PIVOTAL GREENPLUM For SSA

Lab Guide

Date 5/3/2017

Greenplum Administrator Lab Guide – Table Of Contents

GPDB Administrator – Lab 1

Greenplum Product Overview

Purpose

Review your understanding of Greenplum Database concepts, architecture, and components.

Success Learning Skills

- Read and follow the instructions in each lab carefully. While we have taken every effort to make the labs free of errors, we're sure that we missed some. So, please, when you to the labs, take a little time to read ahead a little to see if there is a critical task in the next step that explains a dilemma you may be facing.
- Learn to go to the documentation to find answers. (This is particularly valuable in this first section because we didn't necessarily cover every topic that provides the answers to the questions below.)

Main Documentation Page

http://gpdb.docs.pivotal.io/43110/common/welcome.html

- If you have any particular issues that you are attempting to resolve, examining the Knowledge Base might be useful.

Master Knowledge Base:

https://discuss.pivotal.io/hc/en-us

Greenplum Knowledge Base:

https://discuss.pivotal.io/hc/en-us/categories/200072608

Tasks:

Provide answers to the following review questions.

Greenplum Product Review

1. What is the master instance and what is its purpose?
2. Name three clients that can connect to a Greenplum database.
3. What is the role of the segment instances in a Greenplum Database system?
4. What is the purpose of mirroring in a Greenplum Database?
5. What is the Interconnect?
6. Where is table data stored in a Greenplum Database system?
7. How are system management tasks performed in a Greenplum Database?
8. How is data distributed in a Greenplum Database?
9. Which redundant components should you deploy in order to have a Greenplum Database system running without a single point of failure?

GPDB Administrator – Lab 2

System Preparation and Verification

Purpose

Perform a series of exercises necessary to prepare the cluster for the installation of the Greenplum Database.

If you have any particular issues that you are attempting to resolve, examine the GPDB Knowledge Base or the documentation.

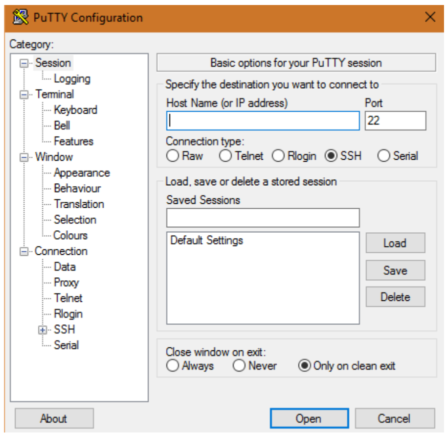
Estimated completion time: 45 minutes

Prepare and Install Greenplum on the master server (mdw)

Set system parameters

1. Use the Putty tool to connect to **mdw.**

   Use the hostname (mdw) or IP address (172.16.1.11) of the Master Server



Login using the following credentials:

Login: root
Password: Piv0tal

2. Add the following lines to your `/etc/sysctl.conf` file.

```
        net.core.rmem_max = 2097152
        net.core.wmem_max = 2097152
```

3. Take a closer look at `/etc/sysctl.conf`.
   - `kernel.shmmax` is the maximum size of a single shared memory segment set in bytes.
   - `kernel.shmmin` is the minimum size of shared memory segment in bytes.
   - `kernel.shmall` is the total amount of shared memory available in bytes or pages.

Find more info about these settings here.

Ensure that any changes that you have made to `/etc/sysctl.conf` take effect immediately with `sysctl -p`. This command would normally accept a file argument, but the `/etc/sysctl.conf` file is the default. The output should look like this:

- **`kernel.shmmax = 500000000`**
- **`kernel.shmmin = 4096`**
- **`kernel.shmall = 4000000000`**
- **`kernel.sem = 250 512000 100 2048`**
- **`kernel.sysrq = 1`**
- **`kernel.core_uses_pid = 1`**
- **`kernel.msgmnb = 65536`**
- **`kernel.msgmax = 65536`**
- **`kernel.msgmni = 2048`**
- **`net.ipv4.tcp_syncookies = 1`**
- **`net.ipv4.conf.default.accept_source_route = 0`**
- **`net.ipv4.tcp_tw_recycle = 1`**
- **`net.ipv4.tcp_max_syn_backlog = 4096`**
- **`net.ipv4.conf.all.arp_filter = 1`**
- **`net.ipv4.ip_local_port_range = 1025 65535`**
- **`net.ipv4.ip_forward = 0`**
- **`net.core.netdev_max_backlog = 10000`**
- **`net.core.rmem_max = 2097152`**
- **`net.core.wmem_max = 2097152`**
- **`vm.overcommit_memory = 2`**

In particular, confirm that `net.core.rmem_max` and `net.core.wmem_max` are correct.

Verify security limit settings

1. Open `/etc/security/limits.conf` and verify that the following settings are set:

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

Modify /etc/hosts and push to all servers

1. Using the editor of your choice, edit the `/etc/hosts` file and add the server names and IP addresses of all servers participating in your Greenplum environment.

```
172.16.1.11     mdw     # Master Server
172.16.1.14     smdw    # Standby Master Server
172.16.1.12     sdw1    # Segment Server One
172.16.1.13     sdw2    # Segment Server Two
```

   Adding these values to your `/etc/hosts` file will enable you to directly connect to these machines by simply typing the short alias name, such as `smdw` rather than the longer command `ssh <user>@174.16.1.14`. This is only possible because we have set the password for both the `root` user and the `gpadmin` user to be the same on all machines. This is not typical in a production environment.

2. Using the `scp` command, copy the file `/etc/hosts` from the master server `mdw` to `smdw`, `smdw1`, and `smdw2`. Recall that the root password for each segment server is `Piv0tal`.

```
scp /etc/hosts smdw:/etc/hosts
scp /etc/hosts sdw1:/etc/hosts
scp /etc/hosts sdw2:/etc/hosts
```

Install the Greenplum Database software on the Master server

1. Typically, you would download or copy the Greenplum Database installer file to the system that will be the Greenplum Master host.

   In this lab environment, the installer was pre-loaded in the `/rawdata/Binaries` directory.

   Make sure you are on `mdw` for this step.

   Change to the `/rawdata/Binaries` directory and display the contents of the directory.

   ```
   cd /rawdata/Binaries
   ls -l
   ```

   You should see two zip files, one for the Greenplum Database and one for the Greenplum Command Center.

   The `greenplum-db-4.3.11.3-rhel5-x86_64.zip` file contains the Greenplum binary that you will use to install the Greenplum Database.

   The `greenplum-cc-web-3.0.1-LINUX-x86_64.zip` file contains the Greenplum Command Center that you will use in a later lab to install the Greenplum Command Center.

2. Unzip `greenplum-db-4.3.11.3-rhel5-x86_64.zip` and view the contents of the directory

   ```
   unzip greenplum-db-4.3.11.3-rhel5-x86_64.zip
   ```

   The `greenplum-db-4.3.11.3-rhel5-x86_64.bin` file is the binary that you will use to install the Greenplum Database.

3. Launch the Greenplum Database installer using `/bin/bash`.

   ```
   /bin/bash greenplum-db-4.3.11.3-rhel5-x86_64.bin
   ```

   The Greenplum Database software has been installed on the Master server (`mdw`) and a symbolic link named `greenplum-db` was created at the same level as your version-specific Greenplum Database installation directory (`/usr/local/greenplum-db-`

4.3.11.3). The symbolic link is used to facilitate patch maintenance and upgrades between versions. The installed location is referred to as $GPHOME.

Install and configure Greenplum on the rest of the servers in the cluster

Source the greenplum_path.sh file

1. Ensure that you are still in the /rawdata/Binaries directory on mdw. Source greenplum_path.sh to ensure that all Greenplum commands are available.

```
source /usr/local/greenplum-db/greenplum_path.sh
```

Create a hostfile_exkeys file

1. Create a file called hostfile_exkeys. Add the hostnames of each server in the Greenplum cluster to this file.

```
mdw
smdw
sdw1
sdw2
```

Use the gpseginstall utility

1. Execute the gpseginstall utility using the -f option to use the hostfile_exkeys you just created. The -u option will create the Greenplum administrative user in all servers. You may or may not be asked for the password for each server. The -p option sets the password on each server for the gpadmin user. Find more about gpseginstall in the documentation.

   The next command may take several minutes. Please be patient.

The `gpseginstall` utility may prompt you to confirm the password for each of the servers on the Greenplum cluster. If prompted you must enter the password for each server in the cluster, except the master.

```
gpseginstall -f hostfile_exkeys -u gpadmin -p changeme
```

When `gpseginstall` finishes with no errors you will see a *SUCCESS* message.

Verify that all servers in the cluster are accessible

1. Verify that all servers in the cluster are accessible and have their own copy of the Greenplum software installed. Use the `gpssh` command to accomplish this task.

   The `gpssh` utility allows you to connect to multiple servers in a single command. Learn more about `gpssh` from the documentation

   The following command **MUST** be entered all on one line.

   ```
   gpssh -f hostfile_exkeys -e 'cd /usr/local/greenplum-
   db/.; ls --classify $GPHOME'
   ```

   The output should clearly show all four servers (`mdw`, `smdw`, `sdw1`, and `sdw2`) and all of the output for each server should contain the same elements.

   ```
   bin/   docs/   ext/   greenplum_path.sh  lib/            sbin/
   demo/  etc/    GPDB-LICENSE.txt*  include/      LICENSE.thirdparty*
   share/
   ```

Create the Data Storage Areas for the Greenplum Database

In this step you will create *directories* for data storage areas on `mdw`, `smdw`, `sdw1` and `sdw2`

Create directories for data storage areas on `mdw`, `smdw`, `sdw1`, and `sdw2`.

Every Greenplum Database Master (`mdw`), Standby Master (`smdw`) and segment instance (`sdw1` and `sdw2`) has a designated storage area on disk that is called the *data directory location*. This is the file system location where the directories that store master or segment instance data will be created. The Master server needs a data storage location for the *master data directory*. If the cluster includes a Standby Master server it also needs a data storage location for a copy of the *master data directory*. Each segment server needs a data directory storage location for its *primary* segment, and another for its *mirror* segment.

