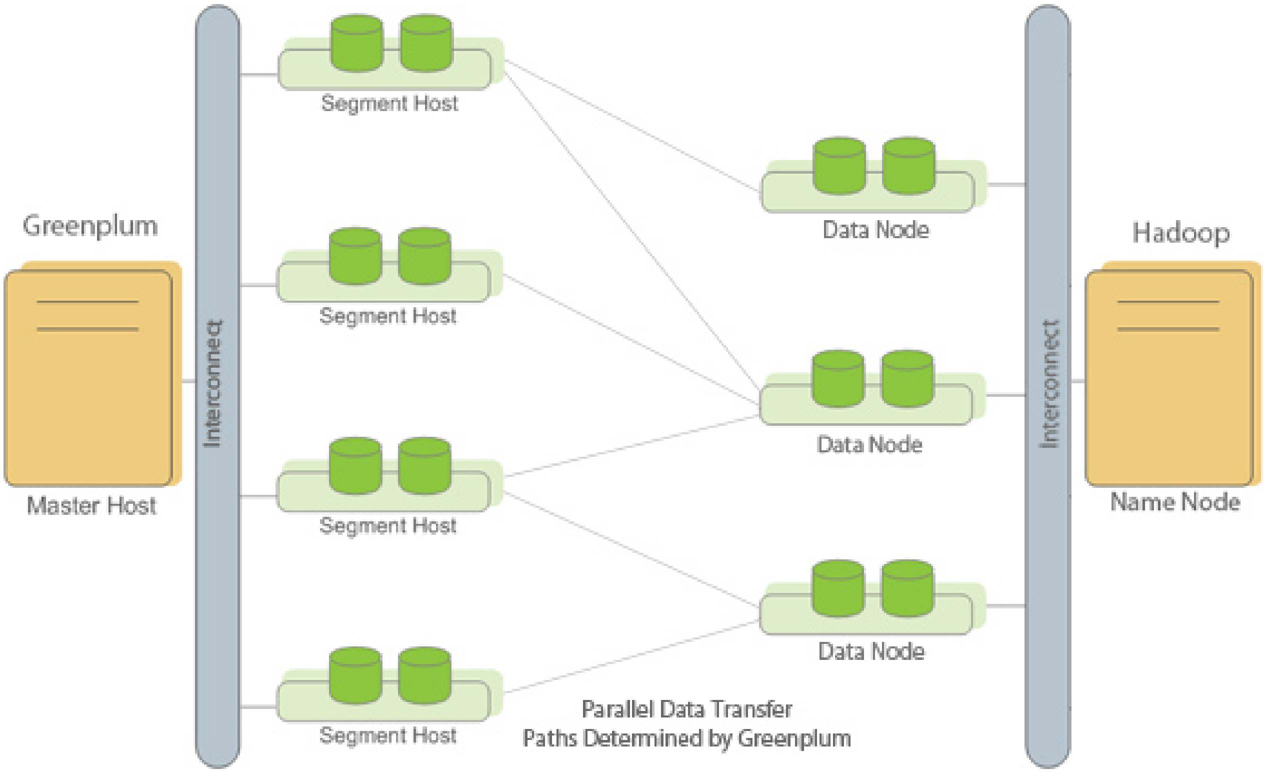


Figure 15: External Table Located on a Hadoop Distributed File System

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. 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*.

s3:// Protocol

Amazon Simple Storage Service (Amazon S3) provides secure, durable, highly-scalable object storage. For information about Amazon S3, see *Amazon S3*.

The `s3` protocol is used in a URL that specifies the location of an Amazon S3 bucket and a prefix to use for reading or writing files in the bucket. You can define read-only external tables that use existing data files in the S3 bucket for table data, or writeable external tables that store the data from INSERT operations to files in the S3 bucket. Greenplum Database uses the S3 URL and prefix specified in the protocol URL either to select one or more files for a read-only table, or to define the location and filename format to use when uploading S3 files for `INSERT` operations to writeable tables.

This topic contains the sections:

- *Configuring and Using S3 External Tables*
- *About the S3 Protocol URL*
- *About S3 Data Files*
- *About the S3 Protocol config Parameter*
- *s3 Protocol Configuration File*
- *s3 Protocol Limitations*
- *Using the gpcheckcloud Utility*

Configuring and Using S3 External Tables

Follow these basic steps to configure the S3 protocol and use S3 external tables, using the available links for more information. See also *s3 Protocol Limitations* to better understand the capabilities and limitations of S3 external tables:

1. Configure each database to support the `s3` protocol:

   a. In each database that will access an S3 bucket with the `s3` protocol, create the read and write functions for the `s3` protocol library:

   ```
   CREATE OR REPLACE FUNCTION write_to_s3() RETURNS integer AS
      '$libdir/gps3ext.so', 's3_export' LANGUAGE C STABLE;
   ```

   ```
   CREATE OR REPLACE FUNCTION read_from_s3() RETURNS integer AS
      '$libdir/gps3ext.so', 's3_import' LANGUAGE C STABLE;
   ```

   b. In each database that will access an S3 bucket, declare the `s3` protocol and specify the read and write functions you created in the previous step:

   ```
   CREATE PROTOCOL s3 (writefunc  = write_to_s3, readfunc = read_from_s3);
   ```

   Note: The protocol name `s3` must be the same as the protocol of the URL specified for the external table you create to access an S3 resource.

   The corresponding function is called by every Greenplum Database segment instance. All segment hosts must have access to the S3 bucket.

2. On each Greenplum Database segment, create and install the `s3` protocol configuration file:

   a. Create a template s3 protocol configuration file using the `gpcheckcloud` utility:

   ```
   gpcheckcloud -t > ./mytest_s3.config
   ```

   b. Edit the template file to specify the `accessid` and `secret` required to connect to the S3 location. See *s3 Protocol Configuration File* for information about configuration other protocol parameters.

   c. Copy the file to the same location and filename for all Greenplum Database segments on all hosts. The default file location is `*gpseg_data_dir*/*gpseg_prefixN*/s3/s3.conf`. *gpseg_data_dir* is the path to the Greenplum Database segment data directory, *gpseg_prefix* is the segment prefix, and *N* is the segment ID. The segment data directory, prefix, and ID are set when you initialize a Greenplum Database system.

   If you copy the file to a different location or filename, then you must specify the location with the `config` parameter in the `s3` protocol URL. See *About the S3 Protocol config Parameter*.

   d. Use the `gpcheckcloud` utility to validate connectivity to the S3 bucket:

   ```
   gpcheckcloud -c "s3://<s3-endpoint>/<s3-bucket> config=./mytest_s3.config"
   ```

Specify the correct path to the configuration file for your system, as well as the S3 endpoint name and bucket that you want to check. `gpcheckcloud` attempts to connect to the S3 endpoint and lists any files in the s3 bucket, if available. A successful connection ends with the message:

```
Your configuration works well.
```

You can optionally use `gpcheckcloud` to validate uploading to and downloading from the S3 bucket, as described in *Using the gpcheckcloud Utility*.

3. After completing the previous steps to create and configure the S3 protocol, you can specify an S3 protocol URL in the `CREATE EXTERNAL TABLE` command to define S3 external tables. For read-only S3 tables, the URL defines the location and prefix used to select existing data files that comprise the S3 table. For example:

```
CREATE READABLE EXTERNAL TABLE S3TBL (date text, time text, amt int)
   location('s3://s3-us-west-2.amazonaws.com/s3test.pivotal.io/dataset1/normal/
      config=/home/gpadmin/aws_s3/s3.conf')
   FORMAT 'csv';
```

For writeable S3 tables, the protocol URL defines the S3 location in which Greenplum database stores data files for the table, as well as a prefix to use when creating files for table `INSERT` operations. For example:

```
CREATE WRITABLE EXTERNAL TABLE S3WRIT (LIKE S3TBL)
   location('s3://s3-us-west-2.amazonaws.com/s3test.pivotal.io/dataset1/normal/
      config=/home/gpadmin/aws_s3/s3.conf')
   FORMAT 'csv';
```

See *About the S3 Protocol URL* for more information.

About the S3 Protocol URL

For the `s3` protocol, you specify a location for files and an optional configuration file location in the `LOCATION` clause of the `CREATE EXTERNAL TABLE` command. This is the syntax:

```
's3://S3_endpoint/bucket_name/[S3_prefix] [config=config_file_location]'
```

The `s3` protocol URL must the AWS S3 endpoint and S3 bucket name. Each Greenplum Database segment instance must have access to the S3 location. The optional *S3_prefix* value is used to select files for read-only S3 tables, or as a filename prefix to use when uploading files for S3 writeable tables.

> **Note:** Although the *S3_prefix* is an optional part of the syntax, you should always include an S3 prefix for both writeable and read-only S3 tables to separate datasets as part of the *CREATE EXTERNAL TABLE* syntax.

For writeable S3 tables, the `s3` protocol URL specifies the endpoint and bucket name where Greenplum Database uploads data files for the table. The S3 bucket permissions must be `Upload/Delete` for the S3 user ID that uploads the files. The S3 file prefix is used for each new file uploaded to the S3 location as a result of inserting data to the table. See *About S3 Data Files*.

For read-only S3 tables, the S3 file prefix is optional. If you specify an *S3_prefix*, then `s3` protocol selects all files that start with the specified prefix as data files for the external table. The `s3` protocol does not use the slash character (`/`) as delimiter, so a slash character follows a prefix is treated as part of the prefix itself.

For example, consider the following 5 files that each have `domain` as the *S3_endpoint*, and `test1` as the *bucket_name*:

```
s3://domain/test1/abc
s3://domain/test1/abc/
s3://domain/test1/abc/xx
s3://domain/test1/abcdef
```

```
s3://domain/test1/abcdefff
```

- If the S3 URL is provided as `s3://domain/test1/abc`, then the `abc` prefix selects all 5 files.
- If the S3 URL is provided as `s3://domain/test1/abc/`, then the `abc/` prefix selects the files `s3://domain/test1/abc/` and `s3://domain/test1/abc/xx`.
- If the S3 URL is provided as `s3://domain/test1/abcd`, then the `abcd` prefix selects the files `s3://domain/test1/abcdef` and `s3://domain/test1/abcdefff`

Wildcard characters are not supported in a *S3_prefix*; however, the S3 prefix functions as if a wildcard character immediately followed the prefix itself.

All of the files selected by the S3 URL (*S3_endpoint*/*bucket_name*/*S3_prefix*) are used as the source for the external table, so they must have the same format. Each file must also contain complete data rows. A data row cannot be split between files. The S3 file permissions must be `Open/Download` and `View` for the S3 user ID that is accessing the files.

For information about the AWS S3 endpoints see *http://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region*. For information about S3 buckets and folders, see the Amazon S3 documentation *http://aws.amazon.com/documentation/s3/*. For information about the S3 file prefix, see the Amazon S3 documentation *Listing Keys Hierarchically Using a Prefix and Delimiter*.

The `config` parameter specifies the location of the required `s3` protocol configuration file that contains AWS connection credentials and communication parameters. See *About the S3 Protocol config Parameter*.

About S3 Data Files

For each `INSERT` operation to a writeable S3 table, each Greenplum Database segment uploads a single file to the configured the S3 bucket using the filename format `<prefix><segment_id><random>.<extension>[.gz]` where:

- `<prefix>` is the prefix specified in the S3 URL.
- `<segment_id>` is the Greenplum Database segment ID.
- `<random>` is a random number that is used to ensure that the filename is unique.
- `<extension>` describes the file type (`.txt` or `.csv`, depending on the value you provide in the `FORMAT` clause of `CREATE WRITEABLE EXTERNAL TABLE`). Files created by the `gpcheckcloud` utility always uses the extension `.data`.
- `.gz` is appended to the filename if compression is enabled for S3 writeable tables (the default).

For writeable S3 tables, you can configure the buffer size and the number of threads that segments use for uploading files. See *s3 Protocol Configuration File*.

For read-only S3 tables, all of the files specified by the S3 file location (*S3_endpoint*/*bucket_name*/*S3_prefix*) are used as the source for the external table and must have the same format. Each file must also contain complete data rows. If the files contain an optional header row, the column names in the header row cannot contain a newline character (`\n`) or a carriage return (`\r`). Also, the column delimiter cannot be a newline character (`\n`) or a carriage return character (`\r`).

The `s3` protocol recognizes the gzip format and uncompress the files. Only the gzip compression format is supported.

The S3 file permissions must be `Open/Download` and `View` for the S3 user ID that is accessing the files. Writeable S3 tables require the S3 user ID to have `Upload/Delete` permissions.

For read-only S3 tables, each segment can download one file at a time from S3 location using several threads. To take advantage of the parallel processing performed by the Greenplum Database segments, the files in the S3 location should be similar in size and the number of files should allow for multiple segments to download the data from the S3 location. For example, if the Greenplum Database system consists of 16 segments and there was sufficient network bandwidth, creating 16 files in the S3 location allows each segment to download a file from the S3 location. In contrast, if the location contained only 1 or 2 files, only 1 or 2 segments download data.

About the S3 Protocol config Parameter

The optional `config` parameter specifies the location of the required `s3` protocol configuration file. The file contains AWS connection credentials and communication parameters. For information about the file, see *s3 Protocol Configuration File*.

The configuration file is required on all Greenplum Database segment hosts. This is default location is a location in the data directory of each Greenplum Database segment instance.

```
gpseg_data_dir/gpseg_prefixN/s3/s3.conf
```

The *gpseg_data_dir* is the path to the Greenplum Database segment data directory, the *gpseg_prefix* is the segment prefix, and *N* is the segment ID. The segment data directory, prefix, and ID are set when you initialize a Greenplum Database system.

If you have multiple segment instances on segment hosts, you can simplify the configuration by creating a single location on each segment host. Then you specify the absolute path to the location with the `config` parameter in the `s3` protocol `LOCATION` clause. This example specifies a location in the `gpadmin` home directory.

```
LOCATION ('s3://s3.amazonaws.com/test/my_data config=/home/gpadmin/s3.conf')
```

All segment instances on the hosts use the file `/home/gpadmin/s3.conf`.

s3 Protocol Configuration File

When using the `s3` protocol, an `s3` protocol configuration file is required on all Greenplum Database segments. The default location is:

```
gpseg_data_dir/gpseg-prefixN/s3/s3.conf
```

The *gpseg_data_dir* is the path to the Greenplum Database segment data directory, the *gpseg-prefix* is the segment prefix, and *N* is the segment ID. The segment data directory, prefix, and ID are set when you initialize a Greenplum Database system.

If you have multiple segment instances on segment hosts, you can simplify the configuration by creating a single location on each segment host. Then you can specify the absolute path to the location with the `config` parameter in the `s3` protocol `LOCATION` clause. However, note that both read-only and writeable S3 external tables use the same parameter values for their connections. If you want to configure protocol parameters differently for read-only and writeable S3 tables, then you must use two different `s3` protocol configuration files and specify the correct file in the `CREATE EXTERNAL TABLE` statement when you create each table.

This example specifies a single file location in the `s3` directory of the `gpadmin` home directory:

```
config=/home/gpadmin/s3/s3.conf
```

All segment instances on the hosts use the file `/home/gpadmin/s3/s3.conf`.

The `s3` protocol configuration file is a text file that consists of a `[default]` section and parameters This is an example configuration file:

```
[default]
secret = "secret"
accessid = "user access id"
threadnum = 3
chunksize = 67108864
```

You can use the Greenplum Database `gpcheckcloud` utility to test the S3 configuration file. See *Using the gpcheckcloud Utility*.

s3 Configuration File Parameters

accessid

Required. AWS S3 ID to access the S3 bucket.

secret

Required. AWS S3 passcode for the S3 ID to access the S3 bucket.

autocompress

For writeable S3 external tables, this parameter specifies whether to compress files (using gzip) before uploading to S3. Files are compressed by default if you do not specify this parameter.

chunksize

The buffer size that each segment thread uses for reading from or writing to the S3 server. The default is 64 MB. The minimum is 8MB and the maximum is 128MB.

When inserting data to a writeable S3 table, each Greenplum Database segment writes the data into its buffer (using multiple threads up to the `threadnum` value) until it is full, after which it writes the buffer to a file in the S3 bucket. This process is then repeated as necessary on each segment until the insert operation completes.

Because Amazon S3 allows a maximum of 10,000 parts for multipart uploads, the minimum `chunksize` value of 8MB supports a maximum insert size of 80GB per Greenplum database segment. The maximum `chunksize` value of 128MB supports a maximum insert size 1.28TB per segment. For writeable S3 tables, you must ensure that the `chunksize` setting can support the anticipated table size of your table. See *Multipart Upload Overview* in the S3 documentation for more information about uploads to S3.

threadnum

The maximum number of concurrent threads a segment can create when uploading data to or downloading data from the S3 bucket. The default is 4. The minimum is 1 and the maximum is 8.

encryption

Use connections that are secured with Secure Sockets Layer (SSL). Default value is `true`. The values `true`, `t`, `on`, `yes`, and `y` (case insensitive) are treated as `true`. Any other value is treated as `false`.

low_speed_limit

The upload/download speed lower limit, in bytes per second. The default speed is 10240 (10K). If the upload or download speed is slower than the limit for longer than the time specified by `low_speed_time`, then the connection is aborted and retried. After 3 retries, the `s3` protocol returns an error. A value of 0 specifies no lower limit.

low_speed_time

When the connection speed is less than `low_speed_limit`, this parameter specified the amount of time, in seconds, to wait before aborting an upload to or a download from the S3 bucket. The default is 60 seconds. A value of 0 specifies no time limit.

Note: Greenplum Database can require up to `threadnum * chunksize` memory on each segment host when uploading or downloading S3 files. Consider this `s3` protocol memory requirement when you configure overall Greenplum Database memory, and increase the value of *gp_vmem_protect_limit* as necessary.

s3 Protocol Limitations

These are `s3` protocol limitations:

- Only the S3 path-style URL is supported.

```
s3://S3_endpoint/bucketname/[S3_prefix]
```

- Only the S3 endpoint is supported. The protocol does not support virtual hosting of S3 buckets (binding a domain name to an S3 bucket).
- AWS signature version 2 and version 4 signing process are supported.

  For information about the S3 endpoints supported by each signing process, see *http://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region*.
- S3 encryption is not supported. The S3 file property **Server Side Encryption** must be `None`.
- For writeable S3 external tables, only the `INSERT` operation is supported. `UPDATE`, `DELETE`, and `TRUNCATE` operations are not supported.
- Because Amazon S3 allows a maximum of 10,000 parts for multipart uploads, the maximum `chunksize` value of 128MB supports a maximum insert size of 1.28TB per Greenplum database segment for writeable s3 tables. You must ensure that the `chunksize` setting can support the anticipated table size of your table. See *Multipart Upload Overview* in the S3 documentation for more information about uploads to S3.
- To take advantage of the parallel processing performed by the Greenplum Database segment instances, the files in the S3 location for read-only S3 tables should be similar in size and the number of files should allow for multiple segments to download the data from the S3 location. For example, if the Greenplum Database system consists of 16 segments and there was sufficient network bandwidth, creating 16 files in the S3 location allows each segment to download a file from the S3 location. In contrast, if the location contained only 1 or 2 files, only 1 or 2 segments download data.

Using the gpcheckcloud Utility

The Greenplum Database utility `gpcheckcloud` helps users create an `s3` protocol configuration file and test a configuration file. You can specify options to test the ability to access an S3 bucket with a configuration file, and optionally upload data to or download data from files in the bucket.

If you run the utility without any options, it sends a template configuration file to `STDOUT`. You can capture the output and create an `s3` configuration file to connect to Amazon S3.

The utility is installed in the Greenplum Database `$GPHOME/bin` directory.

Syntax

```
gpcheckcloud {-c | -d} "s3://S3_endpoint/bucketname/[S3_prefix]
 [config==path_to_config_file]"

gpcheckcloud -u <file_to_upload> "s3://S3_endpoint/bucketname/[S3_prefix]
 [config==path_to_config_file]"
gpcheckcloud -t

gpcheckcloud -h
```

Options

-c

Connect to the specified S3 location with the configuration specified in the `s3` protocol URL and return information about the files in the S3 location.

If the connection fails, the utility displays information about failures such as invalid credentials, prefix, or server address (DNS error), or server not available.

-d

Download data from the specified S3 location with the configuration specified in the `s3` protocol URL and send the output to `STDOUT`.

If files are gzip compressed, the uncompressed data is sent to `STDOUT`.

-u

Upload a file to the S3 bucket specified in the `s3` protocol URL using the specified configuration file if available. Use this option to test compression and `chunksize` and `autocompress` settings for your configuration.

-t

Sends a template configuration file to `STDOUT`. You can capture the output and create an `s3` configuration file to connect to Amazon S3.

-h

Display `gpcheckcloud` help.

Examples

This example runs the utility without options to create a template `s3` configuration file `mytest_s3.config` in the current directory.

```
gpcheckcloud -t > ./mytest_s3.config
```

This example attempts to upload a local file, `test-data.csv` to an S3 bucket location using the `s3` configuration file `s3.mytestconf`:

```
gpcheckcloud -u ./test-data.csv "s3://domain/test1/abc config=s3.mytestconf"
```

A successful upload results in one or more files placed in the S3 bucket using the filename format `abc<segment_id><random>.data[.gz]`. See *About S3 Data Files*.

This example attempts to connect to an S3 bucket location with the `s3` configuration file `s3.mytestconf`.

```
gpcheckcloud -c "s3://domain/test1/abc config=s3.mytestconf"
```

Download all files from the S3 bucket location and send the output to `STDOUT`.

```
gpcheckcloud -d "s3://domain/test1/abc config=s3.mytestconf"
```

Using a Custom Protocol

A custom protocol allows you to connect Greenplum Database to a data source that cannot be accessed with the `file://`, `gpfdist://`, or `gphdfs://` protocols.

Creating a custom protocol requires that you implement a set of C functions with specified interfaces, declare the functions in Greenplum Database, and then use the `CREATE TRUSTED PROTOCOL` command to enable the protocol in the database.

See *Example Custom Data Access Protocol* for an example.

Handling Errors in External Table Data

By default, if external table data contains an error, the command fails and no data loads into the target database table. 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 Load Errors*.

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.

Using the Greenplum Parallel File Server (gpfdist)

The `gpfdist` protocol 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 topic describes the setup and management tasks for using `gpfdist` with external tables.

- *About gpfdist Setup and Performance*
- *Controlling Segment Parallelism*
- *Installing gpfdist*
- *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 *Creating External Tables - Examples*).

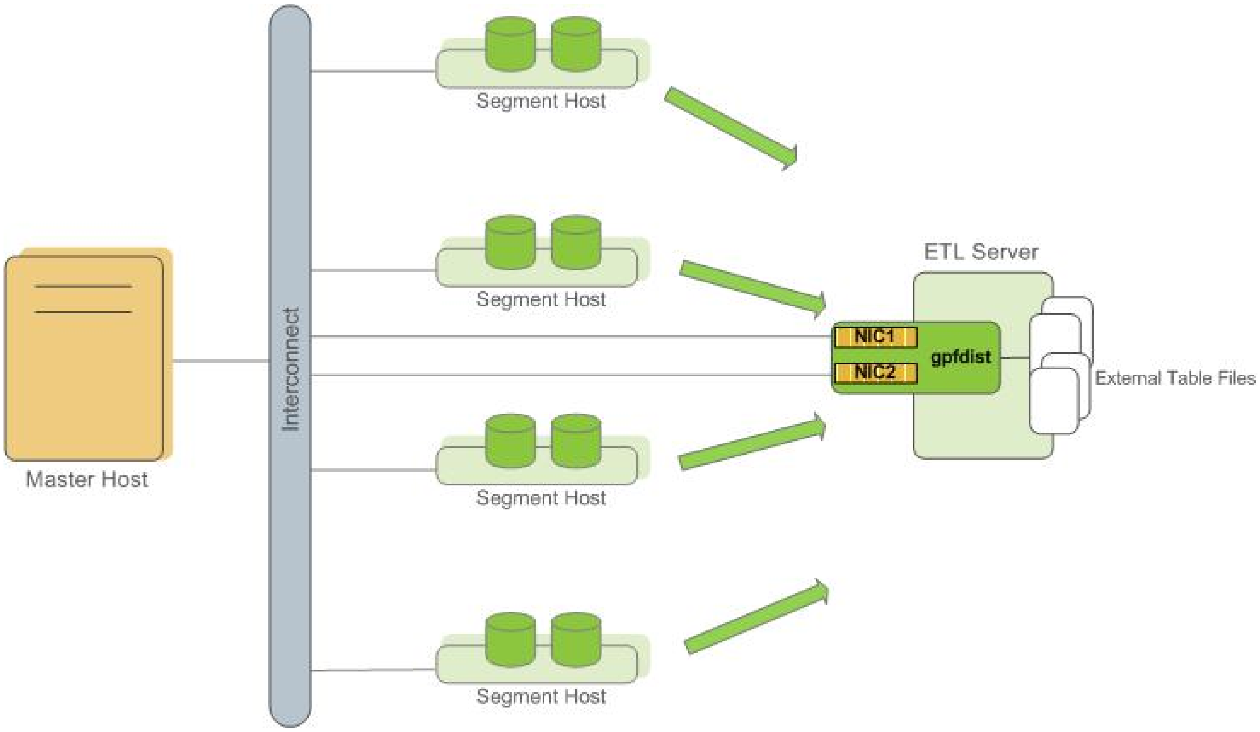


Figure 16: External Table Using Single gpfdist Instance with Multiple NICs

- 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.

