Pivotal

PRODUCT DOCUMENTATION

Pivotal™ Greenplum Database®

Version 4.3

Best Practices Guide

Rev: A08

Notice

Copyright

Revised January, 2018 (4.3.21.0)

Contents

Chapter 1

Introduction

Greenplum Database Best Practices Guide describes best practices for Greenplum Database. A best practice is a method or technique that has consistently shown results superior to those achieved with other means. Best practices are found through experience and are proven to reliably lead to a desired result. Best practices are a commitment to use any product correctly and optimally, by leveraging all the knowledge and expertise available to ensure success.

This document does not teach you how to use Greenplum Database features. Refer to the Greenplum Database documentation at *http://gpdb.docs.pivotal.io* for information on how to use and implement specific Greenplum Database features. This document addresses the most important best practices to follow when designing, implementing, and using Greenplum Database.

It is not the intent of this document to cover the entire product or compendium of features but to provide a summary of *what matters most* in Greenplum Database. This document does not address *edge* use cases that can further leverage and benefit from these Greenplum Database features. Edge use cases require a proficient knowledge and expertise of these features and a deep understanding of your environment including SQL access, query execution, concurrency, workload, and other factors.

By mastering these best practices you will increase the success of your Greenplum Database clusters in the areas of maintenance, support, performance and scalability.

Best Practices Summary

This section contains a summary of best practices for Greenplum Database.

Data Model

- Greenplum Database is an analytical MPP shared nothing database. This model is significantly different from a highly normalized/transactional SMP database. Greenplum Database performs best with a denormalized schema design suited for MPP analytical processing for example, Star or Snowflake schema, with large fact tables and smaller dimension tables.
- Use the same data types for columns used in joins between tables.

See *Schema Design*.

Heap vs. Append-Optimized Storage

- Use heap storage for tables and partitions that will receive iterative batch and singleton `UPDATE`, `DELETE`, and `INSERT` operations.
- Use heap storage for tables and partitions that will receive concurrent `UPDATE`, `DELETE`, and `INSERT` operations.
- Use append-optimized storage for tables and partitions that are updated infrequently after the initial load and have subsequent inserts only performed in large batch operations.
- Never perform singleton `INSERT`, `UPDATE`, or `DELETE` operations on append-optimized tables.
- Never perform concurrent batch `UPDATE` or `DELETE` operations on append-optimized tables. Concurrent batch `INSERT` operations are okay.

See *Heap Storage or Append-Optimized Storage*.

Row vs. Column Oriented Storage

- Use row-oriented storage for workloads with iterative transactions where updates are required and frequent inserts are performed.
- Use row-oriented storage when selects against the table are wide.
- Use row-oriented storage for general purpose or mixed workloads.
- Use column-oriented storage where selects are narrow and aggregations of data are computed over a small number of columns.
- Use column-oriented storage for tables that have single columns that are regularly updated without modifying other columns in the row.

See *Row or Column Orientation*.

Compression

- Use compression on large append-optimized and partitioned tables to improve I/O across the system.
- Set the column compression settings at the level where the data resides.
- Balance higher levels of compression with the time and CPU cycles needed to compress and uncompress data.

See *Compression*.

Distributions

- Explicitly define a column or random distribution for all tables. Do not use the default.
- Use a single column that will distribute data across all segments evenly.

- Do not distribute on columns that will be used in the `WHERE` clause of a query.
- Do not distribute on dates or timestamps.
- Never distribute and partition tables on the same column.
- Achieve local joins to significantly improve performance by distributing on the same column for large tables commonly joined together.
- Validate that data is evenly distributed after the initial load and after incremental loads.
- Ultimately ensure there is no data skew!

See *Distributions*.

Memory Management

- Set `vm.overcommit_memory` to 2.
- Do not configure the OS to use huge pages.
- Use `gp_vmem_protect_limit` to set the maximum memory that the instance can allocate for *all* work being done in each segment database.
- Set the value for `gp_vmem_protect_limit` by calculating:
  - `gp_vmem` – the total memory available to Greenplum Database

    ```
    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 host's RAM in GB
  - `max_acting_primary_segments` – 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
  - `gp_vmem_protect_limit`

    ```
    gp_vmem_protect_limit = gp_vmem / acting_primary_segments
    ```

    Convert to MB to set the value of the configuration parameter.
- In a scenario where a large number of workfiles are generated calculate the `gp_vmem` factor with this formulat to account for the workfiles:

  ```
  gp_vmem = ((SWAP + RAM) – (7.5GB + 0.05 * RAM - (300KB *
        total_#_workfiles))) / 1.7
  ```
- Never set `gp_vmem_protect_limit` too high or larger than the physical RAM on the system.
- Use the calculated `gp_vmem` value to calculate the setting for the `vm.overcommit_ratio` operating system parameter:

  ```
  vm.overcommit_ratio = (RAM - 0.026 * gp_vmem) / RAM
  ```
- Use `statement_mem` to allocate memory used for a query per segment db.
- Use resource queues to set both the numbers of active queries (`ACTIVE_STATEMENTS`) and the amount of memory (`MEMORY_LIMIT`) that can be utilized by queries in the queue.
- Associate all users with a resource queue. Do not use the default queue.
- Set `PRIORITY` to match the real needs of the queue for the workload and time of day.
- Ensure that resource queue memory allocations do not exceed the setting for `gp_vmem_protect_limit`.
- Dynamically update resource queue settings to match daily operations flow.

See *Memory and Workload Management*.

Partitioning

- Partition large tables only. Do not partition small tables.

- Use partitioning only if partition elimination (partition pruning) can be achieved based on the query criteria.
- Choose range partitioning over list partitioning.
- Partition the table based on the query predicate.
- Never partition and distribute tables on the same column.
- Do not use default partitions.
- Do not use multi-level partitioning; create fewer partitions with more data in each partition.
- Validate that queries are selectively scanning partitioned tables (partitions are being eliminated) by examining the query `EXPLAIN` plan.
- Do not create too many partitions with column-oriented storage because of the total number of physical files on every segment: `physical files = segments x columns x partitions`

See *Partitioning*.

Indexes

- In general indexes are not needed in Greenplum Database.
- Create an index on a single column of a columnar table for drill-through purposes for high cardinality tables that require queries with high selectivity.
- Do not index columns that are frequently updated.
- Always drop indexes before loading data into a table. After the load, re-create the indexes for the table.
- Create selective B-tree indexes.
- Do not create bitmap indexes on columns that are updated.
- Do not use bitmap indexes for unique columns, very high or very low cardinality data.
- Do not use bitmap indexes for transactional workloads.
- In general do not index partitioned tables. If indexes are needed, the index columns must be different than the partition columns.

See *Indexes*.

Resource Queues

- Use resource queues to manage the workload on the cluster.
- Associate all roles with a user-defined resource queue.
- Use the `ACTIVE_STATEMENTS` parameter to limit the number of active queries that members of the particular queue can run concurrently.
- Use the `MEMORY_LIMIT` parameter to control the total amount of memory that queries running through the queue can utilize.
- Do not set all queues to `MEDIUM`, as this effectively does nothing to manage the workload.
- Alter resource queues dynamically to match the workload and time of day.

See *Configuring Resource Queues*.

Monitoring and Maintenance

- Implement the "Recommended Monitoring and Maintenance Tasks" in the *Greenplum Database Administrator Guide*.
- Run `gpcheckperf` at install time and periodically thereafter, saving the output to compare system performance over time.
- Use all the tools at your disposal to understand how your system behaves under different loads.
- Examine any unusual event to determine the cause.
- Monitor query activity on the system by running explain plans periodically to ensure the queries are running optimally.

- Review plans to determine whether index are being used and partition elimination is occurring as expected.
- Know the location and content of system log files and monitor them on a regular basis, not just when problems arise.

See *System Monitoring and Maintenance* and *Monitoring Greenplum Database Log Files*.

ANALYZE

- Do not run `ANALYZE` on the entire database. Selectively run `ANALYZE` at the table level when needed.
- Always run `ANALYZE` after loading.
- Always run `ANALYZE` after `INSERT`, `UPDATE`. and `DELETE` operations that significantly changes the underlying data.
- Always run `ANALYZE` after `CREATE INDEX` operations.
- If `ANALYZE` on very large tables takes too long, run `ANALYZE` only on the columns used in a join condition, `WHERE` clause, `SORT`, `GROUP BY`, or `HAVING` clause.

See *Updating Statistics with ANALYZE*.

Vacuum

- Run `VACUUM` after large `UPDATE` and `DELETE` operations.
- Do not run `VACUUM FULL`. Instead run a `CREATE TABLE...AS` operation, then rename and drop the original table.
- Frequently run `VACUUM` on the system catalogs to avoid catalog bloat and the need to run `VACUUM FULL` on catalog tables.
- Never kill `VACUUM` on catalog tables.
- Do not run `VACUUM ANALYZE`.

See *Managing Bloat in the Database*.

Loading

- Use `gpfdist` to load or unload data in Greenplum Database.
- Maximize the parallelism as the number of segments increase.
- Spread the data evenly across as many ETL nodes as possible.
- Split very large data files into equal parts and spread the data across as many file systems as possible.
- Run two `gpfdist` instances per file system.
- Run `gpfdist` on as many interfaces as possible.
- Use `gp_external_max_segs` to control the number of segments each `gpfdist` serves.
- Always keep `gp_external_max_segs` and the number of `gpfdist` processes an even factor.
- Always drop indexes before loading into existing tables and re-create the index after loading.
- Always run `ANALYZE` on the table after loading it.
- Disable automatic statistics collection during loading by setting `gp_autostats_mode` to `NONE`.
- Run `VACUUM` after load errors to recover space.

See *Loading Data*.

gptransfer

- For fastest transfer rates, use `gptransfer` to transfer data to a destination database that is the same size or larger.
- Avoid using the `--full` or `--schema-only` options. Instead, copy schemas to the destination database using a different method, and then transfer the table data.

- Drop indexes before transferring tables and recreate them when the transfer is complete.
- Transfer smaller tables to the destination database using the SQL `COPY` command.
- Transfer larger tables in batches using `gptransfer`.
- Test running `gptransfer` before performing a production migration. Experiment with the `--batch-size` and `--sub-batch-size` options to obtain maximum parallelism. Determine proper batching of tables for iterative `gptransfer` runs.
- Use only fully qualified table names. Periods (.), whitespace, quotes (') and double quotes (") in table names may cause problems.
- If you use the `--validation` option to validate the data after transfer, be sure to also use the `-x` option to place an exclusive lock on the source table.
- Ensure any roles, functions, and resource queues are created in the destination database. These objects are not transferred when you use the `gptransfer -t` option.
- Copy the `postgres.conf` and `pg_hba.conf` configuration files from the source to the destination cluster.
- Install needed extensions in the destination database with `gppkg`.

See *Migrating Data with Gptransfer*.

Security

- Secure the `gpadmin` user id and only allow essential system administrators access to it.
- Administrators should only log in to Greenplum as `gpadmin` when performing certain system maintenance tasks (such as upgrade or expansion).
- 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" in the *Greenplum Database Administrator Guide*.
- 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 who logs in.
- For applications or web services, consider creating a distinct role for each application or service.
- Use groups to manage access privileges.
- Protect the root password.
- Enforce a strong password password policy for operating system passwords.
- Ensure that important operating system files are protected.

See *Security*.

Encryption

- Encrypting and decrypting data has a performance cost; only encrypt data that requires encryption.
- Do performance testing before implementing any encryption solution in a production system.
- Server certificates in a production Greenplum Database system should be signed by a certificate authority (CA) so that clients can authenticate the server. The CA may be local if all clients are local to the organization.
- Client connections to Greenplum Database should use SSL encryption whenever the connection goes through an insecure link.
- A symmetric encryption scheme, where the same key is used to both encrypt and decrypt, has better performance than an asymmetric scheme and should be used when the key can be shared safely.
- Use functions from the pgcrypto package to encrypt data on disk. The data is encrypted and decrypted in the database process, so it is important to secure the client connection with SSL to avoid transmitting unencrypted data.
- Use the gpfdists protocol to secure ETL data as it is loaded into or unloaded from the database. .

See *Encrypting Data and Database Connections*

High Availability

- Use a hardware RAID storage solution with 8 to 24 disks.
- Use RAID 1, 5, or 6 so that the disk array can tolerate a failed disk.
- Configure a hot spare in the disk array to allow rebuild to begin automatically when disk failure is detected.
- Protect against failure of the entire disk array and degradation during rebuilds by mirroring the RAID volume.
- Monitor disk utilization regularly and add additional space when needed.
- Monitor segment skew to ensure that data is distributed evenly and storage is consumed evenly at all segments.
- Set up a standby master instance to take over if the primary master fails.
- Plan how to switch clients to the new master instance when a failure occurs, for example, by updating the master address in DNS.
- Set up monitoring to send notifications in a system monitoring application or by email when the primary fails.
- Set up mirrors for all segments.
- Locate primary segments and their mirrors on different hosts to protect against host failure.
- Set up monitoring to send notifications in a system monitoring application or by email when a primary segment fails.
- Recover failed segments promptly, using the `gprecoverseg` utility, to restore redundancy and return the system to optimal balance.
- Configure Greenplum Database to send SNMP notifications to your network monitor.
- Set up email notification in the `$MASTER_DATA_DIRECTORY/postgresql.conf` configuration file so that the Greenplum system can email administrators when a critical issue is detected.
- Consider a Dual Cluster configuration to provide an additional level of redundancy and additional query processing throughput.
- Backup Greenplum databases regularly unless the data is easily restored from sources.
- Use incremental backups if heap tables are relatively small and few append-optimized or column-oriented partitions are modified between backups.
- If backups are saved to local cluster storage, move the files to a safe, off-cluster location when the backup is complete.
- If backups are saved to NFS mounts, use a scale-out NFS solution such as Dell EMC Isilon to prevent IO bottlenecks.
- Consider using Greenplum integration to stream backups to the Dell EMC Data Domain or Veritas NetBackup enterprise backup platforms.

See *High Availability*.

Chapter 2

System Configuration

The topics in this section describe requirements and best practices for configuring Greenplum Database cluster hosts.

Configuration of the Greenplum Database cluster is usually performed as root.

Preferred Operating System

Red Hat Enterprise Linux (RHEL) is the preferred operating system. The latest supported major version should be used, currently RHEL 7.3.

File System

XFS is the file system used for Greenplum Database data directories. XFS should be mounted with the following mount options:

```
rw,nodev,noatime,nobarrier,inode64
```

Port Configuration

Set up `ip_local_port_range` so it does not conflict with the Greenplum Database port ranges. For example, setting this range in `/etc/sysctl.conf`:

```
net.ipv4.ip_local_port_range = 10000  65535
```

you could set the Greenplum Database base port numbers to these values.

```
PORT_BASE = 6000
MIRROR_PORT_BASE = 7000
REPLICATION_PORT_BASE = 8000
MIRROR_REPLICATION_PORT_BASE = 9000
```

For information about port ranges that are used by Greenplum Database, see *gpinitsystem* in the *Greenplum Database Utility Guide*.

I/O Configuration

Set the blockdev read-ahead size to 16384 on the devices that contain data directories. This command sets the read-ahead size for `/dev/sdb`.

```
# /sbin/blockdev --setra 16384 /dev/sdb
```

This command returns the read-ahead size for `/dev/sdb`.

```
# /sbin/blockdev --getra /dev/sdb
16384
```

The deadline IO scheduler should be set for all data directory devices.

```
 # cat /sys/block/sdb/queue/scheduler
 noop anticipatory [deadline] cfq
```

The maximum number of OS files and processes should be increased in the `/etc/security/limits.conf` file.

```
* soft  nofile 65536
* hard  nofile 65536
* soft  nproc 131072
* hard  nproc 131072
```

Enable core files output to a known location and make sure `limits.conf` allows core files.

```
kernel.core_pattern = /var/core/core.%h.%t
# grep core /etc/security/limits.conf
* soft  core unlimited
```

OS Memory Configuration

The Linux sysctl `vm.overcommit_memory` and `vm.overcommit_ratio` variables affect how the operating system manages memory allocation. These variables should be set as follows:

`vm.overcommit_memory` determines the method the OS uses for determining how much memory can be allocated to processes. This should be always set to 2, which is the only safe setting for the database.

> **Note:** For information on configuration of overcommit memory, refer to:
>
> - *https://en.wikipedia.org/wiki/Memory_overcommitment*
> - *https://www.kernel.org/doc/Documentation/vm/overcommit-accounting*

`vm.overcommit_ratio` is the percent of RAM that is used for application processes. The default is 50 on Red Hat Enterprise Linux. See *Segment Memory Configuration* for a formula to calculate an optimal value.

Do not enable huge pages in the operating system.

See also *Memory and Workload Management*.

Shared Memory Settings

Greenplum Database uses shared memory to communicate between `postgres` processes that are part of the same `postgres` instance. The following shared memory settings should be set in `sysctl` and are rarely modified.

```
kernel.shmmax = 500000000
kernel.shmmni = 4096
kernel.shmall = 4000000000
```

Validate the Operating System

Run `gpcheck` to validate the operating system configuration. See `gpcheck` in the *Greenplum Database Utility Guide*.

Number of Segments per Host

Determining the number of segments to execute on each segment host has immense impact on overall system performance. The segments share the host's CPU cores, memory, and NICs with each other and with other processes running on the host. Over-estimating the number of segments a server can accommodate is a common cause of suboptimal performance.

The factors that must be considered when choosing how many segments to run per host include the following:

- Number of cores
- Amount of physical RAM installed in the server
- Number of NICs

- Amount of storage attached to server
- Mixture of primary and mirror segments
- ETL processes that will run on the hosts
- Non-Greenplum processes running on the hosts

Segment Memory Configuration

The `gp_vmem_protect_limit` server configuration parameter specifies the amount of memory that all active postgres processes for a single segment can consume at any given time. Queries that exceed this amount will fail. Use the following calculations to estimate a safe value for `gp_vmem_protect_limit`.

1. Calculate `gp_vmem`, the host memory available to Greenplum Database, 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.
2. Calculate `max_acting_primary_segments`. This is the maximum number of primary segments that can be running on a host when mirror segments are activated due to a segment or host failure on another host in the cluster. With mirrors arranged in a 4-host block with 8 primary segments per host, for example, a single segment host failure would activate two or three mirror segments on each remaining host in the failed host's block. The `max_acting_primary_segments` value for this configuration is 11 (8 primary segments plus 3 mirrors activated on failure).
3. Calculate `gp_vmem_protect_limit` by dividing the total Greenplum Database memory by the maximum number of acting primaries:

   ```
   gp_vmem_protect_limit = gp_vmem / max_acting_primary_segments
   ```

   Convert to megabytes to find the value to set for the `gp_vmem_protect_limit` system configuration parameter.

For scenarios where a large number of workfiles are generated, adjust the calculation for `gp_vmem` to account for the workfiles:

```
gp_vmem = ((SWAP + RAM) – (7.5GB + 0.05 * RAM - (300KB * total_#_workfiles))) / 1.7
```

For information about monitoring and managing workfile usage, see the *Greenplum Database Administrator Guide*.

You can calculate the value of the `vm.overcommit_ratio` operating system parameter from the value of `gp_vmem`:

```
vm.overcommit_ratio = (RAM - 0.026 * gp_vmem) / RAM
```

See *OS Memory Configuration* for more about about `vm.overcommit_ratio`.

See also *Memory and Workload Management*.

Statement Memory Configuration

The `statement_mem` server configuration parameter is the amount of memory to be allocated to any single query in a segment database. If a statement requires additional memory it will spill to disk. Calculate the value for `statement_mem` with the following formula:

```
(gp_vmem_protect_limit * .9) / max_expected_concurrent_queries
```

For example, for 40 concurrent queries with `gp_vmem_protect_limit` set to 8GB (8192MB), the calculation for `statement_mem` would be:

```
(8192MB * .9) / 40 = 184MB
```

Each query would be allowed 184MB of memory before it must spill to disk.

To increase `statement_mem` safely you must either increase `gp_vmem_protect_limit` or reduce the number of concurrent queries. To increase `gp_vmem_protect_limit`, you must add physical RAM and/or swap space, or reduce the number of segments per host.

Note that adding segment hosts to the cluster cannot help out-of-memory errors unless you use the additional hosts to decrease the number of segments per host.

Explain what spill files are. Then cover `gp_workfile_limit_files_per_query`. Control the maximum number of spill files created per segment per query with the configuration parameter `gp_workfile_limit_files_per_query` . Then explain `gp_workfile_limit_per_segment`.

Also, see *Workload Management* for best practices for managing query memory using resource queues.

Spill File Configuration

Greenplum Database creates *spill files* (also called *workfiles*) on disk if a query is allocated insufficient memory to execute in memory. A single query can create no more than 100,000 spill files, by default, which is sufficient for the majority of queries.

You can control the maximum number of spill files created per query and per segment with the configuration parameter `gp_workfile_limit_files_per_query`. Set the parameter to 0 to allow queries to create an unlimited number of spill files. Limiting the number of spill files permitted prevents run-away queries from disrupting the system.

A query could generate a large number of spill files if not enough memory is allocated to it or if data skew is present in the queried data. 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
```

Before raising the `gp_workfile_limit_files_per_query`, try reducing the number of spill files by changing the query, changing the data distribution, or changing the memory configuration.

The `gp_toolkit` schema includes views that allow you to see information about all the queries that are currently using spill files. This information can be used for troubleshooting and for tuning queries:

- The `gp_workfile_entries` view contains one row for each operator using disk space for workfiles on a segment at the current time. See *How to Read Explain Plans*for information about operators.
- 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.

See the *Greenplum Database Reference Guide* for descriptions of the columns in these views.

The `gp_workfile_compress_algorithm` configuration parameter specifies a compression algorithm to apply to spill files. It can have the value `none` or `zlib`. Setting this parameter to `zlib` can improve performance when spill files are used.

A query could generate a large number of spill files if not enough memory is allocated to it or if data skew is present in the queried data. 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
```

Before raising the `gp_workfile_limit_files_per_query`, try reducing the number of spill files by changing the query, changing the data distribution, or changing the memory configuration.

Chapter 3

Schema Design

This topic contains best practices for designing Greenplum Database schemas.

Greenplum Database is an analytical, shared-nothing database, which is much different than a highly normalized, transactional SMP database. Greenplum Database performs best with a denormalized schema design suited for MPP analytical processing, for example a star or snowflake schema, with large centralized fact tables connected to multiple smaller dimension tables.

Data Types

Use Types Consistently

Use the same data types for columns used in joins between tables. If the data types differ, Greenplum Database must dynamically convert the data type of one of the columns so the data values can be compared correctly. With this in mind, you may need to increase the data type size to facilitate joins to other common objects.