1. On the Master server (`mdw`) change to the `/data` directory and create a sub-directory named `master`. This will be your master data storage area.
2. Change the owner/group permissions of the new directory.

```
cd /data
mkdir master
```

```
chown gpadmin:gpadmin /data/master
```

3. Confirm that the `/master` directory was created in `/data` and that the *owner/group* is *gpadmin* using the Linux `ls` command.

```
ls -l
```

The resulting output should look similar to the following:

```
total 4
drwxr-xr-x 2 gpadmin gpadmin 4096 Feb  1 20:50 master
```

4. Using `gpssh` create the `master data directory` storage area on the Standby Master server.

   The `gpssh` command allows us to connect to any one or more of the servers in the cluster. In our example, we are only connecting to one server. Later in the labs, you will encounter a situation where we use `gpssh` to automatically connect to **ALL** the servers in the cluster.

```
gpssh -h smdw -e 'mkdir /data/master'
mkdir /data/master
```

5. Change the ownership of the master data directory storage area on the Standby Master (`smdw`) to `gpadmin`.

```
gpssh -h smdw -e 'chown gpadmin:gpadmin /data/master'
chown gpadmin:gpadmin /data/master
```

6. Change to the `/rawdata/Binaries` directory on the Master server. You can check where you are by using the Linux `pwd` command.

```
cd /rawdata/Binaries/
```

7. Use the editor of your choice to create a file called `hostfile_gpssh_segonly`. Add the hostnames of each segment server in the Greenplum cluster to this file.

```
sdw1
sdw2
```

   Remember to save your file and exit the editor

8. Use the Linux `cat` command to display the contents of the file.

```
cat hostfile_gpssh_segonly
```

   The results should show these elements.

```
sdw1
sdw2
```

9. Using `gpssh` create the `/primary` data directory storage area and change the `owner/group` permissions to `gpadmin:gpadmin` on Segment Server One (`sdw1`) and Segment Server Two (`*sdw2*`) using the `hostfile_gpssh_segonly` file.

   This is a very long command and should all go on one line

```
gpssh -f hostfile_gpssh_segonly -e 'mkdir
/data/primary; chown gpadmin:gpadmin /data/primary'
```

10. Using `gpssh` change the permissions on the directory `/loaddata` in the `sdw1` server. This directory will be used in future labs.

```
gpssh -h sdw1 -e 'chown -R gpadmin:gpadmin /loaddata'
chown -R gpadmin:gpadmin /loaddata
```

   Good job! You are now ready to synchronize the system clocks!

Synchronize System Clocks

Greenplum recommends using NTP (Network Time Protocol) to synchronize the system clocks on all servers that comprise your Greenplum Database cluster.

NTP on the segment servers should be configured to use the master server as the primary time source and the standby master server as the secondary time source. On the master and standby master server, configure NTP to point to the preferred time server.

Configure NTP on mdw, smdw, sdw1 and sdw2

1. On the Master server (`mdw`) use the editor of your choice to review the `/etc/ntp.conf` file. You will need to *modify* the contents of this file to enable NTP. Modify the four *server* references by commenting them out. Then add the server IP address as shown (**make sure that it corresponds to the IP address of your Master server `mdw`**).

   Note: Do not copy the - or + in the `diff`. The green highlighted lines denote something to add to the file. The red is something you need to take away.

   ```
   # Use public servers from the pool.ntp.org project.
   # Please consider joining the pool (http://www.pool.ntp.org/join.html).
   - server 0.centos.pool.ntp.org iburst
   + #server 0.centos.pool.ntp.org iburst
   - server 1.centos.pool.ntp.org iburst
   + #server 1.centos.pool.ntp.org iburst
   - server 2.centos.pool.ntp.org iburst
   + #server 2.centos.pool.ntp.org iburst
   - server 3.centos.pool.ntp.org iburst
   + #server 3.centos.pool.ntp.org iburst

   + server 172.16.1.11

   #broadcast 192.168.1.255 autokey        # broadcast server
   #broadcastclient                        # broadcast client
   #broadcast 224.0.1.1 autokey            # multicast server
   ```

   Save your changes and exit.

2. Connect to Segment Server One (`sdw1`) using `ssh` from the current terminal window.

   ```
   ssh sdw1
   ```

3. On Segment Server One (`sdw1`) use the editor of your choice to review the `/etc/ntp.conf` file. Modify the same four server references by commenting them out and add the two server lines as shown:

   ```
   # Use public servers from the pool.ntp.org project.
   # Please consider joining the pool (http://www.pool.ntp.org/join.html).
   - server 0.centos.pool.ntp.org iburst
   ```

```
+ #server 0.centos.pool.ntp.org iburst
- server 1.centos.pool.ntp.org iburst
+ #server 1.centos.pool.ntp.org iburst
- server 2.centos.pool.ntp.org iburst
+ #server 2.centos.pool.ntp.org iburst
- server 3.centos.pool.ntp.org iburst
+ #server 3.centos.pool.ntp.org iburst

+ server mdw prefer
+ server smdw

#broadcast 192.168.1.255 autokey        # broadcast server
#broadcastclient                        # broadcast client
#broadcast 224.0.1.1 autokey            # multicast server
```

Save your changes and exit.

4. Using `gpscp` copy the file `/etc/ntp.conf` from Segment Server One (`sdw1`) to Segment Server Two (`sdw2`). Learn more about this utility from the documentation.

   Remember to `source greenplum_path.sh` to make the `gpscp` command available.

```
source /usr/local/greenplum-db/greenplum_path.sh
gpscp -h sdw2 /etc/ntp.conf =:/etc/ntp.conf
```

5. Connect to the Standby Master Server (`smdw`) using `ssh` from the current terminal window.

```
ssh smdw
```

6. On the Standby Master Server (`smdw`) use the editor of your choice to review the `*/etc/ntp.conf*` file. Modify the four server references by commenting them out and add the two server lines as shown:

```
# Use public servers from the pool.ntp.org project.
# Please consider joining the pool (http://www.pool.ntp.org/join.html).
- server 0.centos.pool.ntp.org iburst
+ #server 0.centos.pool.ntp.org iburst
- server 1.centos.pool.ntp.org iburst
+ #server 1.centos.pool.ntp.org iburst
- server 2.centos.pool.ntp.org iburst
+ #server 2.centos.pool.ntp.org iburst
- server 3.centos.pool.ntp.org iburst
+ #server 3.centos.pool.ntp.org iburst

+ server mdw prefer
+ server 172.16.1.14

#broadcast 192.168.1.255 autokey        # broadcast server
#broadcastclient                        # broadcast client
#broadcast 224.0.1.1 autokey            # multicast server
```

Save your changes and exit.

Exit from the Standby Master Server (`smdw`).

Exit from the Segment Server One (`sdw1`) segment server.

7. Synchronize the system clocks on all Greenplum hosts. This effectively starts or resets the `ntpd` service on all servers within the cluster.

```
gpssh -f hostfile_exkeys -v -e 'ntpd'
```

8. Verify the `ntpd` service is executing on all hosts in the cluster.

   Your output will be slightly different as the process ID's for the `ntpd` processes will be different.

```
gpssh -f hostfile_exkeys -e 'pgrep ntp'
```

   The output should look something like this:

```
[ mdw] pgrep ntp
[ mdw] 1279
[sdw1] pgrep ntp
[sdw1] 1224
[sdw2] pgrep ntp
[sdw2] 1173
[smdw] pgrep ntp
[smdw] 1221
```

9. Change to the *root* directory on the Master Server.

```
cd
```

Perform System Verification Tests

The following tests should be executed prior to initializing your Greenplum Database system

In this step you will tun the `gpcheck` command to verify *Operating System* settings on the servers in the cluster. Disable the *firewall* between the servers in the cluster and then verify baseline hardware performance.

If you are are not already connected, open a terminal session to *mdw* and log in with the following credentials:

username: `root`

password: `Piv0tal`

Run the `gpcheck` utility

1. Execute the `gpcheck` command to verify the *Operating System* settings. This command should normally be executed against all servers in the cluster, but due to the amount of information it provides, we will examine it first for the Master Server (`mdw`). Learn more about `gpcheck` from the documentation.

   ```
   gpcheck -h mdw
   ```

   What is the result of the command?

2. There are several errors visible for the Master Server (mdw). The following highlights the *I/O scheduler* and the *blockdev readahead* errors found on this server.

   ```
   20170119:18:42:06:012657 gpcheck:mdw:root-[ERROR]:-GPCHECK_ERROR
   host(mdw): on device (sr0) IO scheduler 'cfq' does not match expected
   value 'deadline'
   20170119:18:42:06:012657 gpcheck:mdw:root-[ERROR]:-GPCHECK_ERROR
   host(mdw): on device (sda) IO scheduler 'cfq' does not match expected
   value 'deadline'
   20170119:18:42:06:012657 gpcheck:mdw:root-[ERROR]:-GPCHECK_ERROR
   host(mdw): on device (sdb) IO scheduler 'cfq' does not match expected
   value 'deadline'
   20170119:18:42:06:012657 gpcheck:mdw:root-[ERROR]:-GPCHECK_ERROR
   host(mdw): on device (/dev/sdb1) blockdev readahead value '256' does
   not match expected value '16384'
   20170119:18:42:06:012657 gpcheck:mdw:root-[ERROR]:-GPCHECK_ERROR
   host(mdw): on device (/dev/sdb) blockdev readahead value '256' does not
   match expected value '16384'
   20170119:18:42:06:012657 gpcheck:mdw:root-[ERROR]:-GPCHECK_ERROR
   host(mdw): on device (/dev/sda1) blockdev readahead value '256' does
   not match expected value '16384'
   20170119:18:42:06:012657 gpcheck:mdw:root-[ERROR]:-GPCHECK_ERROR
   host(mdw): on device (/dev/sda) blockdev readahead value '256' does not
   match expected value '16384'
   ```

3. Update *IO scheduler* value for `sr0` device using the command below.

   ```
   echo deadline > /sys/block/sr0/queue/scheduler
   ```

4. Update *blockdev readahead* value for `sdb1` device using the command below.

   ```
   /sbin/blockdev --setra 16384 /dev/sdb1
   ```

5. Re-execute the `gpcheck` command to confirm that the errors on the devices `sr0` and `sdb1` on `mdw` are no longer present.

```
gpcheck -h mdw
```

The `sr0` *IO scheduler* and the `sdb1` *blockdev readahead* errors have been cleared. However some errors will still exist on `mdw`. It appears we still have these errors on other devices. In addition, all of the errors that were present on `mdw` are also present on `smdw`, `sdw1` and `sdw2` and need to be cleared as well.