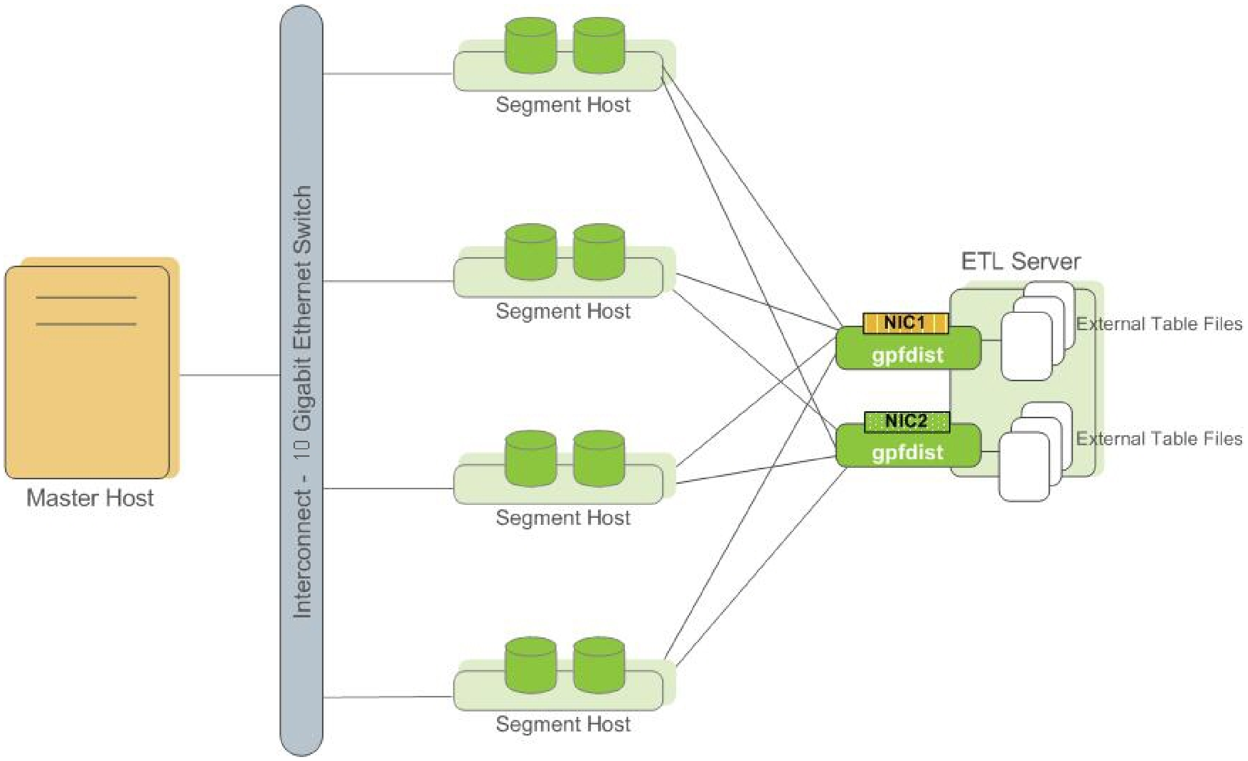


Figure 17: External Tables Using Multiple gpfdist Instances with Multiple NICs

> **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 gpfdist

`gpfdist` is installed in `$GPHOME/bin` of your Greenplum Database master host installation. 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: External Tables Using Multiple gpfdist Instances with Multiple NICs*), 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 *Creating External Tables - Examples*.

If you start gpfdist on your system and IPv6 networking is disabled, gpfdist displays this warning message when testing for an IPv6 port.

```
[WRN gpfdist.c:2050] Creating the socket failed
```

If the corresponding IPv4 port is available, gpfdist uses that port and the warning for IPv6 port can be ignored. To see information about the ports that gpfdist tests, use the -V option.

For information about IPv6 and IPv4 networking, see your operating system documentation.

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 steps to using HDFS:

- *One-time HDFS Protocol Installation*
- *Grant Privileges for the HDFS Protocol*
- *Specify HDFS Data in an External Table Definition*

For information about using Greenplum Database external tables with Amazon EMR when Greenplum Database is installed on Amazon Web Services (AWS), also see *Using Amazon EMR with Greenplum Database installed on AWS* .

For HDFS, Greenplum Database supports these additional HDFS-specific file formats:

- *Avro File Format*
- *Parquet File Format*

One-time HDFS Protocol Installation

Install and configure Hadoop for use with `gphdfs` as follows:

1. Install Java 1.6 or later on **all** Greenplum Database hosts: master, segment, and standby master.
2. Install a supported Hadoop distribution on all hosts. The distribution must be the same on all hosts. For Hadoop installation information, see the Hadoop distribution documentation.

   Greenplum Database supports the following Hadoop distributions:

Table 36: Hadoop Distributions

Hadoop Distribution	Version	gp_hadoop_ target_version
Pivotal HD	Pivotal HD 3.0, 3.0.1	`gphd-3.0`
	Pivotal HD 2.0, 2.1 Pivotal HD 1.0[1]	`gphd-2.0`
Greenplum HD	Greenplum HD 1.2	`gphd-1.2`
	Greenplum HD 1.1	`gphd-1.1` (default)
Cloudera	CDH 5.2, 5.3, 5.4.x - 5.8.x	`cdh5`
	CDH 5.0, 5.1	`cdh4.1`
	CDH 4.1[2] - CDH 4.7	`cdh4.1`
Hortonworks Data Platform	HDP 2.1, 2.2, 2.3	`hdp2`
MapR[3]	MapR 4.x, MapR 5.x	`gpmr-1.2`
	MapR 1.x, 2.x, 3.x	`gpmr-1.0`
Apache Hadoop	2.x	`hadoop2`

Note:

1. Pivotal HD 1.0 is a distribution of Hadoop 2.0.

2. For CDH 4.1, only CDH4 with MRv1 is supported.

3. MapR requires the MapR client software.

For the latest information regarding supported Hadoop distributions, see the *Greenplum Database Release Notes* for your release.

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 .bashrc profile.

```
export JAVA_HOME=/usr/java/default
export HADOOP_HOME=/usr/lib/gphd
```

   The variables must be set in the ~gpadmin/.bashrc or the ~gpadmin/.bash_profile file so that the gpadmin user shell environment can locate the Java home and Hadoop home.
5. Set the following Greenplum Database server configuration parameters and restart Greenplum Database.

Table 37: 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 gphd-2.0 gpmr-1.0 gpmr-1.2 hdp2 cdh3u2 cdh4.1	gphd-1.1	master session reload
gp_hadoop_home	When using Pivotal HD, specify the installation directory for Hadoop. For example, the default installation directory is /usr/lib/gphd. When using Greenplum HD 1.2 or earlier, specify the same value as the HADOOP_HOME environment variable.	NULL	master session reload

For example, the following commands use the Greenplum Database utilities gpconfig and gpstop to set the server configuration parameters and restart Greenplum Database:

```
gpconfig -c gp_hadoop_target_version -v "'gphd-2.0'"
gpconfig -c gp_hadoop_home -v "'/usr/lib/gphd'"
```

```
gpstop -u
```

For information about the Greenplum Database utilities gpconfig and gpstop, see the *Greenplum Database Utility Guide*.

6. If needed, ensure that the CLASSPATH environment variable generated by the $GPHOME/lib/hadoop/hadoop_env.sh file on every Greenplum Database host contains the path to JAR files that contain Java classes that are required for gphdfs.

   For example, if gphdfs returns a class not found exception, ensure the JAR file containing the class is on every Greenplum Database host and update the $GPHOME/lib/hadoop/hadoop_env.sh file so that the CLASSPATH environment variable created by file contains the JAR file.

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 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

The LOCATION clause of the CREATE EXTERNAL TABLE command for HDFS files differs slightly for Hadoop HA (High Availability) clusters, Hadoop clusters without HA, and MapR clusters.

In a Hadoop HA cluster, the LOCATION clause references the logical nameservices id (the dfs.nameservices property in the hdfs-site.xml configuration file). The hdfs-site.xml file with the nameservices configuration must be installed on the Greenplum master and on each segment host.

For example, if dfs.nameservices is set to mycluster the LOCATION clause takes this format:

```
LOCATION ('gphdfs://mycluster/path/filename.txt')
```

A cluster without HA specifies the hostname and port of the name node in the LOCATION clause:

```
LOCATION ('gphdfs://hdfs_host[:port]/path/filename.txt')
```

If you are using MapR clusters, you specify a specific cluster and the file:

- To specify the default cluster, the first entry in the MapR configuration file /opt/mapr/conf/mapr-clusters.conf, specify the location of your table with this syntax:

  ```
  LOCATION ('gphdfs:///file_path')
  ```

  The *file_path* is the path to the file.
- To specify another MapR cluster listed in the configuration file, specify the file with this syntax:

  ```
  LOCATION ('gphdfs:///mapr/cluster_name/file_path')
  ```

  The *cluster_name* is the name of the cluster specified in the configuration file and *file_path* is the path to the file.

For information about MapR clusters, see the MapR documentation.

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.
- The URI of the LOCATION clause cannot contain any of these four characters: \, ', <, >. The CREATE EXTERNAL TABLE returns a an error if the URI contains any of the characters.
- Format restrictions are as follows.
  - 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 (&).

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*.

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*.

MapReduce code 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 `gnet-1.1-javadoc.tar` 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`. Compile and run the MapReduce job with the cross-connect package. For example, compile and run the MapReduce job with `gphd-1.0-gnet-1.0.0.1.jar` if you use the Greenplum HD 1.0 distribution of Hadoop.

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 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());
      }
    }
  }

  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*.

   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.setOutputKeyLClass (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);

}
```

About gphdfs JVM Memory

When Greenplum Database accesses external table data from an HDFS location with gphdfs protocol, each Greenplum Database segment on a host system starts a JVM for use by the protocol. The default JVM heapsize is 1GB and should be enough for most workloads

If the gphdfs JVM runs out of memory, the issue might be related to the density of tuples inside the Hadoop HDFS block assigned to the gphdfs segment worker. A higher density of tuples per block requires more gphdfs memory. HDFS block size is usually 128MB, 256MB, or 512MB depending on the Hadoop cluster configuration.

You can increase the JVM heapsize by changing GP_JAVA_OPT variable in the file $GPHOME/lib/hadoop/hadoop_env.sh. In this example line, the option -Xmx1000m specifies that the JVM consumes 1GB of virtual memory.

```
export GP_JAVA_OPT='-Xmx1000m -XX:+DisplayVMOutputToStderr'
```

The $GPHOME/lib/hadoop/hadoop_env.sh must be updated for every segment instance in the Greenplum Database system.

> **Important:** Before increasing the gphdfs JVM memory, ensure that you have sufficient memory on the host. For example, 8 primary segments consume 8GB of virtual memory for the gphdfs JVM when using default. Increasing the Java -Xmx value to 2GB results in 16GB allocated in that environment of 8 segments per host.

Using Amazon EMR with Greenplum Database installed on AWS

Amazon Elastic MapReduce (EMR) is a managed cluster platform that can run big data frameworks, such as Apache Hadoop and Apache Spark, on Amazon Web Services (AWS) to process and analyze data. For a Greenplum Database system that is installed on Amazon Web Services (AWS), you can define Greenplum Database external tables that use the gphdfs protocol to access files on an Amazon EMR instance HDFS.

In addition to the steps described in *One-time HDFS Protocol Installation*, you must also ensure Greenplum Database can access the EMR instance. If your Greenplum Database system is running on an Amazon Elastic Compute Cloud (EC2) instance, you configure the Greenplum Database system and the EMR security group.

For information about Amazon EMR, see *https://aws.amazon.com/elasticmapreduce/*. For information about Amazon EC2, see *https://aws.amazon.com/ec2/*

Configuring Greenplum Database and Amazon EMR

These steps describe how to set up Greenplum Database system and an Amazon EMR instance to support Greenplum Database external tables:

1. Ensure that the appropriate Java (including JDK) and Hadoop environments are correctly installed on all Greenplum Database segment hosts.

   For example, Amazon EMR Release 4.0.0 includes Apache Hadoop 2.6.0. This Amazon page describes *Amazon EMR Release 4.0.0*.

   For information about Hadoop versions used by EMR and Greenplum Database, see *Table 38: EMR Hadoop Configuration Information*.
2. Ensure the environment variables and Greenplum Database server configuration parameters are set:

   - System environment variables:

     - HADOOP_HOME
     - JAVA_HOME

- Greenplum Database server configuration parameters:
  - `gp_hadoop_target_version`
  - `gp_hadoop_home`

3. Configure communication between Greenplum Database and the EMR instance Hadoop master.

   For example, open port 8020 in the AWS security group.
4. Configure for communication between Greenplum Database and EMR instance Hadoop data nodes. Open a TCP/IP port for so that Greenplum Database segments hosts can communicate with EMR instance Hadoop data nodes.

   For example, open port 50010 in the AWS security manager.

This table lists EMR and Hadooop version information that can be used to configure Greenplum Database.

Table 38: EMR Hadoop Configuration Information

EMR Version	EMR Apache Hadoop Version	EMR Hadoop Master Port	gp_hadoop_target_version	Hadoop Version on Greenplum Database Segment Hosts
4.0	2.6	8020	`hadoop2`	Apache Hadoop 2.x
3.9	2.4	9000	`hadoop2`	Apache Hadoop 2.x
3.8	2.4	9000	`hadoop2`	Apache Hadoop 2.x
3.3	2.4	9000	`hadoop2`	Apache Hadoop 2.x

Support for Avro Files

You can use the Greenplum Database `gphdfs` protocol to access Avro files on a Hadoop file system (HDFS).

About the Avro File Format

An Avro file stores both the data definition (schema) and the data together in one file making it easy for programs to dynamically understand the information stored in an Avro file. The Avro schema is in JSON format, the data is in a binary format making it compact and efficient.

The following example Avro schema defines an Avro record with 3 fields:

- name
- favorite_number
- favorite_color

```
{"namespace": "example.avro",
"type": "record", "name": "User",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "favorite_number", "type": ["int", "null"]},
    {"name": "favorite_color", "type": ["string", "null"]}
  ]
}
```

These are two rows of data based on the schema:

```
{ "name" : "miguno" , "favorite_number" : 6 , "favorite_color" : "red" }
{ "name" : "BlizzardCS" , "favorite_number" : 21 , "favorite_color" : "green" }
```

For information about the Avro file format, see *http://avro.apache.org/docs/1.7.7/*

Required Avro Jar Files

Support for the Avro file format requires these jar files:

avro-1.7.7.jar
avro-tools-1.7.7.jar
avro-mapred-1.7.5-hadoop2.jar (available with Apache Pig)

> **Note:** Hadoop 2 distributions include the Avro jar file `$HADOOP_HOME/share/hadoop/common/lib/avro-1.7.4.jar`. To avoid conflicts, you can rename the file to another file such as `avro-1.7.4.jar.bak`.
>
> For the Cloudera 5.4.x Hadoop distribution, only the jar file `avro-mapred-1.7.5-hadoop2.jar` needs to be downloaded and installed. The distribution contains the other required jar files. The other files are included in the `classpath` used by the `gphdfs` protocol.

For information about downloading the Avro jar files, see *https://avro.apache.org/releases.html*.

On all the Greenplum Database hosts, ensure that the jar files are installed and are on the `classpath` used by the `gphdfs` protocol. The `classpath` is specified by the shell script `$GPHOME/lib/hadoop/hadoop_env.sh`.

As an example, if the directory `$HADOOP_HOME/share/hadoop/common/lib` does not exist, create it on all Greenplum Database hosts as the `gpadmin` user. Then, add the add the jar files to the directory on all hosts.

The `hadoop_env.sh` script file adds the jar files to `classpath` for the `gphdfs` protocol. This fragment in the script file adds the jar files to the `classpath`.

```
if [ -d "${HADOOP_HOME}/share/hadoop/common/lib" ]; then
for f in ${HADOOP_HOME}/share/hadoop/common/lib/*.jar; do
            CLASSPATH=${CLASSPATH}:$f;
done
```

Avro File Format Support

The Greenplum Database `gphdfs` protocol supports the Avro file type as an external table:

- Avro file format - GPDB certified with Avro version 1.7.7
- Reading and writing Avro files
- Support for overriding the Avro schema when reading an Avro file
- Compressing Avro files during writing
- Automatic Avro schema generation when writing an Avro file

Greenplum Database returns an error if the Avro file contains unsupported features or if the specified schema does not match the data.

Reading from and Writing to Avro Files

To read from or write to an Avro file, you create an external table and specify the location of the Avro file in the `LOCATION` clause and `'AVRO'` in the `FORMAT` clause. For example, this is the syntax for a readable external table.

```
CREATE EXTERNAL TABLE tablename (column_spec) LOCATION ( 'gphdfs://location') FORMAT
 'AVRO'
```

The *location* can be an individual Avro file or a directory containing a set of Avro files. If the location specifies multiple files (a directory name or a file name containing wildcard characters), Greenplum Database uses the schema in the first file of the directory as the schema of the whole directory. For the file name you can specify the wildcard character * to match any number of characters.

You can add parameters after the file specified in the *location*. You add parameters with the http query string syntax that starts with `?` and `&` between field and value pairs.

For readable external tables, the only valid parameter is `schema`. The `gphdfs` uses this schema instead of the Avro file schema when reading Avro files. See *Avro Schema Overrides for Readable External Tables*.

For writable external tables, you can specify `schema`, `namespace`, and parameters for compression.

Table 39: Avro External Table Parameters

Parameter	Value	Readable/ Writeable	Default Value
schema	*URL_to_schema_file*	Read and Write	None. For a readable external table • The specified schema overrides the schema in the Avro file. See *Avro Schema Overrides* • If not specified, Greenplum Database uses the Avro file schema. For a writeable external table • Uses the specified schema when creating the Avro file. • If not specified, Greenplum Database creates a schema according to the external table definition.
namespace	*avro_namespace*	Write only	`public.avro` If specified, a valid Avro *namespace*.
compress	`true` or `false`	Write only	`false`
compression_type	`block`	Write only	Optional. For `avro` format, `compression_type` must be `block` if `compress` is `true`.
codec	`deflate` or `snappy`	Write only	`deflate`
codec_level (`deflate` codec only)	integer between 1 and 9	Write only	`6` The level controls the trade-off between speed and compression. Valid values are 1 to 9, where 1 is the fastest and 9 is the most compressed.

This set of parameters specify `snappy` compression:

```
'compress=true&codec=snappy'
```

These two sets of parameters specify `deflate` compression and are equivalent:

```
'compress=true&codec=deflate&codec_level=1'
'compress=true&codec_level=1'
```

Data Conversion When Reading Avro Files

When you create a readable external table to Avro file data, Greenplum Database converts Avro data types to Greenplum Database data types.

> **Note:** When reading an Avro, Greenplum Database converts the Avro field data at the top level of the Avro schema to a Greenplum Database table column. This is how the `gphdfs` protocol converts the Avro data types.

- An Avro primitive data type, Greenplum Database converts the data to a Greenplum Database type.
- An Avro complex data type that is not `map` or `record`, Greenplum Database converts the data to a Greenplum Database type.
- An Avro `record` that is a sub-record (nested within the top level Avro schema record), Greenplum Database converts the data XML.

This table lists the Avro primitive data types and the Greenplum Database type it is converted to.

Table 40: Avro Primitive Data Type Support for Readable External Tables

Avro Data Type	Greenplum Database Data Type
null	Supported only in a Avro union data type. See *Data Conversion when Writing Avro Files*.
boolean	boolean
int	int or smallint
long	bigint
float	real
double	double
bytes	bytea
string	text

> **Note:** When reading the Avro `int` data type as Greenplum Database `smallint` data type, you must ensure that the Avro `int` values do not exceed the Greenplum Database maximum `smallint` value. If the Avro value is too large, the Greenplum Database value will be incorrect.
>
> The `gphdfs` protocol converts performs this conversion for `smallint`: `short result = (short)IntValue;`.

This table lists the Avro complex data types and the and the Greenplum Database type it is converted to.

Table 41: Avro Complex Data Type Support for Readable External Tables

Avro Data Type	Greenplum Database Data Type
enum	int The integer represents the zero-based position of the symbol in the schema.
array	array The Greenplum Database array dimensions match the Avro array dimensions. The element type is converted from the Avro data type to the Greenplum Database data type.

Avro Data Type	Greenplum Database Data Type
maps	Not supported
union	The first non-null data type.
fixed	bytea
record	XML data

Example Avro Schema

This is an example Avro schema. When reading the data from the Avro file the `gphdfs` protocol performs these conversions:

- `name` and `color` data are converted to Greenplum Database `sting`.
- `age` data is converted to Greenplum Database `int`.
- `clist` records are converted to `XML`.

```
{"namespace": "example.avro",
  "type": "record",
  "name": "User",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "number", "type": ["int", "null"]},
    {"name": "color", "type": ["string", "null"]},
    {"name": "clist",
      "type": {
       "type":"record",
        "name":"clistRecord",
        "fields":[
          {"name": "class", "type": ["string", "null"]},
          {"name": "score", "type": ["double", "null"]},
          {"name": "grade",
            "type": {
             "type":"record",
              "name":"inner2",
              "fields":[
                {"name":"a", "type":["double" ,"null"]},
                {"name":"b", "type":["string","null"]}
             ]}
          },
         {"name": "grade2",
           "type": {
            "type":"record",
             "name":"inner",
             "fields":[
               {"name":"a", "type":["double","null"]},
               {"name":"b", "type":["string","null"]},
               {"name":"c", "type":{
                 "type": "record",
                 "name":"inner3",
                 "fields":[
                   {"name":"c1", "type":["string", "null"]},
                   {"name":"c2", "type":["int", "null"]}
               ]}}
           ]}
         }
      ]}
    }
  ]
}
```

This XML is an example of how the `gpfist` protocol converts Avro data from the `clist` field to XML data based on the previous schema. For records nested in the Avro top-level record, `gpfist` protocol converts the Avro element name to the XML element name and the name of the record is an attribute of the XML element. For example, the name of the top most element `clist` and the `type` attribute is the name of the Avro record element `clistRecord`.

```
<clist type="clistRecord">
  <class type="string">math</class>
  <score type="double">99.5</score>
  <grade type="inner2">
    <a type="double">88.8</a>
    <b type="string">subb0</b>
  </grade>
  <grade2 type="inner">
    <a type="double">77.7</a>
    <b type="string">subb20</b>
    <c type="inner3">
       <c1 type="string">subc</c1>
       <c2 type="int& quot;>0</c2>
    </c>
  </grade2>
</clist>
```

Avro Schema Overrides for Readable External Tables

When you specify schema for a readable external table that specifies an Avro file as a source, Greenplum Database uses the schema when reading data from the Avro file. The specified schema overrides the Avro file schema.

You can specify a file that contains an Avro schema as part of the location paramter `CREATE EXTERNAL TABLE` command, to override the Avro file schema. If a set of Avro files contain different, related schemas, you can specify an Avro schema to retrieve the data common to all the files.

Greenplum Database extracts the data from the Avro files based on the field name. If an Avro file contains a field with same name, Greenplum Database reads the data , otherwise a `NULL` is returned.

For example, if a set of Avro files contain one of the two different schemas. This is the original schema.

```
{
 "type":"record",
 "name":"tav2",
 "namespace":"public.avro",
 "doc":"",
 "fields":[
  {"name":"id","type":["null","int"],"doc":""},
  {"name":"name","type":["null","string"],"doc":""},
  {"name":"age","type":["null","long"],"doc":""},
  {"name":"birth","type":["null","string"],"doc":""}
 ]
}
```

This updated schema contains a comment field.

```
{
 "type":"record",
 "name":"tav2",
 "namespace":"public.avro",
 "doc":"",
 "fields":[
  {"name":"id","type":["null","int"],"doc":""},
  {"name":"name","type":["null","string"],"doc":""},
  {"name":"birth","type":["null","string"],"doc":""},
  {"name":"age","type":["null","long"],"doc":""},
  {"name":"comment","type":["null","string"],"doc":""}
 ]
}
```

You can specify an file containing this Avro schema in a `CREATE EXTERNAL TABLE` command, to read the `id`, `name`, `birth`, and `comment` fields from the Avro files.

```
{
 "type":"record",
 "name":"tav2",
 "namespace":"public.avro",
 "doc":"",
 "fields":[
  {"name":"id","type":["null","int"],"doc":""},
  {"name":"name","type":["null","string"],"doc":""},
  {"name":"birth","type":["null","string"],"doc":""},
  {"name":"comment","type":["null","string"],"doc":""}
 ]
}
```

In this example command, the customer data is in the Avro files `tmp/cust*.avro`. Each file uses one of the schemas listed previously. The file `avro/cust.avsc` is a text file that contains the Avro schema used to override the schemas in the customer files.