Choose Data Types that Use the Least Space

You can increase database capacity and improve query execution by choosing the most efficient data types to store your data.

Use `TEXT` or `VARCHAR` rather than `CHAR`. There are no performance differences among the character data types, but using `TEXT` or `VARCHAR` can decrease the storage space used.

Use the smallest numeric data type that will accommodate your data. Using `BIGINT` for data that fits in `INT` or `SMALLINT` wastes storage space.

Storage Model

Greenplum Database provides an array of storage options when creating tables. It is very important to know when to use heap storage versus append-optimized (AO) storage, and when to use row-oriented storage versus column-oriented storage. The correct selection of heap versus AO and row versus column is extremely important for large fact tables, but less important for small dimension tables.

The best practices for determining the storage model are:

1. Design and build an insert-only model, truncating a daily partition before load.
2. For large partitioned fact tables, evaluate and use optimal storage options for different partitions. One storage option is not always right for the entire partitioned table. For example, some partitions can be row-oriented while others are column-oriented.
3. When using column-oriented storage, every column is a separate file on *every* Greenplum Database segment. For tables with a large number of columns consider columnar storage for data often accessed (hot) and row-oriented storage for data not often accessed (cold).
4. Storage options should be set at the partition level or at the level where the data is stored.
5. Compress large tables to improve I/O performance and to make space in the cluster, if needed.

Heap Storage or Append-Optimized Storage

Heap storage is the default model, and is the model PostgreSQL uses for all database tables. Use heap storage for tables and partitions that will receive iterative `UPDATE`, `DELETE`, and singleton `INSERT` operations. Use heap storage for tables and partitions that will receive concurrent `UPDATE`, `DELETE`, and `INSERT` operations.

Use append-optimized storage for tables and partitions that are updated infrequently after the initial load with subsequent inserts only performed in batch operations. Never perform singleton `INSERT`, `UPDATE`, or `DELETE` operations on append-optimized tables. Concurrent batch `INSERT` operations are okay but *never* perform concurrent batch `UPDATE` or `DELETE` operations.

Space occupied by rows that are updated and deleted in append-optimized tables is not recovered and reused as efficiently as with heap tables, so the append-optimized storage model is inappropriate for frequently updated tables. It is intended for large tables that are loaded once, updated infrequently, and queried frequently for analytical query processing.

Row or Column Orientation

Row orientation is the traditional way to store database tuples. The columns that comprise a row are stored on disk contiguously, so that an entire row can be read from disk in a single I/O.

Column orientation stores column values together on disk. A separate file is created for each column. If the table is partitioned, a separate file is created for each column and partition. When a query accesses only a small number of columns in a column-oriented table with many columns, the cost of I/O is substantially reduced compared to a row-oriented table; any columns not referenced do not have to be retrieved from disk.

Row-oriented storage is recommended for transactional type workloads with iterative transactions where updates are required and frequent inserts are performed. Use row-oriented storage when selects against the table are wide, where many columns of a single row are needed in a query. If the majority of columns in the `SELECT` list or `WHERE` clause is selected in queries, use row-oriented storage. Use row-oriented storage for general purpose or mixed workloads, as it offers the best combination of flexibility and performance.

Column-oriented storage is optimized for read operations but it is not optimized for write operations; column values for a row must be written to different places on disk. Column-oriented tables can offer optimal query performance on large tables with many columns where only a small subset of columns are accessed by the queries.

Another benefit of column orientation is that a collection of values of the same data type can be stored together in less space than a collection of mixed type values, so column-oriented tables use less disk space (and consequently less disk I/O) than row-oriented tables. Column-oriented tables also compress better than row-oriented tables.

Use column-oriented storage for data warehouse analytic workloads where selects are narrow or aggregations of data are computed over a small number of columns. Use column-oriented storage for tables that have single columns that are regularly updated without modifying other columns in the row. **Reading a complete row in a wide columnar table requires more time than reading the same row from a row-oriented table.** It is important to understand that each column is a separate physical file on *every* segment in Greenplum Database.

Compression

Greenplum Database offers a variety of options to compress append-optimized tables and partitions. Use compression to improve I/O across the system by allowing more data to be read with each disk read operation. The best practice is to set the column compression settings at the level where the data resides.

Note that new partitions added to a partitioned table do not automatically inherit compression defined at the table level; you must *specifically* define compression when you add new partitions.

Delta and RLE compression provide the best levels of compression. Higher levels of compression usually result in more compact storage on disk, but require additional time and CPU cycles when compressing data on writes and uncompressing on reads. Sorting data, in combination with the various compression options, can achieve the highest level of compression.

Data compression should never be used for data that is stored on a compressed file system.

Test different compression types and ordering methods to determine the best compression for your specific data.

Distributions

An optimal distribution that results in evenly distributed data is the most important factor in Greenplum Database. In an MPP shared nothing environment overall response time for a query is measured by the completion time for all segments. The system is only as fast as the slowest segment. If the data is skewed, segments with more data will take more time to complete, so every segment must have an approximately equal number of rows and perform approximately the same amount of processing. Poor performance and out of memory conditions may result if one segment has significantly more data to process than other segments.

Consider the following best practices when deciding on a distribution strategy:

- Explicitly define a column or random distribution for all tables. Do not use the default.
- Ideally, use a single column that will distribute data across all segments evenly.
- Do not distribute on columns that will be used in the `WHERE` clause of a query.
- Do not distribute on dates or timestamps.
- The distribution key column data should contain unique values or very high cardinality.
- If a single column cannot achieve an even distribution, use a multi-column distribution key, but no more than two columns. Additional column values do not typically yield a more even distribution and they require additional time in the hashing process.
- If a two-column distribution key cannot achieve an even distribution of data, use a random distribution. Multi-column distribution keys in most cases require motion operations to join tables, so they offer no advantages over a random distribution.

Greenplum Database random distribution is not round-robin, so there is no guarantee of an equal number of records on each segment. Random distributions typically fall within a target range of less than ten percent variation.

Optimal distributions are critical when joining large tables together. To perform a join, matching rows must be located together on the same segment. If data is not distributed on the same join column, the rows needed from one of the tables are dynamically redistributed to the other segments. In some cases a broadcast motion is performed rather than a redistribution motion.

Local (Co-located) Joins

Using a hash distribution that evenly distributes table rows across all segments and results in local joins can provide substantial performance gains. When joined rows are on the same segment, much of the processing can be accomplished within the segment instance. These are called *local* or *co-located* joins. Local joins minimize data movement; each segment operates independently of the other segments, without network traffic cor communications between segments.

To achieve local joins for large tables commonly joined together, distribute the tables on the same column. Local joins require that both sides of a join be distributed on the same columns (and in the same order) *and* that all columns in the distribution clause are used when joining tables. The distribution columns must also be the same data type—although some values with different data types may appear to have the same representation, they are stored differently and hash to different values, so they are stored on different segments.

Data Skew

Data skew is often the root cause of poor query performance and out of memory conditions. Skewed data affects scan (read) performance, but it also affects all other query execution operations, for instance, joins and group by operations.

It is very important to *validate* distributions to *ensure* that data is evenly distributed after the initial load. It is equally important to *continue* to validate distributions after incremental loads.

The following query shows the number of rows per segment as well as the variance from the minimum and maximum numbers of rows:

```
SELECT 'Example Table' AS "Table Name",
    max(c) AS "Max Seg Rows", min(c) AS "Min Seg Rows",
    (max(c)-min(c))*100.0/max(c) AS "Percentage Difference Between Max & Min"
FROM (SELECT count(*) c, gp_segment_id FROM facts GROUP BY 2) AS a;
```

The `gp_toolkit` schema has two views that you can use to check for skew.

- The `gp_toolkit.gp_skew_coefficients` view shows data distribution skew by calculating the coefficient of variation (CV) for the data stored on each segment. The `skccoeff` column shows the coefficient of variation (CV), which is calculated as the standard deviation divided by the average. It takes into account both the average and variability around the average of a data series. The lower the value, the better. Higher values indicate greater data skew.
- The `gp_toolkit.gp_skew_idle_fractions` view shows data distribution skew by calculating the percentage of the system that is idle during a table scan, which is an indicator of computational skew. The `siffraction` column shows the percentage of the system that is idle during a table scan. This is an indicator of uneven data distribution or query processing skew. For example, a value of 0.1 indicates 10% skew, a value of 0.5 indicates 50% skew, and so on. Tables that have more than10% skew should have their distribution policies evaluated.

Processing Skew

Processing skew results when a disproportionate amount of data flows to, and is processed by, one or a few segments. It is often the culprit behind Greenplum Database performance and stability issues. It can happen with operations such join, sort, aggregation, and various OLAP operations. Processing skew happens in flight while a query is executing and is not as easy to detect as data skew, which is caused by uneven data distribution due to the wrong choice of distribution keys. Data skew is present at the table level, so it can be easily detected and avoided by choosing optimal distribution keys.

If single segments are failing, that is, not all segments on a host, it may be a processing skew issue. Identifying processing skew is currently a manual process. First look for spill files. If there is skew, but not enough to cause spill, it will not become a performance issue. If you determine skew exists, then find the query responsible for the skew. Following are the steps and commands to use. (Change names like the host file name passed to `gpssh` accordingly):

1. Find the OID for the database that is to be monitored for skew processing:

   ```
   SELECT oid, datname FROM pg_database;
   ```

   Example output:

   ```
     oid  |  datname
   -------+-----------
    17088 | gpadmin
    10899 | postgres
        1 | template1
    10898 | template0
    38817 | pws
    39682 | gpperfmon
   (6 rows)
   ```

2. Run a `gpssh` command to check file sizes across all of the segment nodes in the system. Replace `<OID>` with the OID of the database from the prior command:

   ```
   [gpadmin@mdw kend]$ gpssh -f ~/hosts -e \
       "du -b /data[1-2]/primary/gpseg*/base/<OID>/pgsql_tmp/*" | \
       grep -v "du -b" | sort | awk -F" " '{ arr[$1] = arr[$1] + $2 ; tot = tot +
    $2 }; END \
       { for ( i in arr ) print "Segment node" i, arr[i], "bytes (" arr[i]/(1024**3)"
    GB)"; \
   ```

```
    print "Total", tot, "bytes (" tot/(1024**3)" GB)" }' -
```

Example output:

```
Segment node[sdw1] 2443370457 bytes (2.27557 GB)
Segment node[sdw2] 1766575328 bytes (1.64525 GB)
Segment node[sdw3] 1761686551 bytes (1.6407 GB)
Segment node[sdw4] 1780301617 bytes (1.65804 GB)
Segment node[sdw5] 1742543599 bytes (1.62287 GB)
Segment node[sdw6] 1830073754 bytes (1.70439 GB)
Segment node[sdw7] 1767310099 bytes (1.64594 GB)
Segment node[sdw8] 1765105802 bytes (1.64388 GB)
Total 14856967207 bytes (13.8366 GB)
```

If there is a *significant and sustained* difference in disk usage, then the queries being executed should be investigated for possible skew (the example output above does not reveal significant skew). In monitoring systems, there will always be some skew, but often it is *transient* and will be *short in duration*.

3. If significant and sustained skew appears, the next task is to identify the offending query.

   The command in the previous step sums up the entire node. This time, find the actual segment directory. You can do this from the master or by logging into the specific node identified in the previous step. Following is an example run from the master.

   This example looks specifically for sort files. Not all spill files or skew situations are caused by sort files, so you will need to customize the command:

```
$ gpssh -f ~/hosts -e
    "ls -l /data[1-2]/primary/gpseg*/base/19979/pgsql_tmp/*"
    | grep -i sort | sort
```

   Here is output from this command:

```
[sdw1] -rw------- 1 gpadmin gpadmin 1002209280 Jul 29 12:48
        /data1/primary/gpseg2/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_19791_0001.0
[sdw1] -rw------- 1 gpadmin gpadmin 1003356160 Jul 29 12:48
        /data1/primary/gpseg1/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_19789_0001.0
[sdw1] -rw------- 1 gpadmin gpadmin  288718848 Jul 23 14:58
        /data1/primary/gpseg2/base/19979/pgsql_tmp/
pgsql_tmp_slice0_sort_17758_0001.0
[sdw1] -rw------- 1 gpadmin gpadmin  291176448 Jul 23 14:58
        /data2/primary/gpseg5/base/19979/pgsql_tmp/
pgsql_tmp_slice0_sort_17764_0001.0
[sdw1] -rw------- 1 gpadmin gpadmin  988446720 Jul 29 12:48
        /data1/primary/gpseg0/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_19787_0001.0
[sdw1] -rw------- 1 gpadmin gpadmin  995033088 Jul 29 12:48
        /data2/primary/gpseg3/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_19793_0001.0
[sdw1] -rw------- 1 gpadmin gpadmin  997097472 Jul 29 12:48
        /data2/primary/gpseg5/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_19797_0001.0
[sdw1] -rw------- 1 gpadmin gpadmin  997392384 Jul 29 12:48
        /data2/primary/gpseg4/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_19795_0001.0
[sdw2] -rw------- 1 gpadmin gpadmin 1002340352 Jul 29 12:48
        /data2/primary/gpseg11/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_3973_0001.0
[sdw2] -rw------- 1 gpadmin gpadmin 1004339200 Jul 29 12:48
        /data1/primary/gpseg8/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_3967_0001.0
[sdw2] -rw------- 1 gpadmin gpadmin  989036544 Jul 29 12:48
        /data2/primary/gpseg10/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_3971_0001.0
[sdw2] -rw------- 1 gpadmin gpadmin  993722368 Jul 29 12:48
```

```
        /data1/primary/gpseg6/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_3963_0001.0
[sdw2] -rw------- 1 gpadmin gpadmin  998277120 Jul 29 12:48
        /data1/primary/gpseg7/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_3965_0001.0
[sdw2] -rw------- 1 gpadmin gpadmin  999751680 Jul 29 12:48
        /data2/primary/gpseg9/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_3969_0001.0
[sdw3] -rw------- 1 gpadmin gpadmin 1000112128 Jul 29 12:48
        /data1/primary/gpseg13/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_24723_0001.0
[sdw3] -rw------- 1 gpadmin gpadmin 1004797952 Jul 29 12:48
        /data2/primary/gpseg17/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_24731_0001.0
[sdw3] -rw------- 1 gpadmin gpadmin 1004994560 Jul 29 12:48
        /data2/primary/gpseg15/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_24727_0001.0
[sdw3] -rw------- 1 gpadmin gpadmin 1006108672 Jul 29 12:48
        /data1/primary/gpseg14/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_24725_0001.0
[sdw3] -rw------- 1 gpadmin gpadmin  998244352 Jul 29 12:48
        /data1/primary/gpseg12/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_24721_0001.0
[sdw3] -rw------- 1 gpadmin gpadmin  998440960 Jul 29 12:48
        /data2/primary/gpseg16/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_24729_0001.0
[sdw4] -rw------- 1 gpadmin gpadmin 1001029632 Jul 29 12:48
        /data2/primary/gpseg23/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_29435_0001.0
[sdw4] -rw------- 1 gpadmin gpadmin 1002504192 Jul 29 12:48
        /data1/primary/gpseg20/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_29429_0001.0
[sdw4] -rw------- 1 gpadmin gpadmin 1002504192 Jul 29 12:48
        /data2/primary/gpseg21/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_29431_0001.0
[sdw4] -rw------- 1 gpadmin gpadmin 1009451008 Jul 29 12:48
        /data1/primary/gpseg19/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_29427_0001.0
[sdw4] -rw------- 1 gpadmin gpadmin  980582400 Jul 29 12:48
        /data1/primary/gpseg18/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_29425_0001.0
[sdw4] -rw------- 1 gpadmin gpadmin  993230848 Jul 29 12:48
        /data2/primary/gpseg22/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_29433_0001.0
[sdw5] -rw------- 1 gpadmin gpadmin 1000898560 Jul 29 12:48
        /data2/primary/gpseg28/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_28641_0001.0
[sdw5] -rw------- 1 gpadmin gpadmin 1003388928 Jul 29 12:48
        /data2/primary/gpseg29/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_28643_0001.0
[sdw5] -rw------- 1 gpadmin gpadmin 1008566272 Jul 29 12:48
        /data1/primary/gpseg24/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_28633_0001.0
[sdw5] -rw------- 1 gpadmin gpadmin  987332608 Jul 29 12:48
        /data1/primary/gpseg25/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_28635_0001.0
[sdw5] -rw------- 1 gpadmin gpadmin  990543872 Jul 29 12:48
        /data2/primary/gpseg27/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_28639_0001.0
[sdw5] -rw------- 1 gpadmin gpadmin  999620608 Jul 29 12:48
        /data1/primary/gpseg26/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_28637_0001.0
[sdw6] -rw------- 1 gpadmin gpadmin 1002242048 Jul 29 12:48
        /data2/primary/gpseg33/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_29598_0001.0
[sdw6] -rw------- 1 gpadmin gpadmin 1003683840 Jul 29 12:48
        /data1/primary/gpseg31/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_29594_0001.0
[sdw6] -rw------- 1 gpadmin gpadmin 1004732416 Jul 29 12:48
        /data2/primary/gpseg34/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_29600_0001.0
```

```
[sdw6] -rw------- 1 gpadmin gpadmin  986447872 Jul 29 12:48
        /data2/primary/gpseg35/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_29602_0001.0
[sdw6] -rw------- 1 gpadmin gpadmin  990543872 Jul 29 12:48
        /data1/primary/gpseg30/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_29592_0001.0
[sdw6] -rw------- 1 gpadmin gpadmin  992870400 Jul 29 12:48
        /data1/primary/gpseg32/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_29596_0001.0
[sdw7] -rw------- 1 gpadmin gpadmin 1007321088 Jul 29 12:48
        /data2/primary/gpseg39/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_18530_0001.0
[sdw7] -rw------- 1 gpadmin gpadmin 1011187712 Jul 29 12:48
        /data1/primary/gpseg37/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_18526_0001.0
[sdw7] -rw------- 1 gpadmin gpadmin  987332608 Jul 29 12:48
        /data2/primary/gpseg41/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_18534_0001.0
[sdw7] -rw------- 1 gpadmin gpadmin  994344960 Jul 29 12:48
        /data1/primary/gpseg38/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_18528_0001.0
[sdw7] -rw------- 1 gpadmin gpadmin  996114432 Jul 29 12:48
        /data2/primary/gpseg40/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_18532_0001.0
[sdw7] -rw------- 1 gpadmin gpadmin  999194624 Jul 29 12:48
        /data1/primary/gpseg36/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_18524_0001.0
[sdw8] -rw------- 1 gpadmin gpadmin 1002242048 Jul 29 12:48
        /data2/primary/gpseg46/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15675_0001.0
[sdw8] -rw------- 1 gpadmin gpadmin 1003520000 Jul 29 12:48
        /data1/primary/gpseg43/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15669_0001.0
[sdw8] -rw------- 1 gpadmin gpadmin 1008009216 Jul 29 12:48
        /data1/primary/gpseg44/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15671_0001.0
[sdw8] -rw------- 1 gpadmin gpadmin 1073741824 Jul 29 12:16
        /data2/primary/gpseg45/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15673_0001.0
[sdw8] -rw------- 1 gpadmin gpadmin 1073741824 Jul 29 12:21
        /data2/primary/gpseg45/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15673_0002.1
[sdw8] -rw------- 1 gpadmin gpadmin 1073741824 Jul 29 12:24
        /data2/primary/gpseg45/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15673_0003.2
[sdw8] -rw------- 1 gpadmin gpadmin 1073741824 Jul 29 12:26
        /data2/primary/gpseg45/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15673_0004.3
[sdw8] -rw------- 1 gpadmin gpadmin 1073741824 Jul 29 12:31
        /data2/primary/gpseg45/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15673_0006.5
[sdw8] -rw------- 1 gpadmin gpadmin 1073741824 Jul 29 12:32
        /data2/primary/gpseg45/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15673_0005.4
[sdw8] -rw------- 1 gpadmin gpadmin 1073741824 Jul 29 12:34
        /data2/primary/gpseg45/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15673_0007.6
[sdw8] -rw------- 1 gpadmin gpadmin 1073741824 Jul 29 12:36
        /data2/primary/gpseg45/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15673_0008.7
[sdw8] -rw------- 1 gpadmin gpadmin 1073741824 Jul 29 12:43
        /data2/primary/gpseg45/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15673_0009.8
[sdw8] -rw------- 1 gpadmin gpadmin  924581888 Jul 29 12:48
        /data2/primary/gpseg45/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15673_0010.9
[sdw8] -rw------- 1 gpadmin gpadmin  990085120 Jul 29 12:48
        /data1/primary/gpseg42/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15667_0001.0
[sdw8] -rw------- 1 gpadmin gpadmin  996933632 Jul 29 12:48
```

```
      /data2/primary/gpseg47/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15677_0001.0
```

Scanning this output reveals that segment `gpseg45` on host `sdw8` is the culprit.

4. Log in to the offending node with `ssh` and become root. Use the `lsof` command to find the PID for the process that owns one of the sort files:

```
[root@sdw8 ~]# lsof /data2/primary/gpseg45/base/19979/pgsql_tmp/
pgsql_tmp_slice10_sort_15673_0002.1
COMMAND  PID    USER    FD   TYPE DEVICE  SIZE        NODE        NAME
postgres 15673  gpadmin 11u  REG  8,48    1073741824  64424546751 /data2/primary/
gpseg45/base/19979/pgsql_tmp/pgsql_tmp_slice10_sort_15673_0002.1
```

The PID, `15673`, is also part of the file name, but this may not always be the case.

5. Use the `ps` command with the PID to identify the database and connection information:

```
[root@sdw8 ~]# ps -eaf | grep 15673
gpadmin  15673 27471 28 12:05 ?        00:12:59 postgres: port 40003, sbaskin bdw
        172.28.12.250(21813) con699238 seg45 cmd32 slice10 MPPEXEC SELECT
root     29622 29566  0 12:50 pts/16   00:00:00 grep 15673
```

6. On the master, check the `pg_log` log file for the user in the previous command (`sbaskin`), connection (`con699238`, and command (`cmd32`). The line in the log file with these three values *should* be the line that contains the query, but occasionally, the command number may differ slightly. For example, the `ps` output may show `cmd32`, but in the log file it is `cmd34`. If the query is still running, the last query for the user and connection is the offending query.

The remedy for processing skew in almost all cases is to rewrite the query. Creating temporary tables can eliminate skew. Temporary tables can be randomly distributed to force a two-stage aggregation.

Partitioning

A good partitioning strategy reduces the amount of data to be scanned by reading only the partitions needed to satisfy a query.