6. Execute the script `update_block_devices_IO_scheduler.sh` to update the devices on all servers. This file is located in the `/rawdata/solutions/lab2/` directory.

```
/rawdata/solutions/lab2/update_block_devices_IO_scheduler.sh
```

This script is provided within the training environment to quickly update the Master, Standby Master, and segment servers. This is not provided by default with the Greenplum Database installation.

The script uses `gpssh` to update the block devices on each server. You must be `root` to successfully execute the commands within the script.

7. Execute the `gpcheck` command to verify the configuration in all servers. Use the `/rawdata/Binaries/hostfile_exkeys` file to provide the list of all servers in the cluster.

```
gpcheck -f /rawdata/Binaries/hostfile_exkeys -m mdw -s smdw
```

The `-m` and `-s` options for `gpcheck` have been deprecated and will not exist at some time in the future.

All of the *IO scheduler* and *blockdev readahead* errors have been cleared from all of the servers in the cluster.

Disable the Firewall

1. Using `gpssh`, execute the commands below for all Greenplum servers. These commands will disable the firewall among the Greenplum servers.

   Some useful reading tips are listed in the documentation on prepping your system.

   The commands are:

   - `gpssh -f hostfile_exkeys -v -e 'service iptables save'`
   - `gpssh -f hostfile_exkeys -v -e 'service iptables stop'`
   - `gpssh -f hostfile_exkeys -v -e 'chkconfig iptables off'`

   First, make sure you are in the `/rawdata/Binaries/` directory

   ```
   cd /rawdata/Binaries/
   ```

   Now, run:

   ```
   gpssh -f hostfile_exkeys -v -e 'service iptables save'
   ```

   Once this has completed run the next command.

   ```
   gpssh -f hostfile_exkeys -v -e 'service iptables stop'
   ```

   Finally run:

   ```
   gpssh -f hostfile_exkeys -v -e 'chkconfig iptables off'
   ```

   It is necessary to either disable the firewall or to allow specific ports through the Linux firewall for specific Greenplum services. When testing the servers, the network performance test uses port 23000 by default. If this port is not open on all systems, the test will fail. Greenplum utilities may also occasionally use remote copy (rcp) services to copy files from one system to another.

Verify baseline hardware performance

1. Execute the `gpcheckperf` utility to verify baseline hardware performance of the cluster. This is executed against the segment servers within the cluster. As there are only two segment hosts, we will simply specify the server names on the command line. If you do

create a file, use the -f option followed by the filename to specify the segment servers to include in the validation tests. Learn more about `gpcheckperf` from the documentation.

The purpose of this command at this time is to capture a baseline of the system performance, **BEFORE** we initialize the database and start adding data. This is a **CRITICAL** and **ESSENTIAL** task to accomplish to ensure appropriate support.

Either of the following commands will work:

```
gpcheckperf -h sdw1 -h sdw2 -d /data -D
```

or

```
gpcheckperf -f hostfile_gpssh_segonly -d /data -D
```

2. If you wish to perform a network only test, execute the `gpcheckperf` command with the `-r n` option (which only does network tests). You will also need to specify a segment directory using the -d option.

   What is the total bandwidth for:

   - Disk Writes? ________________________________
   - Disk Reads? _________________________________
   - Sustainable Memory? __________________________
   - Network? ____________________________________

   Network performance in the virtual environment may be slightly degraded and so you may see messages that the connection between systems does not meet the guidelines established by Greenplum as adequate for a production environment.

   In a single-host environment, or virtual environment, performance will be lower than on a distributed cluster. On production systems, you should run `gpcheckperf` when the system is idle to get accurate performance metrics.

Summary

As with any database system, the performance of the Greenplum Database is dependent upon the hardware and IT infrastructure on which it is running. The Greenplum Database is comprised of several servers (or hosts) acting together as one cohesive system. The Greenplum Database's runtime performance will be as fast as the slowest segment host in the array. It is important to know your systems’ expected level of performance before setting database performance expectations.

The Greenplum Database requires that the operating systems of the hosts on which it runs be properly tuned. These tuning parameters are especially important on large systems with complex query workloads, as queries can fail when they do not get the resources they need from the operating system. The `gpcheck` utility checks the OS environment of each host to ensure that they have the Greenplum recommended settings.

The expected results of the `gpcheckperf` tests depend on the total capacity of the server hardware you are using. If the expected disk I/O of a system is 2 GBytes per second, multiply the expected rate by the number of segment servers for the total bandwidth. If there are two segment servers, as in this environment, the expected total bandwidth is 4 GB/s.

When looking at the output from `gpcheckperf`, you want to make sure that your disk I/O rate is what you would expect from your hardware platform and that the memory and network bandwidth are not bottlenecks to optimal performance. (They should be comparable to or greater than disk I/O.)

Congratulations! You have completed this lab!

GPDB Administrator – Lab 3

Purpose

In this lab you will perform several setup and initialization tasks required for the Greenplum Database software to run. You will initialize a Greenplum Database cluster by examining the initialization configuration file and executing the `gpinitsystem` utility. You will also troubleshoot errors that may occur during initialization of a Greenplum Database array.

Estimated completion time: 45 minutes

Initialize the Database on Master & Segment Servers

If you are not already connected, open a Putty session to `mdw` and log in with the following credentials:

username: `root`

password: `Piv0tal`

If you are still connected to `mdw`, you may be in the `/rawdata/Binaries` directory. Change to the root directory using the `cd` command.

Review/modify the initialization configuration file

1. Switch to `gpadmin` user.

```
su - gpadmin
```

2. Create a directory named `gpconfigs` in the home directory of the `gpadmin` user.

```
mkdir gpconfigs
```

3. Copy the file `/usr/local/greenplum-db/docs/cli_help/gpconfigs/gpinitsystem_config` to the `/home/gpadmin/gpconfigs` directory.

```
cp /usr/local/greenplum-
db/docs/cli_help/gpconfigs/gpinitsystem_config ~/gpconfigs
```

4. Change to the `gpconfigs` directory and confirm that `gpinitsystem_config` was copied successfully.

```
cd gpconfigs/
ls
```

   The output should show the `gpinitsystem_config` file.

5. Copy the `/rawdata/Binaries/hostfile_gpssh_segonly` file you created in the previous lab exercise to `hostfile_gpinitsystem` in the `gpconfigs` directory. You are copying and changing the name of this file at the same time. This file should contain the hostname of all segment servers in the cluster. This file cannot contain any extra lines or spaces. Confirm that `hostfile_gpssh_segonly` was copied successfully and that it has the names of the two segments servers, `sdw1` and `sdw2` in it.

```
cp /rawdata/Binaries/hostfile_gpssh_segonly
hostfile_gpinitsystem
```

   The results should show `sdw1` and `sdw2`

   Edit the `gpinitsystem_config` file. Make the following changes:

```
################################################
#### REQUIRED PARAMETERS
################################################

#### Name of this Greenplum system enclosed in quotes.
- ARRAY_NAME=""
+ ARRAY_NAME="Pivotal Greenplum DW"

#### Naming convention for utility-generated data directories.
SEG_PREFIX=gpseg

#### Base number by which primary segment port numbers
#### are calculated.
PORT_BASE=40000

#### File system location(s) where primary segment data directories
#### will be created. The number of locations in the list dictate
#### the number of primary segments that will get created per
#### physical host (if multiple addresses for a host are listed in
#### the hostfile, the number of segments will be spread evenly across
#### the specified interface addresses).
- declare -a DATA_DIRECTORY=(/data1/primary /data1/primary
/data1/primary /data2/primary /data2/primary /data2/primary)
+ declare -a DATA_DIRECTORY=(/data/primary)
```

Initialize the Greenplum Database on the Master and Segment servers

1. Source `greenplum_path.sh` on `mdw` to make Greenplum commands available.

```
source /usr/local/greenplum-db/greenplum_path.sh
```

2. Run the `gpinitsystem` utility to create a Greenplum Database system using the values defined in `gpinitsystem_config`.

```
gpinitsystem -c gpinitsystem_config -h hostfile_gpinitsystem
```

Answer `y` if the utility asks permission to overwrite the configuration file.

The utility displays the configuration for the master and primary segments. It also asks you whether or not you want to continue with the Greenplum creation.

Answer `y` to `Continue with Greenplum Creation`.