```
CREATE WRITABLE EXTERNAL TABLE cust_avro(id int, name text, birth date)
   LOCATION ('gphdfs://my_hdfs:8020/tmp/cust*.avro
      ?schema=hdfs://my_hdfs:8020/avro/cust.avsc')
   FORMAT 'avro';
```

When reading the Avro data, if Greenplum Database reads a file that does not contain a `comment` field, a `NULL` is returned for the `comment` data.

Data Conversion when Writing Avro Files

When you create a writable external table to write data to an Avro file, each table row is an Avro record and each table column is an Avro field. When writing an Avro file, the default compression algorithm is `deflate`.

For a writable external table, if the `schema` option is not specified, Greenplum Database creates an Avro schema for the Avro file based on the Greenplum Database external table definition. The name of the table column is the Avro field name. The data type is a union data type. See the following table:

Table 42: Avro Data Types Generated by Greenplum Database

Greenplum Database Data Type	Avro Union Data Type Definition
boolean	`["boolean", "null"]`
int	`["int", "null"]`
bigint	`["long", "null"]`
smallint	`["int", "null"]`
real	`["float", "null"]`
double	`["double", "null"]`
bytea	`["bytes", "null"]`
text	`["string", "null"]`
array	`[{`*`array`*`}, "null"]` The Greenplum Database array is converted to an Avro array with same dimensions and same element type as the Greenplum Database array.

Greenplum Database Data Type	Avro Union Data Type Definition
other data types	["string", "null"] Data are formatted strings. The gphdfs protocol casts the data to Greenplum Database text and writes the text to the Avro file as an Avro string. For example, date and time data are formatted as date and time strings and converted to Avro string type.

You can specify a schema with the schema option. When you specify a schema, the file can be on the segment hosts or a file on the HDFS that is accessible to Greenplum Database. For a local file, the file must exist in all segment hosts in the same location. For a file on the HDFS, the file must exist in the same cluster as the data file.

This example schema option specifies a schema on an HDFS.

```
'schema=hdfs://mytest:8000/avro/array_simple.avsc'
```

This example schema option specifies a schema on the host file system.

```
'schema=file:///mydata/avro_schema/array_simple.avsc'
```

gphdfs Limitations for Avro Files

For a Greenplum Database writeable external table definition, columns cannot specify the NOT NULL clause.

Greenplum Database supports only a single top-level schema in Avro files or specified with the schema parameter in the CREATE EXTERNAL TABLE command. An error is returned if Greenplum Database detects multiple top-level schemas.

Greenplum Database does not support the Avro map data type and returns an error when encountered.

When Greenplum Database reads an array from an Avro file, the array is converted to the literal text value. For example, the array [1,3] is converted to '{1,3}'.

User defined types (UDT), including array UDT, are supported. For a writable external table, the type is converted to string.

Examples

Simple CREATE EXTERNAL TABLE command that reads data from the two Avro fields id and ba.

```
 CREATE EXTERNAL TABLE avro1 (id int, ba bytea[])
   LOCATION ('gphdfs://my_hdfs:8020/avro/singleAvro/array2.avro')
   FORMAT 'avro';
```

CREATE WRITABLE EXTERNAL TABLE command specifies the Avro schema that is the gphdfs protocol uses to create the Avro file.

```
CREATE WRITABLE EXTERNAL TABLE at1w(id int, names text[], nums int[])
   LOCATION ('gphdfs://my_hdfs:8020/tmp/at1
      ?schema=hdfs://my_hdfs:8020/avro/array_simple.avsc')
   FORMAT 'avro';
```

CREATE WRITEABLE EXTERNAL TABLE command that writes to an Avro file and specifies a namespace for the Avro schema.

```
CREATE WRITABLE EXTERNAL TABLE atudt1 (id int, info myt, birth date, salary
 numeric )
   LOCATION ('gphdfs://my_hdfs:8020/tmp/emp01.avro
     ?namespace=public.pivotal.avro')
   FORMAT 'avro';
```

Support for Parquet Files

You can use the Greenplum Database `gphdfs` protocol to access Parquet files on a Hadoop file system (HDFS).

About the Parquet File Format

The Parquet file format is designed to take advantage of compressed, efficient columnar data representation available to projects in the Hadoop ecosystem. Parquet supports complex nested data structures and uses Dremel record shredding and assembly algorithms. Parquet supports very efficient compression and encoding schemes. Parquet allows compression schemes to be specified on a per-column level, and supports adding more encodings as they are invented and implemented.

For information about the Parquet, see the Parquet documentation *http://parquet.apache.org/documentation/latest/*.

For an overview of columnar data storage and the Parquet file format, see *https://blog.twitter.com/2013/dremel-made-simple-with-parquet*.

Required Parquet Jar Files

Support for the Parquet file format requires these jar files:

parquet-hadoop-1.7.0.jar
parquet-common-1.7.0.jar
parquet-encoding-1.7.0.jar
parquet-column-1.7.0.jar
parquet-generator-1.7.0.jar
parquet-format-2.3.0-incubating.jar

> **Note:** The Cloudera 5.4.x Hadoop distribution includes some Parquet jar files. However, the Java class names in the jar files are `parquet.xxx`. The `gphdfs` protocol uses the Java class names `org.apache.parquet.xxx`. The jar files with the class name `org.apache.parquet` can be downloaded and installed on the Greenplum Database hosts.

The `gphdfs` protocol also supports using `parquet-hadoop-bundle-1.7.0.jar` that contains the classes required to use Parquet within a Hadoop environment. These versions of `parquet-hadoop-bundle` are not supported:

- Version 1.6 and earlier. The versions do not use the Java class names `org.apache.parquet`
- Version 1.8 and later. The versions contain the class `VersionParser` that is not supported by `gphdfs`.

For information about downloading the Parquet jar files, see *http://parquet.apache.org/downloads/*

On all the Greenplum Database hosts, ensure that the jar files are installed and are on the `classpath` used by the `gphdfs` protocol. The `classpath` is specified by the shell script `$GPHOME/lib/hadoop/hadoop_env.sh`. As a Hadoop 2 example, you can install the jar files in `$HADOOP_HOME/share/hadoop/common/lib`. The `hadoop_env.sh` script file adds the jar files to the `classpath`.

As an example, if the directory `$HADOOP_HOME/share/hadoop/common/lib` does not exist, create it on all Greenplum Database hosts as the `gpadmin` user. Then, add the add the jar files to the directory on all hosts.

The `hadoop_env.sh` script file adds the jar files to `classpath` for the `gphdfs` protocol. This fragment in the script file adds the jar files to the `classpath`.

```
if [ -d "${HADOOP_HOME}/share/hadoop/common/lib" ]; then
for f in ${HADOOP_HOME}/share/hadoop/common/lib/*.jar; do
```

```
            CLASSPATH=${CLASSPATH}:$f;
done
```

Parquet File Format Support

The Greenplum Database `gphdfs` protocol supports the Parquet file format version 1 or 2. Parquet takes advantage of compressed, columnar data representation on HDFS. In a Parquet file, the metadata (Parquet schema definition) contains data structure information is written after the data to allow for single pass writing.

This is an example of the Parquet schema definition format:

```
message test {
    repeated byte_array binary_field;
    required int32 int32_field;
    optional int64 int64_field;
    required boolean boolean_field;
    required fixed_len_byte_array(3) flba_field;
    required byte_array someDay (utf8);
    };
```

The definition for last field `someDay` specifies the `binary` data type with the `utf8` annotation. The data type and annotation defines the data as a UTF-8 encoded character string.

Reading from and Writing to Parquet Files

To read from or write to a Parquet file, you create an external table and specify the location of the parquet file in the `LOCATION` clause and `'PARQUET'` in the `FORMAT` clause. For example, this is the syntax for a readable external table.

```
CREATE EXTERNAL TABLE tablename (column_spec) LOCATION ( 'gphdfs://location') FORMAT
 'PARQUET'
```

The *location* can be an Parquet file or a directory containing a set of Parquet files. For the file name you can specify the wildcard character * to match any number of characters. If the location specifies multiple files when reading Parquet files, Greenplum Database uses the schema in the first file that is read as the schema for the other files.

Reading a Parquet File

The following table lists how Greenplum database converts the Parquet data type if the Parquet schema definition does not contain an annotation.

Table 43: Data Type Conversion when Reading a Parquet File

Parquet Data Type	Greenplum Database Data Type
boolean	boolean
int32	int or smallint
int64	long
int96	bytea
float	real
double	double
fixed_lenth_byte_array	bytea
byte_array	bytea

Note: When reading the Parquet `int` data type as Greenplum Database `smallint` data type, you must ensure that the Parquet `int` values do not exceed the Greenplum Database maximum `smallint` value. If the value is too large, the Greenplum Database value will be incorrect.

The `gphdfs` protocol considers Parquet schema annotations for these cases. Otherwise, data conversion is based on the parquet schema primitive type:

Table 44: Data Type (with Annotation) Conversion when Reading Parquet File

Parquet Schema Data Type and Annotation	Greenplum Database Data Type
binary with `json` or `utf8` annotation	text
binary and the Greenplum Database column data type is text	text
int32 with `int_16` annotation	smallint
int32, int64, fixed_lenth_byte_array, or binary with `decimal` annotation	decimal
`repeated`	array column - The data type is converted according to *Table 43: Data Type Conversion when Reading a Parquet File*
`optional`, `required`	Data type is converted according to *Table 43: Data Type Conversion when Reading a Parquet File*

Note: See *Limitations and Notes* and the Parquet documentation when specifying `decimal`, `date`, `interval`, or `time*` annotations.

The `gphdfs` protocol converts the field data to text if the Parquet field type is binary without any annotation, and the data type is defined as text for the corresponding Greenplum Database external table column.

When reading Parquet type `group`, the `gphdfs` protocol converts the `group` data into an XML document.

This schema contains a required group with the name `inner`.

```
message test {
    required byte_array binary_field;
    required int64 int64_field;
    required group inner {
       int32 age;
       required boolean test;
       required byte_array name (UTF8);
       }
    };
```

This how a single row of the group data would be converted to XML.

```
<inner type="group">
  <age type="int">50</age>
  <test type="boolean">true</test>
  <name type="string">fred</name>
</inner>
```

This example schema contains a repeated group with the name `inner`.

```
message test {
    required byte_array binary_field;
    required int64 int64_field;
    repeated group inner {
       int32 age;
```

```
        required boolean test;
        required byte_array name (UTF8);
        }
    };
```

For a repeated `group`, the Parquet file can contain multiple sets of the group data in a single row. For the example schema, the data for the `inner` group is converted into XML data.

This is sample output if the data in the Parquet file contained two sets of data for the `inner` group.

```
<inner type="repeated">
  <inner type="group">
    <age type="int">50</age>
    <test type="boolean">true</test>
    <name type="string">fred</name>
  </inner>
  <inner>
    <age type="int">23</age>
    <test type="boolean">false</test>
    <name type="string">sam</name>
  </inner>
</inner>
```

Reading a Hive Generated Parquet File

The Apache Hive data warehouse software can manage and query large datasets that reside in distributed storage. Apache Hive 0.13.0 and later can store data in Parquet format files. For information about Parquet used by Apache Hive, see *https://cwiki.apache.org/confluence/display/Hive/Parquet*.

For Hive 1.1 data stored in Parquet files, this table lists how Greenplum database converts the data. The conversion is based on the Parquet schema that is generated by Hive. For information about the Parquet schema generated by Hive, see *Notes on the Hive Generated Parquet Schema*.

Table 45: Data Type Conversion when Reading a Hive Generated Parquet File

Hive Data Type	Greenplum Database Data Type
tinyint	int
smallint	int
int	int
bigint	bigint
decimal	numeric
float	real
double	float
boolean	boolean
string	text
char	text or char
varchar	text or varchar
timestamp	bytea
binary	bytea
array	xml
map	xml

Hive Data Type	Greenplum Database Data Type
struct	xml

Notes on the Hive Generated Parquet Schema

- When writing data to Parquet files, Hive treats all integer data types `tinyint`, `smallint`, `int` as `int32`. When you create an external table in Greenplum Database for a Hive generated Parquet file, specify the column data type as `int`. For example, this Hive `CREATE TABLE` command stores data in Parquet files.

  ```
  CREATE TABLE hpSimple(c1 tinyint, c2 smallint, c3 int, c4 bigint,
      c5 float, c6 double, c7 boolean, c8 string)
    STORED AS PARQUET;
  ```

  This is the Hive generated Parquet schema for the `hpSimple` table data.

  ```
  message hive_schema {
    optional int32 c1;
    optional int32 c2;
    optional int32 c3;
    optional int64 c4;
    optional float c5;
    optional double c6;
    optional boolean c7;
    optional binary c8 (UTF8);
  }
  ```

  The `gphdfs` protocol converts the Parquet integer data types to the Greenplum Database data type `int`.
- For the Hive `char` data type, the Greenplum Database column data types can be either `text` or `char`. For the Hive `varchar` data type, the Greenplum Database column data type can be either `text` or `varchar`.
- Based on the Hive generated Parquet schema, some Hive data is converted to Greenplum Database XML data. For example, Hive array column data that is stored in a Parquet file is converted to XML data. As an example, this the Hive generated Parquet schema for a Hive column `col1` of data type `array[int]`.

  ```
  optional group col1 (LIST) {
    repeated group bag {
      optional int32 array_element;
    }
  }
  ```

  The `gphdfs` protocol converts the Parquet `group` data to the Greenplum Database data type `XML`.
- For the Hive `timestamp` data type, the Hive generated Parquet schema for the data type specifies that the data is stored as data type `int96`. The `gphdfs` protocol converts the `int96` data type to the Greenplum Database `bytea` data type.

Writing a Parquet File

For writable external tables, you can add parameters after the file specified in the *location*. You add parameters with the http query string syntax that starts with `?` and `&` between field and value pairs.

Table 46: Parquet Format External Table location Parameters

Option	Values	Readable/ Writeable	Default Value
schema	*URL_to_schema*	Write only	None. If not specified, the `gphdfs` protocol creates a schema according to the external table definition.
pagesize	> 1024 Bytes	Write only	1 MB
rowgroupsize	> 1024 Bytes	Write only	8 MB
version	`v1`, `v2`	Write only	`v1`
codec	`UNCOMPRESSED`, `GZIP`, `LZO`, `snappy`	Write only	`UNCOMPRESSED`
dictionaryenable[1]	`true`, `false`	Write only	false
dictionarypagesize[1]	> 1024 Bytes	Write only	512 KB

Note:

1. Creates an internal dictionary. Enabling a dictionary can improve Parquet file compression if text columns contain similar or duplicate data.

When writing a Parquet file, the `gphdfs` protocol can generate a Parquet schema based on the table definition.

- The table name is used as the Parquet `message` name.
- The column name is uses as the Parquet `field` name.

When creating the Parquet schema from a Greenplum Database table definition, the schema is generated based on the column data type.

Table 47: Schema Data Type Conversion when Writing a Parquet File

Greenplum Database Data Type	Parquet Schema Data Type
boolean	optional boolean
smallint	optional int32 with annotation `int_16`
int	optional int32
bigint	optional int64
real	optional float
double	optional double
numeric or decimal	binary with annotation `decimal`
bytea	optional binary
array column	repeated field - The data type is the same data type as the Greenplum Database the array. For example, `array[int]` is converted to `repeated int`
Others	binary with annotation `utf8`

> **Note:** To support `Null` data, `gphdfs` protocol specifies the Parquet `optional` schema annotation when creating a Parquet schema.

A simple example of a Greenplum Database table definition and the Parquet schema generated by the `gphdfs` protocol.

An example external table definition for a Parquet file.

```
CREATE WRITABLE EXTERNAL TABLE films (
   code char(5),
   title varchar(40),
   id integer,
   date_prod date,
   subtitle boolean
) LOCATION ( 'gphdfs://my-films') FORMAT 'PARQUET' ;
```

This is the Parquet schema for the Parquet file `my-films` generated by the `gphdfs` protocol.

```
message films {
    optional byte_array code;
    optional byte_array title (utf8);
    optional int32 id;
    optional binary date_prod (utf8);
    optional boolean subtitle;
    };
```

Limitations and Notes

- For writable external tables, column definitions in Greenplum Database external table cannot specify `NOT NULL` to support automatically generating a Parquet schema. When the `gphdfs` protocol automatically generates a Parquet schema, the `gphdfs` protocol specifies the field attribute `optional` to support `null` in the Parquet schema. Repeated fields can be `null` in Parquet.
- The `gphdfs` protocol supports Parquet nested `group` structures only for readable external files. The nested structures are converted to an XML document.
- Greenplum Database does not have an unsigned `int` data type. Greenplum Database converts the Parquet unsigned `int` data type to the next largest Greenplum Database `int` type. For example, Parquet `uint_8` is converted to Greenplum Database `int` (32 bit).
- Greenplum Database supports any UDT data type or UDT array data type. Greenplum Database attempts to convert the UDT to a sting. If the UDT cannot be converted to a sting, Greenplum Database returns an error.
- The definition of the `Interval` data type in Parquet is significantly different than the `Interval` definition in Greenplum Database and cannot be converted. The Parquet `Interval` data is formatted as `bytea`.
- The `Date` data type in Parquet is starts from 1970.1.1, while `Date` in Greenplum Database starts from 4173 BC, Greenplum Database cannot convert `date` data types because largest values are different. A similar situation occurs between `Timestamp_millis` in Parquet and `Timestamp` in Greenplum Database.

Creating and Using Web External Tables

`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 host(s) 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 using `http://`. The web data file(s) 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 two external files to a Greenplum Database system with eight primary segments, two of the eight 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 );
```

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;
```

Loading and Writing Non-HDFS Custom Data

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.

- *Using a Custom Format*
- *Using a Custom Protocol*

For information about importing custom data from HDFS, see *Reading and Writing Custom-Formatted HDFS Data*.

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/students_out.txt')
FORMAT 'CUSTOM' (formatter=fixedwidth_out,
        name=20, address=30, age=4, line_delim=E'\r\n');
```

Using 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`.

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* .
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>
  ```

Creating 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 &
```

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:// Protocol*.

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 captured internally in Greenplum Database with a description of the error. See *Viewing Bad Rows in the Error Table or Error Log* for information about investigating error rows. You can view the errors, fix the issues, and then 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 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 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 '|');
```

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*.

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/data/international/*',
             'file://filehost/data/regional/*',
             'file://filehost/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 recieved 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*.

Handling Load Errors

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 and optionally logs formatting errors for erroneous rows, if the number of error rows does not reach the `SEGMENT REJECT LIMIT`.

The `LOG ERRORS` clause allows you to keep error rows for further examination. For information about the `LOG ERRORS` clause, see the `CREATE EXTERNAL TABLE` command in the *Greenplum Database Reference Guide*.

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;
```

> **Note:** When loading data with the `COPY` command or an external table, the value of the server configuration parameter `gp_initial_bad_row_limit` limits the initial number of rows that are processed that are not formatted properly. The default is to stop processing if the first 1000 rows contain formatting errors. See the *Greenplum Database Reference Guide* for information about the parameter.

Define an External Table with Single Row Error Isolation

The following example logs errors internally in Greenplum Database and sets an error threshold of 10 errors.

```
=# 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 SEGMENT REJECT LIMIT 10
     ROWS;
```

Use the built-in SQL function `gp_read_error_log('`*`external_table`*`')` to read the error log data. This example command displays the log errors for *ext_expenses*:

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

For information about the format of the error log, see *Viewing Bad Rows in the Error Table or Error Log*.

The built-in SQL function `gp_truncate_error_log('external_table')` deletes the error data. This example deletes the error log data created from the previous external table example :

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

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;
```

> **Note:** The clause `INTO err_table` that creates error tables is deprecated in Greenplum Database and will not be supported in the next major release.

Capture Row Formatting Errors and Declare a Reject Limit

The following SQL fragment captures formatting errors internally in Greenplum Database and declares a reject limit of 10 rows.

```
LOG ERRORS SEGMENT REJECT LIMIT 10 ROWS
```

Use the built-in SQL function `gp_read_error_log()` to read the error log data. For information about viewing log errors, see *Viewing Bad Rows in the Error Table or Error Log*.

Viewing Bad Rows in the Error Table or Error Log

If you use single row error isolation (see *Define an External Table with Single Row Error Isolation* or *Running COPY in Single Row Error Isolation Mode*), any rows with formatting errors are either logged internally or logged into an error table.

> **Note:** Storing row formatting in a error table is deprecated in Greenplum Database and will not be supported in a the next major release. Only internal error logs will be supported.

Greenplum Database captures the following error information in a table format:

Table 48: 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.

column	type	description
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 the SQL function `gp_read_error_log()` to display formatting errors that were logged internally in Greenplum Database. For example, this command displays the error log information for the table *ext_expenses*:

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

For information about managing formatting errors that are logged internally, see the command `COPY` or `CREATE EXTERNAL TABLE` in the *Greenplum Database Reference Guide*.

If you created an error table, 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 *Escaping in CSV Formatted Files* for more information on the correct use of quotes in CSV files.

Moving Data between Tables

You can use `CREATE TABLE AS` or `INSERT...SELECT` to load external and web external table data into another (non-external) database table, and the data will be 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 will happen, 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)

Loading Data with gpload

The Greenplum `gpload` utility loads data using readable external tables and the Greenplum parallel file server (`gpfdist` or `gpfdists`). It handles parallel file-based external table setup and allows users to configure their data format, external table definition, and `gpfdist` or `gpfdists` setup in a single configuration file.

To use gpload

1. Ensure that your environment is set up to run `gpload`. Some dependent files from your Greenplum Database installation are required, such as `gpfdist` and Python, as well as network access to the Greenplum segment hosts.

   See the *Greenplum Database Reference Guide* for details.
2. Create your load control file. This is a YAML-formatted file that specifies the Greenplum Database connection information, `gpfdist` configuration information, external table options, and data format.

   See the *Greenplum Database Reference Guide* for details.

   For example:

```
---
VERSION: 1.0.0.1
DATABASE: ops
USER: gpadmin
HOST: mdw-1
PORT: 5432
GPLOAD:
   INPUT:
    - SOURCE:
         LOCAL_HOSTNAME:
           - etl1-1
           - etl1-2
           - etl1-3
           - etl1-4
         PORT: 8081
         FILE:
           - /var/load/data/*
    - COLUMNS:
           - name: text
- amount: float4
           - category: text
           - desc: text
           - date: date
    - FORMAT: text
    - DELIMITER: '|'
    - ERROR_LIMIT: 25
    - ERROR_TABLE: payables.err_expenses
   OUTPUT:
    - TABLE: payables.expenses
    - MODE: INSERT
SQL:
   - BEFORE: "INSERT INTO audit VALUES('start', current_timestamp)"
   - AFTER: "INSERT INTO audit VALUES('end', current_timestamp)"
```

3. Run `gpload`, passing in the load control file. For example:

```
gpload -f my_load.yml
```

Loading Data with COPY

`COPY FROM` copies data from a file or standard input into a table and appends the data to the table contents. `COPY` is non-parallel: data is loaded in a single process using the Greenplum master instance. Using `COPY` is only recommended for very small data files.

The `COPY` source file must be accessible to the master host. Specify the `COPY` source file name relative to the master host location.

Greenplum copies data from `STDIN` or `STDOUT` using the connection between the client and the master server.

Running COPY in Single Row Error Isolation Mode

By default, `COPY` stops an operation at the first error: if the data contains an error, the operation fails and no data loads. If you run `COPY FROM` in *single row error isolation mode*, Greenplum skips rows that contain format errors and loads properly formatted rows. Single row error isolation mode applies only to rows in the input file that contain format errors. If the data contains a contraint error such as violation of a `NOT NULL`, `CHECK`, or `UNIQUE` constraint, the operation fails and no data loads.

Specifying `SEGMENT REJECT LIMIT` runs the `COPY` operation in single row error isolation mode. Specify the acceptable number of error rows on each segment, after which the entire `COPY FROM` operation fails and no rows load. The error row count is for each Greenplum segment, not for the entire load operation.

If the `COPY` operation does not reach the error limit, Greenplum loads all correctly-formatted rows and discards the error rows. The `LOG ERRORS INTO` clause allows you to keep error rows for further examination. Use `LOG ERRORS` to capture data formatting errors internally in Greenplum Database. For example:

```
=> COPY country FROM '/data/gpdb/country_data'
   WITH DELIMITER '|' LOG ERRORS
   SEGMENT REJECT LIMIT 10 ROWS;
```

See *Viewing Bad Rows in the Error Table or Error Log* for information about investigating error rows.

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 optimizer. Current statistics ensure that the optimizer 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.

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 topic describes how to unload data from Greenplum Database using parallel unload (writable external tables) and non-parallel unload (`COPY`).

- *Defining a File-Based Writable External Table*
- *Defining a Command-Based Writable External Web Table*
- *Unloading Data Using a Writable External Table*
- *Unloading Data Using COPY*

Defining a File-Based Writable External Table

Writable external tables that output data to files use 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 specify the location and format of the output files. See *Using the Greenplum Parallel File Server (gpfdist)* for instructions on setting up `gpfdist` for use with an external table and *Using Hadoop Distributed File System (HDFS) Tables* for instructions on setting up `gphdfs` for use with an external table.

- 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 column(s) 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);
```

Example 2—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.