Each partition is a separate physical file on *every* segment. Just as reading a complete row in a wide columnar table requires more time than reading the same row from a heap table, *reading all partitions in a partitioned table requires more time than reading the same data from a non-partitioned table*.

Following are partitioning best practices:

- Partition large tables only, do not partition small tables.
- Use partitioning on large tables *only* when partition elimination (partition pruning) can be achieved based on query criteria and is accomplished by partitioning the table based on the query predicate. Use range partitioning over list partitioning.
- Choose range partitioning over list partitioning.
- The query planner 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 planner to selectively scan partitioned tables, but a `WHERE` clause such as

  ```
  time > TIMEOFDAY
  ```

  does not. It is important to validate that queries are selectively scanning partitioned tables (partitions are being eliminated) by examining the query `EXPLAIN` plan.
- Do not use default partitions. The default partition is always scanned but, more importantly, in many environments they tend to overfill resulting in poor performance.
- *Never* partition and distribute tables on the same column.
- Do not use multi-level partitioning. While sub-partitioning is supported, it is not recommended because typically subpartitions contain little or no data. It is a myth that performance increases as the number of partitions or subpartitions increases; the administrative overhead of maintaining many partitions and subpartitions will outweigh any performance benefits. For performance, scalability and manageability, balance partition scan performance with the number of overall partitions.
- Beware of using too many partitions with column-oriented storage.
- Consider workload concurrency and the average number of partitions opened and scanned for all concurrent queries.

Number of Partition and Columnar Storage Files

The only hard limit for the number of files Greenplum Database supports is the operating system's open file limit. It is important, however, to consider the total number of files in the cluster, the number of files on every segment, and the total number of files on a host. In an MPP shared nothing environment, every node operates independently of other nodes. Each node is constrained by its disk, CPU, and memory. CPU and I/O constraints are not common with Greenplum Database, but memory is often a limiting factor because the query execution model optimizes query performance in memory.

The optimal number of files per segment also varies based on the number of segments on the node, the size of the cluster, SQL access, concurrency, workload, and skew. There are generally six to eight segments per host, but large clusters should have fewer segments per host. When using partitioning and columnar storage it is important to balance the total number of files in the cluster ,but it is *more* important to consider the number of files per segment and the total number of files on a node.

Example DCA V2 64GB Memory per Node

- Number of nodes: 16
- Number of segments per node: 8
- Average number of files per segment: 10,000

The total number of files per node is `8*10,000 = 80,000` and the total number of files for the cluster is `8*16*10,000 = 1,280,000`. The number of files increases quickly as the number of partitions and the number of columns increase.

As a general best practice, limit the total number of files per node to under 100,000. As the previous example shows, the optimal number of files per segment and total number of files per node depends on the hardware configuration for the nodes (primarily memory), size of the cluster, SQL access, concurrency, workload and skew.

Indexes

Indexes are not generally needed in Greenplum Database. Most analytical queries operate on large volumes of data. In Greenplum Database, a sequential scan is an efficient method to read data as each segment contains an equal portion of the data and all segments work in parallel to read the data.

For queries with high selectivity, indexes may improve query performance. Create an index on a single column of a columnar table for drill through purposes for high cardinality tables that are required for selective queries.

If it is determined that indexes are needed, do not index columns that are frequently updated. Creating an index on a column that is frequently updated increases the number of writes required on updates.

Indexes on expressions should be used only if the expression is used frequently in queries.

An index with a predicate creates a partial index that can be used to select a small number of rows from large tables.

Avoid overlapping indexes. Indexes that have the same leading column are redundant.

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 pages are uncompressed.

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.

If adding an index does not produce performance gains, drop it. Verify that every index you create is used by the optimizer.

Always drop indexes before loading data into a table. The load will run an order of magnitude faster than loading data into a table with indexes. After the load, re-create the indexes.

Bitmap indexes are suited for querying and not updating. Bitmap indexes perform best when the column has a low cardinality—100 to 100,000 distinct values. Do not use bitmap indexes for unique columns, very high, or very low cardinality data. Do not use bitmap indexes for transactional workloads.

In general, do not index partitioned tables. If indexes are needed, the index columns must be different than the partition columns. A benefit of indexing partitioned tables is that because the b-tree performance degrades exponentially as the size of the b-tree grows, creating indexes on partitioned tables creates smaller b-trees that perform better than with non-partitioned tables.

Column Sequence and Byte Alignment

For optimum performance lay out the columns of a table to achieve data type byte alignment. Lay out the columns in heap tables in the following order:

1. Distribution and partition columns
2. Fixed numeric types
3. Variable data types

Lay out the data types from largest to smallest, so that `BIGINT` and `TIMESTAMP` come before `INT` and `DATE`, and all of these types come before `TEXT`, `VARCHAR`, or `NUMERIC(x,y)`. For example, 8-byte types first (`BIGINT`, `TIMESTAMP`), 4-byte types next (`INT`, `DATE`), 2-byte types next (`SMALLINT`), and variable data type last (`VARCHAR`).

Instead of defining columns in this sequence:

`Int`, `Bigint`, `Timestamp`, `Bigint`, `Timestamp`, `Int` (distribution key), `Date` (partition key), `Bigint`, `Smallint`

define the columns in this sequence:

`Int` (distribution key), `Date` (partition key), `Bigint`, `Bigint`, `Timestamp`, `Bigint`, `Timestamp`, `Int`, `Smallint`

Chapter 4

Memory and Workload Management

Memory management has a significant impact on performance in a Greenplum Database cluster. The default settings are suitable for most environments. Do not change the default settings until you understand the memory characteristics and usage on your system.

Resolving Out of Memory Errors

An out of memory error message identifies the Greenplum segment, host, and process that experienced the out of memory error. For example:

```
Out of memory (seg27 host.example.com pid=47093)
VM Protect failed to allocate 4096 bytes, 0 MB available
```

Some common causes of out-of-memory conditions in Greenplum Database are:

- Insufficient system memory (RAM) available on the cluster
- Improperly configured memory parameters
- Data skew at the segment level
- Operational skew at the query level

Following are possible solutions to out of memory conditions:

- Tune the query to require less memory
- Reduce query concurrency using a resource queue
- Decrease the number of segments per host in the Greenplum cluster
- Increase memory on the host
- Validate the `gp_vmem_protect_limit` configuration parameter at the database level. See calculations for the maximum safe setting in *Configuring Memory for Greenplum Database*.
- Use a session setting to reduce the `statement_mem` used by specific queries
- Decrease `statement_mem` at the database level
- Set the memory quota on a resource queue to limit the memory used by queries executed within the resource queue

Adding segment hosts to the cluster will not in itself alleviate out of memory problems. The memory used by each query is determined by the `statement_mem` parameter and it is set when the query is invoked. However, if adding more hosts allows decreasing the number of segments per host, then the amount of memory allocated in `gp_vmem_protect_limit` can be raised.

Configuring Memory for Greenplum Database

Most out-of-memory conditions can be avoided if memory is thoughtfully managed.

It is not always possible to increase system memory, but you can prevent out-of-memory conditions by configuring memory use correctly and setting up resource queues to manage expected workloads.

It is important to include memory requirements for mirror segments that become primary segments during a failure to ensure that the cluster can continue when primary segments or segment hosts fail.

The following are recommended operating system and Greenplum Database memory settings:

- Do not configure the OS to use huge pages.
- **vm.overcommit_memory**

  This is a Linux kernel parameter, set in `/etc/sysctl.conf`. It should aways be set to 2. It determines the method the OS uses for determining how much memory can be allocated to processes and 2 is the only safe setting for Greenplum Database.
- **vm.overcommit_ratio**

  This is a Linux kernel parameter, set in `/etc/sysctl.conf`. It is the percentage of RAM that is used for application processes. The remainder is reserved for the operating system. The default on Red Hat is 50.

  Setting `vm.overcommit_ratio` too high may result in not enough memory being reserved for the operating system, which can result in segment host failure or database failure. Setting the value too low reduces the amount of concurrency and query complexity that can be run by reducing the amount of memory available to Greenplum Database. When increasing the setting it is important to remember to always reserve some memory for operating system activities.

  See *Segment Memory Configuration* for instructions to calculate a value for `vm.overcommit_ratio`.
- **gp_vmem_protect_limit**

  Use `gp_vmem_protect_limit` to set the maximum memory that the instance can allocate for *all* work being done in each segment database. Never set this value larger than the physical RAM on the system. If `gp_vmem_protect_limit` is too high, it is possible for memory to become exhausted on the system and normal operations may fail, causing segment failures. If `gp_vmem_protect_limit` is set to a safe lower value, true memory exhaustion on the system is prevented; queries may fail for hitting the limit, but system disruption and segment failures are avoided, which is the desired behavior.

  See *Segment Memory Configuration* for instructions to calculate a safe value for `gp_vmem_protect_limit`.
- **runaway_detector_activation_percent**

  Runaway Query Termination, introduced in Greenplum Database 4.3.4, prevents out of memory conditions. The `runaway_detector_activation_percent` system parameter controls the percentage of `gp_vmem_protect_limit` memory utilized that triggers termination of queries. It is set on by default at 90%. If the percentage of `gp_vmem_protect_limit memory` that is utilized for a segment exceeds the specified value, Greenplum Database terminates queries based on memory usage, beginning with the query consuming the largest amount of memory. Queries are terminated until the utilized percentage of `gp_vmem_protect_limit` is below the specified percentage.
- **statement_mem**

  Use `statement_mem` to allocate memory used for a query per segment database. If additional memory is required it will spill to disk. Set the optimal value for `statement_mem` as follows:

  ```
  (vmprotect * .9) / max_expected_concurrent_queries
  ```

  The default value of `statement_mem` is 125MB. For example, a query running on a Dell EMC DCA V2 system using the default `statement_mem` value will use 1GB of memory on each segment server (8 segments # 125MB). Set `statement_mem` at the session level for specific queries that require additional memory to complete. This setting works well to manage query memory on clusters with low concurrency. For clusters with high concurrency also use resource queues to provide additional control on what and how much is running on the system.
- **gp_workfile_limit_files_per_query**

  Set `gp_workfile_limit_files_per_query` to limit the maximum number of temporary spill files (workfiles) allowed per query. Spill files are created when a query requires more memory than it is

allocated. When the limit is exceeded the query is terminated. The default is zero, which allows an unlimited number of spill files and may fill up the file system.

- **gp_workfile_compress_algorithm**

  If there are numerous spill files then set `gp_workfile_compress_algorithm` to compress the spill files. Compressing spill files may help to avoid overloading the disk subsystem with IO operations.

Example Memory Configuration Calculations

- Total RAM = 256GB
- SWAP = 64GB
- 8 primary segments and 8 mirror segments per host, in blocks of 4 hosts
- Maximum number of primaries per host during failure is 11

vm.overcommit_ratio calculation

```
gp_vmem = ((SWAP + RAM) – (7.5GB + 0.05 * RAM)) / 1.7
        = ((64 + 256) - (7.5 + 0.05 * 256)) / 1.7
        = 176

vm.overcommit_ratio = (RAM - (0.026 * gp_vmem)) / RAM
                    = (256 - (0.026 * 176)) / 256
                    = .982
```

Set `vm.overcommit_ratio` to 98.

gp_vmem_protect_limit calculation

```
gp_vmem_protect_limit = gp_vmem / maximum_acting_primary_segments
                      = 176 / 11
                      = 16GB
                      = 16384MB
```

Configuring Resource Queues

Greenplum Database resource queues provide a powerful mechanism for managing the workload of the cluster. Queues can be used to limit both the numbers of active queries and the amount of memory that can be used by queries in the queue. When a query is submitted to Greenplum Database, it is added to a resource queue, which determines if the query should be accepted and when the resources are available to execute it.

- Do not use the default queue. Associate all roles with a user-defined resource queue.

  Each login user (role) is associated with a single resource queue; any query the user submits is handled by the associated resource queue. If a queue is not explicitly assigned the user's queries are handed by the default queue, `pg_default`.
- Do not run queries with the gpadmin role or other superuser roles.

  Superusers are exempt from resource queue limits, therefore superuser queries always run regardless of the limits set on their assigned queue.
- Use the `ACTIVE_STATEMENTS` resource queue parameter to limit the number of active queries that members of a particular queue can run concurrently.
- Use the `MEMORY_LIMIT` parameter to control the total amount of memory that queries running through the queue can utilize. By combining the `ACTIVE_STATEMENTS` and `MEMORY_LIMIT` attributes an administrator can fully control the activity emitted from a given resource queue.

  The allocation works as follows: Suppose a resource queue, `sample_queue`, has `ACTIVE_STATEMENTS` set to 10 and `MEMORY_LIMIT` set to 2000MB. This limits the queue to approximately 2 gigabytes of memory per segment. For a cluster with 8 segments per server, the total usage per server of 16 GB for

`sample_queue` (2GB * 8 segments/server). If a segment server has 64GB of RAM, there could be no more than four of this type of resource queue on the system before there is a chance of running out of memory (4 queues * 16GB per queue).

Note that by using `STATEMENT_MEM`, individual queries running in the queue can allocate more than their "share" of memory, thus reducing the memory available for other queries in the queue.

- Resource queue priorities can be used to align workloads with desired outcomes. Queues with `MAX` priority throttle activity in all other queues until the `MAX` queue completes running all queries.
- Alter resource queues dynamically to match the real requirements of the queue for the workload and time of day.

  Typical environments have an operational flow that changes based on the time of day and type of usage of the system. You can script these changes and add crontab entries to execute the scripts.
- Use gptoolkit to view resource queue usage and to understand how the queues are working.

Chapter 5

System Monitoring and Maintenance

This section contains best practices for regular maintenance that will ensure Greenplum Database high availability and optimal performance.

Monitoring

Greenplum Database includes utilities that are useful for monitoring the system.

The `gp_toolkit` schema conatains several views that can be accessed using SQL commands to query system catalogs, log files, and operating environment for system status information.

The `gp_stats_missing` view shows tables that do not have statistics and require `ANALYZE` to be run.

For additional information on `gpstate` and `gpcheckperf` refer to the *Greenplum Database Utility Guide*. For information about the gp_`toolkit` schema, see the *Greenplum Database Reference Guide*.

gpstate

The `gpstate` utility program displays the status of the Greenplum system, including which segments are down, master and segment configuration information (hosts, data directories, etc.), the ports used by the system, and mapping of primary segments to their corresponding mirror segments.

Run `gpstate -Q` to get a list of segments that are marked "down" in the master system catalog.

To get detailed status information for the Greenplum system, run `gpstate -s`.

gpcheckperf

The `gpcheckperf` utility tests baseline hardware performance for a list of hosts. The results can help identify hardware issues. It performs the following checks:

- disk I/O test – measures I/O performance by writing and reading a large file using the `dd` operating system command. It reports read and write rates in megabytes per second.
- memory bandwidth test – measures sustainable memory bandwidth in megabytes per second using the STREAM benchmark.
- network performance test – runs the `gpnetbench` network benchmark program (optionally `netperf)` to test network performance. The test is run in one of three modes: parallel pair test (`-r N`), serial pair test (`-r n`), or full-matrix test (`-r M)`.The minimum, maximum, average, and median transfer rates are reported in megabytes per second.

To obtain valid numbers from `gpcheckperf`, the database system must be stopped. The numbers from `gpcheckperf` can be inaccurate even if the system is up and running with no query activity.

`gpcheckperf` requires a trusted host setup between the hosts involved in the performance test. It calls `gpssh` and `gpscp`, so these utilities must also be in your `PATH`. Specify the hosts to check individually (`-h` *`host1`* `-h` *`host2`* `...`) or with `-f` *`hosts_file`*, where *`hosts_file`* is a text file containing a list of the hosts to check. If you have more than one subnet, create a separate host file for each subnet so that you can test the subnets separately.

By default, `gpcheckperf` runs the disk I/O test, the memory test, and a serial pair network performance test. With the disk I/O test, you must use the `-d` option to specify the file systems you want to test. The following command tests disk I/O and memory bandwidth on hosts listed in the `subnet_1_hosts` file:

```
$ gpcheckperf -f subnet_1_hosts -d /data1 -d /data2 -r ds
```

The `-r` option selects the tests to run: disk I/O (`d`), memory bandwidth (`s`), network parallel pair (`N`), network serial pair test (`n`), network full-matrix test (`M`). Only one network mode can be selected per execution. See the *Greenplum Database Reference Guide* for the detailed `gpcheckperf` reference.

Monitoring with Operating System Utilities

The following Linux/UNIX utilities can be used to assess host performance:

- `iostat` allows you to monitor disk activity on segment hosts.
- `top` displays a dynamic view of operating system processes.
- `vmstat` displays memory usage statistics.

You can use `gpssh` to run utilities on multiple hosts.

Best Practices

- Implement the "Recommended Monitoring and Maintenance Tasks" in the *Greenplum Database Administrator Guide*.
- Run `gpcheckperf` at install time and periodically thereafter, saving the output to compare system performance over time.
- Use all the tools at your disposal to understand how your system behaves under different loads.
- Examine any unusual event to determine the cause.
- Monitor query activity on the system by running explain plans periodically to ensure the queries are running optimally.
- Review plans to determine whether index are being used and partition elimination is occurring as expected.

Additional Information

- `gpcheckperf` reference in the *Greenplum Database Utility Guide*.
- "Recommended Monitoring and Maintenance Tasks" in the *Greenplum Database Administrator Guide*.
- *Sustainable Memory Bandwidth in Current High Performance Computers*. John D. McCalpin. Oct 12, 1995.
- *www.netperf.org* to use `netperf`, netperf must be installed on each host you test. See `gpcheckperf` reference for more information.

Updating Statistics with ANALYZE

The most important prerequisite for good query performance is to begin with accurate statistics for the tables. Updating stastistics with the `ANALYZE` statement enables the query planner to generate optimal query plans. When a table is analyzed, information about the data is stored in the system catalog tables. If the stored information is out of date, the planner can generate inefficient plans.

Generating Statistics Selectively

Running `ANALYZE` with no arguments updates statistics for all tables in the database. This can be a very long-running process and it is not recommended. You should `ANALYZE` tables selectively when data has changed.

Running `ANALYZE` on a large table can take a long time. If it is not feasible to run `ANALYZE` on all columns of a very large table, you can generate statistics for selected columns only using `ANALYZE table(column, ...)`. Be sure to include columns used in joins, `WHERE` clauses, `SORT` clauses, `GROUP BY` clauses, or `HAVING` clauses.

For a partitioned table, you can run `ANALYZE` on just partitions that have changed, for example, if you add a new partition. Note that for partitioned tables, you can run `ANALYZE` on the parent (main) table, or on the leaf nodes—the partition files where data and statistics are actually stored. The intermediate files for sub-partitioned tables store no data or statistics, so running `ANALYZE` on them does not work. You can find the names of the partition tables in the `pg_partitions` system catalog:

```
SELECT partitiontablename from pg_partitions WHERE tablename='parent_table;
```

Improving Statistics Quality

There is a trade-off between the amount of time it takes to generate statistics and the quality, or accuracy, of the statistics.

To allow large tables to be analyzed in a reasonable amount of time, `ANALYZE` takes a random sample of the table contents, rather than examining every row. To increase sampling for all table columns adjust the `default_statistics_target` configuration parameter. The target value ranges from 1 to 1000; the default target value is 25. The `default_statistics_target` variable applies to all columns by default. A larger target value increases the time needed to perform the `ANALYZE`, but may improve the quality of the query planner's estimates. This is especially true for columns with irregular data patterns. `default_statistics_target` can be set at the master/session level and requires a reload.

The `gp_analyze_relative_error` configuration parameter affects the sampling rate during statistics collection to determine cardinality in a column. For example, a value of .5 is equivalent to an acceptable error of 50%. The default is .25. Use the `gp_analyze_relative_error` parameter to set the acceptable estimated relative error in the cardinality of a table. If statistics do not produce good estimates of cardinality for a particular table attribute, decreasing the relative error fraction (accepting less errors) tells the system to sample more rows. However, it is not recommended to reduce this below 0.1 as it will increase `ANALYZE` time substantially.

When to Run ANALYZE

Run `ANALYZE`:

- after loading data,
- after `CREATE INDEX` operations,
- and after `INSERT`, `UPDATE`, and `DELETE` operations that significantly change the underlying data.

`ANALYZE` requires only a read lock on the table, so it may be run in parallel with other database activity, but do not run `ANALYZE` while performing loads, `INSERT`, `UPDATE`, `DELETE`, and `CREATE INDEX` operations.

Configuring Automatic Statistics Collection

The `gp_autostats_mode` configuration parameter, together with the `gp_autostats_on_change_threshold` parameter, determines when an automatic analyze operation is triggered. When automatic statistics collection is triggered, the planner adds an `ANALYZE` step to the query.

By default, `gp_autostats_mode` is `on_no_stats`, which triggers statistics collection for `CREATE TABLE AS SELECT`, `INSERT`, or `COPY` operations on any table that has no existing statistics.

Setting `gp_autostats_mode` to `on_change` triggers statistics collection only when the number of rows affected exceeds the threshold defined by `gp_autostats_on_change_threshold`, which has a default value of 2147483647. Operations that can trigger automatic statistics collection with `on_change` are: `CREATE TABLE AS SELECT`, `UPDATE`, `DELETE`, `INSERT`, and `COPY`.

Setting `gp_autostats_mode` to `none` disables automatics statistics collection.

For partitioned tables, automatic statistics collection is not triggered if data is inserted from the top-level parent table of a partitioned table. But automatic statistics collection *is* triggered if data is inserted directly in a leaf table (where the data is stored) of the partitioned table.

Managing Bloat in the Database

Greenplum Database heap tables use the PostgreSQL Multiversion Concurrency Control (MVCC) storage implementation. A deleted or updated row is logically deleted from the database, but a non-visible image of the row remains in the table. These deleted rows, also called expired rows, are tracked in a free space map. Running `VACUUM` marks the expired rows as free space that is available for reuse by subsequent inserts.

If the free space map is not large enough to accommodate all of the expired rows, the `VACUUM` command is unable to reclaim space for expired rows that overflowed the free space map. The disk space may only be recovered by running `VACUUM FULL`, which locks the table, copies rows one-by-one to the beginning of the file, and truncates the file. This is an expensive operation that can take an exceptional amount of time to complete with a large table. It should be used only on smaller tables. If you attempt to kill a `VACUUM FULL` operation, the system can be disrupted.

> **Important:**
>
> It is very important to run `VACUUM` after large `UPDATE` and `DELETE` operations to avoid the necessity of ever running `VACUUM FULL`.

If the free space map overflows and it is necessary to recover the space it is recommended to use the `CREATE TABLE...AS SELECT` command to copy the table to a new table, which will create a new compact table. Then drop the original table and rename the copied table.