Initialization will take a few minutes.

```
.
.
.
20170201:22:32:56:024054 gpstart:mdw:gpadmin-[INFO]:-Process results...
20170201:22:32:56:024054 gpstart:mdw:gpadmin-[INFO]:--------------------
----------------------------------
20170201:22:32:56:024054 gpstart:mdw:gpadmin-[INFO]:-   Successful
segment starts                                            = 2
20170201:22:32:56:024054 gpstart:mdw:gpadmin-[INFO]:-   Failed segment
starts                                                = 0
20170201:22:32:56:024054 gpstart:mdw:gpadmin-[INFO]:-   Skipped segment
starts (segments are marked down in configuration)   = 0
20170201:22:32:56:024054 gpstart:mdw:gpadmin-[INFO]:--------------------
----------------------------------
20170201:22:32:56:024054 gpstart:mdw:gpadmin-[INFO]:-
20170201:22:32:56:024054 gpstart:mdw:gpadmin-[INFO]:-Successfully
started 2 of 2 segment instances
.
.
.
170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-To complete the
environment configuration, please
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-update
gpadmin .bashrc file with the following
```

```
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-1. Ensure
that the greenplum_path.sh file is sourced
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-2. Add
"export MASTER_DATA_DIRECTORY=/data/master/gpseg-1"
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-   to access
the Greenplum scripts for this instance:
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-   or, use -d
/data/master/gpseg-1 option for the Greenplum scripts
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-   Example
gpstate -d /data/master/gpseg-1
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-Script log
file = /home/gpadmin/gpAdminLogs/gpinitsystem_20170201.log
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-To remove
instance, run gpdeletesystem utility
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-To initialize
a Standby Master Segment for this Greenplum instance
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-Review
options for gpinitstandby
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:--------------
-----------------------------------------
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-The Master
/data/master/gpseg-1/pg_hba.conf post gpinitsystem
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-has been
configured to allow all hosts within this new
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-array to
intercommunicate. Any hosts external to this
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-new array
must be explicitly added to this file
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-Refer to the
Greenplum Admin support guide which is
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:-located in
the /usr/local/greenplum-db/./docs directory
20170201:22:33:05:013473 gpinitsystem:mdw:gpadmin-[INFO]:--------------
-----------------------------------------
```

The above output indicates that `gpinitsystem` ended with no errors.

Notice the output also indicates that there are some steps that need to be performed to complete the environment configuration.

Complete the environment configuration

1. On the Master server (`mdw`) change directory to `/home/gpadmin`.

   Edit the file **/home/gpadmin/.bash_profile**.

```
cd
vi /home/gpadmin/.bash_profile
```

Add the following entries to the end of the file.

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin

export PATH

+ MASTER_DATA_DIRECTORY=/data/master/gpseg-1
+ export MASTER_DATA_DIRECTORY
+ source /usr/local/greenplum-db/greenplum_path.sh
```

2. Source `.bash_profile` to make the changes you just made available immediately.

```
source .bash_profile
```

Run the `gpstate -f` command to see the state of the system after initialization

1. Run the `gpstate -f` command

```
gpstate -f
```

The output below clearly shows, the standby master is not configured.

```
20170201:22:40:21:025506 gpstate:mdw:gpadmin-[INFO]:-Starting gpstate
with args: -f
20170201:22:40:21:025506 gpstate:mdw:gpadmin-[INFO]:-local Greenplum
Version: 'postgres (Greenplum Database) 4.3.11.3 build 1'
20170201:22:40:21:025506 gpstate:mdw:gpadmin-[INFO]:-master Greenplum
Version: 'PostgreSQL 8.2.15 (Greenplum Database 4.3.11.3 build 1) on
x86_64-unknown-linux-gnu, compiled by GCC gcc (GCC) 4.4.2 compiled on
Jan 24 2017 20:28:18'
20170201:22:40:21:025506 gpstate:mdw:gpadmin-[INFO]:-Obtaining Segment
details from master...
20170201:22:40:21:025506 gpstate:mdw:gpadmin-[INFO]:-Standby master
instance not configured
20170201:22:40:21:025506 gpstate:mdw:gpadmin-[INFO]:--------------------
-------------------------------------------
20170201:22:40:21:025506 gpstate:mdw:gpadmin-[INFO]:--
pg_stat_replication
```

```
20170201:22:40:21:025506 gpstate:mdw:gpadmin-[INFO]:--------------------
-----------------------------------------
20170201:22:40:21:025506 gpstate:mdw:gpadmin-[INFO]:-No entries found.
20170201:22:40:21:025506 gpstate:mdw:gpadmin-[INFO]:--------------------
-----------------------------------------
```

Add a Standby Master Server

Manually add the Standby Master server to the Greenplum Database

1. Run the `gpinitstandby` utility to add the Standby Master server.

```
gpinitstandby -s smdw
```

Answer `y` when asked if you want to continue with `standby master initialization`.

```
Do you want to continue with standby master initialization? Yy|Nn
(default=N):
> Y
20170201:22:47:21:025624 gpinitstandby:mdw:gpadmin-[INFO]:-Syncing
Greenplum Database extensions to standby
20170201:22:47:23:025624 gpinitstandby:mdw:gpadmin-[INFO]:-The packages
on smdw are consistent.
20170201:22:47:23:025624 gpinitstandby:mdw:gpadmin-[INFO]:-Adding
standby master to catalog...
20170201:22:47:23:025624 gpinitstandby:mdw:gpadmin-[INFO]:-Database
catalog updated successfully.
20170201:22:47:23:025624 gpinitstandby:mdw:gpadmin-[INFO]:-Updating
pg_hba.conf file...
20170201:22:47:29:025624 gpinitstandby:mdw:gpadmin-[INFO]:-pg_hba.conf
files updated successfully.
20170201:22:47:44:025624 gpinitstandby:mdw:gpadmin-[INFO]:-Updating
filespace flat files...
20170201:22:47:44:025624 gpinitstandby:mdw:gpadmin-[INFO]:-Filespace
flat file updated successfully.
20170201:22:47:44:025624 gpinitstandby:mdw:gpadmin-[INFO]:-Starting
standby master
20170201:22:47:44:025624 gpinitstandby:mdw:gpadmin-[INFO]:-Checking if
standby master is running on host: smdw  in directory:
/data/master/gpseg-1
20170201:22:47:46:025624 gpinitstandby:mdw:gpadmin-[INFO]:-Cleaning up
pg_hba.conf backup files...
20170201:22:47:52:025624 gpinitstandby:mdw:gpadmin-[INFO]:-Backup files
of pg_hba.conf cleaned up successfully.
20170201:22:47:52:025624 gpinitstandby:mdw:gpadmin-[INFO]:-Successfully
created standby master on smdw
```

The output indicates that `gpinitstandby` successfully created the Standby Master.

Confirm the Standby Master server was created and is synchronized with the Master server

1. Run `gpstate -f` (the `-f` option displays the details of the standby master `smdw`) to verify that the Standby Master was created and is synchronized with the Master.

```
gpstate -f
```

```
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:-Starting gpstate
with args: -f
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:-local Greenplum
Version: 'postgres (Greenplum Database) 4.3.11.3 build 1'
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:-master Greenplum
Version: 'PostgreSQL 8.2.15 (Greenplum Database 4.3.11.3 build 1) on
x86_64-unknown-linux-gnu, compiled by GCC gcc (GCC) 4.4.2 compiled on
Jan 24 2017 20:28:18'
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:-Obtaining Segment
details from master...
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:-Standby master
details
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:-------------------
----
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:-   Standby address
= smdw
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:-   Standby data
directory   = /data/master/gpseg-1
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:-   Standby port
= 5432
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:-   Standby PID
= 12235
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:-   Standby status
= Standby host passive
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:-------------------
-------------------------------------------
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:--
pg_stat_replication
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:-------------------
-------------------------------------------
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:--WAL Sender State:
streaming
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:--Sync state: sync
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:--Sent Location:
0/C000000
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:--Flush Location:
0/C000000
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:--Replay Location:
0/C000000
20170201:22:49:36:025795 gpstate:mdw:gpadmin-[INFO]:-------------------
-------------------------------------------
```

As you can see, the standby master now exists and shows that it is `smdw`, and the `Sync state = sync`.

2. Copy the file `/home/gpadmin/.bash_profile` to `smdw` using the `scp`:

```
scp /home/gpadmin/.bash_profile smdw:
```

Add Mirrors on the Segment Servers

Configure mirror directories on the Segment Servers

1. Switch to the `root` user by typing `exit`

```
exit
```

2. Source `greenplum_path.sh` on `mdw` to make Greenplum commands available.

```
source /usr/local/greenplum-db/greenplum_path.sh
```

3. Use `gpssh` to connect to `sdw1` and `sdw2`. Create the mirror directory `/data/mirror`.

```
gpssh -h sdw1 -h sdw2 -e 'mkdir /data/mirror'
```

4. Use `gpssh` to connect to `sdw1` and `sdw2` and change the ownership of the `/data/mirror` directories to `gpadmin`.

```
gpssh -h sdw1 -h sdw2 -e 'chown gpadmin:gpadmin /data/mirror'
```

5. Switch to the `gpadmin` user typing `su - gpadmin`

```
su - gpadmin
```

Run the `gpstate -s` command to see that mirrors are not configured

1. Run the `gpstate -s` command

```
gpstate -s
```

You can clearly see that there are no mirrors defined and that both segments (`sdw1` and *`sdw2`) have primary **Datadir** only!