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 command(s). 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 49: External Table EXECUTE Variables

Variable	Description
$GP_CID	Command count of the transaction 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.

Variable	Description
$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
```

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 table(s) 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;
```

Unloading Data Using COPY

COPY TO copies data from a table to a file (or standard input) on the Greenplum master host using a single process on the Greenplum master instance. Use COPY to output a table's entire contents, or filter the output using a SELECT statement. For example:

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

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.

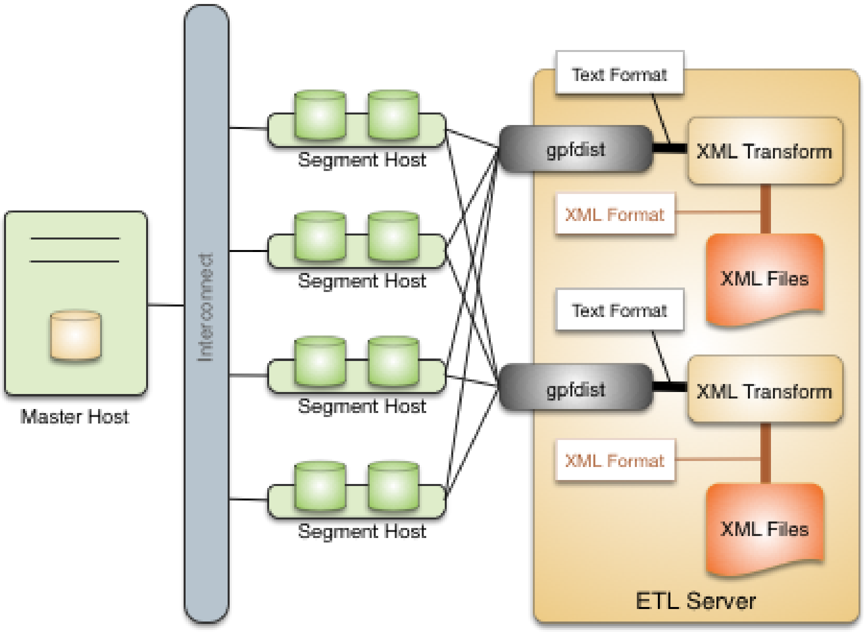


Figure 18: External Tables using XML Transformations

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 Pivotal 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.

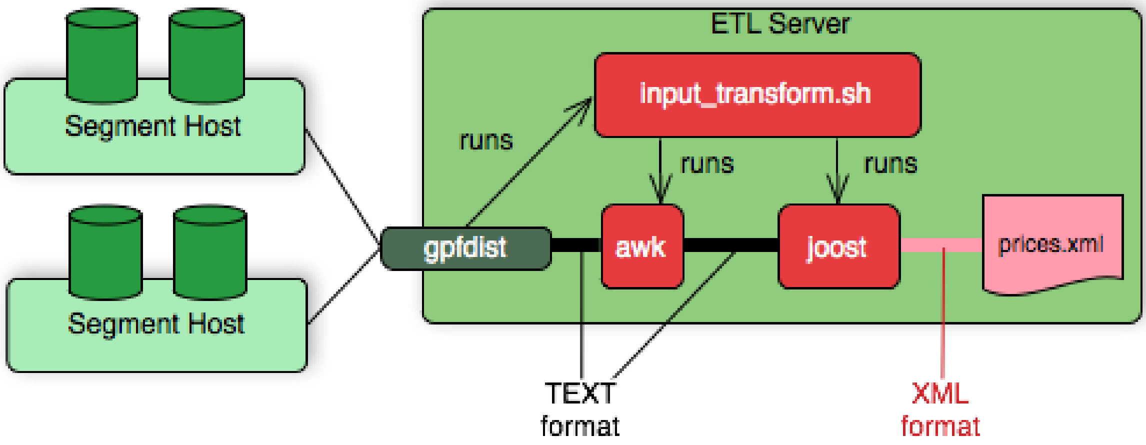
```
---
VERSION: 1.0.0.1
TRANSFORMATIONS:
  prices_input:
    TYPE:     input
    COMMAND:  /bin/bash input_transform.sh %filename%
```

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

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.

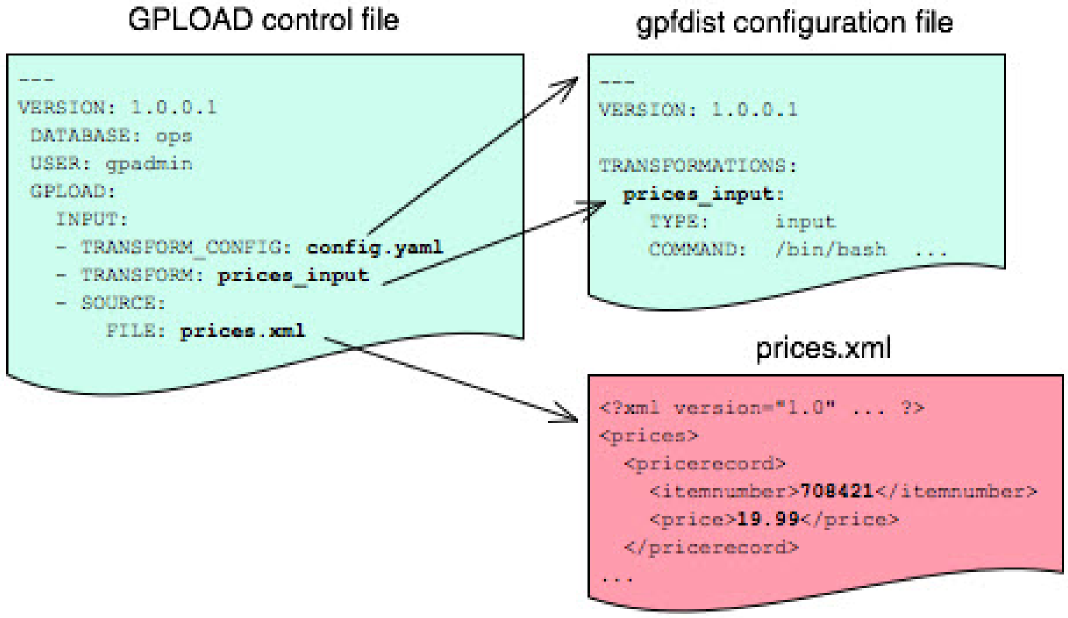
```
---
VERSION: 1.0.0.1
DATABASE: ops
USER: gpadmin
GPLOAD:
INPUT:
- TRANSFORM_CONFIG: config.yaml
- TRANSFORM: prices_input
- SOURCE:
```

```
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.



```
---
VERSION: 1.0.0.1
 DATABASE: ops
 USER: gpadmin
 GPLOAD:
   INPUT:
   - TRANSFORM_CONFIG: config.yaml
   - TRANSFORM: prices_input
   - SOURCE:
       FILE: prices.xml
```

```
---
VERSION: 1.0.0.1

TRANSFORMATIONS:
  prices_input:
    TYPE:     input
    COMMAND:  /bin/bash  ...
```

```
<?xml version="1.0" ... ?>
<prices>
  <pricerecord>
    <itemnumber>708421</itemnumber>
    <price>19.99</price>
  </pricerecord>
...
```

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 SEGMENT REJECT LIMIT 10;
```

In the command above, change `hostname` to your hostname. `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.

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 host(s) 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 '|');
```

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> element contains general details about the tax return such as the taxpayer's name, the tax year of the return, and the preparer. The <ReturnData> element 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 of 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@example.com|
```

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@example.com
- `doc xml`

Then, load the data into Greenplum Database.

Formatting 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 topic explains the format of data files expected by Greenplum Database.

- *Formatting Rows*
- *Formatting Columns*
- *Representing NULL Values*
- *Escaping*
- *Character Encoding*

Formatting Rows

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 X 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*.

Formatting Columns

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.

Escaping

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.

Escaping 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 stransmission 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.

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 *Using a Custom Protocol*.

```
/* ---- 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 *Using a Custom Protocol*. 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 Database 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);
}
```

Chapter 23

Querying Data

This topic provides information about using SQL in Greenplum databases.

You enter SQL statements called queries to view, change, and analyze data in a database using the `psql` interactive SQL client and other client tools.

- *About Greenplum Query Processing*
- *About the Pivotal Query Optimizer*
- *Defining Queries*
- *Using Functions and Operators*
- *Query Performance*
- *Managing Spill Files Generated by Queries*
- *Query Profiling*

About Greenplum Query Processing

This topic provides an overview of how Greenplum Database processes queries. Understanding this process can be useful when writing and tuning queries.

Users issue queries to Greenplum Database as they would to any database management system. 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 19: Dispatching the Parallel Query Plan*. The master dispatches targeted query plans to a single segment, as shown in *Figure 20: Dispatching a Targeted Query Plan*. Each segment is responsible for executing local database operations on its own set of data.query plans

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.

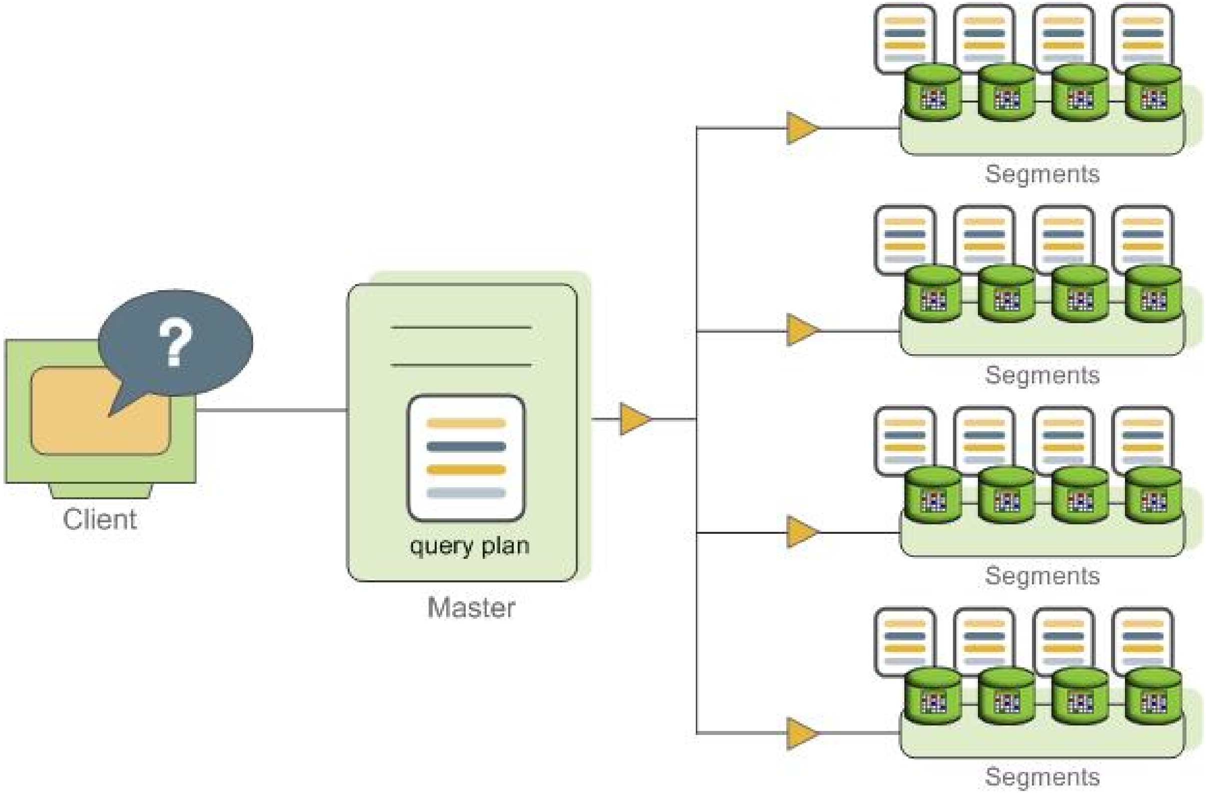


Figure 19: Dispatching the Parallel Query Plan

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).

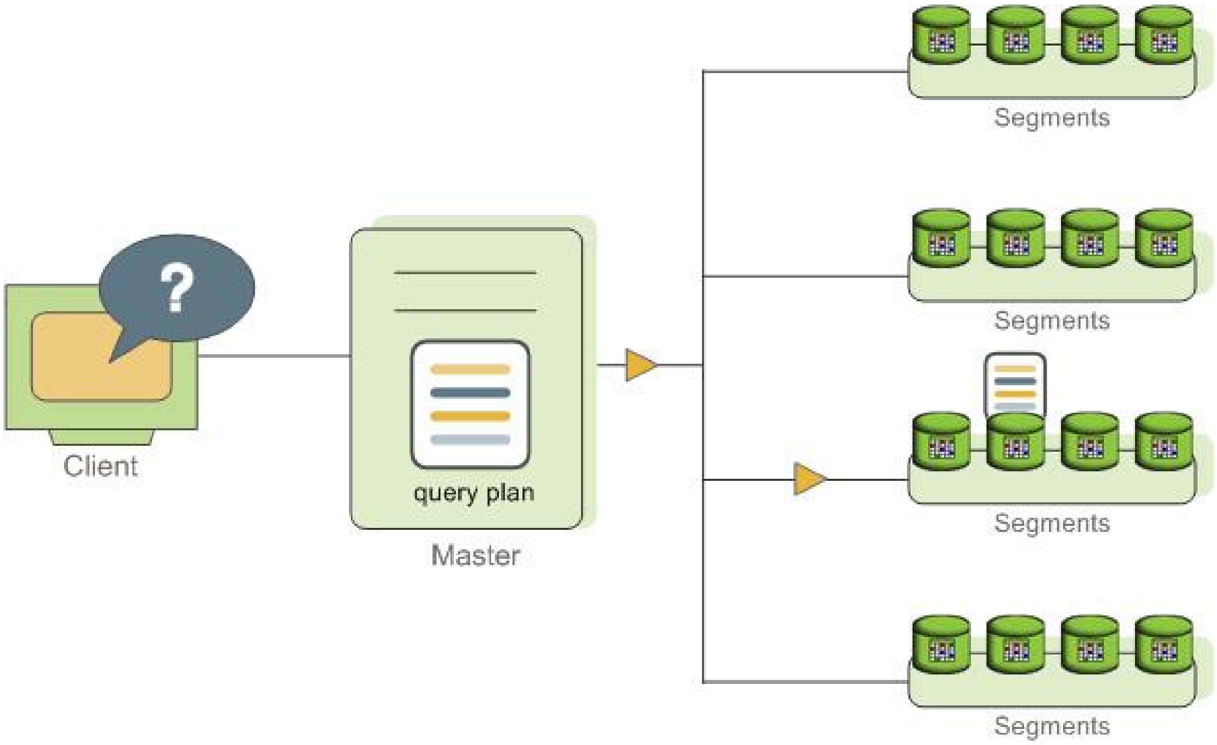


Figure 20: Dispatching a Targeted Query Plan

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 21: Query Slice Plan 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.

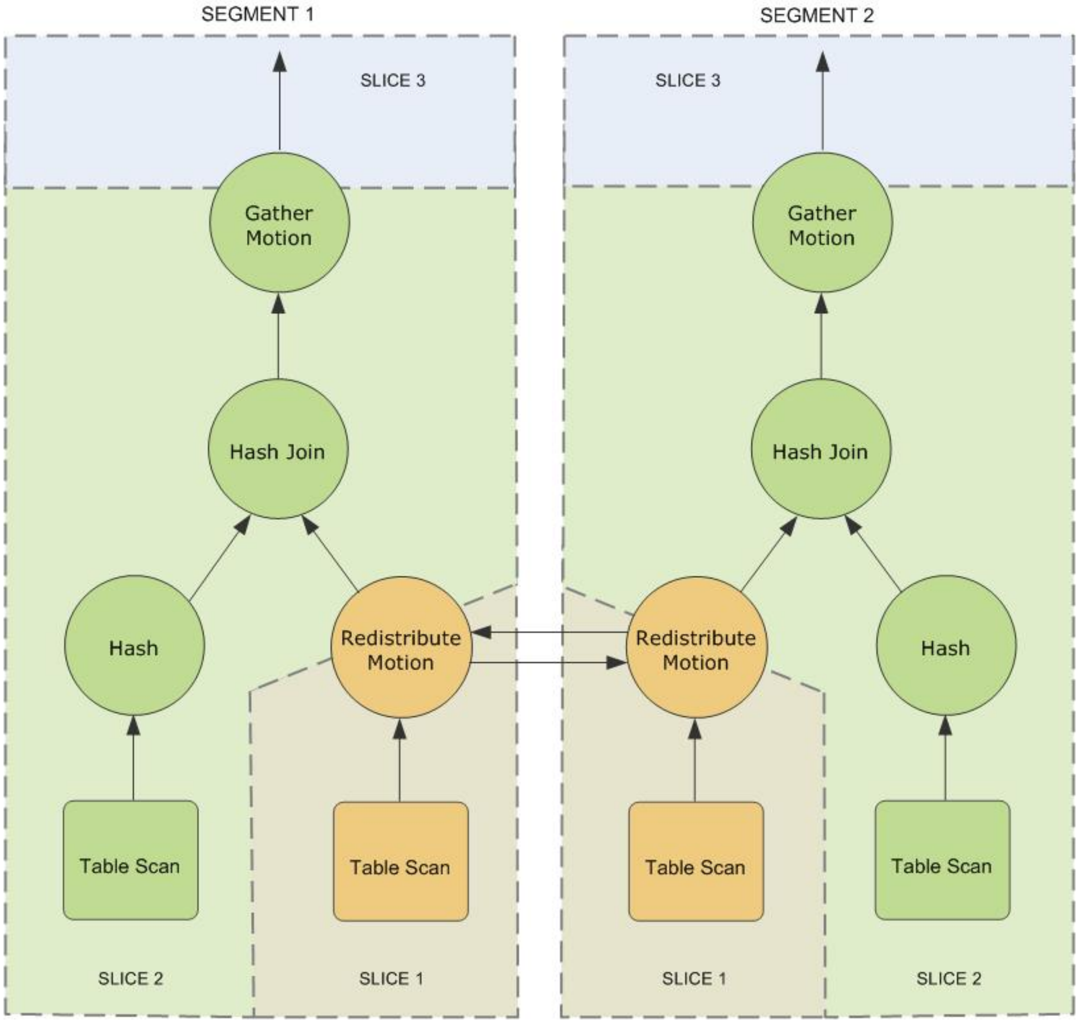


Figure 21: Query Slice Plan

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 22: Query Worker Processes shows the query worker processes on the master and two segment instances for the query plan illustrated in *Figure 21: Query Slice Plan*.

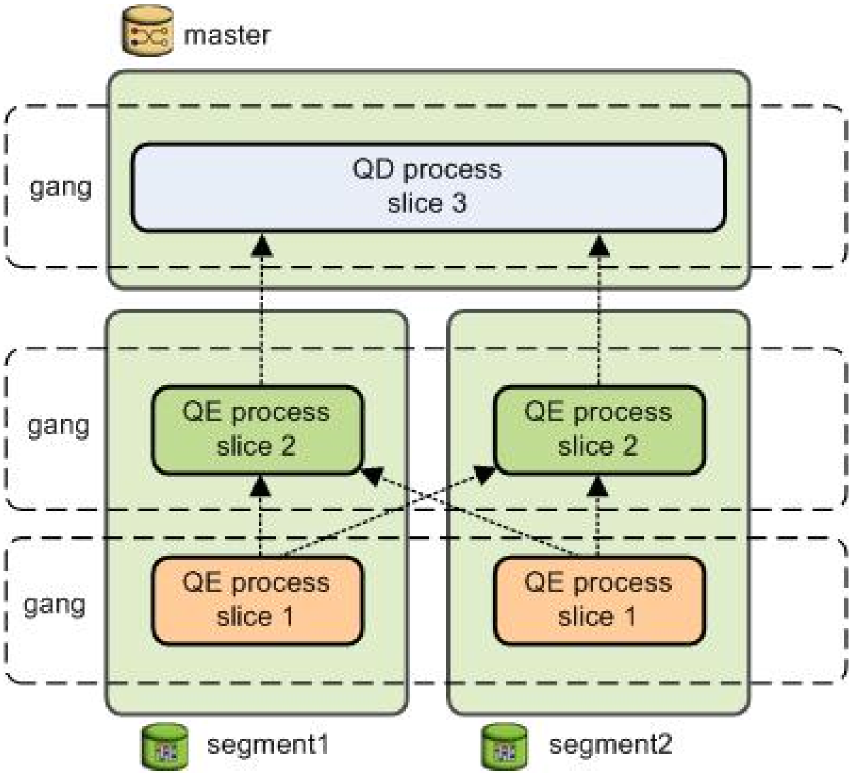


Figure 22: Query Worker Processes

About the Pivotal Query Optimizer

In Greenplum Database 4.3.5.0 and later, the Pivotal Query Optimizer co-exists with the legacy query optimizer.

These sections describe the Pivotal Query Optimizer functionality and usage:

- *Overview of the Pivotal Query Optimizer*
- *Enabling the Pivotal Query Optimizer*
- *Considerations when Using the Pivotal Query Optimizer*
- *Pivotal Query Optimizer Features and Enhancements*
- *Changed Behavior with the Pivotal Query Optimizer*
- *Pivotal Query Optimizer Limitations*
- *Determining the Query Optimizer that is Used*
- *About Uniform Multi-level Partitioned Tables*

Overview of the Pivotal Query Optimizer

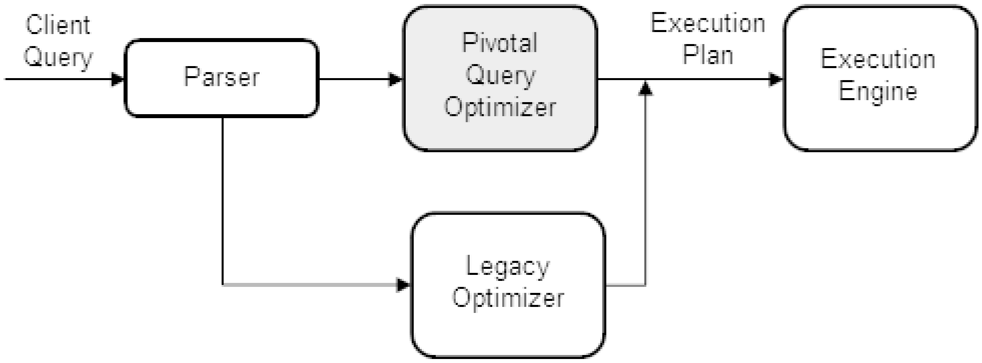
The Pivotal Query Optimizer extends the planning and optimization capabilities of the Greenplum Database legacy optimizer. The Pivotal Query Optimizer is extensible and achieves better optimization in multi-core architecture environments. When the Pivotal Query Optimizer is enabled, Greenplum Database uses the Pivotal Query Optimizer to generate an execution plan for a query when possible.

The Pivotal Query Optimizer also enhances Greenplum Database query performance tuning in the following areas:

- Queries against partitioned tables
- Queries that contain a common table expression (CTE)
- Queries that contain subqueries

In Greenplum Database 4.3.5.0 and later, the Pivotal Query Optimizer co-exists with the legacy query optimizer. By default, Greenplum Database uses the legacy query optimizer. When the Pivotal Query Optimizer is enabled, Greenplum Database uses the Pivotal Query Optimizer to generate an execution plan for a query when possible. If the Pivotal Query Optimizer cannot be used, the legacy query optimizer is used.

The following figure shows how the Pivotal Query Optimizer fits into the query planning architecture.



Note: All legacy query optimizer (planner) server configuration parameters are ignored by the Pivotal Query Optimizer. However, if Greenplum Database falls back to the legacy optimizer, the planner server configuration parameters will impact the query plan generation. For a list of legacy query optimizer (planner) server configuration parameters, see *Query Tuning Parameters*.

Enabling the Pivotal Query Optimizer

To enable Pivotal Query Optimizer, you set Greenplum Database server configuration parameters.

- *Set the `optimizer_analyze_root_partition` parameter to `on`* to enable statistics collection for the root partition of a partitioned table.
- Set the `optimizer` parameter to `on` to enable the Pivotal Query Optimizer. You can set the parameter at these levels:
  - *A Greenplum Database system*
  - *A specific Greenplum database*
  - *A session or query*

  Note: You can disable the ability to enable or disable the Pivotal Query Optimizer with the server configuration parameter `optimizer_control`. For information about the server configuration parameters, see the *Greenplum Database Reference Guide*.

  Important: If you intend to execute queries on partitioned tables with the Pivotal Query Optimizer enabled, you must collect statistics on the partitioned table root partition with the `ANALYZE ROOTPARTITION` command. The command `ANALYZE ROOTPARTITION` collects statistics on the root partition of a partitioned table without collecting statistics on the leaf partitions. If you specify a list of column names for a partitioned table, the statistics for the columns and the root partition are collected. For information on the `ANALYZE` command, see the *Greenplum Database Reference Guide*.