It is normal for tables that have frequent updates to have a small or moderate amount of expired rows and free space that will be reused as new data is added. But when the table is allowed to grow so large that active data occupies just a small fraction of the space, the table has become significantly "bloated." Bloated tables require more disk storage and additional I/O that can slow down query execution.

Bloat affects heap tables, system catalogs, and indexes.

Running the `VACUUM` statement on tables regularly prevents them from growing too large. If the table does become significantly bloated, the `VACUUM FULL` statement (or an alternative procedure) must be used to compact the file. If a large table becomes significantly bloated, it is better to use one of the alternative methods described in *Removing Bloat from Database Tables* to remove the bloat.

> **Caution: Never** run `VACUUM FULL <database_name>` and do not run `VACUUM FULL` on large tables in a Greenplum Database.

Sizing the Free Space Map

Expired rows in heap tables are added to a shared free space map when you run `VACUUM`. The free space map must be adequately sized to accommodate these rows. If the free space map is not large enough, any space occupied by rows that overflow the free space map cannot be reclaimed by a regular `VACUUM` statement. You will have to use `VACUUM FULL` or an alternative method to recover the space.

You can avoid overflowing the free space map by running the `VACUUM` statement regularly. The more bloated a table becomes, the more rows that must be tracked in the free space map. For very large databases with many objects, you may need to increase the size of the free space map to prevent overflow.

The `max_fsm_pages` configuration parameter sets the maximum number of disk pages for which free space will be tracked in the shared free-space map. Each page slot consumes six bytes of shared memory. The default value for `max_fsm_pages` is 200,000.

The `max_fsm_relations` configuration parameter sets the maximum number of relations for which free space will be tracked in the shared memory free-space map. It should be set to a value larger than the total number of tables, indexes, and system tables in the database. It costs about 60 bytes of memory for each relation per segment instance. The default value is 1000.

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

Detecting Bloat

The statistics collected by the `ANALYZE` statement can be used to calculate the expected number of disk pages required to store a table. The difference between the expected number of pages and the actual number of pages is a measure of bloat. The `gp_toolkit` schema provides a `gp_bloat_diag` view that identifies table bloat by comparing the ratio of expected to actual pages. To use it, make sure statistics are up to date for all of the tables in the database, then run the following SQL:

```
gpadmin=# SELECT * FROM gp_toolkit.gp_bloat_diag;
 bdirelid | bdinspname | bdirelname | bdirelpages | bdiexppages |
 bdidiag
----------+------------+------------+-------------+-------------
+----------------------------------------
    21488 | public     | t1         |          97 |           1 | significant amount
 of bloat suspected
(1 row)
```

The results include only tables with moderate or significant bloat. Moderate bloat is reported when the ratio of actual to expected pages is greater than four and less than ten. Significant bloat is reported when the ratio is greater than ten.

The `gp_toolkit.gp_bloat_expected_pages` view lists the actual number of used pages and expected number of used pages for each database object.

```
gpadmin=# SELECT * FROM gp_toolkit.gp_bloat_expected_pages LIMIT 5;
 btdrelid | btdrelpages | btdexppages
----------+-------------+-------------
    10789 |           1 |           1
    10794 |           1 |           1
    10799 |           1 |           1
     5004 |           1 |           1
     7175 |           1 |           1
(5 rows)
```

The `btdrelid` is the object ID of the table. The `btdrelpages` column reports the number of pages the table uses; the `btdexppages` column is the number of pages expected. Again, the numbers reported are based on the table statistics, so be sure to run `ANALYZE` on tables that have changed.

Removing Bloat from Database Tables

The `VACUUM` command adds expired rows to the shared free space map so that the space can be reused. When `VACUUM` is run regularly on a table that is frequently updated, the space occupied by the expired rows can be promptly reused, preventing the table file from growing larger. It is also important to run `VACUUM` before the free space map is filled. For heavily updated tables, you may need to run `VACUUM` at least once a day to prevent the table from becoming bloated.

> **Warning:** When a table is significantly bloated, it is better to run `ANALYZE` *before* running `VACUUM`. Because `ANALYZE` uses block-level sampling, a table with a high ratio of blocks containing no valid rows can cause `ANALYZE` to set the `reltuples` column of the `pg_class` system catalog to an inaccurate value or 0, which can lead to poorly optimized queries. The `VACUUM` command produces a more accurate count and when run after `ANALYZE` will correct an inaccurate row count estimate.

When a table accumulates significant bloat, running the `VACUUM` command is insufficient. For small tables, running `VACUUM FULL <table_name>` can reclaim space used by rows that overflowed the free space map and reduce the size of the table file. However, a `VACUUM FULL` statement is an expensive operation that requires an `ACCESS EXCLUSIVE` lock and may take an exceptionally long and unpredictable amount of time to finish. Rather than run `VACUUM FULL` on a large table, an alternative method is required to remove bloat from a large file. Note that every method for removing bloat from large tables is resource intensive and should be done only under extreme circumstances.

The first method to remove bloat from a large table is to create a copy of the table excluding the expired rows, drop the original table, and rename the copy. This method uses the `CREATE TABLE <table_name> AS SELECT` statement to create the new table, for example:

```
gpadmin=# CREATE TABLE mytable_tmp AS SELECT * FROM mytable;
gpadmin=# DROP TABLE mytable;
gpadmin=# ALTER TABLE mytabe_tmp RENAME TO mytable;
```

A second way to remove bloat from a table is to redistribute the table, which rebuilds the table without the expired rows. Follow these steps:

1. Make a note of the table's distribution columns.
2. Change the table's distribution policy to random:

   ```
   ALTER TABLE mytable SET WITH (REORGANIZE=false)
   DISTRIBUTED randomly;
   ```

   This changes the distribution policy for the table, but does not move any data. The command should complete instantly.
3. Change the distribution policy back to its initial setting:

   ```
   ALTER TABLE mytable SET WITH (REORGANIZE=true)
   DISTRIBUTED BY (<original distribution columns>);
   ```

   This step redistributes the data. Since the table was previously distributed with the same distribution key, the rows are simply rewritten on the same segment, excluding expired rows.

Removing Bloat from Indexes

The `VACUUM` command only recovers space from tables. To recover the space from indexes, recreate them using the `REINDEX` command.

To rebuild all indexes on a table run `REINDEX table_name;`. To rebuild a particular index, run `REINDEX index_name;`. `REINDEX` does not update the `reltuples` and `relpages` statistics for the index, so it is important to `ANALYZE` the table to update these statistics after reindexing.

Removing Bloat from System Catalogs

Greenplum Database system catalogs are also heap tables and can become bloated over time. As database objects are created, altered, or dropped, expired rows are left in the system catalogs. Using `gpload` to load data contributes to the bloat since `gpload` creates and drops external tables. (Rather than use `gpload`, it is recommended to use `gpfdist` to load data.)

Bloat in the system catalogs increases the time require to scan the tables, for example, when creating explain plans. System catalogs are scanned frequently and if they become bloated, overall system performance is degraded.

It is recommended to run `VACUUM` on the system catalog nightly and at least weekly. At the same time, running `REINDEX SYSTEM` removes bloat from the indexes. Alternatively, you can reindex system tables using the `reindexdb` utility with the `-s` (`--system`) option. After reindexing, it is also important to run `ANALYZE`, because the `REINDEX` command rebuilds indexes with empty statistics.

The following script runs `VACUUM`, `REINDEX`, and `ANALYZE` on the system catalogs.

```
#!/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 -s -d $DBNAME
analyzedb -s pg_catalog -d $DBNAME
```

If the system catalogs become significantly bloated, you must perform an intensive system catalog maintenance procedure. The `CREATE TABLE AS SELECT` and redistribution key methods for removing bloat cannot be used with system catalogs. You must instead run `VACUUM FULL` during a scheduled downtime period. During this period, stop all catalog activity on the system; `VACUUM FULL` takes exclusive locks against the system catalog. Running `VACUUM` regularly can prevent the need for this more costly procedure.

Removing Bloat from Append-Optimized Tables

Append-optimized tables are handled much differently than heap tables. Although append-optimized tables allow updates, inserts, and deletes, they are not optimized for these operations and it is recommended to not use them with append-optimized tables. If you heed this advice and use append-optimized for *load-once/read-many* workloads, `VACUUM` on an append-optimized table runs almost instantaneously.

If you do run `UPDATE` or `DELETE` commands on an append-optimized table, expired rows are tracked in an auxiliary bitmap instead of the free space map. `VACUUM` is the only way to recover the space. Running `VACUUM` on an append-optimized table with expired rows compacts a table by rewriting the entire table without the expired rows. However, no action is performed if the percentage of expired rows in the table exceeds the value of the `gp_appendonly_compaction_threshold` configuration parameter, which is 10 (10%) by default. The threshold is checked on each segment, so it is possible that a `VACUUM` statement will compact an append-only table on some segments and not others. Compacting append-only tables can be disabled by setting the `gp_appendonly_compaction` parameter to `no`.

Monitoring Greenplum Database Log Files

Know the location and content of system log files and monitor them on a regular basis and not just when problems arise.

The following table shows the locations of the various Greenplum Database log files. In file paths, *date* is a date in the format YYYYMMDD, *instance* is the current instance name, and *n* is the segment number.

Path	Description
/var/gpadmin/gpadminlogs/*	Many different types of log files, directory on each server
/var/gpadmin/gpadminlogs/gpstart_*date*.log	start log
/var/gpadmin/gpadminlogs/gpstop_*date*.log	stop log
/var/gpadmin/gpadminlogs/gpsegstart.py_idb*gpadmin_*date*.log	segment start log
/var/gpadmin/gpadminlogs/gpsegstop.py_idb*gpadmin_*date*.log	segment stop log
/var/gpdb/*instance*/datamaster/gpseg-1/pg_log/startup.log	instance start log
/var/gpdb/*instance*/datamaster/gpseg-1/gpperfmon/logs/gpmon.*.log	gpperfmon logs
/var/gpdb/*instance*/datamirror/gpseg*n*/pg_log/*.csv	mirror segment logs
/var/gpdb/*instance*/dataprimary/gpseg*n*/pg_log/*.csv	primary segment logs
/var/log/messages	Global Linux system messages

Use `gplogfilter -t` (`--trouble`) first to search the master log for messages beginning with `ERROR:`, `FATAL:`, or `PANIC:`. Messages beginning with `WARNING` may also provide useful information.

To search log files on the segment hosts, use the Greenplum `gplogfilter` utility with `gpssh` to connect to segment hosts from the master host. You can identify corresponding log entries in segment logs by the `statement_id`.

The `log_rotation_age` configuration parameter specifies when a new log file is automatically created while a database instance is running. By default, a new log file is created every day.

Chapter 6

Loading Data

There are several ways to add data to Greenplum Database, each with its appropriate uses.

INSERT Statement with Column Values

A singleton `INSERT` statement with values adds a single row to a table. The row flows through the master and is distributed to a segment. This is the slowest method and is not suitable for loading large amounts of data.

COPY Statement

The PostgreSQL `COPY` statement copies data from an external file into a database table. It can insert multiple rows more efficiently than an `INSERT` statement, but the rows are still passed through the master. All of the data is copied in one command; it is not a parallel process.

Data input to the `COPY` command is from a file or the standard input. For example:

```
COPY table FROM '/data/mydata.csv' WITH CSV HEADER;
```

Use `COPY` to add relatively small sets of data, for example dimension tables with up to ten thousand rows, or one-time data loads.

Use `COPY` when scripting a process that loads small amounts of data, less than 10 thousand rows.

Since COPY is a single command, there is no need to disable autocommit when you use this method to populate a table.

You can run multiple concurrent `COPY` commands to improve performance.

External Tables

External tables provide access to data in sources outside of Greenplum Database. They can be accessed with `SELECT` statements and are commonly used with the Extract, Load, Transform (ELT) pattern, a variant of the Extract, Transform, Load (ETL) pattern that takes advantage of Greenplum Database's fast parallel data loading capability.

With ETL, data is extracted from its source, transformed outside of the database using external transformation tools, such as Informatica or Datastage, and then loaded into the database.

With ELT, Greenplum external tables provide access to data in external sources, which could be read-only files (for example, text, CSV, or XML files), Web servers, Hadoop file systems, executable OS programs, or the Greenplum `gpfdist` file server, described in the next section. External tables support SQL operations such as select, sort, and join so the data can be loaded and transformed simultaneously, or loaded into a *load table* and transformed in the database into target tables.

The external table is defined with a `CREATE EXTERNAL TABLE` statement, which has a `LOCATION` clause to define the location of the data and a `FORMAT` clause to define the formatting of the source data so that the system can parse the input data. Files use the `file://` protocol, and must reside on a segment host in a location accessible by the Greenplum super user. The data can be spread out among the segment hosts with no more than one file per primary segment on each host. The number of files listed in the `LOCATION` clause is the number of segments that will read the external table in parallel.

External Tables with Gpfdist

The fastest way to load large fact tables is to use external tables with `gpdist`. `gpfdist` is a file server program using an HTTP protocol that serves external data files to Greenplum Database segments in parallel. A `gpfdist` instance can serve 200 MB/second and many `gpfdist` processes can run simultaneously, each serving up a portion of the data to be loaded. When you begin the load using a statement such as `INSERT INTO <table> SELECT * FROM <external_table>`, the `INSERT` statement is parsed by the master and distributed to the primary segments. The segments connect to the `gpfdist` servers and retrieve the data in parallel, parse and validate the data, calculate a hash from the distribution key data and, based on the hash key, send the row to its destination segment. By default, each `gpfdist` instance will accept up to 64 connections from segments. With many segments and `gpfdist` servers participating in the load, data can be loaded at very high rates.

Primary segments access external files in parallel when using `gpfdist` up to the value of `gp_external_max_segments`. When optimizing `gpfdist` performance, maximize the parallelism as the number of segments increase. Spread the data evenly across as many ETL nodes as possible. Split very large data files into equal parts and spread the data across as many file systems as possible.

Run two `gpfdist` instances per file system. `gpfdist` tends to be CPU bound on the segment nodes when loading. But if, for example, there are eight racks of segment nodes, there is lot of available CPU on the segments to drive more `gpfdist` processes. Run `gpfdist` on as many interfaces as possible. Be aware of bonded NICs and be sure to start enough `gpfdist` instances to work them.

It is important to keep the work even across all these resources. The load is as fast as the slowest node. Skew in the load file layout will cause the overall load to bottleneck on that resource.

The `gp_external_max_segs` configuration parameter controls the number of segments each `gpfdist` process serves. The default is 64. You can set a different value in the `postgresql.conf` configuration file on the master. Always keep `gp_external_max_segs` and the number of `gpfdist` processes an even factor; that is, the `gp_external_max_segs` value should be a multiple of the number of `gpfdist` processes. For example, if there are 12 segments and 4 `gpfdist` processes, the planner round robins the segment connections as follows:

```
Segment 1  - gpfdist 1
Segment 2  - gpfdist 2
Segment 3  - gpfdist 3
Segment 4  - gpfdist 4
Segment 5  - gpfdist 1
Segment 6  - gpfdist 2
Segment 7  - gpfdist 3
Segment 8  - gpfdist 4
Segment 9  - gpfdist 1
Segment 10 - gpfdist 2
Segment 11 - gpfdist 3
Segment 12 - gpfdist 4
```

Drop indexes before loading into existing tables and re-create the index after loading. Creating an index on pre-existing data is faster than updating it incrementally as each row is loaded.

Run `ANALYZE` on the table after loading. Disable automatic statistics collection during loading by setting `gp_autostats_mode` to `NONE`. Run `VACUUM` after load errors to recover space.

Performing small, high frequency data loads into heavily partitioned column-oriented tables can have a high impact on the system because of the number of physical files accessed per time interval.

Gploload

`gpload` is a data loading utility that acts as an interface to the Greenplum external table parallel loading feature.

Beware of using `gpload` as it can cause catalog bloat by creating and dropping external tables. Use `gpfdist` instead, since it provides the best performance.

`gpload` executes a load using a specification defined in a YAML-formatted control file. It performs the following operations:

- Invokes `gpfdist` processes
- Creates a temporary external table definition based on the source data defined
- Executes an `INSERT`, `UPDATE`, or `MERGE` operation to load the source data into the target table in the database
- Drops the temporary external table
- Cleans up `gpfdist` processes

The load is accomplished in a single transaction.

Best Practices

- Drop any indexes on an existing table before loading data and recreate the indexes after loading. Newly creating an index is faster than updating an index incrementally as each row is loaded.
- Disable automatic statistics collection during loading by setting the `gp_autostats_mode` configuration parameter to `NONE`.
- External tables are not intended for frequent or ad hoc access.
- External tables have no statistics to inform the optimizer. You can set rough estimates for the number of rows and disk pages for the external table in the `pg_class` system catalog with a statement like the following:

  ```
  UPDATE pg_class SET reltuples=400000, relpages=400
  WHERE relname='myexttable';
  ```

- When using `gpfdist`, maximize network bandwidth by running one `gpfdist` instance for each NIC on the ETL server. Divide the source data evenly between the `gpfdist` instances.
- When using `gpload`, run as many simultaneous `gpload` instances as resources allow. Take advantage of the CPU, memory, and networking resources available to increase the amount of data that can be transferred from ETL servers to the Greenplum Database.
- Use the `SEGMENT REJECT LIMIT` clause of the `COPY FROM` statement to set a limit for the number of rows or percentage of rows that can have errors before the `COPY FROM` command is aborted. The reject limit is per segment; when any one segment exceeds the limit, the command is aborted and no rows are added.
- Use the `LOG ERRORS` clause of the `COPY FROM` statement to save error rows. If a row has errors in the formatting—for example missing or extra values, or incorrect data types—Greenplum Database stores the error information and row internally and the load continues. Use the `gp_read_error_log()` built-in SQL function to access this stored information.

  Note: Specifying an error table for error rows with `LOG ERRORS INTO` *`error_table`* is deprecated and will be unsupported in future releases.
- If the load has errors, run `VACUUM` on the table to recover space.
- After you load data into a table, run `VACUUM` on heap tables, including system catalogs, and `ANALYZE` on all tables. It is not necessary to run `VACUUM` on append-optimized tables. If the table is partitioned, you can vacuum and analyze just the partitions affected by the data load. These steps clean up any rows from aborted loads, deletes, or updates and update statistics for the table.
- Recheck for segment skew in the table after loading a large amount of data. You can use a query like the following to check for skew:

  ```
  SELECT gp_segment_id, count(*)
  FROM schema.table
  GROUP BY gp_segment_id ORDER BY 2;
  ```

- By default, `gpfdist` assumes a maximum record size of 32K. To load data records larger than 32K, you must increase the maximum row size parameter by specifying the `-m <bytes>` option on the `gpfdist` command line. If you use `gpload`, set the `MAX_LINE_LENGTH` parameter in the `gpload` control file.

  Note: Integrations with Informatica Power Exchange are currently limited to the default 32K record length.

Additional Information

See the *Greenplum Database Reference Guide* for detailed instructions for loading data using `gpfdist` and `gpload`.

Chapter 7

Migrating Data with Gptransfer

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. `gptransfer` moves data 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 `gptransfer` 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.

What gptransfer Does

`gptransfer` uses writable 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 writable external table. Segments in the destination cluster select from a readable external table and insert into the destination database table. The writable 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 writable 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 writable 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

Prerequisites

- The `gptransfer` utility can only be used with Greenplum Database, including the Dell 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.

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 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 writable external web 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.

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.

Full Mode and Table Mode

When run with the `--full` option, `gptransfer` copies all tables, views, indexes, roles, user-defined functions, and resource queues in the source database to the destination database. Databases to be transferred must not already exist on the destination cluster. If `gptransfer` finds the database on the destination it fails with a message like the following:

```
[ERROR]:- gptransfer: error: --full option specified but tables exist on destination
 system
```

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
Functions	Yes	No
Resource Queues	Yes	No

Object	Full Mode	Table Mode
`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 a database in phases, for example, during scheduled periods of downtime or low activity. Run `gptransfer --full --schema-only ...` to create the full database schema on the destination cluster, but with no data. You can then 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

`gptransfer` creates a configuration that allows transferring large amounts of data at very high rates. For small or empty tables, however, the `gptransfer` set up and clean up are too expensive. The best practice is to use `gptransfer` for large tables and to use other methods for copying smaller tables.

1. Before you begin to transfer data, replicate the schema or schemas from the source cluster to the destination cluster. Options for copying the schema include:
    - Use the `gpsd` (Greenplum Statistics Dump) support utility. This method includes statistics, so be sure to run `ANALYZE` after creating the schema on the destination cluster.
    - Use the PostgreSQL `pg_dump` or `pg_dumpall` utility with the `-schema-only` option.
    - DDL scripts, or any other method for recreating schema in the destination database.

2. Divide the non-empty tables to be transferred into large and small categories, using criteria of your own choice. For example, you could decide large tables have more than one million rows or a raw data size greater than 1GB.
3. Transfer data for small tables using the SQL `COPY` command. This eliminates the warm-up/cool-down time each table incurs when using the `gptransfer` utility.
    - Optionally, write or reuse existing shell scripts to loop through a list of table names to copy with the `COPY` command.
4. Use `gptransfer` to transfer the large table data in batches of tables.
    - It is best to transfer to the same size cluster or to a larger cluster so that `gptransfer` runs in fast mode.
    - If any indexes exist, drop them before starting the transfer process.
    - Use the `gptransfer` table (`-t`) or file (`-f`) options to execute the migration in batches of tables. Do not run `gptransfer` using the full mode; the schema and smaller tables have already been transferred.
    - Perform test runs of the `gptransfer` process before the production migration. This ensures tables can be transferred successfully. You can experiment with the `--batch-size` and `--sub-batch-size` options to obtain maximum parallelism. Determine proper batching of tables for iterative `gptransfer` runs.
    - Include the `--skip-existing` option because the schema already exists on the destination cluster.
    - Use only fully qualified table names. Be aware that periods (.), whitespace, quotes (') and double quotes (") in table names may cause problems.
    - If you decide to use the `--validation` option to validate the data after transfer, be sure to also use the `-x` option to place an exclusive lock on the source table.
5. After all tables are transferred, perform the following tasks:
    - Check for and correct any failed transfers.
    - Recreate the indexes that were dropped before the transfer.
    - Ensure any roles, functions, and resource queues are created in the destination database. These objects are not transferred when you use the `gptransfer -t` option.
    - Copy the `postgres.conf` and `pg_hba.conf` configuration files from the source to the destination cluster.
    - Install needed extensions in the destination database with `gppkg`.

Chapter 8

Security

This section describes basic security best practices that Pivotal recommends you follow to ensure the highest level of system security.