```
20170201:22:56:41:026032 gpstate:mdw:gpadmin-[INFO]:-Starting gpstate
with args: -s
20170201:22:56:41:026032 gpstate:mdw:gpadmin-[INFO]:-local Greenplum
Version: 'postgres (Greenplum Database) 4.3.11.3 build 1'
20170201:22:56:41:026032 gpstate:mdw:gpadmin-[INFO]:-master Greenplum
Version: 'PostgreSQL 8.2.15 (Greenplum Database 4.3.11.3 build 1) on
x86_64-unknown-linux-gnu, compiled by GCC gcc (GCC) 4.4.2 compiled on
Jan 24 2017 20:28:18'
20170201:22:56:41:026032 gpstate:mdw:gpadmin-[INFO]:-Obtaining Segment
details from master...
20170201:22:56:41:026032 gpstate:mdw:gpadmin-[INFO]:-Gathering data
from segments...
.
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-------------------
----------------------------------
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:--Master
Configuration & Status
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-------------------
----------------------------------
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-   Master host
= mdw
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-   Master postgres
process ID     = 24110
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-   Master data
directory          = /data/master/gpseg-1
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-   Master port
= 5432
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-   Master current
role            = dispatch
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-   Greenplum
initsystem version   = 4.3.11.3 build 1
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-   Greenplum
current version      = PostgreSQL 8.2.15 (Greenplum Database 4.3.11.3
build 1) on x86_64-unknown-linux-gnu, compiled by GCC gcc (GCC) 4.4.2
compiled on Jan 24 2017 20:28:18
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-   Postgres
version               = 8.2.15
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-   Master standby
= smdw
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-   Standby master
state            = Standby host passive
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-------------------
----------------------------------
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-Segment Instance
Status Report
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-------------------
----------------------------------
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-   Segment Info
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-      Hostname
= sdw1
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-      Address
= sdw1
```

```
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-      Datadir
= /data/primary/gpseg0
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-      Port
= 40000
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-   Status
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-      PID
= 13235
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-
Configuration reports status as    = Up
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-      Database
status                        = Up
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-------------------
----------------------------------
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-   Segment Info
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-      Hostname
= sdw2
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-      Address
= sdw2
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-      Datadir
= /data/primary/gpseg1
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-      Port
= 40000
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-   Status
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-      PID
= 12991
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-
Configuration reports status as    = Up
20170201:22:56:42:026032 gpstate:mdw:gpadmin-[INFO]:-      Database
status                        = Up
```

Manually add mirrors using the `gpaddmirror` utility

1. Run the `gpaddmirrors` utility to manually add the mirrors to the segment servers.

```
gpaddmirrors
```

   You will be asked to enter the mirror segment data directory location

2. Respond to the request by entering `/data/mirror`.

```
/data/mirror
```

   Answer `y` when prompted with the `add mirrors procedure`.

```
.
20170201:23:02:34:026239 gpaddmirrors:mdw:gpadmin-[INFO]:-
******************************************************************
20170201:23:02:34:026239 gpaddmirrors:mdw:gpadmin-[INFO]:-Mirror
segments have been added; data synchronization is in progress.
20170201:23:02:34:026239 gpaddmirrors:mdw:gpadmin-[INFO]:-Data
synchronization will continue in the background.
```

```
20170201:23:02:34:026239 gpaddmirrors:mdw:gpadmin-[INFO]:-
20170201:23:02:34:026239 gpaddmirrors:mdw:gpadmin-[INFO]:-Use  gpstate
-s  to check the resynchronization progress.
20170201:23:02:34:026239 gpaddmirrors:mdw:gpadmin-[INFO]:-
*****************************************************************
```

The output indicates that the `gpaddmirrors` utility completed successfully.

Confirm the mirrors were created and are synchronized with their primary segments

1. Run `gpstate -s` (the `-s` option displays the details of the segment servers `sdw1` and `sdw2`) to verify that the mirrors were created and are synchronized with their primary segments.

   You can use the `gpstate -s` utility to check that the mirror segments are being synchronized with their primary segments. This utility will also indicate when the mirrors are fully synchronized.

   ```
   gpstate -s
   ```

   The output should look something like this:

   ```
   20170201:23:04:26:026436 gpstate:mdw:gpadmin-[INFO]:-Starting gpstate
   with args: -s
   20170201:23:04:26:026436 gpstate:mdw:gpadmin-[INFO]:-local Greenplum
   Version: 'postgres (Greenplum Database) 4.3.11.3 build 1'
   20170201:23:04:26:026436 gpstate:mdw:gpadmin-[INFO]:-master Greenplum
   Version: 'PostgreSQL 8.2.15 (Greenplum Database 4.3.11.3 build 1) on
   x86_64-unknown-linux-gnu, compiled by GCC gcc (GCC) 4.4.2 compiled on
   Jan 24 2017 20:28:18'
   20170201:23:04:26:026436 gpstate:mdw:gpadmin-[INFO]:-Obtaining Segment
   details from master...
   20170201:23:04:26:026436 gpstate:mdw:gpadmin-[INFO]:-Gathering data
   from segments...
   .
   20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-------------------
   ----------------------------------
   20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:--Master
   Configuration & Status
   20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-------------------
   ----------------------------------
   20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Master host
   = mdw
   20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Master postgres
   process ID     = 24110
   20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Master data
   directory          = /data/master/gpseg-1
   ```

```
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Master port
= 5432
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Master current
role           = dispatch
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Greenplum
initsystem version   = 4.3.11.3 build 1
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Greenplum
current version      = PostgreSQL 8.2.15 (Greenplum Database 4.3.11.3
build 1) on x86_64-unknown-linux-gnu, compiled by GCC gcc (GCC) 4.4.2
compiled on Jan 24 2017 20:28:18
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Postgres
version               = 8.2.15
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Master standby
= smdw
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Standby master
state           = Standby host passive
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:--------------------
----------------------------------
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-Segment Instance
Status Report
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:--------------------
----------------------------------
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Segment Info
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Hostname
= sdw1
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Address
= sdw1
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Datadir
= /data/primary/gpseg0
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Port
= 40000
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Mirroring Info
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Current role
= Primary
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Preferred
role                    = Primary
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Mirror
status                     = Synchronized
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Status
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      PID
= 13235
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-
Configuration reports status as   = Up
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Database
status                   = Up
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:--------------------
----------------------------------
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Segment Info
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Hostname
= sdw2
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Address
= sdw2
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Datadir
= /data/mirror/gpseg0
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Port
= 41000
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Mirroring Info
```

```
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Current role
= Mirror
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Preferred
role                    = Mirror
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Mirror
status                     = Synchronized
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Status
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      PID
= 13291
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-
Configuration reports status as   = Up
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Segment
status                    = Up
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-------------------
----------------------------------
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Segment Info
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Hostname
= sdw2
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Address
= sdw2
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Datadir
= /data/primary/gpseg1
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Port
= 40000
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Mirroring Info
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Current role
= Primary
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Preferred
role                    = Primary
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Mirror
status                     = Synchronized
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Status
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      PID
= 12991
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-
Configuration reports status as   = Up
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Database
status                  = Up
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-------------------
----------------------------------
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Segment Info
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Hostname
= sdw1
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Address
= sdw1
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Datadir
= /data/mirror/gpseg1
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Port
= 41000
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Mirroring Info
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Current role
= Mirror
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Preferred
role                    = Mirror
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Mirror
status                     = Synchronized
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-   Status
```

```
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      PID
= 13567
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-
Configuration reports status as   = Up
20170201:23:04:27:026436 gpstate:mdw:gpadmin-[INFO]:-      Segment
status                          = Up
```

Delete the Greenplum database (Optional)

Delete the Greenplum Database you have installed & configured

1. You will now delete the Greenplum Database using the `gpdeletesystem` utility.

   The `-d` option identifies the directory where the master database instance was created.

   ```
   gpdeletesystem -d /data/master/gpseg-1
   ```

   Enter `y` when asked multiple times whether you want to continue.

   ```
   .
   .
   .
   Continue with Greenplum instance deletion? Yy|Nn (default=N):
   > Y
   20170201:23:07:43:026697 gpdeletesystem:mdw:gpadmin-[INFO]:-Stopping
   database...
   20170201:23:08:12:026697 gpdeletesystem:mdw:gpadmin-[INFO]:-Deleting
   segments and removing data directories...
   20170201:23:08:12:026697 gpdeletesystem:mdw:gpadmin-[INFO]:-Waiting for
   worker threads to complete...
   20170201:23:08:17:026697 gpdeletesystem:mdw:gpadmin-[INFO]:-Delete
   system successful.
   ```

   The output indicates that `gpdeletesystem` successfully deleted the Greenplum Database.

Initialize a new Greenplum Database in One Step

Modify the `gpinitsystem_conf` configuration file to create a Greenplum Database in one step

1. Change to the `/home/gpadmin/gpconfigs` directory.

```
cd gpconfigs/
```

2. Edit the `gpinitsystem_config` file. Locate the `OPTIONAL MIRROR PARAMETERS` section.

   Recall that you reviewed / modified the `REQUIRED PARAMETERS` defined in this configuration file in an earlier step. In this step you are examining and modifying the `OPTIONAL MIRROR PARAMETERS`.

   Review each of the '**OPTIONAL MIRROR PARAMETERS**' defined in this configuration file and make the following edits:

```
.
.
.
################################################
#### OPTIONAL MIRROR PARAMETERS
################################################

#### Base number by which mirror segment port numbers
#### are calculated.
- #MIRROR_PORT_BASE=50000
+ MIRROR_PORT_BASE=50000

#### Base number by which primary file replication port
#### numbers are calculated.
- #REPLICATION_PORT_BASE=41000
+ REPLICATION_PORT_BASE=41000

#### Base number by which mirror file replication port
#### numbers are calculated.
- #MIRROR_REPLICATION_PORT_BASE=51000
+ MIRROR_REPLICATION_PORT_BASE=51000

#### File system location(s) where mirror segment data directories
#### will be created. The number of mirror locations must equal the
#### number of primary locations as specified in the
```

```
#### DATA_DIRECTORY parameter.
#declare -a MIRROR_DATA_DIRECTORY=(/data1/mirror /data1/mirror
/data1/mirror /data2/mirror /data2/mirror /data2/mirror)
+ declare -a MIRROR_DATA_DIRECTORY=(/data/mirror)
.
.
.
```

The **OPTIONAL MIRROR PARAMETERS** section should appear as follows:

```
.
.
.