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

  As part of routine database maintenance, Pivotal recommends refreshing statistics on the root partition when there are significant changes to child leaf partition data.

Setting the optimizer_analyze_root_partition Parameter

When the configuration parameter `optimizer_analyze_root_partition` is set to `on`, root partition statistics will be collected when `ANALYZE` is run on a partitioned table. Root partition statistics are required by the Pivotal Query Optimizer.

1. Log into the Greenplum Database master host as `gpadmin`, the Greenplum Database administrator.
2. Set the values of the server configuration parameters. These Greenplum Database `gpconfig` utility commands sets the value of the parameters to `on`:

```
$ gpconfig -c optimizer_analyze_root_partition -v on --masteronly
```

3. Restart Greenplum Database. This Greenplum Database `gpstop` utility command reloads the `postgresql.conf` files of the master and segments without shutting down Greenplum Database.

```
gpstop -u
```

Enabling the Pivotal Query Optimizer for a System

Set the server configuration parameter `optimizer` for the Greenplum Database system.

1. Log into the Greenplum Database master host as `gpadmin`, the Greenplum Database administrator.
2. Set the values of the server configuration parameters. These Greenplum Database `gpconfig` utility commands sets the value of the parameters to `on`:

```
$ gpconfig -c optimizer -v on --masteronly
```

3. Restart Greenplum Database. This Greenplum Database `gpstop` utility command reloads the `postgresql.conf` files of the master and segments without shutting down Greenplum Database.

   ```
   gpstop -u
   ```

Enabling the Pivotal Query Optimizer for a Database

Set the server configuration parameter `optimizer` for individual Greenplum databases with the `ALTER DATABASE` command. For example, this command enables Pivotal Query Optimizer for the database *test_db*.

```
> ALTER DATABASE test_db SET OPTIMIZER = ON ;
```

Enabling the Pivotal Query Optimizer for a Session or a Query

You can use the `SET` command to set `optimizer` server configuration parameter for a session. For example, after you use the `psql` utility to connect to Greenplum Database, this `SET` command enables the Pivotal Query Optimizer:

```
> set optimizer = on ;
```

To set the parameter for a specific query, include the `SET` command prior to running the query.

Considerations when Using the Pivotal Query Optimizer

To execute queries optimally with the Pivotal Query Optimizer, query criteria to consider.

Ensure the following criteria are met:

- The table does not contain multi-column partition keys.
- The multi-level partitioned table is a uniform multi-level partitioned table. See *About Uniform Multi-level Partitioned Tables*.
- The server configuration parameter `optimizer_enable_master_only_queries` is set to `on` when running against master only tables such as the system table *pg_attribute*. For information about the parameter, see the *Greenplum Database Reference Guide*.

  > **Note:** Enabling this parameter decreases performance of short running catalog queries. To avoid this issue, set this parameter only for a session or a query.
- Statistics have been collected on the root partition of a partitioned table.

If the partitioned table contains more than 20,000 partitions, consider a redesign of the table schema.

These server configuration parameters affect Pivotal Query Optimizer query processing.

- The parameter `optimizer_join_order_threshold` specifies the maximum number of join children for which the Pivotal Query Optimizer uses the dynamic programming-based join ordering algorithm.
- The parameter `optimizer_parallel_union` controls the amount of parallelization that occurs for queries that contain a `UNION` or `UNION ALL` clause. When the value is `on`, Pivotal Query Optimizer can generate a query plan the child operations of a `UNION` or `UNION ALL` operation execute in parallel on segment instances.
- The parameter `optimizer_sort_factor` controls the cost factor that Pivotal Query Optimizer applies to sorting operations during query optimization. The cost factor can be adjusted for queries when data skew is present.

For information about the parameter, see the *Greenplum Database Reference Guide*.

The Pivotal Query Optimizer generates minidumps to describe the optimization context for a given query. The minidump files are used by Pivotal support to analyze Greenplum Database issues. The information in the file is not in a format that can be easily used by customers for debugging or troubleshooting. The minidump file is located under the master data directory and uses the following naming format:

Minidump_*date*_*time*.mdp

For information about the minidump file, see the server configuration parameter `optimizer_minidump` in the *Greenplum Database Reference Guide*.

When the `EXPLAIN ANALYZE` command uses the Pivotal Query Optimizer, the `EXPLAIN` plan shows only the number of partitions that are being eliminated. The scanned partitions are not shown. To show name of the scanned partitions in the segment logs set the server configuration parameter `gp_log_dynamic_partition_pruning` to `on`. This example `SET` command enables the parameter.

```
SET gp_log_dynamic_partition_pruning = on;
```

Pivotal Query Optimizer Features and Enhancements

The Pivotal Query Optimizer includes enhancements for specific types of queries and operations:

- *Queries Against Partitioned Tables*
- *Queries that Contain Subqueries*
- *Queries that Contain Common Table Expressions*
- *DML Operation Enhancements with Pivotal Query Optimizer*

Pivotal Query Optimizer also includes these optimization enhancements:

- Improved join ordering
- Join-Aggregate reordering
- Sort order optimization
- Data skew estimates included in query optimization

Queries Against Partitioned Tables

The Pivotal Query Optimizer includes these enhancements for queries against partitioned tables:

- Partition elimination is improved.
- Uniform multi-level partitioned tables are supported. For information about uniform multi-level partitioned tables, see *About Uniform Multi-level Partitioned Tables*
- Query plan can contain the `Partition selector` operator.
- Partitions are not enumerated in `EXPLAIN` plans.

  For queries that involve static partition selection where the partitioning key is compared to a constant, the Pivotal Query Optimizer lists the number of partitions to be scanned in the `EXPLAIN` output under the Partition Selector operator. This example Partition Selector operator shows the filter and number of partitions selected:

  ```
  Partition Selector for Part_Table (dynamic scan id: 1)
         Filter: a > 10
         Partitions selected:  1 (out of 3)
  ```

  For queries that involve dynamic partition selection where the partitioning key is compared to a variable, the number of partitions that are scanned will be known only during query execution. The partitions selected are not shown in the `EXPLAIN` output.
- Plan size is independent of number of partitions.
- Out of memory errors caused by number of partitions are reduced.

This example `CREATE TABLE` command creates a range partitioned table.

```
CREATE TABLE sales(order_id int, item_id int, amount numeric(15,2),
      date date, yr_qtr int)
   range partitioned by yr_qtr;
```

Pivotal Query Optimizer improves on these types of queries against partitioned tables:

- Full table scan. Partitions are not enumerated in plans.

  ```
  SELECT * FROM sales;
  ```

- Query with a constant filter predicate. Partition elimination is performed.

  ```
  SELECT * FROM sales WHERE yr_qtr = 201201;
  ```

- Range selection. Partition elimination is performed.

  ```
  SELECT * FROM sales WHERE yr_qtr BETWEEN 201301 AND 201404 ;
  ```

- Joins involving partitioned tables. In this example, the partitioned dimension table *date_dim* is joined with fact table *catalog_sales*:

  ```
  SELECT * FROM catalog_sales
     WHERE date_id IN (SELECT id FROM date_dim WHERE month=12);
  ```

Queries that Contain Subqueries

Pivotal Query Optimizer handles subqueries more efficiently. A subquery is query that is nested inside an outer query block. In the following query, the `SELECT` in the `WHERE` clause is a subquery.

```
SELECT * FROM part
  WHERE price > (SELECT avg(price) FROM part);
```

Pivotal Query Optimizer also handles queries that contain a correlated subquery (CSQ) more efficiently. A correlated subquery is a subquery that uses values from the outer query. In the following query, the `price` column is used in both the outer query and the subquery.

```
SELECT * FROM part p1
  WHERE price > (SELECT avg(price) FROM part p2
  WHERE  p2.brand = p1.brand);
```

Pivotal Query Optimizer generates more efficient plans for the following types of subqueries:

- CSQ in the `SELECT` list.

  ```
  SELECT *,
   (SELECT min(price) FROM part p2 WHERE p1.brand = p2.brand)
   AS foo
  FROM part p1;
  ```

- CSQ in disjunctive (`OR`) filters.

  ```
  SELECT FROM part p1 WHERE p_size > 40 OR
        p_retailprice >
        (SELECT avg(p_retailprice)
            FROM part p2
            WHERE p2.p_brand = p1.p_brand)
  ```

- Nested CSQ with skip level correlations

  ```
  SELECT * FROM part p1 WHERE p1.p_partkey
  IN (SELECT p_partkey FROM part p2 WHERE p2.p_retailprice =
       (SELECT min(p_retailprice)
         FROM part p3
         WHERE p3.p_brand = p1.p_brand)
  );
  ```

  Note: Nested CSQ with skip level correlations are not supported by the legacy query optimizer.

- CSQ with aggregate and inequality. This example contains a CSQ with an inequality.

  ```
  SELECT * FROM part p1 WHERE p1.p_retailprice =
   (SELECT min(p_retailprice) FROM part p2 WHERE p2.p_brand <> p1.p_brand);
  ```

- CSQ that must return one row.

  ```
  SELECT p_partkey,
    (SELECT p_retailprice FROM part p2 WHERE p2.p_brand = p1.p_brand )
  FROM part p1;
  ```

Queries that Contain Common Table Expressions

Pivotal Query Optimizer handles queries that contain the `WITH` clause. The `WITH` clause, also known as a common table expression (CTE), generates temporary tables that exist only for the query. This example query contains a CTE.

```
WITH v AS (SELECT a, sum(b) as s FROM T where c < 10 GROUP BY a)
  SELECT *FROM  v AS v1 ,  v AS v2
  WHERE v1.a <> v2.a AND v1.s < v2.s;
```

As part of query optimization, the Pivotal Query Optimizer can push down predicates into a CTE. For example query, Pivotal Query Optimizer pushes the equality predicates to the CTE.

```
WITH v AS (SELECT a, sum(b) as s FROM T GROUP BY a)
  SELECT *
  FROM v as v1, v as v2, v as v3
  WHERE v1.a < v2.a
    AND v1.s < v3.s
    AND v1.a = 10
    AND v2.a = 20
    AND v3.a = 30;
```

Pivotal Query Optimizer can handle these types of CTEs:

- CTE that defines one or multiple tables. In this query, the CTE defines two tables.

  ```
  WITH cte1 AS (SELECT a, sum(b) as s FROM T
                 where c < 10 GROUP BY a),
        cte2 AS (SELECT a, s FROM cte1 where s > 1000)
    SELECT *
    FROM cte1 as v1, cte2 as v2, cte2 as v3
    WHERE v1.a < v2.a AND v1.s < v3.s;
  ```

- Nested CTEs.

  ```
  WITH v AS (WITH w AS (SELECT a, b FROM foo
                        WHERE b < 5)
             SELECT w1.a, w2.b
             FROM w AS w1, w AS w2
             WHERE w1.a = w2.a AND w1.a > 2)
    SELECT v1.a, v2.a, v2.b
    FROM v as v1, v as v2
    WHERE v1.a < v2.a;
  ```

DML Operation Enhancements with Pivotal Query Optimizer

Pivotal Query Optimizer contains enhancements for DML operations such as `INSERT`, `UPDATE`, and `DELETE`.

- A DML node in a query plan is a query plan operator.
  - Can appear anywhere in the plan, as a regular node (top slice only for now)
  - Can have consumers

- UPDATE operations use the query plan operator Split and supports these operations:
    - UPDATE operations on the table distribution key columns.
    - UPDATE operations on the table on the partition key column.

  This example plan shows the Split operator.

  ```
  QUERY PLAN
  --------------------------------------------------------------
  Update  (cost=0.00..5.46 rows=1 width=1)
     ->  Redistribute Motion 2:2  (slice1; segments: 2)
           Hash Key: a
           ->  Result  (cost=0.00..3.23 rows=1 width=48)
                 ->  Split  (cost=0.00..2.13 rows=1 width=40)
                       ->  Result  (cost=0.00..1.05 rows=1 width=40)
                             ->  Table Scan on dmltest
  ```
- New query plan operator Assert is used for constraints checking.

  This example plan shows the Assert operator.

  ```
  QUERY PLAN
  ------------------------------------------------------------
   Insert  (cost=0.00..4.61 rows=3 width=8)
     ->  Assert  (cost=0.00..3.37 rows=3 width=24)
           Assert Cond: (dmlsource.a > 2) IS DISTINCT FROM
  false
           ->  Assert  (cost=0.00..2.25 rows=3 width=24)
                 Assert Cond: NOT dmlsource.b IS NULL
                 ->  Result  (cost=0.00..1.14 rows=3 width=24)
                       ->  Table Scan on dmlsource
  ```

Changed Behavior with the Pivotal Query Optimizer

There are changes to Greenplum Database behavior when Pivotal Query Optimizer is enabled.

- UPDATE operations on distribution keys are allowed.
- UPDATE operations on partitioned keys are allowed.
- Queries against uniform partitioned tables are supported.
- Queries against partitioned tables that are altered to use an external table as a leaf child partition fall back to the legacy query optimizer.
- Except for INSERT, DML operations directly on partition (child table) of a partitioned table are not supported (as of Greenplum Database 4.3.0.0).

  For the INSERT command, you can specify a leaf child table of the partitioned table to insert data into a partitioned table. An error is returned if the data is not valid for the specified leaf child table. Specifying a child table that is not a leaf child table is not supported.
- The command CREATE TABLE AS distributes table data randomly if the DISTRIBUTED BY clause is not specified and no primary or unique keys are specified.
- Non-deterministic updates not allowed. The following UPDATE command returns an error.

  ```
  update r set b =  r.b  + 1 from s where  r.a  in (select a from s);
  ```
- Statistics are required on the root table of a partitioned table. The ANALYZE command generates statistics on both root and individual partition tables (leaf child tables). See the ROOTPARTITION clause for ANALYZE command.
- Additional Result nodes in the query plan:
    - Query plan Assert operator.
    - Query plan Partition selector operator.
    - Query plan Split operator.

- When running EXPLAIN, the query plan generated by Pivotal Query Optimizer is different than the plan generated by the legacy query optimizer.
- Greenplum Database adds the log file message `Planner produced plan` when the Pivotal Query Optimizer is enabled and Greenplum Database falls back to the legacy query optimizer to generate the query plan.
- Greenplum Database issues a warning when statistics are missing from one or more table columns. When executing an SQL command with the Pivotal Query Optimizer, Greenplum Database issues a warning if the command performance could be improved by collecting statistics on a column or set of columns referenced by the command. The warning is issued on the command line and information is added to the Greenplum Database log file. For information about collecting statistics on table columns, see the ANALYZE command in the *Greenplum Database Reference Guide*.

Pivotal Query Optimizer Limitations

There are limitations in Greenplum Database 4.3.5.0 and later when the Pivotal Query Optimizer is enabled. The Pivotal Query Optimizer and the legacy query optimizer currently coexist in Greenplum Database 4.3.5.0 and later because the Pivotal Query Optimizer does not support all Greenplum Database features.

This section describes the limitations.

- *Unsupported SQL Query Features*
- *Performance Regressions*
- *Greenplum Command Center Database Limitation*

Unsupported SQL Query Features

These are unsupported features when the Pivotal Query Optimizer is enabled:

- Indexed expressions (an index defined as expression based on one or more columns of the table)
- PERCENTILE window function
- External parameters
- These types of partitioned tables:
  - Non-uniform partitioned tables.
  - Partitioned tables that have been altered to use an external table as a leaf child partition.
- SortMergeJoin (SMJ)
- Ordered aggregations
- These analytics extensions:
  - CUBE
  - Multiple grouping sets
- These scalar operators:
  - ROW
  - ROWCOMPARE
  - FIELDSELECT
- Multiple DISTINCT qualified aggregate functions
- Inverse distribution functions

Performance Regressions

When the Pivotal Query Optimizer is enabled in Greenplum Database, the following features are known performance regressions:

- Short running queries - For the Pivotal Query Optimizer, short running queries might encounter additional overhead due to the Pivotal Query Optimizer enhancements for determining an optimal query execution plan.
- `ANALYZE` - For Pivotal Query Optimizer, the `ANALYZE` command generates root partition statistics for partitioned tables. For the legacy optimizer, these statistics are not generated.
- DML operations - For Pivotal Query Optimizer, DML enhancements including the support of updates on partition and distribution keys might require additional overhead.

Also, enhanced functionality of the features from previous versions could result in additional time required when Pivotal Query Optimizer executes SQL statements with the features.

Greenplum Command Center Database Limitation

For Greenplum Command Center monitoring performance, Pivotal recommends the default setting for Pivotal Query Optimizer (`off`) for the *gpperfmon* database that is used by Greenplum Command Center. Enabling Pivotal Query Optimizer for the *gpperfmon* database is not supported. To ensure that the Pivotal Query Optimizer is disabled for the *gpperfmon* database, run this command on the system where the database is installed:

```
ALTER DATABASE gpperfmon SET OPTIMIZER = OFF
```

Determining the Query Optimizer that is Used

When the Pivotal Query Optimizer is enabled, you can determine if Greenplum Database is using the Pivotal Query Optimizer or is falling back to the legacy query optimizer.

You can examine the `EXPLAIN` query plan for the query determine which query optimizer was used by Greenplum Database to execute the query:

- When the Pivotal Query Optimizer generates the query plan, the setting `optimizer=on` and the Pivotal Query Optimizer version are displayed at the end of the query plan. For example.

  ```
  Settings:  optimizer=on
  Optimizer status: PQO version 1.584
  ```

  When Greenplum Database falls back to the legacy optimizer to generate the plan, the setting `optimizer=on` and `legacy query optimizer` are displayed at the end of the query plan. For example.

  ```
  Settings:  optimizer=on
  Optimizer status: legacy query optimizer
  ```

  When the server configuration parameter `OPTIMIZER` is `off`, these lines are displayed at the end of a query plan.

  ```
  Settings:  optimizer=off
  Optimizer status: legacy query optimizer
  ```
- These plan items appear only in the `EXPLAIN` plan output generated by the Pivotal Query Optimizer. The items are not supported in a legacy optimizer query plan.
  - Assert operator
  - Sequence operator
  - DynamicIndexScan
  - DynamicTableScan
  - Table Scan
- When a query against a partitioned table is generated by Pivotal Query Optimizer, the `EXPLAIN` plan displays only the number of partitions that are being eliminated is listed. The scanned partitions are not shown. The `EXPLAIN` plan generated by the legacy optimizer lists the scanned partitions.

The log file contains messages that indicate which query optimizer was used. If Greenplum Database falls back to the legacy optimizer, a message with `NOTICE` information is added to the log file that indicates the unsupported feature. Also, the label `Planner produced plan:` appears before the query in the query execution log message when Greenplum Database falls back to the legacy optimizer.

> **Note:** You can configure Greenplum Database to display log messages on the psql command line by setting the Greenplum Database server configuration parameter `client_min_messages` to `LOG`. See the *Greenplum Database Reference Guide* for information about the parameter.

Examples

This example shows the differences for a query that is run against partitioned tables when the Pivotal Query Optimizer is enabled.

This `CREATE TABLE` statement creates a table with single level partitions:

```
CREATE TABLE sales (trans_id int, date date,
    amount decimal(9,2), region text)
   DISTRIBUTED BY (trans_id)
   PARTITION BY RANGE (date)
      (START (date '20110101')
       INCLUSIVE END (date '20120101')
       EXCLUSIVE EVERY (INTERVAL '1 month'),
   DEFAULT PARTITION outlying_dates );
```

This query against the table is supported by the Pivotal Query Optimizer and does not generate errors in the log file:

```
select * from sales ;
```

The `EXPLAIN` plan output lists only the number of selected partitions.

```
 ->  Partition Selector for sales (dynamic scan id: 1)  (cost=10.00..100.00 rows=50
 width=4)
       Partitions selected:  13 (out of 13)
```

If a query against a partitioned table is not supported by the Pivotal Query Optimizer. Greenplum Database falls back to the legacy optimizer. The `EXPLAIN` plan generated by the legacy optimizer lists the selected partitions. This example shows a part of the explain plan that lists some selected partitions.

```
 ->  Append  (cost=0.00..0.00 rows=26 width=53)
     ->  Seq Scan on sales2_1_prt_7_2_prt_usa sales2  (cost=0.00..0.00 rows=1
 width=53)
     ->  Seq Scan on sales2_1_prt_7_2_prt_asia sales2  (cost=0.00..0.00 rows=1
 width=53)
     ...
```

This example shows the log output when the Greenplum Database falls back to the legacy query optimizer from the Pivotal Query Optimizer.

When this query is run, Greenplum Database falls back to the legacy query optimizer.

```
explain select * from pg_class;
```

A message is added to the log file. The message contains this `NOTICE` information that indicates the reason the Pivotal Query Optimizer did not execute the query:

```
NOTICE,""Feature not supported by the Pivotal Query Optimizer: Queries on master-only
 tables"
```

About Uniform Multi-level Partitioned Tables

Pivotal Query Optimizer supports queries on a multi-level partitioned (MLP) table if the MLP table is a *uniform partitioned table*. A multi-level partitioned table is a partitioned table that was created with the `SUBPARTITION` clause. A uniform partitioned table must meet these requirements.

- The partitioned table structure is uniform. Each partition node at the same level must have the same hierarchical structure.
- The partition key constraints must be consistent and uniform. At each subpartition level, the sets of constraints on the child tables created for each branch must match.

You can display information about partitioned tables in several ways, including displaying information from these sources:

- The `pg_partitions` system view contains information on the structure of a partitioned table.
- The `pg_constraint` system catalog table contains information on table constraints.
- The psql meta command \d+ *tablename* displays the table constraints for child leaf tables of a partitioned table.

Example

This `CREATE TABLE` command creates a uniform partitioned table.

```
CREATE TABLE mlp (id  int,  year int,  month int,  day int,
   region  text)
   DISTRIBUTED  BY (id)
    PARTITION BY RANGE ( year)
      SUBPARTITION  BY LIST (region)
        SUBPARTITION TEMPLATE (
          SUBPARTITION usa  VALUES ( 'usa'),
          SUBPARTITION europe  VALUES ( 'europe'),
          SUBPARTITION asia  VALUES ( 'asia'))
   (  START ( 2000)  END ( 2010) EVERY ( 5));
```

These are child tables and the partition hierarchy that are created for the table `mlp`. This hierarchy consists of one subpartition level that contains two branches.

```
mlp_1_prt_11
   mlp_1_prt_11_2_prt_usa
   mlp_1_prt_11_2_prt_europe
   mlp_1_prt_11_2_prt_asia

mlp_1_prt_21
   mlp_1_prt_21_2_prt_usa
   mlp_1_prt_21_2_prt_europe
   mlp_1_prt_21_2_prt_asia
```

The hierarchy of the table is uniform, each partition contains a set of three child tables (subpartitions). The constraints for the region subpartitions are uniform, the set of constraints on the child tables for the branch table `mlp_1_prt_11` are the same as the constraints on the child tables for the branch table `mlp_1_prt_21`.

As a quick check, this query displays the constraints for the partitions.

```
WITH tbl AS (SELECT oid, partitionlevel AS level,
             partitiontablename AS part
         FROM pg_partitions, pg_class
         WHERE tablename = 'mlp' AND partitiontablename=relname
            AND partitionlevel=1 )
  SELECT tbl.part, consrc
    FROM tbl, pg_constraint
    WHERE tbl.oid = conrelid ORDER BY consrc;
```

> **Note:** You will need modify the query for more complex partitioned tables. For example, the query does not account for table names in different schemas.

The `consrc` column displays constraints on the subpartitions. The set of region constraints for the subpartitions in `mlp_1_prt_1` match the constraints for the subpartitions in `mlp_1_prt_2`. The constraints for year are inherited from the parent branch tables.

```
           part           |               consrc
--------------------------+------------------------------------
 mlp_1_prt_2_2_prt_asia   | (region = 'asia'::text)
 mlp_1_prt_1_2_prt_asia   | (region = 'asia'::text)
 mlp_1_prt_2_2_prt_europe | (region = 'europe'::text)
 mlp_1_prt_1_2_prt_europe | (region = 'europe'::text)
 mlp_1_prt_1_2_prt_usa    | (region = 'usa'::text)
 mlp_1_prt_2_2_prt_usa    | (region = 'usa'::text)
 mlp_1_prt_1_2_prt_asia   | ((year >= 2000) AND (year < 2005))
 mlp_1_prt_1_2_prt_usa    | ((year >= 2000) AND (year < 2005))
 mlp_1_prt_1_2_prt_europe | ((year >= 2000) AND (year < 2005))
 mlp_1_prt_2_2_prt_usa    | ((year >= 2005) AND (year < 2010))
 mlp_1_prt_2_2_prt_asia   | ((year >= 2005) AND (year < 2010))
 mlp_1_prt_2_2_prt_europe | ((year >= 2005) AND (year < 2010))
(12 rows)
```

If you add a default partition to the example partitioned table with this command:

```
ALTER TABLE mlp ADD DEFAULT PARTITION def
```

The partitioned table remains a uniform partitioned table. The branch created for default partition contains three child tables and the set of constraints on the child tables match the existing sets of child table constraints.