Security Best Practices

- 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. The `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. The 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 the `gpadmin` user id. This `gpadmin` user can bypass all security features of Greenplum Database. Anyone who logs on to a Greenplum host with 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 allow essential system administrators access to it. 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 who logs in. For logging and auditing purposes, each user who 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)" in the *Greenplum Database Administrator Guide*.
- Use groups to manage access privileges. See "Creating Groups (Role Membership)" in the *Greenplum Database Administrator Guide*.
- 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" in the *Greenplum Database Administrator Guide*.

Password Strength Guidelines

To protect the network from intrusion, system administrators should verify the passwords used within an organization are strong ones. The following recommendations can strengthen a password:

- Minimum password length recommendation: At least 9 characters. MD5 passwords should be 15 characters or longer.
- Mix upper and lower case letters.
- Mix letters and numbers.
- Include non-alphanumeric characters.
- Pick a password you can remember.

The following are recommendations for password cracker software that you can use to determine the strength of a password.

- John The Ripper. A fast and flexible password cracking program. It allows the use of multiple word lists and is capable of brute-force password cracking. It is available online at *http://www.openwall.com/john/*.
- Crack. Perhaps the most well-known password cracking software, Crack is also very fast, though not as easy to use as John The Ripper. It can be found online at *http://www.crypticide.com/alecm/security/crack/c50-faq.html*.

The security of the entire system depends on the strength of the root password. This password should be at least 12 characters long and include a mix of capitalized letters, lowercase letters, special characters, and numbers. It should not be based on any dictionary word.

Password expiration parameters should be configured.

Ensure the following line exists within the file `/etc/libuser.conf` under the `[import]` section.

```
login_defs = /etc/login.defs
```

Ensure no lines in the `[userdefaults]` section begin with the following text, as these words override settings from `/etc/login.defs`:

- `LU_SHADOWMAX`
- `LU_SHADOWMIN`
- `LU_SHADOWWARNING`

Ensure the following command produces no output. Any accounts listed by running this command should be locked.

```
grep "^+:" /etc/passwd /etc/shadow /etc/group
```

Note: We strongly recommend that customers change their passwords after initial setup.

```
cd /etc
chown root:root passwd shadow group gshadow
chmod 644 passwd group
chmod 400 shadow gshadow
```

Find all the files that are world-writable and that do not have their sticky bits set.

```
find / -xdev -type d \( -perm -0002 -a ! -perm -1000 \) -print
```

Set the sticky bit (`# chmod +t {dir}`) for all the directories that result from running the previous command.

Find all the files that are world-writable and fix each file listed.

```
find / -xdev -type f -perm -0002 -print
```

Set the right permissions (`# chmod o-w {file}`) for all the files generated by running the aforementioned command.

Find all the files that do not belong to a valid user or group and either assign an owner or remove the file, as appropriate.

```
find / -xdev \( -nouser -o -nogroup \) -print
```

Find all the directories that are world-writable and ensure they are owned by either root or a system account (assuming only system accounts have a User ID lower than 500). If the command generates any output, verify the assignment is correct or reassign it to root.

```
find / -xdev -type d -perm -0002 -uid +500 -print
```

Authentication settings such as password quality, password expiration policy, password reuse, password retry attempts, and more can be configured using the Pluggable Authentication Modules (PAM) framework.

PAM looks in the directory `/etc/pam.d` for application-specific configuration information. Running `authconfig` or `system-config-authentication` will re-write the PAM configuration files, destroying any manually made changes and replacing them with system defaults.

The default `pam_cracklib` PAM module provides strength checking for passwords. To configure `pam_cracklib` to require at least one uppercase character, lowercase character, digit, and special character, as recommended by the U.S. Department of Defense guidelines, edit the file `/etc/pam.d/system-auth` to include the following parameters in the line corresponding to password requisite `pam_cracklib.so try_first_pass`.

```
retry=3:
dcredit=-1. Require at least one digit
ucredit=-1. Require at least one upper case character
ocredit=-1. Require at least one special character
lcredit=-1. Require at least one lower case character
minlen-14. Require a minimum password length of 14.
```

For example:

```
password required pam_cracklib.so try_first_pass retry=3\minlen=14 dcredit=-1
 ucredit=-1 ocredit=-1 lcredit=-1
```

These parameters can be set to reflect your security policy requirements. Note that the password restrictions are not applicable to the root password.

The `pam_tally2` PAM module provides the capability to lock out user accounts after a specified number of failed login attempts. To enforce password lockout, edit the file `/etc/pam.d/system-auth` to include the following lines:

- The first of the auth lines should include:

  ```
  auth required pam_tally2.so deny=5 onerr=fail unlock_time=900
  ```

- The first of the account lines should include:

  ```
  account required pam_tally2.so
  ```

Here, the deny parameter is set to limit the number of retries to 5 and the `unlock_time` has been set to 900 seconds to keep the account locked for 900 seconds before it is unlocked. These parameters may be configured appropriately to reflect your security policy requirements. A locked account can be manually unlocked using the `pam_tally2` utility:

```
/sbin/pam_tally2 --user {username} -reset
```

You can use PAM to limit the reuse of recent passwords. The remember option for the `pam_ unix` module can be set to remember the recent passwords and prevent their reuse. To accomplish this, edit the appropriate line in `/etc/pam.d/system-auth` to include the remember option.

For example:

```
password sufficient pam_unix.so [ … existing_options …]
remember=5
```

You can set the number of previous passwords to remember to appropriately reflect your security policy requirements.

```
cd /etc
chown root:root passwd shadow group gshadow
chmod 644 passwd group
```

```
chmod 400 shadow gshadow
```

Chapter 9

Encrypting Data and Database Connections

Encryption can be used to protect data in a Greenplum Database system in the following ways:

- Connections between clients and the master database can be encrypted with SSL. This is enabled by setting the `ssl` server configuration parameter to `on` and editing the `pg_hba.conf` file. See "Encrypting Client/Server Connections" in the *Greenplum Database Administrator Guide* for information about enabling SSL in Greenplum Database.
- Greenplum Database 4.2.1 and above allow SSL encryption of data in transit between the Greenplum parallel file distribution server, `gpfdist`, and segment hosts. See*Encrypting gpfdist Connections* for more information.
- Network communications between hosts in the Greenplum Database cluster can be encrypted using IPsec. An authenticated, encrypted VPN is established between every pair of hosts in the cluster. See "Configuring IPsec for Greenplum Database" in the *Greenplum Database Adminitrator Guide*.
- The `pgcrypto` package of encryption/decryption functions protects data at rest in the database. Encryption at the column level protects sensitive information, such as passwords, Social Security numbers, or credit card numbers. See *Encrypting Data in Tables using PGP* for an example.

Best Practices

- Encryption ensures that data can be seen only by users who have the key required to decrypt the data.
- Encrypting and decrypting data has a performance cost; only encrypt data that requires encryption.
- Do performance testing before implementing any encryption solution in a production system.
- Server certificates in a production Greenplum Database system should be signed by a certificate authority (CA) so that clients can authenticate the server. The CA may be local if all clients are local to the organization.
- Client connections to Greenplum Database should use SSL encryption whenever the connection goes through an insecure link.
- A symmetric encryption scheme, where the same key is used to both encrypt and decrypt, has better performance than an asymmetric scheme and should be used when the key can be shared safely.
- Use functions from the pgcrypto package to encrypt data on disk. The data is encrypted and decrypted in the database process, so it is important to secure the client connection with SSL to avoid transmitting unencrypted data.
- Use the gpfdists protocol to secure ETL data as it is loaded into or unloaded from the database. See *Encrypting gpfdist Connections*.

Key Management

Whether you are using symmetric (single private key) or asymmetric (public and private key) cryptography, it is important to store the master or private key securely. There are many options for storing encryption keys, for example, on a file system, key vault, encrypted USB, trusted platform module (TPM), or hardware security module (HSM).

Consider the following questions when planning for key management:

- Where will the keys be stored?
- When should keys expire?
- How are keys protected?
- How are keys accessed?

- How can keys be recovered and revoked?

The Open Web Application Security Project (OWASP) provides a very comprehensive *guide to securing encryption keys*.

Encrypting Data at Rest with pgcrypto

The pgcrypto package for Greenplum Database provides functions for encrypting data at rest in the database. Administrators can encrypt columns with sensitive information, such as social security numbers or credit card numbers, to provide an extra layer of protection. Database data stored in encrypted form cannot be read by users who do not have the encryption key, and the data cannot be read directly from disk.

pgcrypto allows PGP encryption using symmetric and asymmetric encryption. Symmetric encryption encrypts and decrypts data using the same key and is faster than asymmetric encryption. It is the preferred method in an environment where exchanging secret keys is not an issue. With asymmetric encryption, a public key is used to encrypt data and a private key is used to decrypt data. This is slower then symmetric encryption and it requires a stronger key.

Using pgcrypto always comes at the cost of performance and maintainability. It is important to use encryption only with the data that requires it. Also, keep in mind that you cannot search encrypted data by indexing the data.

Before you implement in-database encryption, consider the following PGP limitations.

- No support for signing. That also means that it is not checked whether the encryption sub-key belongs to the master key.
- No support for encryption key as master key. This practice is generally discouraged, so this limitation should not be a problem.
- No support for several subkeys. This may seem like a problem, as this is common practice. On the other hand, you should not use your regular GPG/PGP keys with pgcrypto, but create new ones, as the usage scenario is rather different.

Greenplum Database is compiled with zlib by default; this allows PGP encryption functions to compress data before encrypting. When compiled with OpenSSL, more algorithms will be available.

Because pgcrypto functions run inside the database server, the data and passwords move between pgcrypto and the client application in clear-text. For optimal security, you should connect locally or use SSL connections and you should trust both the system and database administrators.

The pgcrypto package is not installed by default with Greenplum Database. You can download a pgcrypto package from *Pivotal Network* and use the Greenplum Package Manager (`gppkg`) to install `pgcrypto` across your cluster.

pgcrypto configures itself according to the findings of the main PostgreSQL configure script.

When compiled with `zlib`, pgcrypto encryption functions are able to compress data before encrypting.

You can enable support for Federal Information Processing Standards (FIPS) 140-2 in pgcrypto. FIPS 140-2 requires pgcrypto package version 1.2. The Greenplum Database `pgcrypto.fips` server configuration parameter controls the FIPS 140-2 support in pgcrypto. See "Server Configuration Parameters" in the *Greenplum Database Reference Guide*.

Pgcrypto has various levels of encryption ranging from basic to advanced built-in functions. The following table shows the supported encryption algorithms.

Table 1: Pgcrypto Supported Encryption Functions

Value Functionality	Built-in	With OpenSSL	OpenSSL with FIPS 140-2
MD5	yes	yes	no

Value Functionality	Built-in	With OpenSSL	OpenSSL with FIPS 140-2
SHA1	yes	yes	no
SHA224/256/384/512	yes	yes [1].	yes
Other digest algorithms	no	yes [2]	no
Blowfish	yes	yes	no
AES	yes	yes[3]	yes
DES/3DES/CAST5	no	yes	yes[4]
Raw Encryption	yes	yes	yes
PGP Symmetric-Key	yes	yes	yes
PGP Public Key	yes	yes	yes

Creating PGP Keys

To use PGP asymmetric encryption in Greenplum Database, you must first create public and private keys and install them.

This section assumes you are installing Greenplum Database on a Linux machine with the Gnu Privacy Guard (`gpg`) command line tool. Pivotal recommends using the latest version of GPG to create keys. Download and install Gnu Privacy Guard (GPG) for your operating system from *https://www.gnupg.org/download/*. On the GnuPG website you will find installers for popular Linux distributions and links for Windows and Mac OS X installers.

1. As root, execute the following command and choose option 1 from the menu:

```
# gpg --gen-key
gpg (GnuPG) 2.0.14; Copyright (C) 2009 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

gpg: directory `/root/.gnupg' created
gpg: new configuration file `/root/.gnupg/gpg.conf' created
gpg: WARNING: options in `/root/.gnupg/gpg.conf' are not yet active during this
 run
gpg: keyring `/root/.gnupg/secring.gpg' created
gpg: keyring `/root/.gnupg/pubring.gpg' created
Please select what kind of key you want:
  (1) RSA and RSA (default)
  (2) DSA and Elgamal
  (3) DSA (sign only)
  (4) RSA (sign only)
Your selection? 1
```

2. Respond to the prompts and follow the instructions, as shown in this example:

```
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048) Press enter to accept default key size
Requested keysize is 2048 bits
Please specify how long the key should be valid.
```

[1] SHA2 algorithms were added to OpenSSL in version 0.9.8. For older versions, pgcrypto will use built-in code

[2] Any digest algorithm OpenSSL supports is automatically picked up. This is not possible with ciphers, which need to be supported explicitly.

[3] AES is included in OpenSSL since version 0.9.7. For older versions, pgcrypto will use built-in code.

[4] 3DES is supported, DES and CAST5 are not

```
 0 = key does not expire
 <n> = key expires in n days
 <n>w = key expires in n weeks
 <n>m = key expires in n months
 <n>y = key expires in n years
 Key is valid for? (0) 365
Key expires at Wed 13 Jan 2016 10:35:39 AM PST
Is this correct? (y/N) y

GnuPG needs to construct a user ID to identify your key.

Real name: John Doe
Email address: jdoe@email.com
Comment:
You selected this USER-ID:
 "John Doe <jdoe@email.com>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
You need a Passphrase to protect your secret key.
(For this demo the passphrase is blank.)
can't connect to `/root/.gnupg/S.gpg-agent': No such file or directory
You don't want a passphrase - this is probably a *bad* idea!
I will do it anyway.  You can change your passphrase at any time,
using this program with the option "--edit-key".

We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
gpg: /root/.gnupg/trustdb.gpg: trustdb created
gpg: key 2027CC30 marked as ultimately trusted
public and secret key created and signed.

gpg:  checking the trustdbgpg:
      3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg:  depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
gpg:  next trustdb check due at 2016-01-13
pub   2048R/2027CC30 2015-01-13 [expires: 2016-01-13]
      Key fingerprint = 7EDA 6AD0 F5E0 400F 4D45   3259 077D 725E 2027 CC30
uid                  John Doe <jdoe@email.com>
sub   2048R/4FD2EFBB 2015-01-13 [expires: 2016-01-13]
```

3. List the PGP keys by entering the following command:

```
gpg --list-secret-keys
/root/.gnupg/secring.gpg
------------------------
sec   2048R/2027CC30 2015-01-13 [expires: 2016-01-13]
uid                  John Doe <jdoe@email.com>
ssb   2048R/4FD2EFBB 2015-01-13
```

   2027CC30 is the public key and will be used to *encrypt* data in the database. 4FD2EFBB is the private (secret) key and will be used to *decrypt* data.

4. Export the keys using the following commands:

```
# gpg -a --export 4FD2EFBB > public.key
# gpg -a --export-secret-keys 2027CC30 > secret.key
```

See the *pgcrypto* documentation for for more information about PGP encryption functions.

Encrypting Data in Tables using PGP

This section shows how to encrypt data inserted into a column using the PGP keys you generated.

1. Dump the contents of the `public.key` file and then copy it to the clipboard:

```
# cat public.key
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2.0.14 (GNU/Linux)

mQENBFS1Zf0BCADNw8Qvk1V1C36Kfcwd3Kpm/dijPfRyyEwB6PqKyA05jtWiXZTh
2His1ojSP6LI0cSkIqMU9LAlncecZhRIhBhuVgKlGSgd9texg2nnSL9Admqik/yX
R5syVKG+qcdWuvyZg9oOOmeyjhc3n+kkbRTEMuM3flbMs8shOwzMvstCUVmuHU/V
vG5rJAe8PuYDSJCJ74I6w7SOH3RiRIc7IfL6xYddV42l3ctd44bl8/i71hq2UyN2
/Hbsjii2ymg7ttw3jsWAx2gP9nssDgoy8QDy/o9nNqC8EGlig96ZFnFnE6Pwbhn+
ic8MD0lK5/GAlR6Hc0ZIHf8KEcavruQlikjnABEBAAG0HHRlc3Qga2V5IDx0ZXN0
a2V5QGVtYWlsLmNvbT6JAT4EEwECACgFAlS1Zf0CGwMFCQHhM4AGCwkIBwMCBhUI
AgkKCwQWAgMBAh4BAheAAAoJEAd9cl4gJ8wwbfwH/3VyVsPkQl1owRJNxvXGt1bY
7BfrvU52yk+PPZYoes9UpdL3CMRk8gAM9bx5Sk08q2UXSZLC6fFOpEW4uWgmGYf8
JRoC3ooezTkmCBW8I1bU0qGetzVxopdXLuPGCE7hVWQe9HcSntiTLxGov1mJAwO7
TAoccXLbyuZh9Rf5vLoQdKzcCyOHh5IqXaQOT100TeFeEpb9TIiwcntg3WCSU5P0
DGoUAOanjDZ3KE8Qp7V74fhG1EZVzHb8FajR62CXSHFKqpBgiNxnTOk45NbXADn4
eTUXPSnwPi46qoAp9UQogsfGyB1XDOTB2UOqhutAMECaM7VtpePv79i0Z/NfnBe5
AQ0EVLVl/QEIANabFdQ+8QMCADOipM1bF/JrQt3zUoc4BTqICaxdyzAfz0tUSf/7
Zro2us99GlARqLWd8EqJcl/xmfcJiZyUam6ZAzzFXCgnH5Y1sdtMTJZdLp5WeOjw
gCWG/ZLu4wzxOFFzDkiPv9RDw6e5MNLtJrSp4hS5o2apKdbO4Ex83O4mJYnav/rE
iDDCWU4T0lhv3hSKCpke6LcwsX+7liozp+aNmP0Ypwfi4hR3UUMP70+V1beFqW2J
bVLz3lLLouHRgpCzla+PzzbEKs16jq77vG9kqZTCIzXoWaLljuitRlfJkO3vQ9hO
v/8yAnkcAmowZrIBlyFg2KBzhunYmN2YvkUAEQEAAYkBJQQYAQIADwUCVLVl/QIb
DAUJAeEzgAAKCRAHfXJeICfMMOHYCACFhInZA9uAM3TC44l+MrgMUJ3rW9izrO48
WrdTsxR8WkSNbIxJoWnYxYuLyPb/shc9k65huw2SSDkj//0fRrI61FPHQNPSvz62
WH+N2lasoUaoJjb2kQGhLOnFbJuevkyBylRz+hI/+8rJKcZOjQkmmK8Hkk8qb5x/
HMUc55H0g2qQAY0BpnJHgOOQ45Q6pk3G2/7Dbek5WJ6K1wUrFy51sNlGWE8pvgEx
/UUZB+dYqCwtvX0nnBu1KNCmk2AkEcFK3YoliCxomdOxhFOv9AKjjojDyC65KJci
Pv2MikPS2fKOAg1R3LpMa8zDEtl4w3vckPQNrQNnYuUtfj6ZoCxv
=XZ8J
-----END PGP PUBLIC KEY BLOCK-----
```

2. Create a table called `userssn` and insert some sensitive data, social security numbers for Bob and Alice, in this example. Paste the public.key contents after "dearmor(".

```
CREATE TABLE userssn( ssn_id SERIAL PRIMARY KEY,
    username varchar(100), ssn bytea);

INSERT INTO userssn(username, ssn)
SELECT robotccs.username, pgp_pub_encrypt(robotccs.ssn, keys.pubkey) AS ssn
FROM (
        VALUES ('Alice', '123-45-6788'), ('Bob', '123-45-6799'))
            AS robotccs(username, ssn)
CROSS JOIN  (SELECT  dearmor('-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2.0.14 (GNU/Linux)

mQENBFS1Zf0BCADNw8Qvk1V1C36Kfcwd3Kpm/dijPfRyyEwB6PqKyA05jtWiXZTh
2His1ojSP6LI0cSkIqMU9LAlncecZhRIhBhuVgKlGSgd9texg2nnSL9Admqik/yX
R5syVKG+qcdWuvyZg9oOOmeyjhc3n+kkbRTEMuM3flbMs8shOwzMvstCUVmuHU/V
vG5rJAe8PuYDSJCJ74I6w7SOH3RiRIc7IfL6xYddV42l3ctd44bl8/i71hq2UyN2
/Hbsjii2ymg7ttw3jsWAx2gP9nssDgoy8QDy/o9nNqC8EGlig96ZFnFnE6Pwbhn+
ic8MD0lK5/GAlR6Hc0ZIHf8KEcavruQlikjnABEBAAG0HHRlc3Qga2V5IDx0ZXN0
a2V5QGVtYWlsLmNvbT6JAT4EEwECACgFAlS1Zf0CGwMFCQHhM4AGCwkIBwMCBhUI
AgkKCwQWAgMBAh4BAheAAAoJEAd9cl4gJ8wwbfwH/3VyVsPkQl1owRJNxvXGt1bY
7BfrvU52yk+PPZYoes9UpdL3CMRk8gAM9bx5Sk08q2UXSZLC6fFOpEW4uWgmGYf8
JRoC3ooezTkmCBW8I1bU0qGetzVxopdXLuPGCE7hVWQe9HcSntiTLxGov1mJAwO7
TAoccXLbyuZh9Rf5vLoQdKzcCyOHh5IqXaQOT100TeFeEpb9TIiwcntg3WCSU5P0
DGoUAOanjDZ3KE8Qp7V74fhG1EZVzHb8FajR62CXSHFKqpBgiNxnTOk45NbXADn4
eTUXPSnwPi46qoAp9UQogsfGyB1XDOTB2UOqhutAMECaM7VtpePv79i0Z/NfnBe5
AQ0EVLVl/QEIANabFdQ+8QMCADOipM1bF/JrQt3zUoc4BTqICaxdyzAfz0tUSf/7
Zro2us99GlARqLWd8EqJcl/xmfcJiZyUam6ZAzzFXCgnH5Y1sdtMTJZdLp5WeOjw
gCWG/ZLu4wzxOFFzDkiPv9RDw6e5MNLtJrSp4hS5o2apKdbO4Ex83O4mJYnav/rE
iDDCWU4T0lhv3hSKCpke6LcwsX+7liozp+aNmP0Ypwfi4hR3UUMP70+V1beFqW2J
bVLz3lLLouHRgpCzla+PzzbEKs16jq77vG9kqZTCIzXoWaLljuitRlfJkO3vQ9hO
v/8yAnkcAmowZrIBlyFg2KBzhunYmN2YvkUAEQEAAYkBJQQYAQIADwUCVLVl/QIb
DAUJAeEzgAAKCRAHfXJeICfMMOHYCACFhInZA9uAM3TC44l+MrgMUJ3rW9izrO48
WrdTsxR8WkSNbIxJoWnYxYuLyPb/shc9k65huw2SSDkj//0fRrI61FPHQNPSvz62
WH+N2lasoUaoJjb2kQGhLOnFbJuevkyBylRz+hI/+8rJKcZOjQkmmK8Hkk8qb5x/
```

```
HMUc55H0g2qQAY0BpnJHgOOQ45Q6pk3G2/7Dbek5WJ6K1wUrFy51sNlGWE8pvgEx
/UUZB+dYqCwtvX0nnBu1KNCmk2AkEcFK3YoliCxomdOxhFOv9AKjjojDyC65KJci
Pv2MikPS2fKOAg1R3LpMa8zDEtl4w3vckPQNrQNnYuUtfj6ZoCxv
=XZ8J
-----END PGP PUBLIC KEY BLOCK-----' AS pubkey) AS keys;
```