################################################
#### OPTIONAL MIRROR PARAMETERS
################################################

#### Base number by which mirror segment port numbers
#### are calculated.
MIRROR_PORT_BASE=50000

#### Base number by which primary file replication port
#### numbers are calculated.
REPLICATION_PORT_BASE=41000

#### Base number by which mirror file replication port
#### numbers are calculated.
MIRROR_REPLICATION_PORT_BASE=51000

#### File system location(s) where mirror segment data directories
#### will be created. The number of mirror locations must equal the
#### number of primary locations as specified in the
#### DATA_DIRECTORY parameter.
#declare -a MIRROR_DATA_DIRECTORY=(/data1/mirror /data1/mirror
/data1/mirror /data2/mirror /data2/mirror /data2/mirror)
declare -a MIRROR_DATA_DIRECTORY=(/data/mirror)

################################################
#### OTHER OPTIONAL PARAMETERS
################################################

#### Create a database of this name after initialization.
#DATABASE_NAME=name_of_database

#### Specify the location of the host address file here instead of
#### with the the -h option of gpinitsystem.
#MACHINE_LIST_FILE=/home/gpadmin/gpconfigs/hostfile_gpinitsystem
```

Run `gpinitsystem` to create the database

1. Run the `gpinitsystem` utility to create a Greenplum Database system using the values defined in `gpinitsystem_config`.

```
gpinitsystem -c gpinitsystem_config -h hostfile_gpinitsystem -s
smdw -S
```

   Answer `y` when asked if you want to continue.

```
.
20161022:18:33:38:004053 gpstart:mdw:gpadmin-[INFO]:-------------------
----------------------------------
20161022:18:33:38:004053 gpstart:mdw:gpadmin-[INFO]:-   Successful
segment starts                                            = 4
20161022:18:33:38:004053 gpstart:mdw:gpadmin-[INFO]:-   Failed segment
starts                                                = 0
20161022:18:33:38:004053 gpstart:mdw:gpadmin-[INFO]:-   Skipped segment
starts (segments are marked down in configuration)   = 0
20161022:18:33:38:004053 gpstart:mdw:gpadmin-[INFO]:-------------------
----------------------------------
20161022:18:33:38:004053 gpstart:mdw:gpadmin-[INFO]:-
20161022:18:33:38:004053 gpstart:mdw:gpadmin-[INFO]:-Successfully
started 4 of 4 segment instances
20161022:18:33:38:004053 gpstart:mdw:gpadmin-[INFO]:-------------------
----------------------------------
.
.
.
```

   The output indicates that `gpinitsystem` completed successfully with no errors.

2. Run the `gpstate` utility to verify the Greenplum instance status summary.

```
gpstate
```

   Your output should look something like this:

```
20161022:18:36:27:005646 gpstate:mdw:gpadmin-[INFO]:-Starting gpstate
with args:
20161022:18:36:27:005646 gpstate:mdw:gpadmin-[INFO]:-local Greenplum
Version: 'postgres (Greenplum Database) 4.3.11.0 build
commit:095e92e4ecbbbb6764897ca98b1cb7705ad0c805'
20161022:18:36:27:005646 gpstate:mdw:gpadmin-[INFO]:-master Greenplum
Version: 'PostgreSQL 8.2.15 (Greenplum Database 4.3.11.0 build
commit:095e92e4ecbbbb6764897ca98b1cb7705ad0c805) on x86_64-unknown-
```

```
linux-gnu, compiled by GCC gcc (GCC) 4.4.2 compiled on Aug 24 2016
06:30:45'
20161022:18:36:27:005646 gpstate:mdw:gpadmin-[INFO]:-Obtaining Segment
details from master...
20161022:18:36:27:005646 gpstate:mdw:gpadmin-[INFO]:-Gathering data
from segments...
.
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-Greenplum instance
status summary
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-------------------
----------------------------------
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Master instance
= Active
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Master standby
= smdw
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Standby master
state                                      = Standby host passive
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total segment
instance count from metadata                = 4
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-------------------
----------------------------------
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Primary Segment
Status
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-------------------
----------------------------------
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total primary
segments                                    = 2
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total primary
segment valid (at master)                   = 2
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total primary
segment failures (at master)                = 0
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number of
postmaster.pid files missing              = 0
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number of
postmaster.pid files found                = 2
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number of
postmaster.pid PIDs missing               = 0
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number of
postmaster.pid PIDs found                 = 2
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number of
/tmp lock files missing                   = 0
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number of
/tmp lock files found                     = 2
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number
postmaster processes missing                 = 0
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number
postmaster processes found                   = 2
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-------------------
----------------------------------
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Mirror Segment
Status
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-------------------
----------------------------------
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total mirror
segments                                     = 2
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total mirror
segment valid (at master)                    = 2
```

```
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total mirror
segment failures (at master)                     = 0
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number of
postmaster.pid files missing                  = 0
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number of
postmaster.pid files found                    = 2
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number of
postmaster.pid PIDs missing                   = 0
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number of
postmaster.pid PIDs found                     = 2
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number of
/tmp lock files missing                       = 0
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number of
/tmp lock files found                         = 2
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number
postmaster processes missing                     = 0
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number
postmaster processes found                       = 2
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number
mirror segments acting as primary segments   = 0
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-   Total number
mirror segments acting as mirror segments    = 2
20161022:18:36:28:005646 gpstate:mdw:gpadmin-[INFO]:-------------------
----------------------------------
```

GPDB Administrator – Lab 4

psql CLI utility

Purpose

In this lab you will learn how to connect to the database using psql, the command line client interface to the Greenplum Database. You will learn how to use psql to run SQL commands both in interactive mode and non-interactive mode. You will also learn how to view the help in psql and about psql meta-commands.

Helpful Resources:

- Main Documentation Page
- Master Knowledge Base
- Greenplum Knowledge Base
- pSQL Documentation link

Access the Database with psql

1. Use the `psql` cli to connect to a database named `postgres`:

```
psql postgres
```

2. The `psql` prompt displays the database you are connected to followed by `=#`, if you are logged in as the superuser account.

   `psql` is a *cli* built into PostgreSQL , including Pivotal Greenplum. This allows you to do a number of things:

   - `psql` can execute SQL commands in three ways:
     - interactive (like we just did)
     - using the `-e` flag to execute single line commands: `psql -e 'SELECT * FROM planes_table'`
     - using the `-f` flag to execute a file with multiple SQL commands: `psql -f <filename>`

- `psql` has an internal command language (meta-commands) that uses a special notation to access in interactive mode. You type a `\` followed by a special character. For example, to get help you would type `\?`.

3. At the `psql` prompt, type the `\?` meta-command to see a list of available `psql` meta-commands:

```
\?
```

4. At the `psql` prompt, type the `\h` meta-command to see a list of available `SQL` commands:

```
\h
```

   You can also get help on specific SQL commands, For Example:

```
\h SELECT
```

5. Run a `SELECT` statement on the `gp_segment_configuration` table. This system catalog table shows the master and segment instances in your array.

```
SELECT * FROM gp_segment_configuration;
```

6. Exit the `psql` session:

```
\q
```

7. Execute the same `SELECT` statement you ran earlier, but this time, in non-interactive mode:

```
psql postgres -c "SELECT * FROM gp_segment_configuration;"
```

8. Create a database with the same name as the user `gpadmin`. By creating a database with the same name as the current user, you will automatically be connected to this database if you do not specify the database name as part of the `psql` command line.

   Connect to the `template1` database:

```
psql template1
```

   Create a new database and name it `gpadmin`:

```
CREATE DATABASE gpadmin;
```

   Exit the database session

9. Edit `.bash_profile` and add the `PGDATABASE` variable as shown below:

```
vi .bash_profile
```

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin

export PATH

MASTER_DATA_DIRECTORY=/data/master/gpseg-1
export MASTER_DATA_DIRECTORY

source /usr/local/greenplum-db/greenplum_path.sh

+ export PGDATABASE=gpadmin
```

This variable contains your `default login database name`. PSQL chooses a default database based on your username. The `PGDATABASE` environment variable takes precedence over this action, so if it is set, you will automatically be logged in to the database specified in the variable if you do not specify one on the command line.

10. Source `.bash_profile` to make the changes active immediately.

```
source .bash_profile
```

11. Log in to Greenplum with the default database of `gpadmin`.

```
psql
```

12. From within the database session, connect to the `postgres` database.

```
\c postgres
```

13. Exit the database session.

```
\q
```

GPDB Administrator – Lab 5

Command Center

Purpose

In this lab you will perform several installation and configuration tasks required for the Greenplum Command Center software to run.

Helpful Resources:

Main Greenplum Documentation Page

Master Knowledge Base

Greenplum Knowledge Base

Greenplum Command Center Documentation

Estimated completion time: 45 minutes

Install Command Center software on the Master server (`mdw`)

This task will install the Command Center software on the Master server.

1. If you are not already connected, open a Putty session to `mdw` and log in with the following credentials:

   username: `root`
   password: `Piv0tal`

   If you are still connected to `mdw` from the previous lab then you need to become the `root` user rather than the `gpadmin` user. You should be able to simply type `exit` to logout of the `gpadmin` user and this should deposit you back into the `root` user account.

```
exit
```

2. Change to the `/rawdata/Binaries/Greenplum_Command_Center` and list the contents.

```
cd /rawdata/Binaries/Greenplum_Command_Center
ls -1
```

You should see the following contents:

```
greenplum-cc-web-3.0.1-x86_64.zip
```

3. Unzip the `greenplum-cc-web-3.0.1-x86_64.zip` file and list the contents.

```
unzip greenplum-cc-web-3.0.1-x86_64.zip
ls -1
```

```
greenplum-cc-web-3.0.1-x86_64.bin
greenplum-cc-web-3.0.1-x86_64.zip
```

The `greenplum-cc-web-3.0.1-x86_64.bin` file is the binary that you will use to install the Greenplum Command Center.

4. Launch the `Greenplum Command Center` installer.

```
/bin/bash greenplum-cc-web-3.0.1-x86_64.bin
```

You will be prompted to accept the license agreement.

After the license agreement accept the default installation path by pushing *Enter*. When prompted to install in the default installation path type `yes` and press *Enter*.

The Greenplum Command Center software has been installed on the Master server (`mdw`) and a symbolic link named `greenplum-cc-web` was created at the same level as your version-specific Greenplum Command Center installation directory (`/usr/local/greenplum-cc-web-3.0.1`). The symbolic link is used to facilitate patch maintenance and upgrades between versions. The installed location is referred to as `$GPPERFMONHOME`.