In the above example, if you drop the subpartition `mlp_1_prt_21_2_prt_asia` and add another subpartition for the region `canada`, the constraints are no longer uniform.

```
ALTER TABLE mlp ALTER PARTITION FOR (RANK(2))
  DROP PARTITION asia ;

ALTER TABLE mlp ALTER PARTITION FOR (RANK(2))
  ADD PARTITION canada VALUES ('canada');
```

Also, if you add a partition `canada` under `mlp_1_prt_21`, the partitioning hierarchy is not uniform.

However, if you add the subpartition `canada` to both `mlp_1_prt_21` and `mlp_1_prt_11` the of the original partitioned table, it remains a uniform partitioned table.

> **Note:** Only the constraints on the sets of partitions at a partition level must be the same. The names of the partitions can be different.

Defining Queries

Greenplum Database is based on the PostgreSQL implementation of the SQL standard.

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

- *SQL Lexicon*
- *SQL Value Expressions*

SQL Lexicon

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 PostgreSQL's structure and syntax, with some 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. For more information about these constructs, see *Using Functions and Operators*.

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*.

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)
```

See the *Greenplum Database Reference Guide* for lists of the built-in functions by category. 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* [, ...]) — operates across all input rows for which the expected result value is non-null. `ALL` is the default.
- *aggregate_name*(ALL *expression* [, ...]) — operates identically to the first form because `ALL` is the default.
- *aggregate_name*(DISTINCT *expression* [, ...]) — operates across all distinct non-null values of input rows.
- *aggregate_name*(*) — 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`.

For predefined aggregate functions, see *Built-in Functions and Operators*. 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 54: Advanced Aggregate Functions* for more details.
- 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* and *Table 51: Built-in functions and operators*.
- 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 *Table 52: Window functions*, *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*.

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 the `TABLE` function with the `SCATTER BY` clause in the the GPText function `gptext.index()` to populate the index `mytest.articles` with data from the messages table:

```
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

Greenplum Database evaluates functions and operators used in SQL expressions. Some functions and operators are only allowed to execute on the master since they could lead to inconsistencies in segment databases.

- *Using Functions in Greenplum Database*
- *User-Defined Functions*
- *Built-in Functions and Operators*
- *Window Functions*
- *Advanced Analytic Functions*

Using Functions in Greenplum Database

Table 50: 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` statement to register user-defined functions that are used as described in *Using Functions in Greenplum Database*. 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*. See the *Functions and Operators* section of the PostgreSQL documentation for more information about these built-in functions and operators.

Table 51: 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	

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
Date/Time Functions and Operators	timeofday	age current_date current_time current_timestamp localtime localtimestamp now	
Geometric Functions and Operators			
Network Address Functions and Operators			
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*	

Operator/Function Category	VOLATILE Functions	STABLE Functions	Restrictions
System Administration Functions	set_config pg_cancel_backend pg_reload_conf pg_rotate_logfile pg_start_backup pg_stop_backup pg_size_pretty pg_ls_dir pg_read_file pg_stat_file	current_setting *All database object size functions*	**Note:** The function `pg_column_size` displays bytes required to store the value, perhaps with TOAST compression.

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*.

Table 52: Window functions

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

Function	Return Type	Full Syntax	Description
`lead(`*expr* `[,`*offset*`]` `[,`*default*`])`	same as input *expr* type	`LEAD(`*expr* `[,`*offset*`]` `[,`*exprdefault*`]) OVER ( [PARTITION BY` *expr*`] ORDER BY` *expr* `)`	Provides access to more than one row of the same table without doing a self join. Given a series of rows returned from a query and a position of the cursor, `lead` provides access to a row at a given physical offset after that position. If *offset* is not specified, the default offset is 1. *default* sets the value that is returned if the offset goes beyond the scope of the window. If *default* is not specified, the default value is null.
`ntile(`*expr*`)`	bigint	`NTILE(`*expr*`) OVER ( [PARTITION BY` *expr*`] ORDER BY` *expr* `)`	Divides an ordered data set into a number of buckets (as defined by *expr*) and assigns a bucket number to each row.
`percent_rank()`	`double precision`	`PERCENT_RANK () OVER ( [PARTITION BY` *expr*`] ORDER BY` *expr* `)`	Calculates the rank of a hypothetical row `R` minus 1, divided by 1 less than the number of rows being evaluated (within a window partition).
`rank()`	bigint	`RANK () OVER ( [PARTITION BY` *expr*`] ORDER BY` *expr* `)`	Calculates the rank of a row in an ordered group of values. Rows with equal values for the ranking criteria receive the same rank. The number of tied rows are added to the rank number to calculate the next rank value. Ranks may not be consecutive numbers in this case.
`row_number()`	`bigint`	`ROW_NUMBER () OVER ( [PARTITION BY` *expr*`] ORDER BY` *expr* `)`	Assigns a unique number to each row to which it is applied (either each row in a window partition or each row of the query).

Advanced Analytic Functions

The following built-in advanced analytic functions are Greenplum extensions of the PostgreSQL database. Analytic functions are *immutable*.

Table 53: Advanced Analytic Functions

Function	Return Type	Full Syntax	Description
`matrix_add(array[], array[])`	`smallint[], int[], bigint[], float[]`	`matrix_add( array[[1,1],[2,2]], array[[3,4],[5,6]])`	Adds two two-dimensional matrices. The matrices must be conformable.
`matrix_multiply( array[], array[])`	`smallint[]int[], bigint[], float[]`	`matrix_multiply( array[[2,0,0],[0,2,0],[0,0,2]], array[[3,0,3],[0,3,0],[0,0,3]] )`	Multiplies two, three-dimensional arrays. The matrices must be conformable.
`matrix_multiply( array[],` *`expr`*`)`	`int[], float[]`	`matrix_multiply( array[[1,1,1], [2,2,2], [3,3,3]], 2)`	Multiplies a two-dimensional array and a scalar numeric value.
`matrix_transpose( array[])`	Same as input `array` type.	`matrix_transpose( array [[1,1,1],[2,2,2]])`	Transposes a two-dimensional array.
`pinv(array [])`	`smallint[]int[], bigint[], float[]`	`pinv(array[[2.5,0,0],[0,1,0],[0,0,.5]])`	Calculates the Moore-Penrose pseudoinverse of a matrix.
`unnest (array[])`	set of `anyelement`	`unnest( array['one', 'row', 'per', 'item'])`	Transforms a one dimensional array into rows. Returns a set of `anyelement`, a polymorphic *pseudotype in PostgreSQL*.

Table 54: Advanced Aggregate Functions

Function	Return Type	Full Syntax	Description

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

Function	Return Type	Full Syntax	Description

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

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

sum(array[])	smallint[]int[], bigint[], float[]	sum(array[[1,2],[3,4]])
		Example:
		CREATE TABLE mymatrix (myvalue
		 int[]);
		INSERT INTO mymatrix VALUES
		 (array[[1,2],[3,4]]);
		INSERT INTO mymatrix VALUES
		 (array[[0,1],[1,0]]);
		SELECT sum(myvalue) FROM
		 mymatrix;
		 sum
		--------------- 	
		 {{1,3},{4,4}}
			Performs matrix summation. Can take as input a two-dimensional array that is treated as a matrix.

pivot_sum (label[], label, expr)	int[], bigint[], float[]	pivot_sum(array['A1','A2'], attr, value)	A pivot aggregation using sum to resolve duplicate entries.

Function	Return Type	Full Syntax	Description
mregr_coef(expr, array[])	float[]	mregr_coef(y, array[1, x1, x2])	The four mregr_* aggregates perform linear regressions using the ordinary-least-squares method. mregr_coef calculates the regression coefficients. The size of the return array for mregr_coef is the same as the size of the input array of independent variables, since the return array contains the coefficient for each independent variable.
mregr_r2 (*expr*, array[])	float	mregr_r2(y, array[1, x1, x2])	The four mregr_* aggregates perform linear regressions using the ordinary-least-squares method. mregr_r2 calculates the r-squared error value for the regression.
mregr_pvalues(*expr*, array[])	float[]	mregr_pvalues(y, array[1, x1, x2])	The four mregr_* aggregates perform linear regressions using the ordinary-least-squares method. mregr_pvalues calculates the p-values for the regression.
mregr_tstats(*expr*, array[])	float[]	mregr_tstats(y, array[1, x1, x2])	The four mregr_* aggregates perform linear regressions using the ordinary-least-squares method. mregr_tstats calculates the t-statistics for the regression.
nb_classify(text[], bigint, bigint[], bigint[])	text	nb_classify(classes, attr_count, class_count, class_total)	Classify rows using a Naive Bayes Classifier. This aggregate uses a baseline of training data to predict the classification of new rows and returns the class with the largest likelihood of appearing in the new rows.

Function	Return Type	Full Syntax	Description
`nb_probabilities(text[], bigint, bigint[], bigint[])`	text	`nb_probabilities(classes, attr_count, class_count, class_total)`	Determine probability for each class using a Naive Bayes Classifier. This aggregate uses a baseline of training data to predict the classification of new rows and returns the probabilities that each class will appear in new rows.

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:

5.
   ```
    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 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.

> **Note:** Greenplum Database supports the Pivotal Query Optimizer. The Pivotal Query Optimizer extends the planning and optimization capabilities of the Greenplum Database legacy optimizer. For information about the features and limitations of the Pivotal Query Optimizer, see *Overview of the Pivotal Query Optimizer*.

Managing Spill Files Generated by Queries

Greenplum Database creates spill files, also known as workfiles, on disk if it does not have sufficient memory to execute an SQL query in memory. The default value of 100,000 spill files is sufficient for the majority of queries. However, if a query creates more than the specified number of spill files, Greenplum Database returns this error:

```
ERROR: number of workfiles per query limit exceeded
```

Reasons that cause a large number of spill files to be generated include:

- Data skew is present in the queried data.
- The amount memory allocated for the query is too low.

You might be able to run the query successfully by changing the query, changing the data distribution, or changing the system memory configuration. You can use the *gp_workfile_** views to see spill file usage information. You can control the maximum amount of memory that can used by a query with the Greenplum Database server configuration parameters `max_statement_mem`, `statement_mem`, or through resource queues.

Monitoring a Greenplum System contains the following information:

- Information about skew and how to check for data skew
- Information about using the *gp_workfile_* views*

For information about server configuration parameters, see the *Greenplum Database Reference Guide*. For information about resource queues, see *Workload Management with Resource Queues*.

If you have determined that the query must create more spill files than allowed by the value of server configuration parameter `gp_workfile_limit_files_per_query`, you can increase the value of the parameter.

Query Profiling

Examine the query plans of poorly performing queries to identify possible performance tuning opportunities.

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.

The query optimizer 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 optimizer'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 optimizer's estimates are to reality. For example:

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

> **Note:** In Greenplum Database 4.3.5.0 and later, the Pivotal Query Optimizer co-exists with the legacy query optimizer. The `EXPLAIN` output generated by the Pivotal Query Optimizer is different than the output generated by the legacy query optimizer.
>
> By default, Greenplum Database uses the legacy query optimizer. To enable the Pivotal Query Optimizer, set the Greenplum Database server configuration parameter `optimizer` to `on`. When the Pivotal Query Optimizer is enabled, Greenplum Database uses the Pivotal Query Optimizer to generate an execution plan for a query when possible.
>
> When the `EXPLAIN ANALYZE` command uses the Pivotal Query Optimizer, the `EXPLAIN` plan shows only the number of partitions that are being eliminated. The scanned partitions are not shown. To show name of the scanned partitions in the segment logs set the server configuration parameter `gp_log_dynamic_partition_pruning` to `on`. This example `SET` command enables the parameter.
>
> ```
> SET gp_log_dynamic_partition_pruning = on;
> ```
>
> For information about the Pivotal Query Optimizer, see *Querying Data*.

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 optimizer intends to minimize.
- The cost reflects only the aspects of plan execution that the query optimizer 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 optimizer 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 optimizer 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 optimizer's estimates. This allows you to see if the optimizer'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 Examples

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 optimizer's estimate: the query optimizer 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.

Determining the Query Optimizer

You can view `EXPLAIN` output to determine if the Pivotal Query Optimizer is enabled for the query plan and whether the Pivotal Query Optimizer or the legacy query optimizer generated the explain plan. The information appears at the end of the `EXPLAIN` output. The `Settings` line displays the setting of the server configuration parameter `OPTIMIZER`. The `Optimizer status` line displays whether the Pivotal Query Optimizer or the legacy query optimizer generated the explain plan.

For these two example query plans, Pivotal Query Optimizer is enabled, the server configuration parameter `OPTIMIZER` is `on`. For the first plan, the Pivotal Query Optimizer generated the `EXPLAIN` plan. For the second plan, Greenplum Database fell back to the legacy query optimizer to generate the query plan.

```
                       QUERY PLAN
------------------------------------------------------------------------------------
 Aggregate  (cost=0.00..296.14 rows=1 width=8)
   ->  Gather Motion 2:1  (slice1; segments: 2)  (cost=0.00..295.10 rows=1 width=8)
         ->  Aggregate  (cost=0.00..294.10 rows=1 width=8)
               ->  Table Scan on part  (cost=0.00..97.69 rows=100040 width=1)
 Settings:  optimizer=on
 Optimizer status: PQO version 1.584
(5 rows)
```

```
explain select count(*) from part;
```

```
                       QUERY PLAN
------------------------------------------------------------------------------------
 Aggregate  (cost=3519.05..3519.06 rows=1 width=8)
   ->  Gather Motion 2:1  (slice1; segments: 2)  (cost=3518.99..3519.03 rows=1
 width=8)
         ->  Aggregate  (cost=3518.99..3519.00 rows=1 width=8)
               ->  Seq Scan on part  (cost=0.00..3018.79 rows=100040 width=1)
 Settings:  optimizer=on
 Optimizer status: legacy query optimizer
(5 rows)
```

For this query, the server configuration parameter `OPTIMIZER` is `off`.

```
explain select count(*) from part;

                       QUERY PLAN
------------------------------------------------------------------------------------
 Aggregate  (cost=3519.05..3519.06 rows=1 width=8)
   ->  Gather Motion 2:1  (slice1; segments: 2)  (cost=3518.99..3519.03 rows=1
 width=8)
         ->  Aggregate  (cost=3518.99..3519.00 rows=1 width=8)
               ->  Seq Scan on part  (cost=0.00..3018.79 rows=100040 width=1)
 Settings: optimizer=off
 Optimizer status: legacy query optimizer
(5 rows)
```

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 legacy query optimizer (planner) to choose a different plan by disabling a particular query plan operator for that query.
- **Are the optimizer's estimates close to reality?** Run `EXPLAIN ANALYZE` and see if the number of rows the optimizer 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 optimizer choose the best join order?** When you have a query that joins multiple tables, make sure that the optimizer 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 legacy query optimizer (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 optimizer selectively scan partitioned tables?** If you use table partitioning, is the optimizer 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* for an example of a query plan that shows a selective partition scan.

- **Does the optimizer 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 optimizer 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.

Part V

Managing Performance

The topics in this section cover Greenplum Database performance management, including how to monitor performance and how to configure workloads to prioritize resource utilization.

This section contains the following topics:

- *Defining Database Performance*
- *Common Causes of Performance Issues*
- *Workload Management with Resource Queues*
- *Investigating a Performance Problem*

Chapter 24

Defining Database Performance

Managing system performance includes measuring performance, identifying the causes of performance problems, and applying the tools and techniques available to you to remedy the problems.

Greenplum measures database performance based on the rate at which the database management system (DBMS) supplies information to requesters.

Understanding the Performance Factors

Several key performance factors influence database performance. Understanding these factors helps identify performance opportunities and avoid problems:

- *System Resources*
- *Workload*
- *Throughput*
- *Contention*
- *Optimization*

System Resources

Database performance relies heavily on disk I/O and memory usage. To accurately set performance expectations, you need to know the baseline performance of the hardware on which your DBMS is deployed. Performance of hardware components such as CPUs, hard disks, disk controllers, RAM, and network interfaces will significantly affect how fast your database performs.

Workload

The workload equals the total demand from the DBMS, and it varies over time. The total workload is a combination of user queries, applications, batch jobs, transactions, and system commands directed through the DBMS at any given time. For example, it can increase when month-end reports are run or decrease on weekends when most users are out of the office. Workload strongly influences database performance. Knowing your workload and peak demand times helps you plan for the most efficient use of your system resources and enables processing the largest possible workload.

Throughput

A system's throughput defines its overall capability to process data. DBMS throughput is measured in queries per second, transactions per second, or average response times. DBMS throughput is closely related to the processing capacity of the underlying systems (disk I/O, CPU speed, memory bandwidth, and so on), so it is important to know the throughput capacity of your hardware when setting DBMS throughput goals.

Contention

Contention is the condition in which two or more components of the workload attempt to use the system in a conflicting way — for example, multiple queries that try to update the same piece of data at the same time or multiple large workloads that compete for system resources. As contention increases, throughput decreases.

Optimization

DBMS optimizations can affect the overall system performance. SQL formulation, database configuration parameters, table design, data distribution, and so on enable the database query optimizer to create the most efficient access plans.

Determining Acceptable Performance

When approaching a performance tuning initiative, you should know your system's expected level of performance and define measurable performance requirements so you can accurately evaluate your system's performance. Consider the following when setting performance goals:

- *Baseline Hardware Performance*
- *Performance Benchmarks*

Baseline Hardware Performance

Most database performance problems are caused not by the database, but by the underlying systems on which the database runs. I/O bottlenecks, memory problems, and network issues can notably degrade database performance. Knowing the baseline capabilities of your hardware and operating system (OS) will help you identify and troubleshoot hardware-related problems before you explore database-level or query-level tuning initiatives.

See the *Greenplum Database Reference Guide* for information about running the `gpcheckperf` utility to validate hardware and network performance.

Performance Benchmarks

To maintain good performance or fix performance issues, you should know the capabilities of your DBMS on a defined workload. A benchmark is a predefined workload that produces a known result set. Periodically run the same benchmark tests to help identify system-related performance degradation over time. Use benchmarks to compare workloads and identify queries or applications that need optimization.

Many third-party organizations, such as the Transaction Processing Performance Council (TPC), provide benchmark tools for the database industry. TPC provides TPC-H, a decision support system that examines large volumes of data, executes queries with a high degree of complexity, and gives answers to critical business questions. For more information about TPC-H, go to:

http://www.tpc.org/tpch

Chapter 25

Common Causes of Performance Issues

This section explains the troubleshooting processes for common performance issues and potential solutions to these issues.

Identifying Hardware and Segment Failures

The performance of Greenplum Database depends on the hardware and IT infrastructure on which it runs. Greenplum Database is comprised of several servers (hosts) acting together as one cohesive system (array); as a first step in diagnosing performance problems, ensure that all Greenplum Database segments are online. Greenplum Database's performance will be as fast as the slowest host in the array. Problems with CPU utilization, memory management, I/O processing, or network load affect performance. Common hardware-related issues are:

- **Disk Failure** – Although a single disk failure should not dramatically affect database performance if you are using RAID, disk resynchronization does consume resources on the host with failed disks. The `gpcheckperf` utility can help identify segment hosts that have disk I/O issues.
- **Host Failure** – When a host is offline, the segments on that host are nonoperational. This means other hosts in the array must perform twice their usual workload because they are running the primary segments and multiple mirrors. If mirrors are not enabled, service is interrupted. Service is temporarily interrupted to recover failed segments. The `gpstate` utility helps identify failed segments.
- **Network Failure** – Failure of a network interface card, a switch, or DNS server can bring down segments. If host names or IP addresses cannot be resolved within your Greenplum array, these manifest themselves as interconnect errors in Greenplum Database. The `gpcheckperf` utility helps identify segment hosts that have network issues.
- **Disk Capacity** – Disk capacity on your segment hosts should never exceed 70 percent full. Greenplum Database needs some free space for runtime processing. To reclaim disk space that deleted rows occupy, run `VACUUM` after loads or updates. The *gp_toolkit* administrative schema has many views for checking the size of distributed database objects.

  See the *Greenplum Database Reference Guide* for information about checking database object sizes and disk space.

Managing Workload

A database system has a limited CPU capacity, memory, and disk I/O resources. When multiple workloads compete for access to these resources, database performance suffers. Workload management maximizes system throughput while meeting varied business requirements. With role-based resource queues, Greenplum Database workload management limits active queries and conserves system resources.

A resource queue limits the size and/or total number of queries that users or roles can execute in the particular queue. By assigning database roles to the appropriate resource queues, administrators can control concurrent user queries and prevent system overload. See *Workload Management with Resource Queues* for more information about setting up resource queues.

Greenplum Database administrators should run maintenance workloads such as data loads and `VACUUM ANALYZE` operations after business hours. Do not compete with database users for system resources; perform administrative tasks at low-usage times.

Avoiding Contention

Contention arises when multiple users or workloads try to use the system in a conflicting way; for example, contention occurs when two transactions try to update a table simultaneously. A transaction that seeks a table-level or row-level lock will wait indefinitely for conflicting locks to be released. Applications should not hold transactions open for long periods of time, for example, while waiting for user input.

Maintaining Database Statistics

Greenplum Database uses a cost-based query optimizer that relies on database statistics. Accurate statistics allow the query optimizer to better estimate the number of rows retrieved by a query to choose the most efficient query plan. Without database statistics, the query optimizer cannot estimate how many records will be returned. The optimizer does not assume it has sufficient memory to perform certain operations such as aggregations, so it takes the most conservative action and does these operations by reading and writing from disk. This is significantly slower than doing them in memory. `ANALYZE` collects statistics about the database that the query optimizer needs.

> **Note:** When executing an SQL command with the Pivotal Query Optimizer, Greenplum Database issues a warning if the command performance could be improved by collecting statistics on a column or set of columns referenced by the command. The warning is issued on the command line and information is added to the Greenplum Database log file. For information about collecting statistics on table columns, see the `ANALYZE` command in the *Greenplum Database Reference Guide*

Identifying Statistics Problems in Query Plans

Before you interpret a query plan for a query using EXPLAIN or `EXPLAIN ANALYZE`, familiarize yourself with the data to help identify possible statistics problems. Check the plan for the following indicators of inaccurate statistics:

- **Are the optimizer's estimates close to reality?** Run `EXPLAIN ANALYZE` and see if the number of rows the optimizer estimated is close to the number of rows the query operation returned .
- **Are selective predicates applied early in the plan?** The most selective filters should be applied early in the plan so fewer rows move up the plan tree.
- **Is the optimizer choosing the best join order?** When you have a query that joins multiple tables, make sure the optimizer 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.

See *Query Profiling* for more information about reading query plans.

Tuning Statistics Collection

The following configuration parameters control the amount of data sampled for statistics collection:

- `default_statistics_target`
- `gp_analyze_relative_error`

These parameters control statistics sampling at the system level. It is better to sample only increased statistics for columns used most frequently in query predicates. You can adjust statistics for a particular column using the command:

```
ALTER TABLE...SET STATISTICS
```

For example:

```
ALTER TABLE sales ALTER COLUMN region SET STATISTICS 50;
```

This is equivalent to increasing `default_statistics_target` for a particular column. Subsequent `ANALYZE` operations will then gather more statistics data for that column and produce better query plans as a result.

Optimizing Data Distribution

When you create a table in Greenplum Database, you must declare a distribution key that allows for even data distribution across all segments in the system. Because the segments work on a query in parallel, Greenplum Database will always be as fast as the slowest segment. If the data is unbalanced, the segments that have more data will return their results slower and therefore slow down the entire system.

Optimizing Your Database Design

Many performance issues can be improved by database design. Examine your database design and consider the following:

- Does the schema reflect the way the data is accessed?
- Can larger tables be broken down into partitions?
- Are you using the smallest data type possible to store column values?
- Are columns used to join tables of the same datatype?
- Are your indexes being used?

Greenplum Database Maximum Limits

To help optimize database design, review the maximum limits that Greenplum Database supports:

Table 55: Maximum Limits of Greenplum Database

Dimension	Limit
Database Size	Unlimited
Table Size	Unlimited, 128 TB per partition per segment
Row Size	1.6 TB (1600 columns * 1 GB)
Field Size	1 GB
Rows per Table	281474976710656 (2^48)
Columns per Table/View	1600
Indexes per Table	Unlimited
Columns per Index	32
Table-level Constraints per Table	Unlimited
Table Name Length	63 Bytes (Limited by *name* data type)

Dimensions listed as unlimited are not intrinsically limited by Greenplum Database. However, they are limited in practice to available disk space and memory/swap space. Performance may suffer when these values are unusually large.

Note:

There is a maximum limit on the number of objects (tables, indexes, and views, but not rows) that may exist at one time. This limit is 4294967296 (2^32).

Chapter 26

Workload Management with Resource Queues

Use Greenplum Database workload management to prioritize and allocate resources to queries according to business requirements and to prevent queries from starting when resources are unavailable.

This section describes Greenplum Database workload management, and explains how to use resource queues to manage resources. Using resource queues, the available memory and CPU resources can be allocated to the different types of queries that execute on your Greenplum Database system. You can limit the number of concurrent queries, the amount of memory used to execute a query, and the relative amount of CPU devoted to processing a query.

The primary resource management concerns are the number of queries that can execute concurrently and the amount of memory to allocate to each query. Without limiting concurrency and memory usage, it is not possible to guarantee acceptable performance. Memory is the resource most likely to limit the processing capacity of the system. Therefore, we begin with an overview of Greenplum Database memory usage.

Pivotal Greenplum Workload Manager contains additional, flexible workload management features that can be used to monitor and manage queries and to manage resource queues. Pivotal Greenplum Workload Manager is included with Pivotal Greenplum Command Center. See *http://gpcc.docs.pivotal.io/* for information about Pivotal Greenplum Command Center and Pivotal Greenplum Workload Manager.

Overview of Memory Usage in Greenplum Database

Memory is a key resource for a Greenplum Database system and, when used efficiently, can ensure high performance and throughput. This topic describes how segment host memory is allocated between segments and the options available to administrators to configure memory.

A Greenplum Database segment host runs multiple PostgreSQL instances, all sharing the host's memory. The segments have an identical configuration and they consume similar amounts of memory, CPU, and disk IO simultaneously, while working on queries in parallel.

For best query throughput, the memory configuration should be managed carefully. There are memory configuration options at every level in Greenplum Database, from operating system parameters, to managing workloads with resource queues, to setting the amount of memory allocated to an individual query.

Segment Host Memory

On a Greenplum Database segment host, the available host memory is shared among all the processes executing on the computer, including the operating system, Greenplum Database segment instances, and other application processes. Administrators must determine what Greenplum Database and non-Greenplum Database processes share the hosts' memory and configure the system to use the memory efficiently. It is equally important to monitor memory usage regularly to detect any changes in the way host memory is consumed by Greenplum Database or other processes.

The following figure illustrates how memory is consumed on a Greenplum Database segment host.

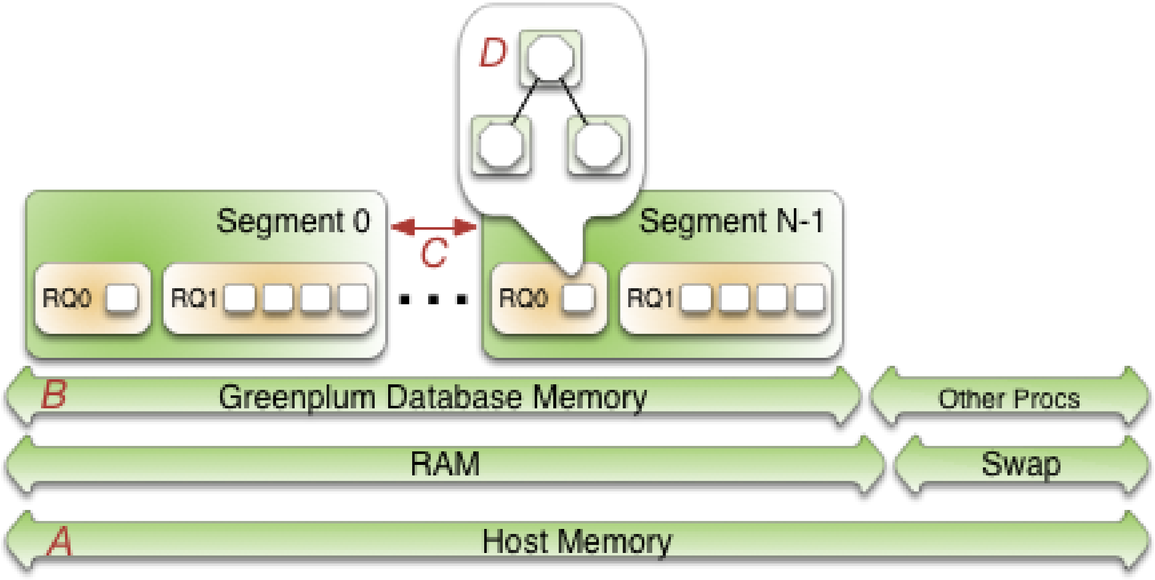


Figure 23: Greenplum Database Segment Host Memory

Beginning at the bottom of the illustration, the line labeled *A* represents the total host memory. The line directly above line *A* shows that the total host memory comprises both physical RAM and swap space.

The line labelled *B* shows that the total memory available must be shared by Greenplum Database and all other processes on the host. Non-Greenplum Database processes include the operating system and any other applications, for example system monitoring agents. Some applications may use a significant portion of memory and, as a result, you may have to adjust the number of segments per Greenplum Database host or the amount of memory per segment.

The segments (*C*) each get an equal share of the Greenplum Database Memory (*B*).

Within a segment, *resource queues* govern how memory is allocated to execute a SQL statement. Resource queues allow you to translate business requirements into execution policies in your Greenplum Database system and to guard against queries that could degrade performance.

Each statement submitted by a non-administrative user to a Greenplum Database system is associated with a resource queue. The queue determines if the statement will be allowed to execute and, when resources are available, allows it to execute. The statement may be rejected, executed immediately, or queued to execute when resources are available.

You can create resource queues for different types of queries and reserve a fixed portion of the segment memory for each queue. Alternatively, you can set a server configuration parameter to specify how much memory to allocate for each query and place no maximum memory restriction on the resource queue.

The query optimizer produces a query execution plan, consisting of a series of tasks called *operators* (labeled *D* in the diagram). Operators perform tasks such as table scans or joins, and typically produce intermediate query results by processing one or more sets of input rows. Operators receive a share of the memory the resource queue allocates to a query. If an operator cannot perform all of its work in the memory allocated to it, it caches data on disk in *spill files*.

Options for Configuring Segment Host Memory

Host memory is the total memory shared by all applications on the segment host. The amount of host memory can be configured using any of the following methods:

- Add more RAM to the nodes to increase the physical memory.
- Allocate swap space to increase the size of virtual memory.

- Set the kernel parameters *vm.overcommit_memory* and *vm.overcommit_ratio* to configure how the operating system handles large memory allocation requests.

The physical RAM and OS configuration are usually managed by the platform team and system administrators. See the *Greenplum Database Installation Guide* for the recommended kernel parameter settings.

The amount of memory to reserve for the operating system and other processes is workload dependent. The minimum recommendation for operating system memory is 32GB, but if there is much concurrency in Greenplum Database, increasing to 64GB of reserved memory may be required. The largest user of operating system memory is SLAB, which increases as Greenplum Database concurrency and the number of sockets used increases.

The `vm.overcommit_memory` kernel parameter should always be set to 2, the only safe value for Greenplum Database.

The `vm.overcommit_ratio` kernel parameter sets the percentage of RAM that is used for application processes, the remainder reserved for the operating system. The default for Red Hat is 50 (50%). Setting this parameter too high may result in insufficient memory reserved for the operating system, which can cause segment host failure or database failure. Leaving the setting at the default of 50 is generally safe, but conservative. Setting the value too low reduces the amount of concurrency and query complexity that can be run at the same time by reducing the amount of memory available to Greenplum Database. When increasing `vm.overcommit_ratio` it is important to remember to always reserve some memory for operating system activities.

To calculate a safe value for `vm.overcommit_ratio`, first determine the total memory available to Greenplum Database processes, called `gp_vmem`, with this formula:

```
gp_vmem = ((SWAP + RAM) – (7.5GB + 0.05 * RAM)) / 1.7
```

where `SWAP` is the swap space on the host in GB, and `RAM` is the number of GB of RAM installed on the host.

Calculate the `vm.overcommit_ratio` value with this formula:

```
vm.overcommit_ratio = (RAM - 0.026 * gp_vmem) / RAM
```

Configuring Greenplum Database Memory

Greenplum Database Memory is the amount of memory available to all Greenplum Database segment instances.

When you set up the Greenplum Database cluster, you determine the number of primary segments to run per host and the amount of memory to allocate for each segment. Depending on the CPU cores, amount of physical RAM, and workload characteristics, the number of segments is usually a value between 4 and 8. With segment mirroring enabled, it is important to allocate memory for the maximum number of primary segments executing on a host during a failure. For example, if you use the default grouping mirror configuration, a segment host failure doubles the number of acting primaries on the host that has the failed host's mirrors. Mirror configurations that spread each host's mirrors over multiple other hosts can lower the maximum, allowing more memory to be allocated for each segment. For example, if you use a block mirroring configuration with 4 hosts per block and 8 primary segments per host, a single host failure would cause other hosts in the block to have a maximum of 11 active primaries, compared to 16 for the default grouping mirror configuration.

The *gp_vmem_protect_limit* value is the amount of memory to allocate each segment. It is estimated by calculating the memory available for all Greenplum Database processes and dividing by the maximum number of primary segments during a failure. If *gp_vmem_protect_limit* is set too high, queries can fail. Use the following formulas to calculate a safe value for *gp_vmem_protect_limit*.

Calculate gp_vmem, the memory available for all Greenplum Database processes, using this formula:

```
gp_vmem = ((SWAP + RAM) – (7.5GB + 0.05 * RAM)) / 1.7
```

where SWAP is the host's swap space in GB and RAM is the RAM installed on the host in GB.

Calculate gp_vmem_protect_limit using this formula:

```
gp_vmem_protect_limit = gp_vmem / max_acting_primary_segments
```

where max_acting_primary_segments is the maximum number of primary segments that could be running on a host when mirror segments are activated due to a host or segment failure.

Another important Greenplum Database server configuration parameter is *statement_mem*. This parameter sets the maximum amount of memory to allocate to execute a query. To determine a value for this parameter, divide the amount of memory per segment (*gp_vmem_protect_limit*), less a 10% safety margin, by the maximum number of queries you expect to execute concurrently. The default Greenplum Database resource queue allows a maximum of 20 concurrent queries. Here is a formula to calculate *statement_mem*:

```
(gp_vmem_protect_limit * .9 ) / max_expected_concurrent_queries
```

Resource queues allow greater control of the amount of memory allocated for queries. See *Configuring Workload Management* for details.

Overview of Managing Workloads with Resource Queues

Resource queues are the main tool for managing the degree of concurrency in a Greenplum Database system. Resource queues are database objects that you create with the CREATE RESOURCE QUEUE SQL statement. You can use them to manage the number of active queries that may execute concurrently, the amount of memory each type of query is allocated, and the relative priority of queries. Resource queues can also guard against queries that would consume too many resources and degrade overall system performance.

Each database role is associated with a single resource queue; multiple roles can share the same resource queue. Roles are assigned to resource queues using the RESOURCE QUEUE phrase of the CREATE ROLE or ALTER ROLE statements. If a resource queue is not specified, the role is associated with the default resource queue, pg_default.

When the user submits a query for execution, the query is evaluated against the resource queue's limits. If the query does not cause the queue to exceed its resource limits, then that query will run immediately. If the query causes the queue to exceed its limits (for example, if the maximum number of active statement slots are currently in use), then the query must wait until queue resources are free before it can run. Queries are evaluated on a first in, first out basis. If query prioritization is enabled, the active workload on the system is periodically assessed and processing resources are reallocated according to query priority (see *How Priorities Work*). Roles with the SUPERUSER attribute are exempt from resource queue limits. Superuser queries always run immediately regardless of limits imposed by their assigned resource queue.

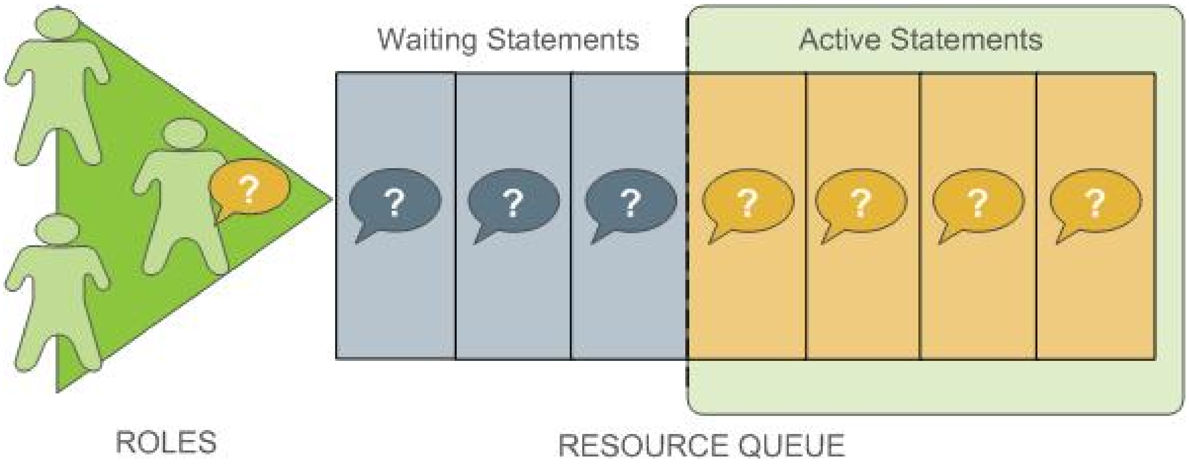


Figure 24: Resource Queue Process

Resource queues define classes of queries with similar resource requirements. Pivotal recommends that administratrors create resource queues for the various types of workloads in their organization. For example, you could create resource queues for the following classes of queries, corresponding to different service level agreements:

- ETL queries
- Reporting queries
- Executive queries

A resource queue has the following characteristics:

MEMORY_LIMIT

The amount of memory used by all the queries in the queue (per segment). For example, setting *MEMORY_LIMIT* to 2GB on the ETL queue allows ETL queries to use up to 2GB of memory in each segment.

ACTIVE_STATEMENTS

The number of *slots* for a queue; the maximum concurrency level for a queue. When all slots are used, new queries must wait. Each query uses an equal amount of memory by default.

For example, the `pg_default` resource queue has *ACTIVE_STATEMENTS* = 20.

PRIORITY

The relative CPU usage for queries. This may be one of the following levels: `LOW`, `MEDIUM`, `HIGH`, `MAX`. The default level is `MEDIUM`. The query prioritization mechanism monitors the CPU usage of all the queries running in the system, and adjusts the CPU usage for each to conform to its priority level. For example, you could set `MAX` priority to the `executive` resource queue and `MEDIUM` to other queues to ensure that executive queries receive a greater share of CPU.

MAX_COST

Query plan cost limit.

The Greenplum Database optimizer assigns a numeric cost to each query. If the cost exceeds the `MAX_COST` value set for the resource queue, the query is rejected as too expensive.

> **Note:** Pivotal recommends that you use `MEMORY_LIMIT` and `ACTIVE_STATEMENTS` to set limits for resource queues rather than `MAX_COST`.

The default configuration for a Greenplum Database system has a single default resource queue named `pg_default`. The `pg_default` resource queue has an `ACTIVE_STATEMENTS` setting of 20, no `MEMORY_LIMIT`, medium `PRIORITY`, and no set `MAX_COST`. This means that all queries are accepted and run immediately, at the same priority and with no memory limitations; however, only twenty queries may execute concurrently.

The number of concurrent queries a resource queue allows depends on whether the `MEMORY_LIMIT` parameter is set:

- If no `MEMORY_LIMIT` is set for a resource queue, the amount of memory allocated per query is the value of the `statement_mem` server configuration parameter. The maximum memory the resource queue can use is the product of `statement_mem` and `ACTIVE_STATEMENTS`.
- When a `MEMORY_LIMIT` is set on a resource queue, the number of queries that the queue can execute concurrently is limited by the queue's available memory.

A query admitted to the system is allocated an amount of memory and a query plan tree is generated for it. Each node of the tree is an operator, such as a sort or hash join. Each operator is a separate execution thread and is allocated a fraction of the overall statement memory, at minimum 100KB. If the plan has a large number of operators, the minimum memory required for operators can exceed the available memory and the query will be rejected with an insufficient memory error. Operators determine if they can complete their tasks in the memory allocated, or if they must spill data to disk, in work files. The mechanism that allocates and controls the amount of memory used by each operator is called *memory quota*.

Not all SQL statements submitted through a resource queue are evaluated against the queue limits. By default only `SELECT`, `SELECT INTO`, `CREATE TABLE AS SELECT`, and `DECLARE CURSOR` statements are evaluated. If the server configuration parameter `resource_select_only` is set to *off*, then `INSERT`, `UPDATE`, and `DELETE` statements will be evaluated as well.

Also, an SQL statement that is run during the execution of an `EXPLAIN ANALYZE` command is excluded from resource queues.

Resource Queue Example

The default resource queue, `pg_default`, allows a maximum of 20 active queries and allocates the same amount of memory to each. This is generally not adequate resource control for production systems. To ensure that the system meets performance expectations, you can define classes of queries and assign them to resource queues configured to execute them with the concurrency, memory, and CPU resources best suited for that class of query.

The following illustration shows an example resource queue configuration for a Greenplum Database system with `gp_vmem_protect_limit` set to 8GB:

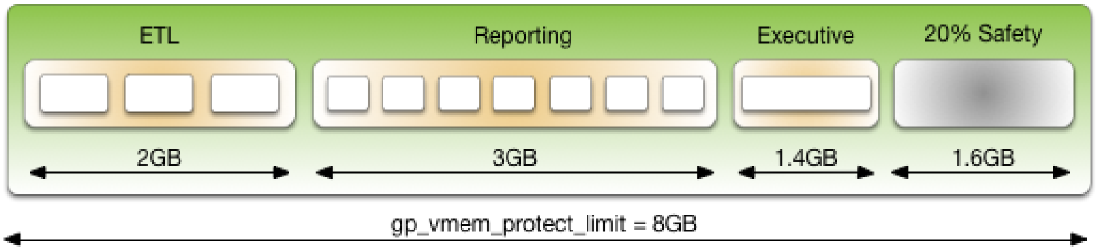


Figure 25: Resource Queue Configuration Example

This example has three classes of queries with different characteristics and service level agreements (SLAs). Three resource queues are configured for them. A portion of the segment memory is reserved as a safety margin.

Resource Queue Name	Active Statements	Memory Limit	Memory per Query
ETL	3	2GB	667MB
Reporting	7	3GB	429MB
Executive	1	1.4GB	1.4GB

The total memory allocated to the queues is 6.4GB, or 80% of the total segment memory defined by the gp_vmem_protect_limit server configuration parameter. Allowing a safety margin of 20% accommodates some operators and queries that are known to use more memory than they are allocated by the resource queue.

See the CREATE RESOURCE QUEUE and CREATE/ALTER ROLE statements in the *Greenplum Database Reference Guide* for help with command syntax and detailed reference information.

How Memory Limits Work

Setting MEMORY_LIMIT on a resource queue sets the maximum amount of memory that all active queries submitted through the queue can consume for a segment instance. The amount of memory allotted to a query is the queue memory limit divided by the active statement limit. (Pivotal recommends that memory limits be used in conjunction with statement-based queues rather than cost-based queues.) For example, if a queue has a memory limit of 2000MB and an active statement limit of 10, each query submitted through the queue is allotted 200MB of memory by default. The default memory allotment can be overridden on a per-query basis using the statement_mem server configuration parameter (up to the queue memory limit). Once a query has started executing, it holds its allotted memory in the queue until it completes, even if during execution it actually consumes less than its allotted amount of memory.

You can use the statement_mem server configuration parameter to override memory limits set by the current resource queue. At the session level, you can increae statement_mem up to the resource queue's MEMORY_LIMIT. This will allow an individual query to use all of the memory allocated for the entire queue without affecting other resource queues.

The value of statement_mem is capped using the max_statement_mem configuration parameter (a superuser parameter). For a query in a resource queue with MEMORY_LIMIT set, the maximum value for statement_mem is min(MEMORY_LIMIT, max_statement_mem). When a query is admitted, the memory allocated to it is subtracted from MEMORY_LIMIT. If MEMORY_LIMIT is exhausted, new queries in the same resource queue must wait. This happens even if ACTIVE_STATEMENTS has not yet been reached. Note that this can happen only when statement_mem is used to override the memory allocated by the resource queue.

For example, consider a resource queue named adhoc with the following settings:

- MEMORY_LIMIT is 1.5GB
- ACTIVE_STATEMENTS is 3

By default each statement submitted to the queue is allocated 500MB of memory. Now consider the following series of events:

1. User ADHOC_1 submits query Q1, overridingSTATEMENT_MEM to 800MB. The Q1 statement is admitted into the system.
2. User ADHOC_2 submits query Q2, using the default 500MB.
3. With Q1 and Q2 still running, user ADHOC3 submits query Q3, using the default 500MB.

Queries Q1 and Q2 have used 1300MB of the queue's 1500MB. Therefore, Q3 must wait for Q1 or Q2 to complete before it can run.

If MEMORY_LIMIT is not set on a queue, queries are admitted until all of the ACTIVE_STATEMENTS slots are in use, and each query can set an arbitrarily high statement_mem. This could lead to a resource queue using unbounded amounts of memory.

For more information on configuring memory limits on a resource queue, and other memory utilization controls, see *Creating Queues with Memory Limits*.

How Priorities Work

The PRIORITY setting for a resource queue differs from the MEMORY_LIMIT and ACTIVE_STATEMENTS settings, which determine whether a query will be admitted to the queue and eventually executed. The PRIORITY setting applies to queries after they become active. Active queries share available CPU resources as determined by the priority settings for its resource queue. When a statement from a high-priority queue enters the group of actively running statements, it may claim a greater share of the available CPU, reducing the share allocated to already-running statements in queues with a lesser priority setting.

The comparative size or complexity of the queries does not affect the allotment of CPU. If a simple, low-cost query is running simultaneously with a large, complex query, and their priority settings are the same, they will be allocated the same share of available CPU resources. When a new query becomes active, the CPU shares will be recalculated, but queries of equal priority will still have equal amounts of CPU.

For example, an administrator creates three resource queues: *adhoc* for ongoing queries submitted by business analysts, *reporting* for scheduled reporting jobs, and *executive* for queries submitted by executive user roles. The administrator wants to ensure that scheduled reporting jobs are not heavily affected by unpredictable resource demands from ad-hoc analyst queries. Also, the administrator wants to make sure that queries submitted by executive roles are allotted a significant share of CPU. Accordingly, the resource queue priorities are set as shown:

- *adhoc* — Low priority
- *reporting* — High priority
- *executive* — Maximum priority

At runtime, the CPU share of active statements is determined by these priority settings. If queries 1 and 2 from the reporting queue are running simultaneously, they have equal shares of CPU. When an ad-hoc query becomes active, it claims a smaller share of CPU. The exact share used by the reporting queries is adjusted, but remains equal due to their equal priority setting:

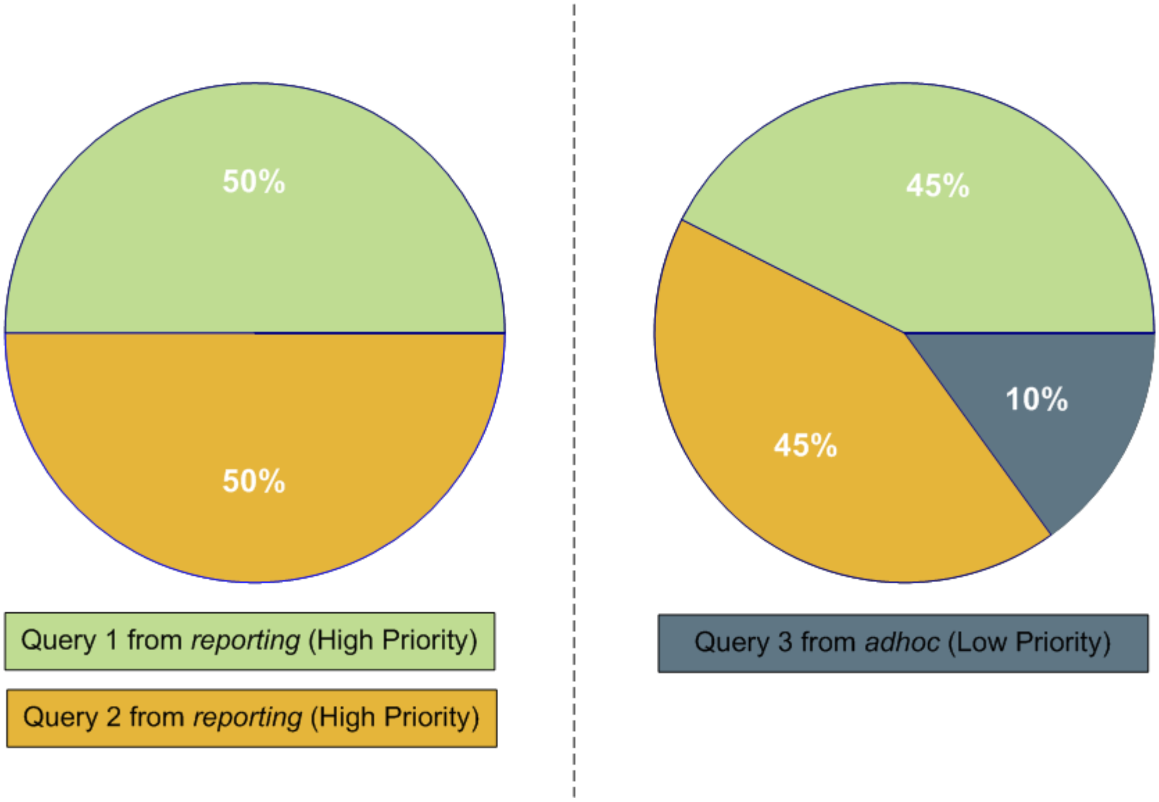


Figure 26: CPU share readjusted according to priority

Note:

The percentages shown in these illustrations are approximate. CPU usage between high, low and maximum priority queues is not always calculated in precisely these proportions.

When an executive query enters the group of running statements, CPU usage is adjusted to account for its maximum priority setting. It may be a simple query compared to the analyst and reporting queries, but until it is completed, it will claim the largest share of CPU.

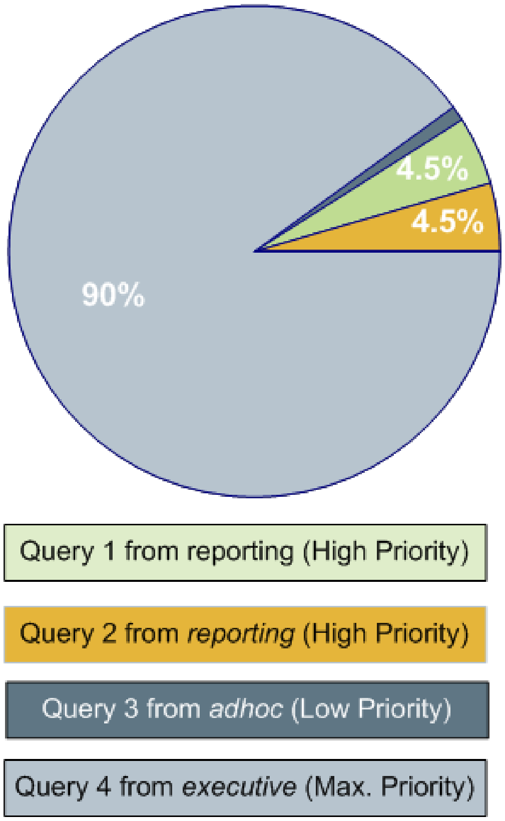


Figure 27: CPU share readjusted for maximum priority query

For more information about commands to set priorities, see *Setting Priority Levels*.

Steps to Enable Workload Management

Enabling and using workload management in Greenplum Database involves the following high-level tasks:

1. Configure workload managment. See *Configuring Workload Management*.
2. Create the resource queues and set limits on them. See *Creating Resource Queues* and *Modifying Resource Queues*.
3. Assign a queue to one or more user roles. See *Assigning Roles (Users) to a Resource Queue*.
4. Use the workload management system views to monitor and manage the resource queues. See *Checking Resource Queue Status*.

Configuring Workload Management

Resource scheduling is enabled by default when you install Greenplum Database, and is required for all roles. The default resource queue, `pg_default`, has an active statement limit of 20, no memory limit, and a medium priority setting. Pivotal recommends that you create resource queues for the various types of workloads.

To configure workload management

1. The following parameters are for the general configuration of resource queues:
    - `max_resource_queues` - Sets the maximum number of resource queues.

- max_resource_portals_per_transaction - Sets the maximum number of simultaneously open cursors allowed per transaction. Note that an open cursor will hold an active query slot in a resource queue.
- resource_select_only - If set to *on*, then SELECT, SELECT INTO, CREATE TABLE ASSELECT, and DECLARE CURSOR commands are evaluated. If set to *off*INSERT, UPDATE, and DELETE commands will be evaluated as well.
- resource_cleanup_gangs_on_wait - Cleans up idle segment worker processes before taking a slot in the resource queue.
- stats_queue_level - Enables statistics collection on resource queue usage, which can then be viewed by querying the pg_stat_resqueues system view.

2. The following parameters are related to memory utilization:

- gp_resqueue_memory_policy - Enables Greenplum Database memory management features.

  In Greenplum Database 4.2 and later, the distribution algorithm eager_free takes advantage of the fact that not all operators execute at the same time. The query plan is divided into stages and Greenplum Database eagerly frees memory allocated to a previous stage at the end of that stage's execution, then allocates the eagerly freed memory to the new stage.

  When set to none, memory management is the same as in Greenplum Database releases prior to 4.1. When set to auto, query memory usage is controlled by statement_mem and resource queue memory limits.
- statement_mem and max_statement_mem - Used to allocate memory to a particular query at runtime (override the default allocation assigned by the resource queue). max_statement_mem is set by database superusers to prevent regular database users from over-allocation.
- gp_vmem_protect_limit - Sets the upper boundary that all query processes can consume and should not exceed the amount of physical memory of a segment host. When a segment host reaches this limit during query execution, the queries that cause the limit to be exceeded will be cancelled.
- gp_vmem_idle_resource_timeout and gp_vmem_protect_segworker_cache_limit - used to free memory on segment hosts held by idle database processes. Administrators may want to adjust these settings on systems with lots of concurrency.
- shared_buffers - Sets the amount of memory a Greenplum server instance uses for shared memory buffers. This setting must be at least 128 kilobytes and at least 16 kilobytes times max_connections. The value must not exceed the operating system shared memory maximum allocation request size, shmmax on Linux. See the *Greenplum Database Installation Guide* for recommended OS memory settings for your platform.

3. The following parameters are related to query prioritization. Note that the following parameters are all *local* parameters, meaning they must be set in the postgresql.conf files of the master and all segments:

- gp_resqueue_priority - The query prioritization feature is enabled by default.
- gp_resqueue_priority_sweeper_interval - Sets the interval at which CPU usage is recalculated for all active statements. The default value for this parameter should be sufficient for typical database operations.
- gp_resqueue_priority_cpucores_per_segment - Specifies the number of CPU cores allocated per segment instance. The default value is 4 for the master and segments. For Greenplum Data Computing Appliance Version 2, the default value is 4 for segments and 25 for the master.

  Each host checks its own postgresql.conf file for the value of this parameter. This parameter also affects the master node, where it should be set to a value reflecting the higher ratio of CPU cores. For example, on a cluster that has 10 CPU cores per host and 4 segments per host, you would specify these values for gp_resqueue_priority_cpucores_per_segment:

  10 for the master and standby master. Typically, only the master instance is on the master host.

  2.5 for segment instances on the segment hosts.

If the parameter value is not set correctly, either the CPU might not be fully utilized, or query prioritization might not work as expected. For example, if the Greenplum Database cluster has fewer than one segment instance per CPU core on your segment hosts, make sure you adjust this value accordingly.

Actual CPU core utilization is based on the ability of Greenplum Database to parallelize a query and the resources required to execute the query.

Note: Any CPU core that is available to the operating system is included in the number of CPU cores. For example, virtual CPU cores are included in the number of CPU cores.

4. If you wish to view or change any of the workload management parameter values, you can use the `gpconfig` utility.
5. For example, to see the setting of a particular parameter:

   ```
   $ gpconfig --show gp_vmem_protect_limit
   ```
6. For example, to set one value on all segment instances and a different value on the master:

   ```
   $ gpconfig -c gp_resqueue_priority_cpucores_per_segment -v 2 -m 8
   ```
7. Restart Greenplum Database to make the configuration changes effective:

   ```
   $ gpstop -r
   ```

Creating Resource Queues

Creating a resource queue involves giving it a name, setting an active query limit, and optionally a query priority on the resource queue. Use the `CREATE RESOURCE QUEUE` command to create new resource queues.

Creating Queues with an Active Query Limit

Resource queues with an `ACTIVE_STATEMENTS` setting limit the number of queries that can be executed by roles assigned to that queue. For example, to create a resource queue named *adhoc* with an active query limit of three:

```
=# CREATE RESOURCE QUEUE adhoc WITH (ACTIVE_STATEMENTS=3);
```

This means that for all roles assigned to the *adhoc* resource queue, only three active queries can be running on the system at any given time. If this queue has three queries running, and a fourth query is submitted by a role in that queue, that query must wait until a slot is free before it can run.

Creating Queues with Memory Limits

Resource queues with a `MEMORY_LIMIT` setting control the amount of memory for all the queries submitted through the queue. The total memory should not exceed the physical memory available per-segment. Pivotal recommends that you set `MEMORY_LIMIT` to 90% of memory available on a per-segment basis. For example, if a host has 48 GB of physical memory and 6 segment instances, then the memory available per segment instance is 8 GB. You can calculate the recommended `MEMORY_LIMIT` for a single queue as 0.90*8=7.2 GB. If there are multiple queues created on the system, their total memory limits must also add up to 7.2 GB.

When used in conjunction with `ACTIVE_STATEMENTS`, the default amount of memory allotted per query is: `MEMORY_LIMIT / ACTIVE_STATEMENTS`. When used in conjunction with `MAX_COST`, the default amount of memory allotted per query is: `MEMORY_LIMIT * (query_cost / MAX_COST)`. Pivotal recommends that `MEMORY_LIMIT` be used in conjunction with `ACTIVE_STATEMENTS` rather than with `MAX_COST`.

For example, to create a resource queue with an active query limit of 10 and a total memory limit of 2000MB (each query will be allocated 200MB of segment host memory at execution time):

```
=# CREATE RESOURCE QUEUE myqueue WITH (ACTIVE_STATEMENTS=20,
MEMORY_LIMIT='2000MB');
```

The default memory allotment can be overridden on a per-query basis using the `statement_mem` server configuration parameter, provided that `MEMORY_LIMIT` or `max_statement_mem` is not exceeded. For example, to allocate more memory to a particular query:

```
=> SET statement_mem='2GB';
=> SELECT * FROM my_big_table WHERE column='value' ORDER BY id;
=> RESET statement_mem;
```

As a general guideline, `MEMORY_LIMIT` for all of your resource queues should not exceed the amount of physical memory of a segment host. If workloads are staggered over multiple queues, it may be OK to oversubscribe memory allocations, keeping in mind that queries may be cancelled during execution if the segment host memory limit (`gp_vmem_protect_limit`) is exceeded.

Setting Priority Levels

To control a resource queue's consumption of available CPU resources, an administrator can assign an appropriate priority level. When high concurrency causes contention for CPU resources, queries and statements associated with a high-priority resource queue will claim a larger share of available CPU than lower priority queries and statements.

Priority settings are created or altered using the `WITH` parameter of the commands `CREATE RESOURCE QUEUE` and `ALTER RESOURCE QUEUE`. For example, to specify priority settings for the *adhoc* and *reporting* queues, an administrator would use the following commands:

```
=# ALTER RESOURCE QUEUE adhoc WITH (PRIORITY=LOW);
=# ALTER RESOURCE QUEUE reporting WITH (PRIORITY=HIGH);
```

To create the *executive* queue with maximum priority, an administrator would use the following command:

```
=# CREATE RESOURCE QUEUE executive WITH (ACTIVE_STATEMENTS=3, PRIORITY=MAX);
```

When the query prioritization feature is enabled, resource queues are given a `MEDIUM` priority by default if not explicitly assigned. For more information on how priority settings are evaluated at runtime, see *How Priorities Work*.

> **Important:** In order for resource queue priority levels to be enforced on the active query workload, you must enable the query prioritization feature by setting the associated server configuration parameters. See *Configuring Workload Management*.

Assigning Roles (Users) to a Resource Queue

Once a resource queue is created, you must assign roles (users) to their appropriate resource queue. If roles are not explicitly assigned to a resource queue, they will go to the default resource queue, `pg_default`. The default resource queue has an active statement limit of 20, no cost limit, and a medium priority setting.

Use the `ALTER ROLE` or `CREATE ROLE` commands to assign a role to a resource queue. For example:

```
=# ALTER ROLE name RESOURCE QUEUE queue_name;
=# CREATE ROLE name WITH LOGIN RESOURCE QUEUE queue_name;
```

A role can only be assigned to one resource queue at any given time, so you can use the `ALTER ROLE` command to initially assign or change a role's resource queue.

Resource queues must be assigned on a user-by-user basis. If you have a role hierarchy (for example, a group-level role) then assigning a resource queue to the group does not propagate down to the users in that group.

Superusers are always exempt from resource queue limits. Superuser queries will always run regardless of the limits set on their assigned queue.

Removing a Role from a Resource Queue

All users *must* be assigned to a resource queue. If not explicitly assigned to a particular queue, users will go into the default resource queue, `pg_default`. If you wish to remove a role from a resource queue and put them in the default queue, change the role's queue assignment to `none`. For example:

```
=# ALTER ROLE role_name RESOURCE QUEUE none;
```

Modifying Resource Queues

After a resource queue has been created, you can change or reset the queue limits using the `ALTER RESOURCE QUEUE` command. You can remove a resource queue using the `DROP RESOURCE QUEUE` command. To change the roles (users) assigned to a resource queue, *Assigning Roles (Users) to a Resource Queue*.

Altering a Resource Queue

The `ALTER RESOURCE QUEUE` command changes the limits of a resource queue. To change the limits of a resource queue, specify the new values you want for the queue. For example:

```
=# ALTER RESOURCE QUEUE adhoc WITH (ACTIVE_STATEMENTS=5);
=# ALTER RESOURCE QUEUE exec WITH (PRIORITY=MAX);
```

To reset active statements or memory limit to no limit, enter a value of `-1`. To reset the maximum query cost to no limit, enter a value of `-1.0`. For example:

```
=# ALTER RESOURCE QUEUE adhoc WITH (MAX_COST=-1.0, MEMORY_LIMIT='2GB');
```

You can use the `ALTER RESOURCE QUEUE` command to change the priority of queries associated with a resource queue. For example, to set a queue to the minimum priority level:

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

Dropping a Resource Queue

The `DROP RESOURCE QUEUE` command drops a resource queue. To drop a resource queue, the queue cannot have any roles assigned to it, nor can it have any statements waiting in the queue. See *Removing a Role from a Resource Queue* and *Clearing a Waiting Statement From a Resource Queue* for instructions on emptying a resource queue. To drop a resource queue:

```
=# DROP RESOURCE QUEUE name;
```

Checking Resource Queue Status

Checking resource queue status involves the following tasks:

- *Viewing Queued Statements and Resource Queue Status*
- *Viewing Resource Queue Statistics*
- *Viewing the Roles Assigned to a Resource Queue*

- *Viewing the Waiting Queries for a Resource Queue*
- *Clearing a Waiting Statement From a Resource Queue*
- *Viewing the Priority of Active Statements*
- *Resetting the Priority of an Active Statement*

Viewing Queued Statements and Resource Queue Status

The `gp_toolkit.gp_resqueue_status` view allows administrators to see status and activity for a workload management resource queue. It shows how many queries are waiting to run and how many queries are currently active in the system from a particular resource queue. To see the resource queues created in the system, their limit attributes, and their current status:

```
=# SELECT * FROM gp_toolkit.gp_resqueue_status;
```

Viewing Resource Queue Statistics

If you want to track statistics and performance of resource queues over time, you can enable statistics collecting for resource queues. This is done by setting the following server configuration parameter in your master `postgresql.conf` file:

```
stats_queue_level = on
```

Once this is enabled, you can use the `pg_stat_resqueues` system view to see the statistics collected on resource queue usage. Note that enabling this feature does incur slight performance overhead, as each query submitted through a resource queue must be tracked. It may be useful to enable statistics collecting on resource queues for initial diagnostics and administrative planning, and then disable the feature for continued use.

See the Statistics Collector section in the PostgreSQL documentation for more information about collecting statistics in Greenplum Database.

Viewing the Roles Assigned to a Resource Queue

To see the roles assigned to a resource queue, perform the following query of the `pg_roles` and `gp_toolkit.gp_resqueue_status` system catalog tables:

```
=# SELECT rolname, rsqname FROM pg_roles,
          gp_toolkit.gp_resqueue_status
   WHERE pg_roles.rolresqueue=gp_toolkit.gp_resqueue_status.queueid;
```

You may want to create a view of this query to simplify future inquiries. For example:

```
=# CREATE VIEW role2queue AS
   SELECT rolname, rsqname FROM pg_roles, pg_resqueue
   WHERE pg_roles.rolresqueue=gp_toolkit.gp_resqueue_status.queueid;
```

Then you can just query the view:

```
=# SELECT * FROM role2queue;
```

Viewing the Waiting Queries for a Resource Queue

When a slot is in use for a resource queue, it is recorded in the `pg_locks` system catalog table. This is where you can see all of the currently active and waiting queries for all resource queues. To check that statements are being queued (even statements that are not waiting), you can also use the `gp_toolkit`.gp_locks_on_resqueue view. For example:

```
=# SELECT * FROM gp_toolkit.gp_locks_on_resqueue WHERE lorwaiting='true';
```

If this query returns no results, then that means there are currently no statements waiting in a resource queue.

Clearing a Waiting Statement From a Resource Queue

In some cases, you may want to clear a waiting statement from a resource queue. For example, you may want to remove a query that is waiting in the queue but has not been executed yet. You may also want to stop a query that has been started if it is taking too long to execute, or if it is sitting idle in a transaction and taking up resource queue slots that are needed by other users. To do this, you must first identify the statement you want to clear, determine its process id (pid), and then, use `pg_cancel_backend` with the process id to end that process, as shown below.

For example, to see process information about all statements currently active or waiting in all resource queues, run the following query:

```
=# SELECT rolname, rsqname, pid, granted,
          current_query, datname
   FROM pg_roles, gp_toolkit.gp_resqueue_status, pg_locks,
        pg_stat_activity
   WHERE pg_roles.rolresqueue=pg_locks.objid
   AND pg_locks.objid=gp_toolkit.gp_resqueue_status.queueid
   AND pg_stat_activity.procpid=pg_locks.pid;
   AND pg_stat_activity.usename=pg_roles.rolname;
```

If this query returns no results, then that means there are currently no statements in a resource queue. A sample of a resource queue with two statements in it looks something like this:

```
rolname | rsqname |  pid  | granted |     current_query      | datname
-----------------------------------------------------------------------
  sammy | webuser | 31861 | t       | <IDLE> in transaction  | namesdb
  daria | webuser | 31905 | f       | SELECT * FROM topten;  | namesdb
```

Use this output to identify the process id (pid) of the statement you want to clear from the resource queue. To clear the statement, you would then open a terminal window (as the `gpadmin` database superuser or as root) on the master host and cancel the corresponding process. For example:

```
=# pg_cancel_backend(31905)
```

Note:

Do not use any operating system `KILL` command.

Viewing the Priority of Active Statements

The *gp_toolkit* administrative schema has a view called *gp_resq_priority_statement*, which lists all statements currently being executed and provides the priority, session ID, and other information.

This view is only available through the `gp_toolkit` administrative schema. See the *Greenplum Database Reference Guide* for more information.

Resetting the Priority of an Active Statement

Superusers can adjust the priority of a statement currently being executed using the built-in function `gp_adjust_priority(session_id, statement_count, priority)`. Using this function, superusers can raise or lower the priority of any query. For example:

```
=# SELECT gp_adjust_priority(752, 24905, 'HIGH')
```

To obtain the session ID and statement count parameters required by this function, superusers can use the `gp_toolkit` administrative schema view, *gp_resq_priority_statement*. From the view, use these values for the function parameters.

- The value of the `rqpsession` column for the `session_id` parameter
- The value of the `rqpcommand` column for the `statement_count` parameter
- The value of `rqppriority` column is the current priority. You can specify a string value of `MAX`, `HIGH`, `MEDIUM`, or `LOW` as the `priority`.

> **Note:** The `gp_adjust_priority()` function affects only the specified statement. Subsequent statements in the same resource queue are executed using the queue's normally assigned priority.

Chapter 27

Investigating a Performance Problem

This section provides guidelines for identifying and troubleshooting performance problems in a Greenplum Database system.

This topic lists steps you can take to help identify the cause of a performance problem. If the problem affects a particular workload or query, you can focus on tuning that particular workload. If the performance problem is system-wide, then hardware problems, system failures, or resource contention may be the cause.

Checking System State

Use the `gpstate` utility to identify failed segments. A Greenplum Database system will incur performance degradation when segment instances are down because other hosts must pick up the processing responsibilities of the down segments.

Failed segments can indicate a hardware failure, such as a failed disk drive or network card. Greenplum Database provides the hardware verification tool `gpcheckperf` to help identify the segment hosts with hardware issues.

Checking Database Activity

- *Checking for Active Sessions (Workload)*
- *Checking for Locks (Contention)*
- *Checking Query Status and System Utilization*

Checking for Active Sessions (Workload)

The *pg_stat_activity* system catalog view shows one row per server process; it shows the database OID, database name, process ID, user OID, user name, current query, time at which the current query began execution, time at which the process was started, client address, and port number. To obtain the most information about the current system workload, query this view as the database superuser. For example:

```
SELECT * FROM pg_stat_activity;
```

Note the information does not update instantaneously.

Checking for Locks (Contention)

The *pg_locks* system catalog view allows you to see information about outstanding locks. If a transaction is holding a lock on an object, any other queries must wait for that lock to be released before they can continue. This may appear to the user as if a query is hanging.

Examine *pg_locks* for ungranted locks to help identify contention between database client sessions. *pg_locks* provides a global view of all locks in the database system, not only those relevant to the current database. You can join its relation column against `pg_class.oid` to identify locked relations (such as tables), but this works correctly only for relations in the current database. You can join the `pid` column to the `pg_stat_activity.procpid` to see more information about the session holding or waiting to hold a lock. For example:

```
SELECT locktype, database, c.relname, l.relation,
l.transactionid, l.transaction, l.pid, l.mode, l.granted,
a.current_query
        FROM pg_locks l, pg_class c, pg_stat_activity a
        WHERE l.relation=c.oid AND l.pid=a.procpid
        ORDER BY c.relname;
```

If you use resource queues for workload management, queries that are waiting in a queue will also show in *pg_locks*. To see how many queries are waiting to run from a resource queue, use the *gp_resqueue_status* system catalog view. For example:

```
SELECT * FROM gp_toolkit.gp_resqueue_status;
```

Checking Query Status and System Utilization

You can use system monitoring utilities such as `ps`, `top`, `iostat`, `vmstat`, `netstat` and so on to monitor database activity on the hosts in your Greenplum Database array. These tools can help identify Greenplum Database processes (`postgres` processes) currently running on the system and the most resource intensive tasks with regards to CPU, memory, disk I/O, or network activity. Look at these system statistics to identify queries that degrade database performance by overloading the system and consuming excessive resources. Greenplum Database's management tool `gpssh` allows you to run these system monitoring commands on several hosts simultaneously.

You can create and use the Greenplum Database *session_level_memory_consumption* view that provides information about the current memory utilization for sessions that are running queries on Greenplum Database. For information about the view, see *Viewing Session Memory Usage Information*.

The Greenplum Command Center collects query and system utilization metrics. See the *Greenplum Command Center Administrator Guide* for procedures to enable Greenplum Command Center.

Troubleshooting Problem Queries

If a query performs poorly, look at its query plan to help identify problems. The `EXPLAIN` command shows the query plan for a given query. See *Query Profiling* for more information about reading query plans and identifying problems.

When an out of memory event occurs during query execution, the Greenplum Database memory accounting framework reports detailed memory consumption of every query running at the time of the event. The information is written to the Greenplum Database segment logs.

Investigating Error Messages

Greenplum Database log messages are written to files in the `pg_log` directory within the master's or segment's data directory. Because the master log file contains the most information, you should always check it first. Log files roll over daily and use the naming convention: `gpdb-YYYY-MM-DD_hhmmss.csv`. To locate the log files on the master host:

```
$ cd $MASTER_DATA_DIRECTORY/pg_log
```

Log lines have the format of:

```
timestamp | user | database | statement_id | con#cmd#
|:-LOG_LEVEL: log_message
```

You may want to focus your search for `WARNING`, `ERROR`, `FATAL` or `PANIC` log level messages. You can use the Greenplum utility `gplogfilter` to search through Greenplum Database log files. For example, when you run the following command on the master host, it checks for problem log messages in the standard logging locations:

```
$ gplogfilter -t
```

To search for related log entries in the segment log files, you can run gplogfilter on the segment hosts using gpssh. You can identify corresponding log entries by the *statement_id* or *con#* (session identifier). For example, to search for log messages in the segment log files containing the string con6 and save output to a file:

```
gpssh -f seg_hosts_file -e 'source
/usr/local/greenplum-db/greenplum_path.sh ; gplogfilter -f
con6 /gpdata/*/pg_log/gpdb*.csv' > seglog.out
```

Gathering Information for Pivotal Support

The Greenplum Database gpsupport utility can collect information from a Greenplum Database system. You can then send the information file to Pivotal Customer Support to aid the diagnosis of Greenplum Database errors or system failures. For example, this gpsupport command collects all logs from the listed segment hosts in a Greenplum Database cluster:

```
$ gpsupport collect logs segs sdw1,sdw2,sdw3-1
```

The gpsupport utility can also collect other information such as core files, previously run queries, and database schema information. The utility can perform some diagnostics tests on the database catalog the database network.

The gpsupport utility is available from *Pivotal Network* and the documentation is available from the *Pivotal Documentation* site.