3. Verify that the `ssn` column is encrypted.

```
test_db=# select * from userssn;
ssn_id   | 1
username | Alice
ssn      | \301\300L\003\235M%_O
\322\357\273\001\010\000\272\227\010\341\216\360\217C\020\261)_\367
[\227\034\313:C\354d<\337\006Q\351('\2330\031lX\263Qf\341\262\200\3015\235\036AK
\242fL+\315g\322
7u\270*\304\361\355\220\021\330"\200%\264\274}R\213\377\363\235\366\030\023)\364!
\331\303\237t\277=
f \015\004\242\231\263\225%\032\271a\001\035\277\021\375X\232\304\305/
\340\334\0131\325\344[~\362\0
37-\251\336\303\340\377_\011\275\301/MY\334\343\245\244\372y\257S
\374\230\346\277\373W\346\230\276\
017fi\226Q\307\012\326\3646\000\326\005:E\364W\252=zz\010(:\343Y\237\257iqU
\0326\350=v0\362\327\350\
315G^\027:K_9\254\362\354\215<\001\304\357\331\355\323,\302\213Fe
\265\315\232\367\254\245%(\\\373
4\254\230\331\356\006B\257\333\326H\022\013\353\216F?\023\220\370\035vH5/\227\344b
\322\227\026\362=\
42\033\322<\001}\243\224;)\030zqX\214\340\221\035\275U\345\327\214\032\351\223c
\2442\345\304K\016\
011\214\307\227\237\270\026`R\205\205a~1\263\236[\037C
\260\031\205\374\245\317\033k|\366\253\037
---------
+-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------
ssn_id   | 2
username | Bob
ssn      | \301\300L\003\235M%_O\322\357\273\001\007\377t>\345\343,
\200\256\272\300\012\033M4\265\032L
L[v\262k\244\2435\264\232B\357\370d9\375\011\002\327\235<\246\210b\030\012\337@
\226Z\361\246\032\00
7`\012c\353]\355d7\360T\335\314\367\370;X\371\350*\231\212\260B
\010#RQ0\223\253c7\0132b\355\242\233\34
1\000\370\370\366\013\022\357\005i\202~\005\\z\301o\012\230Z\014\362\244\324&\243g
\351\362\325\375
\213\032\226$\2751\256XR\346k\266\030\234\267\201vUh\004\250\337A\231\223u
\247\366/i\022\275\276\350\2
20\316\306|\203+\010\261;\232\254tp\255\243\261\373Rq;\316w\357\006\207\374U
\333\365\365\245hg\031\005
\322\347ea\220\015l\212g\337\264\336b\263\004\311\210.4\340G+\221\274D
\035\375\2216\241`\346a0\273wE\2
12\342y^\202\262|A7\202t\240\333p\345G\373\253\243oCO
\011\360\247\211\014\024{\272\271\322<\001\267
\347\240\005\213\0078\036\210\307$\317\322\311\222\035\354\006<
\266\264\004\376\251q\256\220(+\030\
3270\013c\327\272\212%\363\033\252\322\337\354\276\225\232\201\212^\304\210\2269@
\3230\370{
```

4. Extract the public.key ID from the database:

```
SELECT pgp_key_id(dearmor('-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2.0.14 (GNU/Linux)
```

```
mQENBFS1Zf0BCADNw8Qvk1V1C36Kfcwd3Kpm/dijPfRyyEwB6PqKyA05jtWiXZTh
2His1ojSP6LI0cSkIqMU9LAlncecZhRIhBhuVgKlGSgd9texg2nnSL9Admqik/yX
R5syVKG+qcdWuvyZg9oOOmeyjhc3n+kkbRTEMuM3flbMs8shOwzMvstCUVmuHU/V
vG5rJAe8PuYDSJCJ74I6w7SOH3RiRIc7IfL6xYddV42l3ctd44bl8/i71hq2UyN2
/Hbsjii2ymg7ttw3jsWAx2gP9nssDgoy8QDy/o9nNqC8EGlig96ZFnFnE6Pwbhn+
ic8MD0lK5/GAlR6Hc0ZIHf8KEcavruQlikjnABEBAAG0HHRlc3Qga2V5IDx0ZXN0
a2V5QGVtYWlsLmNvbT6JAT4EEwECACgFAls1Zf0CGwMFCQHhM4AGCwkIBwMCBhUI
AgkKCwQWAgMBAh4BAheAAAoJEAd9cl4gJ8wwbfwH/3VyVsPkQl1owRJNxvXGt1bY
7BfrvU52yk+PPZYoes9UpdL3CMRk8gAM9bx5Sk08q2UXSZLC6fFOpEW4uWgmGYf8
JRoC3ooezTkmCBW8I1bU0qGetzVxopdXLuPGCE7hVWQe9HcSntiTLxGov1mJAwO7
TAoccXLbyuZh9Rf5vLoQdKzcCyOHh5IqXaQOT100TeFeEpb9TIiwcntg3WCSU5P0
DGoUAOanjDZ3KE8Qp7V74fhG1EZVzHb8FajR62CXSHFKqpBgiNxnTOk45NbXADn4
eTUXPSnwPi46qoAp9UQogsfGyB1XDOTB2UOqhutAMECaM7VtpePv79i0Z/NfnBe5
AQ0EVLVl/QEIANabFdQ+8QMCADOipM1bF/JrQt3zUoc4BTqICaxdyzAfz0tUSf/7
Zro2us99GlARqLWd8EqJcl/xmfcJiZyUam6ZAzzFXCgnH5Y1sdtMTJZdLp5WeOjw
gCWG/ZLu4wzxOFFzDkiPv9RDw6e5MNLtJrSp4hS5o2apKdbO4Ex83O4mJYnav/rE
iDDCWU4T0lhv3hSKCpke6LcwsX+7liozp+aNmP0Ypwfi4hR3UUMP70+V1beFqW2J
bVLz3lLLouHRgpCzla+PzzbEKs16jq77vG9kqZTCIzXoWaLljuitRlfJkO3vQ9hO
v/8yAnkcAmowZrIBlyFg2KBzhunYmN2YvkUAEQEAAYkBJQQYAQIADwUCVLVl/QIb
DAUJAeEzgAAKCRAHfXJeICfMMOHYCACFhInZA9uAM3TC44l+MrgMUJ3rW9izrO48
WrdTsxR8WkSNbIxJoWnYxYuLyPb/shc9k65huw2SSDkj//0fRrI61FPHQNPSvz62
WH+N2lasoUaoJjb2kQGhLOnFbJuevkyBylRz+hI/+8rJKcZOjQkmmK8Hkk8qb5x/
HMUc55H0g2qQAY0BpnJHgOOQ45Q6pk3G2/7Dbek5WJ6K1wUrFy51sNlGWE8pvgEx
/UUZB+dYqCwtvX0nnBu1KNCmk2AkEcFK3YoliCxomdOxhFOv9AKjjojDyC65KJci
Pv2MikPS2fKOAg1R3LpMa8zDEtl4w3vckPQNrQNnYuUtfj6ZoCxv
=XZ8J
-----END PGP PUBLIC KEY BLOCK-----'));

pgp_key_id | 9D4D255F4FD2EFBB
```

This shows that the PGP key ID used to encrypt the `ssn` column is 9D4D255F4FD2EFBB. It is recommended to perform this step whenever a new key is created and then store the ID for tracking.

You can use this key to see which key pair was used to encrypt the data:

```
SELECT username, pgp_key_id(ssn) As key_used

 FROM userssn;

           username | Bob
key_used | 9D4D255F4FD2EFBB
----------+-----------------
username | Alice
key_used | 9D4D255F4FD2EFBB
```

> **Note:** Different keys may have the same ID. This is rare, but is a normal event. The client application should try to decrypt with each one to see which fits — like handling `ANYKEY`. See *pgp_key_id()* in the pgcrypto documentation.

5. Decrypt the data using the private key.

```
SELECT username, pgp_pub_decrypt(ssn, keys.privkey)
                 AS decrypted_ssn FROM userssn
                 CROSS JOIN
                 (SELECT dearmor('-----BEGIN PGP PRIVATE KEY BLOCK-----
Version: GnuPG v2.0.14 (GNU/Linux)

lQOYBFS1Zf0BCADNw8Qvk1V1C36Kfcwd3Kpm/dijPfRyyEwB6PqKyA05jtWiXZTh
2His1ojSP6LI0cSkIqMU9LAlncecZhRIhBhuVgKlGSgd9texg2nnSL9Admqik/yX
R5syVKG+qcdWuvyZg9oOOmeyjhc3n+kkbRTEMuM3flbMs8shOwzMvstCUVmuHU/V
vG5rJAe8PuYDSJCJ74I6w7SOH3RiRIc7IfL6xYddV42l3ctd44bl8/i71hq2UyN2
/Hbsjii2ymg7ttw3jsWAx2gP9nssDgoy8QDy/o9nNqC8EGlig96ZFnFnE6Pwbhn+
ic8MD0lK5/GAlR6Hc0ZIHf8KEcavruQlikjnABEBAAEAB/wNfjjvP1brRfjjIm/j
XwUNm+sI4v2Ur7qZC94VTukPGf67lvqcYZJuqXxvZrZ8bl6mvl65xEUiZYy7BNA8
fe0PaM4Wy+Xr94Cz2bPbWgawnRNN3GAQy4rlBTrvqQWy+kmpbd87iTjwZidZNNmx
02iSzraq41Rt0Zx21Jh4rkpF67ftmzOHOvlrS0bWOvHUeMY7tCwmdPe9HbQeDlPr
n9CllUqBn4/acTtCClWAjREZn0zXAsNixtTIPC1V+9nO9YmecMkVwNfIPkIhymAM
OPFnuZ/Dz1rCRHjNHb5j6ZyUM5zDqUVnnezktxqrOENSxm0gfMGcpxHQogUMzb7c
6UyBBADSCXHPfo/VPVtMm5p1yGrNOR2jR2rUj9+poZzD2gjkt5G/xIKRlkB4uoQl
```

```
emu27wr9dVEX7ms0nvDq58iutbQ4d0JIDlcHMeSRQZluErblB75Vj3HtImblPjpn
4Jx6SWRXPUJPGXGI87u0UoBH0Lwij7M2PW7l1ao+MLEA9jAjQwQA+sr9BKPL4Ya2
r5nE72gsbCCLowkC0rdldf1RGtobwYDMpmYZhOaRKjkOTMG6rCXJxrf6LqiN8w/L
/gNziTmch35MCq/MZzA/bN4VMPyeIlwzxVZkJLsQ7yyqX/A7ac7B7DH0KfXciEXW
MSOAJhMmklW1Q1RRNw3cnYi8w3q7X40EAL/w54FVvvPqp3+sCd86SAAapM4UO2R3
tIsuNVemMWdgNXwvK8AJsz7VreVU5yZ4B8hvCuQj1C7geaN/LXhiT8foRsJC5o71
Bf+iHC/VNEv4k4uDb4lOgnHJYYyifB1wC+nn/EnXCZYQINMia1a4M6Vqc/RIfTH4
nwkZt/89LsAiR/20HHRlc3Qga2V5IDx0ZXN0a2V5QGVtYWlsLmNvbT6JAT4EEwEC
ACgFAlS1Zf0CGwMFCQHhM4AGCwkIBwMCBhUIAgkKCwQWAgMBAh4BAheAAAoJEAd9
cl4gJ8wwbfwH/3VyVsPkQl1owRJNxvXGt1bY7BfrvU52yk+PPZYoes9UpdL3CMRk
8gAM9bx5Sk08q2UXSZLC6fFOpEW4uWgmGYf8JRoC3ooezTkmCBW8I1bU0qGetzVx
opdXLuPGCE7hVWQe9HcSntiTLxGov1mJAwO7TAoccXLbyuZh9Rf5vLoQdKzcCyOH
h5IqXaQOT100TeFeEpb9TIiwcntg3WCSU5P0DGoUAOanjDZ3KE8Qp7V74fhG1EZV
zHb8FajR62CXSHFKqpBgiNxnTOk45NbXADn4eTUXPSnwPi46qoAp9UQogsfGyB1X
DOTB2UOqhutAMECaM7VtpePv79i0Z/NfnBedA5gEVLVl/QEIANabFdQ+8QMCADOi
pM1bF/JrQt3zUoc4BTqICaxdyzAfz0tUSf/7Zro2us99GlARqLWd8EqJcl/xmfcJ
iZyUam6ZAzzFXCgnH5Y1sdtMTJZdLp5WeOjwgCWG/ZLu4wzxOFFzDkiPv9RDw6e5
MNLtJrSp4hS5o2apKdbO4Ex83O4mJYnav/rEiDDCWU4T0lhv3hSKCpke6LcwsX+7
liozp+aNmP0Ypwfi4hR3UUMP70+V1beFqW2JbVLz3lLLouHRgpCzla+PzzbEKs16
jq77vG9kqZTCIzXoWaLljuitRlfJkO3vQ9hOv/8yAnkcAmowZrIBlyFg2KBzhunY
mN2YvkUAEQEAAQAH/A7r4hDrnmzX3QU6FAzePlRB7niJtE2IEN8AufF05Q2PzKU/
c1S72WjtqMAIAgYasDkOhfhcxanTneGuFVYggKT3eSDm1RFKpRjX22m0zKdwy67B
Mu95V2Oklul6OCm8dO6+2fmkGxGqc4ZsKy+jQxtxK3HG9YxMC0dvA2v2C5N4TWi3
Utc7zh//k6IbmaLd7F1d7DXt7Hn2Qsmo8I1rtgPE8grDToomTnRUodToyejEqKyI
ORwsp8n8g2CSFaXSrEyU6HbFYXSxZealhQJGYLFOZdR0MzVtZQCn/7n+IHjupndC
Nd2a8DVx3yQS3dAmvLzhFacZdjXi31wvj0moFOkEAOCz1E63SKNNksniQ11lRMJp
gaov6Ux/zGLMstwTzNouI+Kr8/db0GlSAy1Z3UoAB4tFQXEApoX9A4AJ2KqQjqOX
cZVULenfDZaxrbb9Lid7ZnTDXKVyGTWDF7ZHavHJ4981mCW17lU11zHBB9xMlx6p
dhFvb0gdy0jSLaFMFr/JBAD0fz3RrhP7e6Xll2zdBqGthjC5S/IoKwwBgw6ri2yx
LoxqBr2pl9PotJJ/JUMPhD/LxuTcOZtYjy8PKgm5jhnBDq3Ss0kNKAY1f5EkZG9a
6I4iAX/NekqSyF+OgBfC9aCgS5RG8hYoOCbp8na5R3bgiuS8IzmVmm5OhZ4MDEwg
nQP7BzmR0p5BahpZ8r3Ada7FcK+0ZLLRdLmOYF/yUrZ53SoYCZRzU/GmtQ7LkXBh
Gjqied9Bs1MHdNUolq7GaexcjZmOWHEf6w9+9M4+vxtQq1nkIWqtaphewEmd5/nf
EP3sIY0EAE3mmiLmHLqBju+UJKMNwFNeyMTqgcg50ISH8J9FRIkBJQQYAQIADwUC
VLVl/QIbDAUJAeEzgAAKCRAHfXJeICfMMOHYCACFhInZA9uAM3TC44l+MrgMUJ3r
W9izrO48WrdTsxR8WkSNbIxJoWnYxYuLyPb/shc9k65huw2SSDkj//0fRrI61FPH
QNPSvz62WH+N2lasoUaoJjb2kQGhLOnFbJuevkyBylRz+hI/+8rJKcZOjQkmmK8H
kk8qb5x/HMUc55H0g2qQAY0BpnJHgOOQ45Q6pk3G2/7Dbek5WJ6K1wUrFy51sNlG
WE8pvgEx/UUZB+dYqCwtvX0nnBu1KNCmk2AkEcFK3YoliCxomdOxhFOv9AKjjojD
yC65KJciPv2MikPS2fKOAg1R3LpMa8zDEtl4w3vckPQNrQNnYuUtfj6ZoCxv
=fa+6
-----END PGP PRIVATE KEY BLOCK-----') AS privkey) AS keys;

username | decrypted_ssn
----------+---------------
 Alice    | 123-45-6788
 Bob      | 123-45-6799
(2 rows)
```

If you created a key with passphrase, you may have to enter it here. However for the purpose of this example, the passphrase is blank.

Encrypting gpfdist Connections

The `gpfdists` protocol is a secure version of the `gpfdist` protocol that securely identifies the file server and the Greenplum Database and encrypts the communications between them. Using `gpfdists` protects against eavesdropping and man-in-the-middle attacks.

The `gpfdists` protocol implements client/server SSL security with the following notable features:

- 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. These 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) or for the Greenplum Database (client.key).
- It is the user's responsibility to issue certificates that are appropriate for the operating system in use. Generally, converting certificates to the required format is supported, for example using the SSL Converter at *https://www.sslshopper.com/ssl-converter.html*.

A `gpfdist` server started with the `--ssl` option can only communicate with the `gpfdists` protocol. A `gpfdist` server started without the `--ssl` option can only communicate with the `gpfdist` protocol. For more detail about `gpfdist` refer to the *Greenplum Database Administrator Guide*.

There are two ways to enable the `gpfdists` protocol:

- Run `gpfdist` with the `--ssl` option and then use the `gpfdists` protocol in the `LOCATION` clause of a `CREATE EXTERNAL TABLE` statement.
- Use a YAML control file with the SSL option set to true and run `gpload`. Running `gpload` starts the `gpfdist` server with the `--ssl` option and then uses the `gpfdists` protocol.

When using gpfdists, the following client certificates must be located 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`

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

When using `gpload` with SSL you specify the location of the server certificates in the YAML control file. When using `gpfdist` with SSL, you specify the location of the server certificates with the --ssl option.

The following example shows how to securely load data into an external table. The example creates a readable external table named `ext_expenses` from all files with the `txt` extension, using the `gpfdists` protocol. The files are formatted with a pipe (`|`) as the column delimiter and an empty space as null.

1. Run `gpfdist` with the `--ssl` option on the segment hosts.
2. Log into the database and execute 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 ' ') ;
```

Chapter 10

Accessing a Kerberized Hadoop Cluster

Using external tables and the `gphdfs` protocol, Greenplum Database can read files from and write files to a Hadoop File System (HDFS). Greenplum segments read and write files in parallel from HDFS for fast performance.

When a Hadoop cluster is secured with Kerberos ("Kerberized"), Greenplum Database must be configured to allow the Greenplum Database gpadmin role, which owns external tables in HDFS, to authenticate through Kerberos. This topic provides the steps for configuring Greenplum Database to work with a Kerberized HDFS, including verifying and troubleshooting the configuration.

Prerequisites

Make sure the following components are functioning and accessible on the network:

- Greenplum Database cluster—either a Pivotal Greenplum Database software-only cluster, or a Dell EMC Data Computing Appliance (DCA).
- Kerberos-secured Hadoop cluster. See the *Greenplum Database Release Notes* for supported Hadoop versions.
- Kerberos Key Distribution Center (KDC) server.

Configuring the Greenplum Cluster

The hosts in the Greenplum Cluster must have a Java JRE, Hadoop client files, and Kerberos clients installed.

Follow these steps to prepare the Greenplum Cluster.

1. Install a Java 1.6 or later JRE on all Greenplum cluster hosts.

   Match the JRE version the Hadoop cluster is running. You can find the JRE version by running `java --version` on a Hadoop node.
2. *(Optional)* Confirm that Java Cryptography Extension (JCE) is present.

   The default location of the JCE libraries is *`JAVA_HOME`*`/lib/security`. If a JDK is installed, the directory is *`JAVA_HOME`*`/jre/lib/security`. The files `local_policy.jar` and `US_export_policy.jar` should be present in the JCE directory.

   The Greenplum cluster and the Kerberos server should, preferably, use the same version of the JCE libraries. You can copy the JCE files from the Kerberos server to the Greenplum cluster, if needed.
3. Set the `JAVA_HOME` environment variable to the location of the JRE in the `.bashrc` or `.bash_profile` file for the `gpadmin` account. For example:

   ```
   export JAVA_HOME=/usr/java/default
   ```
4. Source the `.bashrc` or `.bash_profile` file to apply the change to your environment. For example:

   ```
   $ source ~/.bashrc
   ```
5. Install the Kerberos client utilities on all cluster hosts. Ensure the libraries match the version on the KDC server before you install them.

   For example, the following command installs the Kerberos client files on Red Hat or CentOS Linux:

   ```
   $ sudo yum install krb5-libs krb5-workstation
   ```

   Use the `kinit` command to confirm the Kerberos client is installed and correctly configured.
6. Install Hadoop client files on all hosts in the Greenplum Cluster. Refer to the documentation for your Hadoop distribution for instructions.
7. Set the Greenplum Database server configuration parameters for Hadoop. The `gp_hadoop_target_version` parameter specifies the version of the Hadoop cluster. See the *Greenplum Database Release Notes* for the target version value that corresponds to your Hadoop distribution. The `gp_hadoop_home` parameter specifies the Hadoop installation directory.

   ```
   $ gpconfig -c gp_hadoop_target_version -v "hdp2"
   $ gpconfig -c gp_hadoop_home -v "/usr/lib/hadoop"
   ```

   See the *Greenplum Database Reference Guide* for more information.
8. Reload the updated `postgresql.conf` files for master and segments:

   ```
   gpstop -u
   ```

   You can confirm the changes with the following commands:

   ```
   $ gpconfig -s gp_hadoop_target_version
   $ gpconfig -s gp_hadoop_home
   ```

9. Grant Greenplum Database gphdfs protocol privileges to roles that own external tables in HDFS, including `gpadmin` and other superuser roles. Grant `SELECT` privileges to enable creating readable external tables in HDFS. Grant `INSERT` privileges to enable creating writable exeternal tables on HDFS.