5. Change the user and group permissions of the Command Center directory to `gpadmin` and then check to make sure it worked by using `ls`.

```
chown -R gpadmin:gpadmin /usr/local/greenplum-cc-web-3.0.1
ls -ld /usr/local/greenplum-cc-web-3.0.1/
```

You should see:

```
drwxr-xr-x 7 gpadmin gpadmin 4096 Feb  2 18:59 /usr/local/greenplum-cc-
web-3.0.1/
```

Run the `gpperfmon_install` utility

The `gpperfmon_install` utility will:

- Create the Command Center database (`gpperfmon`)
- Create the Command Center superuser (`gpmon`)
- Configure connections from the Command Center superuser
- Set Command Center server configuration parameters in the `postgres.conf` file

1. If you are **not** already connected, open a Putty session to `mdw` and login with the following credentials:

   username: `root`

   password: `Piv0tal`

2. Switch user to `gpadmin`

   ```
   su - gpadmin
   ```

3. Source the setup files `greenplum_path.sh` and `gpcc_path.sh`.

   The file `greenplum_path.sh` can be found in the Greenplum installation directory and it sets up the Greenplum environment.

   The file `gpcc_path.sh` can be found in the Command Center installation directory. Source `gpcc_path.sh` to make Command Center commands available.

   ```
   source /usr/local/greenplum-db/greenplum_path.sh
   source /usr/local/greenplum-cc-web/gpcc_path.sh
   ```

4. Run the `gpperfmon_install` utility.

   ```
   gpperfmon_install --enable --password changeme --port 5432
   ```

5. View the `gpcc_path.sh` file to see that it adds a new variable called `GPPERFMONHOME`. Typically, we would source this file in our local `.bash_profile`.

6. Edit your `.bash_profile` to add an entry to source the `gpcc_path.sh` file. While there, check to make sure you are sourcing the `greenplum_path.sh` file as well.

```
vi .bash_profile
```

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin

export PATH

MASTER_DATA_DIRECTORY=/data/master/gpseg-1
export MASTER_DATA_DIRECTORY
+ source /usr/local/greenplum-db/greenplum_path.sh

export PGDATABASE=gpadmin

+ source /usr/local/greenplum-cc-web/gpcc_path.sh
```

7. Source `/home/gpadmin/.bash_profile` to make the `GPPERFMONHOME` environment variable and the Command Center commands available.

```
source /home/gpadmin/.bash_profile
```

8. Add a `host` connection record to `pg_hba.conf` to connect to `localhost` through the IPV6 address ::1.

```
echo "host     all     gpmon     ::1/128     md5" >>
/data/master/gpseg-1/pg_hba.conf
```

Tab characters are not supported in the `pg_hba.conf` file. You must use spaces between the record entries. It is best to copy the above line verbatim as it is easy to make a mistake and a mistake in the `pg_hba.conf` file will render Greenplum un-operational.

9. Restart the Greenplum Database to make the connection record available.

```
gpstop -a -r
```

Run the `gpcmdr` utility

This task will setup & configure the command center using the `gpcmdr` utility.

1. Login or switch user to `gpadmin` user if you are not already logged in as the `gpadmin` user.
2. Run the `gpcmdr` utility to setup and configure Command Center

```
gpcmdr -setup

You will be prompted to enter a new instance name. Name your instance
training. Type a name, and then hit the ENTER key.

You will be prompted to enter a display name, enter Training. Press
ENTER.

You will be asked if the Greenplum Database is remote. Press ENTER to
select the default (N).

You will be prompted for the Greenplum Database port. Press ENTER to
accept the default port (:5432). A message will appear indicating that
an instance schema is being created.

You will be prompted for the web server port for this instance. Press
ENTER to accept the default port (:28080)

Enable SSL for the Web API Yy/Nn (default=N). Press ENTER to accept
the default (N).

You will be asked if you want to copy the instance to the standby
master host. Enter N and press ENTER.
```

3. Start your instance of Command Center