```
#= GRANT SELECT ON PROTOCOL gphdfs TO gpadmin;
#= GRANT INSERT ON PROTOCOL gphdfs TO gpadmin;
```

10. Grant Greenplum Database external table privileges to external table owner roles:

```
ALTER ROLE HDFS_USER CREATEEXTTABLE (type='readable');
ALTER ROLE HDFS_USER CREATEEXTTABLE (type='writable');
```

Note: It is best practice to review database privileges, including gphdfs external table privileges, at least annually.

Creating and Installing Keytab Files

1. Log in to the KDC server as root.
2. Use the `kadmin.local` command to create a new principal for the `gpadmin` user:

   ```
   # kadmin.local -q "addprinc -randkey gpadmin@LOCAL.DOMAIN"
   ```

3. Use `kadmin.local` to generate a Kerberos service principal for each host in the Greenplum Database cluster. The service principal should be of the form *name*/*role*@*REALM*, where:

   - *name* is the gphdfs service user name. This example uses `gphdfs`.
   - *role* is the DNS-resolvable host name of a Greenplum cluster host (the output of the `hostname -f` command).
   - *REALM* is the Kerberos realm, for example `LOCAL.DOMAIN`.

   For example, the following commands add service principals for four Greenplum Database hosts, mdw.example.com, smdw.example.com, sdw1.example.com, and sdw2.example.com:

   ```
   # kadmin.local -q "addprinc -randkey gphdfs/mdw.example.com@LOCAL.DOMAIN"
   # kadmin.local -q "addprinc -randkey gphdfs/smdw.example.com@LOCAL.DOMAIN"
   # kadmin.local -q "addprinc -randkey gphdfs/sdw1.example.com@LOCAL.DOMAIN"
   # kadmin.local -q "addprinc -randkey gphdfs/sdw2.example.com@LOCAL.DOMAIN"
   ```

   Create a principal for each Greenplum cluster host. Use the same principal name and realm, substituting the fully-qualified domain name for each host.
4. Generate a keytab file for each principal that you created (`gpadmin` and each `gphdfs` service principal). You can store the keytab files in any convenient location (this example uses the directory `/etc/security/keytabs`). You will deploy the service principal keytab files to their respective Greenplum host machines in a later step:

   ```
   # kadmin.local -q "xst -k /etc/security/keytabs/gphdfs.service.keytab
    gpadmin@LOCAL.DOMAIN"
   # kadmin.local -q "xst -k /etc/security/keytabs/mdw.service.keytab gpadmin/mdw
    gphdfs/mdw.example.com@LOCAL.DOMAIN"
   # kadmin.local -q "xst -k /etc/security/keytabs/smdw.service.keytab gpadmin/smdw
    gphdfs/smdw.example.com@LOCAL.DOMAIN"
   # kadmin.local -q "xst -k /etc/security/keytabs/sdw1.service.keytab gpadmin/sdw1
    gphdfs/sdw1.example.com@LOCAL.DOMAIN"
   # kadmin.local -q "xst -k /etc/security/keytabs/sdw2.service.keytab gpadmin/sdw2
    gphdfs/sdw2.example.com@LOCAL.DOMAIN"
   # kadmin.local -q "listprincs"
   ```

5. Change the ownership and permissions on `gphdfs.service.keytab` as follows:

   ```
   # chown gpadmin:gpadmin /etc/security/keytabs/gphdfs.service.keytab
   # chmod 440 /etc/security/keytabs/gphdfs.service.keytab
   ```

6. Copy the keytab file for `gpadmin@LOCAL.DOMAIN` to the Greenplum master host:

   ```
   # scp /etc/security/keytabs/gphdfs.service.keytab mdw_fqdn:/home/gpadmin/
   gphdfs.service.keytab
   ```

7. Copy the keytab file for each service principal to its respective Greenplum host:

   ```
   # scp /etc/security/keytabs/mdw.service.keytab mdw_fqdn:/home/gpadmin/
   mdw.service.keytab
   # scp /etc/security/keytabs/smdw.service.keytab smdw_fqdn:/home/gpadmin/
   smdw.service.keytab
   # scp /etc/security/keytabs/sdw1.service.keytab sdw1_fqdn:/home/gpadmin/
   sdw1.service.keytab
   ```

```
# scp /etc/security/keytabs/sdw2.service.keytab sdw2_fqdn:/home/gpadmin/
sdw2.service.keytab
```

Configuring gphdfs for Kerberos

1. Edit the Hadoop `core-site.xml` client configuration file on all Greenplum cluster hosts. Enable service-level authorization for Hadoop by setting the `hadoop.security.authorization` property to `true`. For example:

```
<property>
    <name>hadoop.security.authorization</name>
    <value>true</value>
</property>
```

2. Edit the `yarn-site.xml` client configuration file on all cluster hosts. Set the resource manager address and yarn Kerberos service principle. For example:

```
<property>
    <name>yarn.resourcemanager.address</name>
    <value>hostname:8032</value>
</property>
<property>
    <name>yarn.resourcemanager.principal</name>
    <value>yarn/hostname@DOMAIN</value>
</property>
```

3. Edit the `hdfs-site.xml` client configuration file on all cluster hosts. Set properties to identify the NameNode Kerberos principals, the location of the Kerberos keytab file, and the principal it is for:

   - `dfs.namenode.kerberos.principal` - the Kerberos principal name the gphdfs protocol will use for the NameNode, for example `gpadmin@LOCAL.DOMAIN`.
   - `dfs.namenode.https.principal` - the Kerberos principal name the gphdfs protocol will use for the NameNode's secure HTTP server, for example `gpadmin@LOCAL.DOMAIN`.
   - `com.emc.greenplum.gpdb.hdfsconnector.security.user.keytab.file` - the path to the keytab file for the Kerberos HDFS service, for example `/home/gpadmin/mdw.service.keytab.`.
   - `com.emc.greenplum.gpdb.hdfsconnector.security.user.name` - the gphdfs service principal for the host, for example `gphdfs/mdw.example.com@LOCAL.DOMAIN`.

   For example:

```
<property>
    <name>dfs.namenode.kerberos.principal</name>
    <value>gphdfs/gpadmin@LOCAL.DOMAIN</value>
</property>
<property>
    <name>dfs.namenode.https.principal</name>
    <value>gphdfs/gpadmin@LOCAL.DOMAIN</value>
</property>
<property>
    <name>com.emc.greenplum.gpdb.hdfsconnector.security.user.keytab.file</name>
    <value>/home/gpadmin/gpadmin.hdfs.keytab</value>
</property>
<property>
    <name>com.emc.greenplum.gpdb.hdfsconnector.security.user.name</name>
    <value>gpadmin/@LOCAL.DOMAIN</value>
</property>
```

Testing Greenplum Database Access to HDFS

Confirm that HDFS is accessible via Kerberos authentication on all hosts in the Greenplum cluster. For example, enter the following command to list an HDFS directory:

```
hdfs dfs -ls hdfs://namenode:8020
```

Create a Readable External Table in HDFS

Follow these steps to verify that you can create a readable external table in a Kerberized Hadoop cluser.

1. Create a comma-delimited text file, `test1.txt`, with contents such as the following:

```
25, Bill
19, Anne
32, Greg
27, Gloria
```

2. Persist the sample text file in HDFS:

```
hdfs dfs -put test1.txt hdfs://namenode:8020/tmp
```

3. Log in to Greenplum Database and create a readable external table that points to the `test1.txt` file in Hadoop:

```
CREATE EXTERNAL TABLE test_hdfs (age int, name text)
LOCATION('gphdfs://namenode:8020/tmp/test1.txt')
FORMAT 'text' (delimiter ',');
```

4. Read data from the external table:

```
SELECT * FROM test_hdfs;
```

Create a Writable External Table in HDFS

Follow these steps to verify that you can create a writable external table in a Kerberized Hadoop cluster. The steps use the `test_hdfs` readable external table created previously.

1. Log in to Greenplum Database and create a writable external table pointing to a text file in HDFS:

```
CREATE WRITABLE EXTERNAL TABLE test_hdfs2 (LIKE test_hdfs)
LOCATION ('gphdfs://namenode:8020/tmp/test2.txt'
FORMAT 'text' (DELIMITER ',');
```

2. Load data into the writable external table:

```
INSERT INTO test_hdfs2
SELECT * FROM test_hdfs;
```

3. Check that the file exists in HDFS:

```
hdfs dfs -ls hdfs://namenode:8020/tmp/test2.txt
```

4. Verify the contents of the external file:

```
hdfs dfs -cat hdfs://namenode:8020/tmp/test2.txt
```

Troubleshooting HDFS with Kerberos

Forcing Classpaths

If you encounter "class not found" errors when executing `SELECT` statements from `gphdfs` external tables, edit the `$GPHOME/lib/hadoop-env.sh` file and add the following lines towards the end of the file, before the `JAVA_LIBRARY_PATH` is set. Update the script on all of the cluster hosts.

```
if [ -d "/usr/hdp/current" ]; then
for f in /usr/hdp/current/**/*.jar; do
    CLASSPATH=${CLASSPATH}:$f;
done
fi
```

Enabling Kerberos Client Debug Messages

To see debug messages from the Kerberos client, edit the `$GPHOME/lib/hadoop-env.sh` client shell script on all cluster hosts and set the `HADOOP_OPTS` variable as follows:

```
export HADOOP_OPTS="-Djava.net.prefIPv4Stack=true -Dsun.security.krb5.debug=true
 ${HADOOP_OPTS}"
```

Adjusting JVM Process Memory on Segment Hosts

Each segment launches a JVM process when reading or writing an external table in HDFS. To change the amount of memory allocated to each JVM process, configure the `GP_JAVA_OPT` environment variable.

Edit the `$GPHOME/lib/hadoop-env.sh` client shell script on all cluster hosts.

For example:

```
export GP_JAVA_OPT=-Xmx1000m
```

Verify Kerberos Security Settings

Review the `/etc/krb5.conf` file:

- If AES256 encryption is not disabled, ensure that all cluster hosts have the JCE Unlimited Strength Jurisdiction Policy Files installed.
- Ensure all encryption types in the Kerberos keytab file match definitions in the `krb5.conf` file.

  ```
  cat /etc/krb5.conf | egrep supported_enctypes
  ```

Test Connectivity on an Individual Segment Host

Follow these steps to test that a single Greenplum Database host can read HDFS data. This test method executes the Greenplum `HDFSReader` Java class at the command-line, and can help to troubleshoot connectivity problems outside of the database.

1. Save a sample data file in HDFS.

   ```
   hdfs dfs -put test1.txt hdfs://namenode:8020/tmp
   ```

2. On the segment host to be tested, create an environment script, `env.sh`, like the following:

   ```
   export JAVA_HOME=/usr/java/default
   export HADOOP_HOME=/usr/lib/hadoop
   export GP_HADOOP_CON_VERSION=hdp2
   ```

```
export GP_HADOOP_CON_JARDIR=/usr/lib/hadoop
```

3. Source all environment scripts:

```
source /usr/local/greenplum-db/greenplum_path.sh
source env.sh
source $GPHOME/lib/hadoop-env.sh
```

4. Test the Greenplum Database HDFS reader:

```
java com.emc.greenplum.gpdb.hdfsconnector.HDFSReader 0 32 TEXT hdp2
 gphdfs://namenode:8020/tmp/test1.txt
```

Chapter 11

Tuning SQL Queries

The Greenplum Database cost-based optimizer evaluates many strategies for executing a query and chooses the least costly method. Like other RDBMS optimizers, the Greenplum optimizer takes into account factors such as the number of rows in tables to be joined, availability of indexes, and cardinality of column data when calculating the costs of alternative execution plans. The optimizer also accounts for the location of the data, preferring to perform as much of the work as possible on the segments and to minimize the amount of data that must be transmitted between segments to complete the query.

When a query runs slower than you expect, you can view the plan the optimizer selected as well as the cost it calculated for each step of the plan. This will help you determine which steps are consuming the most resources and then modify the query or the schema to provide the optimizer with more efficient alternatives. You use the SQL `EXPLAIN` statement to view the plan for a query.

The optimizer produces plans based on statistics generated for tables. It is important to have accurate statistics to produce the best plan. See *Updating Statistics with ANALYZE* in this guide for information about updating statistics.

How to Generate Explain Plans

The `EXPLAIN` and `EXPLAIN ANALYZE` statements are useful tools to identify opportunities to improve query performance. `EXPLAIN` displays the query plan and estimated costs for a query, but does not execute the query. `EXPLAIN ANALYZE` executes the query in addition to displaying the query plan. `EXPLAIN ANALYZE` discards any output from the `SELECT` statement; however, other operations in the statement are performed (for example, `INSERT`, `UPDATE`, or `DELETE`). To use `EXPLAIN ANALYZE` on a DML statement without letting the command affect the data, explicitly use `EXPLAIN ANALYZE` in a transaction (`BEGIN; EXPLAIN ANALYZE ...; ROLLBACK;`).

`EXPLAIN ANALYZE` runs the statement in addition to displaying the plan with additional information as follows:

- Total elapsed time (in milliseconds) to run the query
- Number of workers (segments) involved in a plan node operation
- Maximum number of rows returned by the segment (and its segment ID) that produced the most rows for an operation
- The memory used by the operation
- Time (in milliseconds) it took to retrieve the first row from the segment that produced the most rows, and the total time taken to retrieve all rows from that segment.

How to Read Explain Plans

An explain plan is a report detailing the steps the Greenplum Database optimizer has determined it will follow to execute a query. The plan is a tree of nodes, read from bottom to top, with each node passing its result to the node directly above. Each node represents a step in the plan, and one line for each node identifies the operation performed in that step—for example, a scan, join, aggregation, or sort operation. The node also identifies the method used to perform the operation. The method for a scan operation, for example, may be a sequential scan or an index scan. A join operation may perform a hash join or nested loop join.

Following is an explain plan for a simple query. This query finds the number of rows in the contributions table stored at each segment.

```
gpacmin=# EXPLAIN SELECT gp_segment_id, count(*)
                  FROM contributions
                  GROUP BY gp_segment_id;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Gather Motion 2:1  (slice2; segments: 2)  (cost=0.00..4.44 rows=4 width=16)
   ->  HashAggregate  (cost=0.00..3.38 rows=4 width=16)
         Group By: contributions.gp_segment_id
         ->  Redistribute Motion 2:2  (slice1; segments: 2)
                (cost=0.00..2.12 rows=4 width=8)
               Hash Key: contributions.gp_segment_id
               ->  Sequence  (cost=0.00..1.09 rows=4 width=8)
                     ->  Result  (cost=10.00..100.00 rows=50 width=4)
                           ->  Function Scan on gp_partition_expansion
                                 (cost=10.00..100.00 rows=50 width=4)
                     ->  Dynamic Table Scan on contributions (partIndex: 0)
                           (cost=0.00..0.03 rows=4 width=8)
 Settings:  optimizer=on
(10 rows)
```

This plan has seven nodes – Dynamic Table Scan, Function Scan, Result, Sequence, Redistribute Motion, HashAggregate, and finally Gather Motion. Each node contains three cost estimates: cost (in sequential page reads), the number of rows, and the width of the rows.

The cost is a two-part estimate. A cost of 1.0 is equal to one sequential disk page read. The first part of the estimate is the start-up cost, which is the cost of getting the first row. The second estimate is the total cost, the cost of getting all of the rows.

The rows estimate is the number of rows output by the plan node. The number may be lower than the actual number of rows processed or scanned by the plan node, reflecting the estimated selectivity of `WHERE` clause conditions. The total cost assumes that all rows will be retrieved, which may not always be the case (for example, if you use a `LIMIT` clause).

The width estimate is the total width, in bytes, of all the columns output by the plan node.

The cost estimates in a node include the costs of all its child nodes, so the top-most node of the plan, usually a Gather Motion, has the estimated total execution costs for the plan. This is this number that the query planner seeks to minimize.

Scan operators scan through rows in a table to find a set of rows. There are different scan operators for different types of storage. They include the following:

- Seq Scan on heap tables — scans all rows in the table.
- Append-only Scan — scans rows in row-oriented append-only tables.
- Append-only Columnar Scan — scans rows in column-oriented append-only tables.
- Index Scan — traverses a B-tree index to fetch the rows from the table.

- Bitmap Append-only Row-oriented Scan — gathers pointers to rows in an append-only table from an index and sorts by location on disk.
- Dynamic Table Scan — chooses partitions to scan using a partition selection function. The Function Scan node contains the name of the partition selection function, which can be one of the following:
  - `gp_partition_expansion` — selects all partitions in the table. No partitions are eliminated.
  - `gp_partition_selection` — chooses a partition based on an equality expression.
  - `gp_partition_inversion` — chooses partitions based on a range expression.

  The Function Scan node passes the dynamically selected list of partitions to the Result node which is passed to the Sequence node.

Join operators include the following:

- Hash Join – builds a hash table from the smaller table with the join column(s) as hash key. Then scans the larger table, calculating the hash key for the join column(s) and probing the hash table to find the rows with the same hash key. Hash joins are typically the fastest joins in Greenplum Database. The Hash Cond in the explain plan identifies the columns that are joined.
- Nested Loop – iterates through rows in the larger dataset, scanning the rows in the smaller dataset on each iteration. The Nested Loop join requires the broadcast of one of the tables so that all rows in one table can be compared to all rows in the other table. It performs well for small tables or tables that are limited by using an index. It is also used for Cartesian joins and range joins. There are performance implications when using a Nested Loop join with large tables. For plan nodes that contain a Nested Loop join operator, validate the SQL and ensure that the results are what is intended. Set the `enable_nestloop` server configuration parameter to OFF (default) to favor Hash Join.
- Merge Join – sorts both datasets and merges them together. A merge join is fast for pre-ordered data, but is very rare in the real world. To favor Merge Joins over Hash Joins, set the `enable_mergejoin` system configuration parameter to ON.

Some query plan nodes specify motion operations. Motion operations move rows between segments when required to process the query. The node identifies the method used to perform the motion operation. Motion operators include the following:

- Broadcast motion – each segment sends its own, individual rows to all other segments so that every segment instance has a complete local copy of the table. A Broadcast motion may not be as optimal as a Redistribute motion, so the optimizer typically only selects a Broadcast motion for small tables. A Broadcast motion is not acceptable for large tables. In the case where data was not distributed on the join key, a dynamic redistribution of the needed rows from one of the tables to another segment is performed.
- Redistribute motion – each segment rehashes the data and sends the rows to the appropriate segments according to hash key.
- Gather motion – result data from all segments is assembled into a single stream. This is the final operation for most query plans.

Other operators that occur in query plans include the following:

- Materialize – the planner materializes a subselect once so it does not have to repeat the work for each top-level row.
- InitPlan – a pre-query, used in dynamic partition elimination, performed when the values the planner needs to identify partitions to scan are unknown until execution time.
- Sort – sort rows in preparation for another operation requiring ordered rows, such as an Aggregation or Merge Join.
- Group By – groups rows by one or more columns.
- Group/Hash Aggregate – aggregates rows using a hash.
- Append – concatenates data sets, for example when combining rows scanned from partitions in a partitioned table.
- Filter – selects rows using criteria from a `WHERE` clause.

- Limit – limits the number of rows returned.

Optimizing Greenplum Queries

This topic describes Greenplum Database features and programming practices that can be used to enhance system performance in some situations.

To analyze query plans, first identify the plan nodes where the estimated cost to perform the operation is very high. Determine if the estimated number of rows and cost seems reasonable relative to the number of rows for the operation performed.

If using partitioning, validate that partition elimination is achieved. To achieve partition elimination the query predicate (`WHERE` clause) must be the same as the partitioning criteria. Also, the `WHERE` clause must not contain an explicit value and cannot contain a subquery.

Review the execution order of the query plan tree. Review the estimated number of rows. You want the execution order to build on the smaller tables or hash join result and probe with larger tables. Optimally, the largest table is used for the final join or probe to reduce the number of rows being passed up the tree to the topmost plan nodes. If the analysis reveals that the order of execution builds and/or probes is not optimal ensure that database statistics are up to date. Running `ANALYZE` will likely address this and produce an optimal query plan.

Look for evidence of computational skew. Computational skew occurs during query execution when execution of operators such as Hash Aggregate and Hash Join cause uneven execution on the segments. More CPU and memory are used on some segments than others, resulting in less than optimal execution. The cause could be joins, sorts, or aggregations on columns that have low cardinality or non-uniform distributions. You can detect computational skew in the output of the `EXPLAIN ANALYZE` statement for a query. Each node includes a count of the maximum rows processed by any one segment and the average rows processed by all segments. If the maximum row count is much higher than the average, at least one segment has performed much more work than the others and computational skew should be suspected for that operator.

Identify plan nodes where a Sort or Aggregate operation is performed. Hidden inside an Aggregate operation is a Sort. If the Sort or Aggregate operation involves a large number of rows, there is an opportunity to improve query performance. A HashAggregate operation is preferred over Sort and Aggregate operations when a large number of rows are required to be sorted. Usually a Sort operation is chosen by the optimizer due to the SQL construct; that is, due to the way the SQL is written. Most Sort operations can be replaced with a HashAggregate if the query is rewritten. To favor a HashAggregate operation over a Sort and Aggregate operation ensure that the `enable_groupagg` server configuration parameter is set to `ON`.

When an explain plan shows a broadcast motion with a large number of rows, you should attempt to eliminate the broadcast motion. One way to do this is to use the `gp_segments_for_planner` server configuration parameter to increase the cost estimate of the motion so that alternatives are favored. The `gp_segments_for_planner` variable tells the query planner how many primary segments to use in its calculations. The default value is zero, which tells the planner to use the actual number of primary segments in estimates. Increasing the number of primary segments increases the cost of the motion, thereby favoring a redistribute motion over a broadcast motion. For example, setting `gp_segments_for_planner = 100000` tells the planner that there are 100,000 segments. Conversely, to influence the optimizer to broadcast a table and not redistribute it, set `gp_segments_for_planner` to a low number, for example 2.

Greenplum Grouping Extensions

Greenplum Database aggregation extensions to the `GROUP BY` clause can perform some common calculations in the database more efficiently than in application or procedure code:

- `GROUP BY ROLLUP(`*`col1`*`, `*`col2`*`, `*`col3`*`)`
- `GROUP BY CUBE(`*`col1`*`, `*`col2`*`, `*`col3`*`)`

- `GROUP BY GROUPING SETS((col1, col2), (col1, col3))`

A `ROLLUP` grouping creates aggregate subtotals that roll up from the most detailed level to a grand total, following a list of grouping columns (or expressions). `ROLLUP` takes an ordered list of grouping columns, calculates the standard aggregate values specified in the `GROUP BY` clause, then creates progressively higher-level subtotals, moving from right to left through the list. Finally, it creates a grand total.

A `CUBE` grouping creates subtotals for all of the possible combinations of the given list of grouping columns (or expressions). In multidimensional analysis terms, `CUBE` generates all the subtotals that could be calculated for a data cube with the specified dimensions.

You can selectively specify the set of groups that you want to create using a `GROUPING SETS` expression. This allows precise specification across multiple dimensions without computing a whole `ROLLUP` or `CUBE`.

Refer to the *Greenplum Database Reference Guide* for details of these clauses.

Window Functions

Window functions apply an aggregation or ranking function over partitions of the result set—for example, `sum(population) over (partition by city)`. Window functions are powerful and, because they do all of the work in the database, they have performance advantages over front-end tools that produce similar results by retrieving detail rows from the database and reprocessing them.

- The `row_number()` window function produces row numbers for the rows in a partition, for example, `row_number() over (order by id)`.
- When a query plan indicates that a table is scanned in more than one operation, you may be able to use window functions to reduce the number of scans.
- It is often possible to eliminate self joins by using window functions.

Chapter 12

High Availability

Greenplum Database supports highly available, fault-tolerant database services when you enable and properly configure Greenplum high availability features. To guarantee a required level of service, each component must have a standby ready to take its place if it should fail.

Disk Storage

With the Greenplum Database "shared-nothing" MPP architecture, the master host and segment hosts each have their own dedicated memory and disk storage, and each master or segment instance has its own independent data directory. For both reliability and high performance, Pivotal recommends a hardware RAID storage solution with from 8 to 24 disks. A larger number of disks improves I/O throughput when using RAID 5 (or 6) because striping increases parallel disk I/O. The RAID controller can continue to function with a failed disk because it saves parity data on each disk in a way that it can reconstruct the data on any failed member of the array. If a hot spare is configured (or an operator replaces the failed disk with a new one) the controller rebuilds the failed disk automatically.

RAID 1 exactly mirrors disks, so if a disk fails, a replacement is immediately available with performance equivalent to that before the failure. With RAID 5 each I/O for data on the failed array member must be reconstructed from data on the remaining active drives until the replacement disk is rebuilt, so there is a temporary performance degradation. If the Greenplum master and segments are mirrored, you can switch any affected Greenplum instances to their mirrors during the rebuild to maintain acceptable performance.

A RAID disk array can still be a single point of failure, for example, if the entire RAID volume fails. At the hardware level, you can protect against a disk array failure by mirroring the array, using either host operating system mirroring or RAID controller mirroring, if supported.

It is important to regularly monitor available disk space on each segment host. Query the `gp_disk_free` external table in the `gptoolkit` schema to view disk space available on the segments. This view runs the Linux `df` command. Be sure to check that there is sufficient disk space before performing operations that consume large amounts of disk, such as copying a large table.

See `gp_toolkit.gp_disk_free` in the *Greenplum Database Reference Guide*.

Best Practices

- Use a hardware RAID storage solution with 8 to 24 disks.
- Use RAID 1, 5, or 6 so that the disk array can tolerate a failed disk.
- Configure a hot spare in the disk array to allow rebuild to begin automatically when disk failure is detected.
- Protect against failure of the entire disk array and degradation during rebuilds by mirroring the RAID volume.
- Monitor disk utilization regularly and add additional space when needed.
- Monitor segment skew to ensure that data is distributed evenly and storage is consumed evenly at all segments.

Master Mirroring

The Greenplum Database master instance is clients' single point of access to the system. The master instance stores the global system catalog, the set of system tables that store metadata about the database instance, but no user data. If an unmirrored master instance fails or becomes inaccessible, the Greenplum instance is effectively off-line, since the entry point to the system has been lost. For this reason, a standby master must be ready to take over if the primary master fails.

Master mirroring uses two processes, a sender on the active master host and a receiver on the mirror host, to synchronize the mirror with the master. As changes are applied to the master system catalogs, the active master streams its write-ahead log (WAL) to the mirror so that each transaction applied on the master is applied on the mirror.

The mirror is a *warm standby*. If the primary master fails, switching to the standby requires an administrative user to run the `gpactivatestandby` utility on the standby host so that it begins to accept client connections. Clients must reconnect to the new master and will lose any work that was not committed when the primary failed.

See "Enabling High Availability Features" in the *Greenplum Database Administrator Guide* for more information.

Best Practices

- Set up a standby master instance—a *mirror*—to take over if the primary master fails.
- The standby can be on the same host or on a different host, but it is best practice to place it on a different host from the primary master to protect against host failure.
- Plan how to switch clients to the new master instance when a failure occurs, for example, by updating the master address in DNS.
- Set up monitoring to send notifications in a system monitoring application or by email when the primary fails.

Segment Mirroring

Greenplum Database segment instances each store and manage a portion of the database data, with coordination from the master instance. If any unmirrored segment fails, the database may have to be shutdown and recovered, and transactions occurring after the most recent backup could be lost. Mirroring segments is, therefore, an essential element of a high availability solution.

A segment mirror is a hot standby for a primary segment. Greenplum Database detects when a segment is unavailable and automatically activates the mirror. During normal operation, when the primary segment instance is active, data is replicated from the primary to the mirror in two ways:

- The transaction commit log is replicated from the primary to the mirror before the transaction is committed. This ensures that if the mirror is activated, the changes made by the last successful transaction at the primary are present at the mirror. When the mirror is activated, transactions in the log are applied to tables in the mirror.
- Second, segment mirroring uses physical file replication to update heap tables. Greenplum Server stores table data on disk as fixed-size blocks packed with tuples. To optimize disk I/O, blocks are cached in memory until the cache fills and some blocks must be evicted to make room for newly updated blocks. When a block is evicted from the cache it is written to disk and replicated over the network to the mirror. Because of the caching mechanism, table updates at the mirror can lag behind the primary. However, because the transaction log is also replicated, the mirror remains consistent with the primary. If the mirror is activated, the activation process updates the tables with any unapplied changes in the transaction commit log.

When the acting primary is unable to access its mirror, replication stops and state of the primary changes to "Change Tracking." The primary saves changes that have not been replicated to the mirror in a system table to be replicated to the mirror when it is back on-line.

The master automatically detects segment failures and activates the mirror. Transactions in progress at the time of failure are restarted using the new primary. Depending on how mirrors are deployed on the hosts, the database system may be unbalanced until the original primary segment is recovered. For example, if each segment host has four primary segments and four mirror segments, and a mirror is activated on one host, that host will have five active primary segments. Queries are not complete until the last segment has finished its work, so performance can be degraded until the balance is restored by recovering the original primary.

Administrators perform the recovery while Greenplum Database is up and running by running the `gprecoverseg` 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.

It is important to reserve enough memory and CPU resources on segment hosts to allow for increased activity from mirrors that assume the primary role during a failure. The formulas provided in *Configuring Memory for Greenplum Database* for configuring segment host memory include a factor for the maximum number of primary hosts on any one segment during a failure. The arrangement of mirrors on the segment hosts affects this factor and how the system will respond during a failure. See *Segment Mirroring Configuration* for a discussion of segment mirroring options.

Best Practices

- Set up mirrors for all segments.
- Locate primary segments and their mirrors on different hosts to protect against host failure.
- Mirrors can be on a separate set of hosts or co-located on hosts with primary segments.
- Set up monitoring to send notifications in a system monitoring application or by email when a primary segment fails.

- Recover failed segments promptly, using the `gprecoverseg` utility, to restore redundancy and return the system to optimal balance.

Dual Clusters

For some use cases, an additional level of redundancy can be provided by maintaining two Greenplum Database clusters that store the same data. The decision to implement dual clusters should be made with business requirements in mind.

There are two recommended methods for keeping the data synchronized in a dual cluster configuration. The first method is called Dual ETL. ETL (extract, transform, and load) is the common data warehousing process of cleansing, transforming, validating, and loading data into a data warehouse. With Dual ETL, the ETL processes are performed twice, in parallel on each cluster, and validated each time. Dual ETL provides for a complete standby cluster with the same data. It also provides the capability to query the data on both clusters, doubling the processing throughput. The application can take advantage of both clusters as needed and also ensure that the ETL is successful and validated on both sides.

The second mechanism for maintaining dual clusters is backup and restore. The data is backedup on the primary cluster, then the backup is replicated to and restored on the second cluster. The backup and restore mechanism has higher latency than Dual ETL, but requires less application logic to be developed. Backup and restore is ideal for use cases where data modifications and ETL are done daily or less frequently.

Best Practices

- Consider a Dual Cluster configuration to provide an additional level of redundancy and additional query processing throughput.

Backup and Restore

Backups are recommended for Greenplum Database databases unless the data in the database can be easily and cleanly regenerated from source data. Backups protect from operational, software, or hardware errors.

The `gpcrondump` utility makes backups in parallel across the segments, so that backups scale as the cluster grows in hardware size.

Incremental backups can significantly reduce backup sizes in some cases. An incremental backup saves all heap tables, but only the append-optimized and column-oriented partitions that have changed since the previous backup. When the database has large fact tables with many partitions and modifications are confined to one or a few partitions in a time period, incremental backups can save a large amount of disk space. Conversely, a database with large, unpartitioned fact tables is a poor application for incremental backups.

A backup strategy must consider where the backups will be written and where they will be stored. Backups can be taken to the local cluster disks, but they should not be stored there permanently. If the database and its backup are on the same storage, they can be lost simultaneously. The backup also occupies space that could be used for database storage or operations. After performing a local backup, the files should be copied to a safe, off-cluster location.

An alternative is to back up directly to an NFS mount. If each host in the cluster has an NFS mount, ghe backups can be written directly to NFS storage. A scale-out NFS solution is recommended to ensure that backups do not bottleneck on the IO throughput of the NFS device. Dell EMC Isilon is an example of this type of solution and can scale alongside the Greenplum cluster.

Finally, through native API integration, Greenplum Database can stream backups directly to Dell EMC Data Domain or Veritas NetBackup enterprise backup platforms.

Best Practices

- Back up Greenplum databases regularly unless the data is easily restored from sources.
- Use the `gpcrondump -s`, `-S`, `-t`, or `-T` options to specify only the schema and tables that need to be backed up. See the *gpcrondump* reference in the *Greenplum Database Utility Reference Guide* for more information.
- Back up one schema at a time to reduce the length of time the `pg_class` table is locked.

  At the start of the backup, `gpcrondump` places an EXCLUSIVE lock on the `pg_class` table, which prevents creating, altering, or dropping tables. This lock is held while `gpcrondump` computes the set of tables to back up and then places SHARED ACCESS locks on the tables. Once the tables are locked and backup processes started on the segments, the EXCLUSIVE lock on `pg_class` is released. In a database with many objects the duration of the EXCLUSIVE lock may be disruptive. Backing up one schema at a time can reduce the duration of the EXCLUSIVE lock. Backups with fewer tables are also more efficient for selectively restoring schemas and tables, since `gpdbrestore` does not have to search through the entire database. See *Backup Process and Locks* for more information.
- Use incremental backups when heap tables are relatively small and few append-optimized or column-oriented partitions are modified between backups.
- If backups are saved to local cluster storage, move the files to a safe, off-cluster location when the backup is complete. Backup files and database files that reside on the same storage can be lost simultaneously.
- If your operating system supports direct I/O, set the `gp_backup_directIO` configuration parameter to reduce CPU usage during backups. See *Using Direct I/O* for more information.
- If backups are saved to NFS mounts, use a scale-out NFS solution such as Dell EMC Isilon to prevent IO bottlenecks.

- Consider using Pivotal Greenplum Database integration to stream backups to the Dell EMC Data Domain or Veritas NetBackup enterprise backup platforms.

Detecting Failed Master and Segment Instances

Recovering from system failures requires intervention from a system administrator, even when the system detects a failure and activates a standby for the failed component. In each case, the failed component must be replaced or recovered to restore full redundancy. Until the failed component is recovered, the active component lacks a standby, and the system may not be executing optimally. For these reasons, it is important to perform recovery operations promptly. Constant system monitoring and automated fault notifications through SNMP and email ensure that administrators are aware of failures that demand their attention.

The Greenplum Database server `ftsprobe` subprocess handles fault detection. `ftsprobe` connects to and scans all segments and database processes at intervals that you can configure with the `gp_fts_probe_interval` configuration parameter. If `ftsprobe` cannot connect to a segment, it marks the segment “down” in the Greenplum Database system catalog. The segment remains down until an administrator runs the `gprecoverseg` recovery utility.

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. See "Using SNMP with a Greenplum Database System" in the *Pivotal Greenplum Database Administrator Guide* for instructions to set up SNMP.

Best Practices

- Run the `gpstate` utility to see the overall state of the Greenplum system.
- Configure Greenplum Database to send SNMP notifications to your network monitor.
- Set up email notification in the `$MASTER_DATA_DIRECTORY/postgresql.conf` configuration file so that the Greenplum system can email administrators when a critical issue is detected.

Additional Information

Greenplum Database Administrator Guide:

- Monitoring a Greenplum System
- Recovering a Failed Segment
- Using SNMP with a Greenplum System
- Enabling Email Notifications

Greenplum Database Utility Guide:

- `gpstate`—view state of the Greenplum system
- `gprecoverseg`—recover a failed segment
- `gpactivatestandby`—make the standby master the active master

RDBMS MIB Specification

Segment Mirroring Configuration

Segment mirroring allows database queries to fail over to a backup segment if the primary segment fails or becomes unavailable. Pivotal requires mirroring for supported production Greenplum Database systems.

A primary segment and its mirror must be on different hosts to ensure high availability. Each host in a Greenplum Database system has the same number of primary segments and mirror segments. Multi-homed hosts should have the same numbers of primary and mirror segments on each interface. This ensures that segment hosts and network resources are equally loaded when all primary segments are operational and brings the most resources to bear on query processing.

When a segment becomes unavailable, its mirror segment on another host becomes the active primary and processing continues. The additional load on the host creates skew and degrades performance, but should allow the system to continue. A database query is not complete until all segments return results, so a single host with an additional active primary segment has the same effect as adding an additional primary segment to every host in the cluster.

The least amount of performance degradation in a failover scenario occurs when no host has more than one mirror assuming the primary role. If multiple segments or hosts fail, the amount of degradation is determined by the host or hosts with the largest number of mirrors assuming the primary role. Spreading a host's mirrors across the remaining hosts minimizes degradation when any single host fails.

It is important, too, to consider the cluster's tolerance for multiple host failures and how to maintain a mirror configuration when expanding the cluster by adding hosts. There is no mirror configuration that is ideal for every situation.

You can allow Greenplum Database to arrange mirrors on the hosts in the cluster using one of two standard configurations, or you can design your own mirroring configuration.

The two standard mirroring arrangements are *group mirroring* and *spread mirroring*:

- **Group mirroring** — Each host mirrors another host's primary segments. This is the default for `gpinitsystem` and `gpaddmirrors`.
- **Spread mirroring** — Mirrors are spread across the available hosts. This requires that the number of hosts in the cluster is greater than the number of segments per host.

You can design a custom mirroring configuration and use the Greenplum `gpaddmirrors` or `gpmovemirrors` utilities to set up the configuration.

Block mirroring is a custom mirror configuration that divides hosts in the cluster into equally sized blocks and distributes mirrors evenly to hosts within the block. If a primary segment fails, its mirror on another host within the same block becomes the active primary. If a segment host fails, mirror segments on each of the other hosts in the block become active.

The following sections compare the group, spread, and block mirroring configurations.

Group Mirroring

Group mirroring is easiest to set up and is the default Greenplum mirroring configuration. It is least expensive to expand, since it can be done by adding as few as two hosts. There is no need to move mirrors after expansion to maintain a consistent mirror configuration.

The following diagram shows a group mirroring configuration with eight primary segments on four hosts.

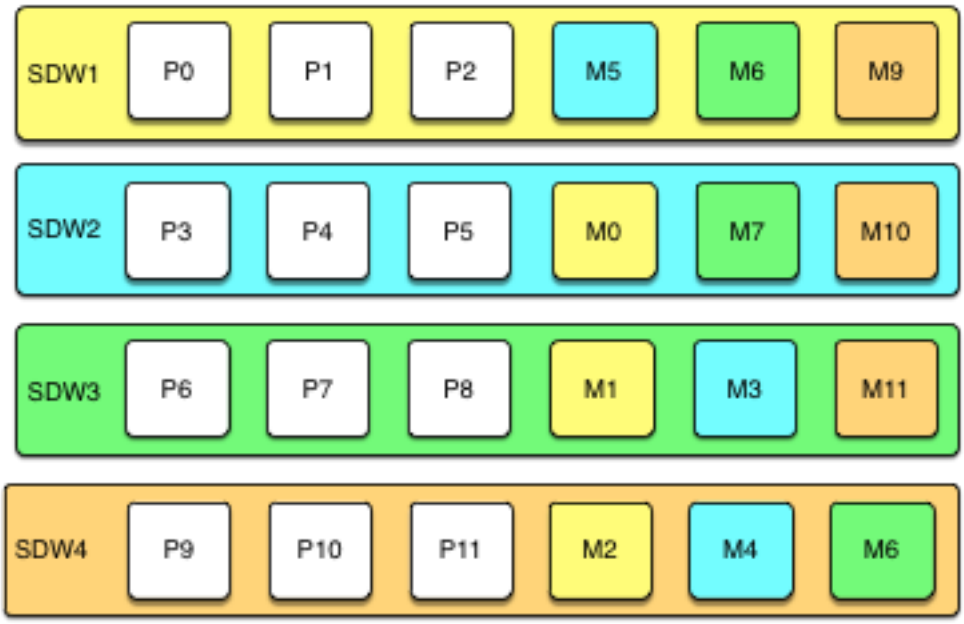
Unless both the primary and mirror of the same segment instance fail, up to half of your hosts can fail and the cluster will continue to run as long as resources (CPU, memory, and IO) are sufficient to meet the needs.

Any host failure will degrade performance by half or more because the host with the mirrors will have twice the number of active primaries. If your resource utilization is normally greater than 50%, you will have to adjust your workload until the failed host is recovered or replaced. If you normally run at less than 50% resource utilization the cluster can continue to operate at a degraded level of performance until the failure is corrected.

Spread Mirroring

With spread mirroring, mirrors for each host's primary segments are spread across as many hosts as there are segments per host. Spread mirroring is easy to set up when the cluster is initialized, but requires that the cluster have at least one more host than there are segments per host.

The following diagram shows the spread mirroring configuration for a cluster with three primaries on four hosts.



Expanding a cluster with spread mirroring requires more planning and may take more time. You must either add a set of hosts equal to the number of primaries per host plus one, or you can add two nodes in a group mirroring configuration and, when the expansion is complete, move mirrors to recreate the spread mirror configuration.

Spread mirroring has the least performance impact for a single failed host because each host's mirrors are spread across the maximum number of hosts. Load is increased by 1/*Nth*, where *N* is the number of primaries per host. Spread mirroring is, however, the most likely configuration to suffer catastrophic failure if two or more hosts fail simultaneously.

Block Mirroring

With block mirroring, nodes are divided into blocks, for example a block of four or eight hosts, and the mirrors for segments on each host are placed on other hosts within the block. Depending on the number of hosts in the block and the number of primary segments per host, each host maintains more than one mirror for each other host's segments.

The following diagram shows a single block mirroring configuration for a block of four hosts, each with eight primary segments:

If there are eight hosts, an additional four-host block is added with the mirrors for primary segments 32 through 63 set up in the same pattern.

A cluster with block mirroring is easy to expand because each block is a self-contained primary mirror group. The cluster is expanded by adding one or more blocks. There is no need to move mirrors after expansion to maintain a consistent mirror setup. This configuration is able to survive multiple host failures as long as the failed hosts are in different blocks.

Because each host in a block has multiple mirror instances for each other host in the block, block mirroring has a higher performance impact for host failures than spread mirroring, but a lower impact than group mirroring. The expected performance impact varies by block size and primary segments per node. As with group mirroring, if the resources are available, performance will be negatively impacted but the cluster will remain available. If resources are insufficient to accommodate the added load you must reduce the workload until the failed node is replaced.

Implementing Block Mirroring

Block mirroring is not one of the automatic options Greenplum Database offers when you set up or expand a cluster. To use it, you must create your own configuration.

For a new Greenplum system, you can initialize the cluster without mirrors, and then run `gpaddmirrors -i mirror_config_file` with a custom mirror configuration file to create the mirrors for each block. You must create the file system locations for the mirror segments before you run `gpaddmirrors`. See the `gpadmirrors` reference page in the *Greenplum Database Management Utility Guide* for details.

If you expand a system that has block mirroring or you want to implement block mirroring at the same time you expand a cluster, it is recommended that you complete the expansion first, using the default grouping mirror configuration, and then use the `gpmovemirrors` utility to move mirrors into the block configuration.

To implement block mirroring with an existing system that has a different mirroring scheme, you must first determine the desired location for each mirror according to your block configuration, and then determine which of the existing mirrors must be relocated. Follow these steps:

1. Run the following query to find the current locations of the primary and mirror segments:

   ```
   SELECT dbid, content, address, port,
      replication_port, fselocation as datadir
      FROM gp_segment_configuration, pg_filespace_entry
      WHERE dbid=fsedbid AND content > -1
      ORDER BY dbid;
   ```

   The `gp_segment_configuration`, `pg_filespace`, and `pg_filespace_entry` system catalog tables contain the current segment configuration.
2. Create a list with the current mirror location and the desired block mirroring location, then remove any mirrors from the list that are already on the correct host.
3. Create an input file for the `gpmovemirrors` utility with an entry for each mirror that must be moved.

   The `gpmovemirrors` input file has the following format:

   ```
   filespaceOrder=[filespace1_fsname[:filespace2_fsname:...]
   old_address:port:fselocation
    new_address:port:replication_port:fselocation[:fselocation:...]
   ```

   The first non-comment line must be a line beginning with `filespaceOrder=`. Do not include the default `pg_system` filespace in the list of filespaces. Leave the list empty if you are using only the `pg_system` filespace.

   The following example `gpmovemirrors` input file specifies three mirror segments to move.

   ```
   filespaceOrder=
   sdw2:50001:/data2/mirror/gpseg1 sdw3:50000:51000:/data/mirror/gpseg1
   sdw2:50001:/data2/mirror/gpseg2 sdw4:50000:51000:/data/mirror/gpseg2
   sdw3:50001:/data2/mirror/gpseg3 sdw1:50000:51000:/data/mirror/gpseg3
   ```
4. Run `gpmovemirrors` with a command like the following:

   ```
   gpmovemirrors -i mirror_config_file
   ```

The `gpmovemirrors` utility validates the input file, calls `gp_recoverseg` to relocate each specified mirror, and removes the original mirror. It creates a backout configuration file which can be used as input to `gpmovemirrors` to undo the changes that were made. The backout file has the same name as the input file, with the suffix `_backout_timestamp` added.

See the *Greenplum Database Management Utility Reference* for complete information about the `gpmovemirrors` utility.