```
gpcmdr --start training
```

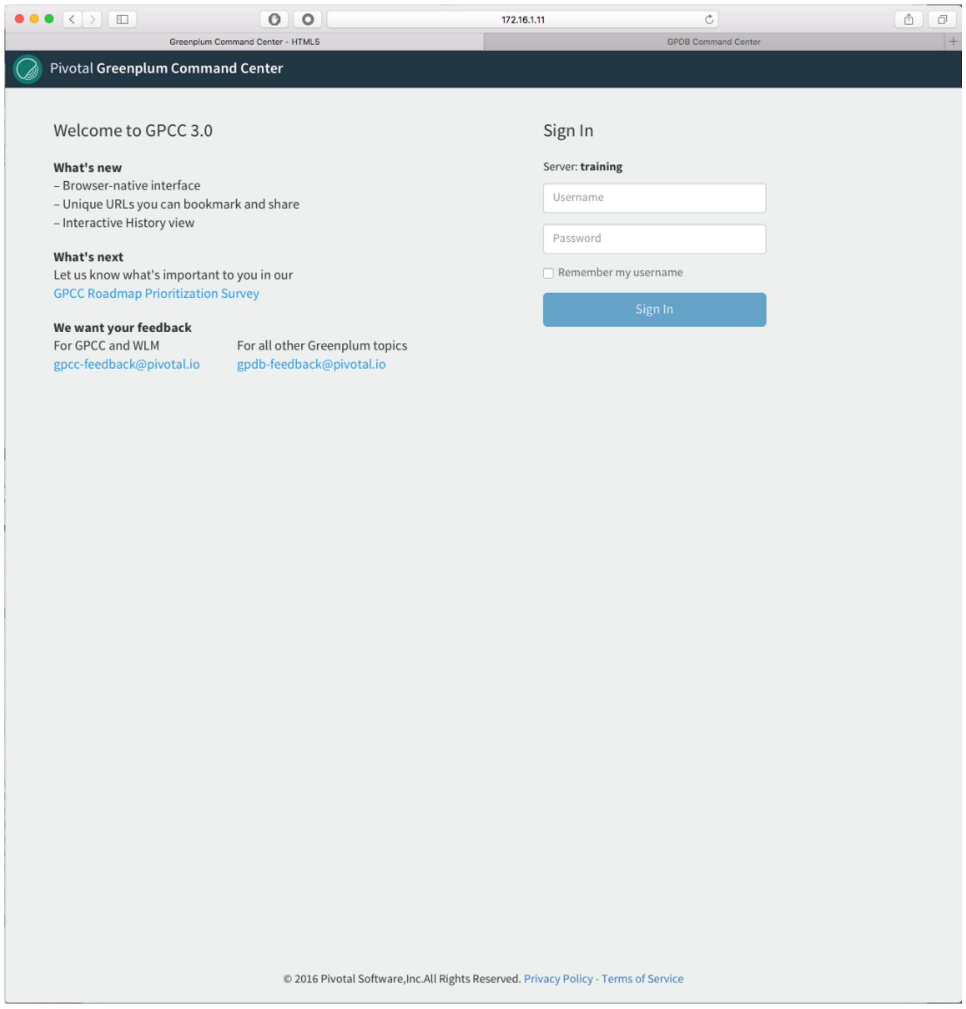
Login to Command Center

1. Open a browser and navigate to [172.16.1.11:28080]() to display the Command Center login screen.

   Login with the following credentials:

   username: `gpmon`

   password: `changeme`



Pivotal Greenplum Command Center

Welcome to GPCC 3.0

What's new
– Browser-native interface
– Unique URLs you can bookmark and share
– Interactive History view

What's next
Let us know what's important to you in our
GPCC Roadmap Prioritization Survey

We want your feedback
For GPCC and WLM gpcc-feedback@pivotal.io
For all other Greenplum topics gpdb-feedback@pivotal.io

Sign In
Server: **training**

© 2016 Pivotal Software,Inc.All Rights Reserved. Privacy Policy - Terms of Service

You will be redirected to the Greenplum Command Center dashboard.

Navigate the Greenplum Command Center

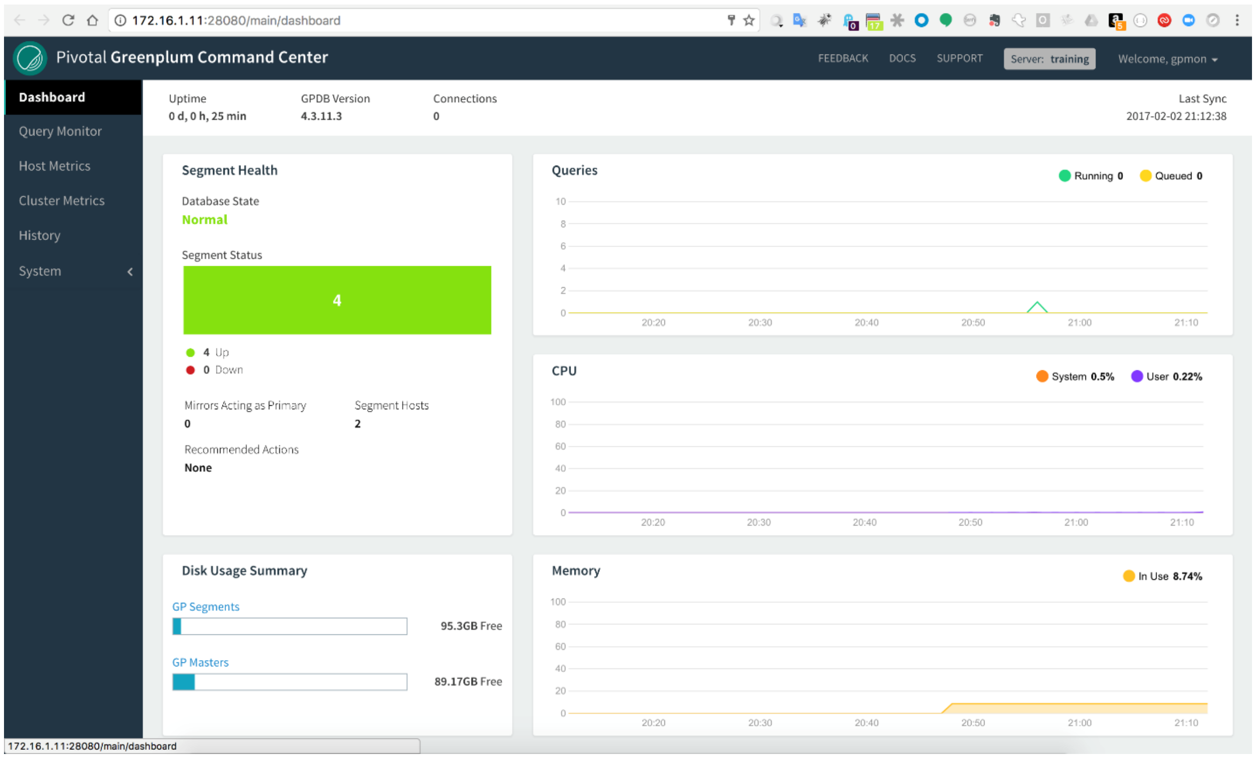
In this step you will:

- Review the *Command Center Dashboard*
- Identify Segment Status
- Review Host Metrics
- Monitor Cluster Storage

Explore the Command Center environment

In this task you will explore the Greenplum Command Center environment

1. Select the *Dashboard* tab.



Uptime 0 d, 0 h, 25 min
GPDB Version 4.3.11.3
Connections 0
Last Sync 2017-02-02 21:12:38

Segment Health
Database State Normal
Segment Status 4
4 Up
0 Down
Mirrors Acting as Primary 0
Segment Hosts 2
Recommended Actions None

Queries: Running 0, Queued 0
CPU: System 0.5%, User 0.22%
Memory: In Use 8.74%

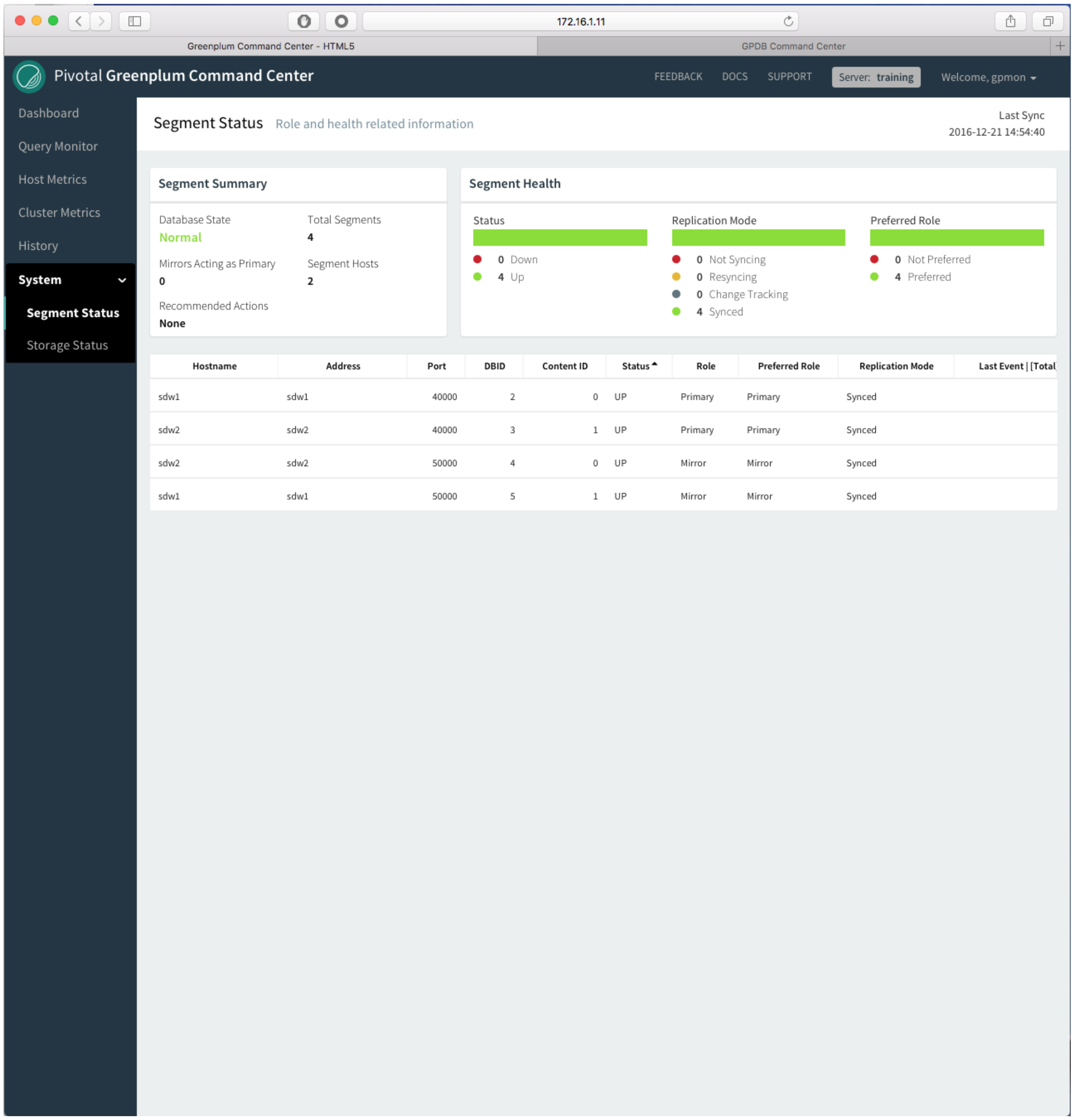
Disk Usage Summary
GP Segments 95.3GB Free
GP Masters 89.17GB Free

What is the Health of the database?

How much space is available on the Master server?

How much space is available on the Segment server(s)?

2. Select the *System* tab.



Segment Status Role and health related information

Last Sync
2016-12-21 14:54:40

Segment Summary

Database State Normal	Total Segments 4
Mirrors Acting as Primary 0	Segment Hosts 2
Recommended Actions None	

Segment Health

Status	Replication Mode	Preferred Role
● 0 Down	● 0 Not Syncing	● 0 Not Preferred
● 4 Up	● 0 Resyncing	● 4 Preferred
	● 0 Change Tracking	
	● 4 Synced	

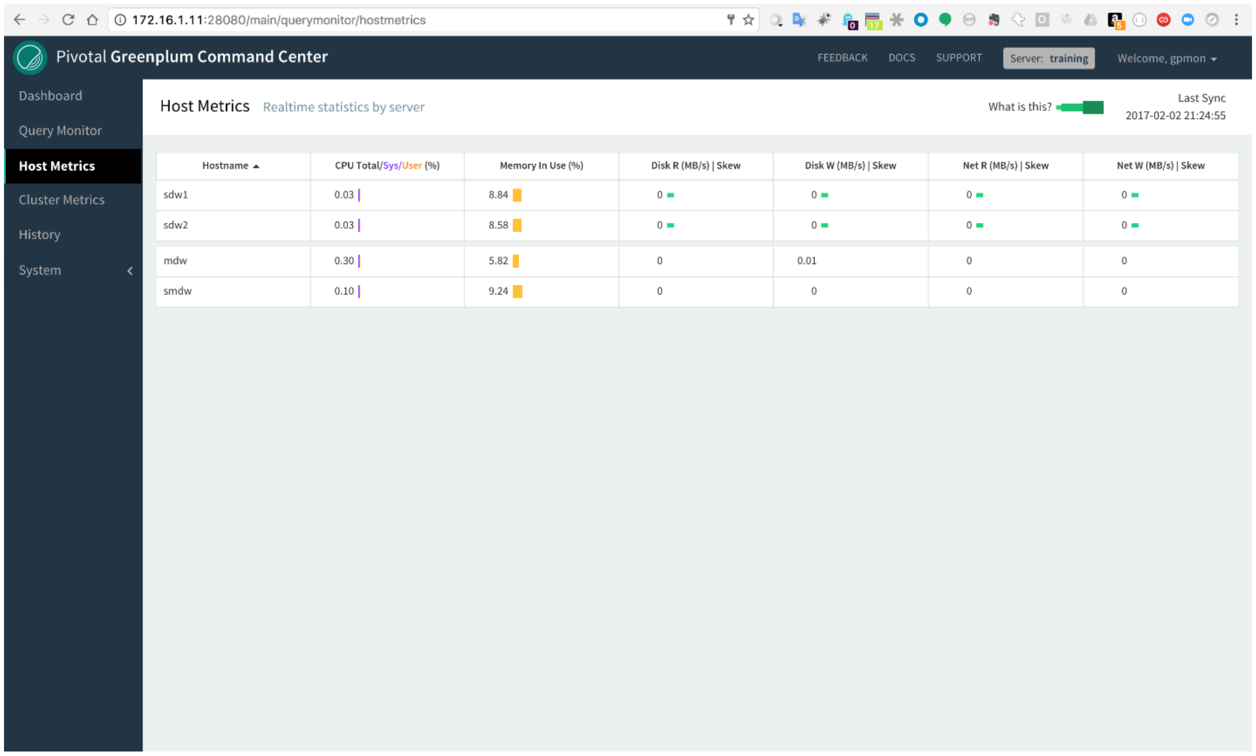
Hostname	Address	Port	DBID	Content ID	Status	Role	Preferred Role	Replication Mode	Last Event | [Total
sdw1	sdw1	40000	2	0	UP	Primary	Primary	Synced	
sdw2	sdw2	40000	3	1	UP	Primary	Primary	Synced	
sdw2	sdw2	50000	4	0	UP	Mirror	Mirror	Synced	
sdw1	sdw1	50000	5	1	UP	Mirror	Mirror	Synced	

How many segment instances are there?

What is the status of the segments?

What is the role of each segment?

3. Select the *Host Metrics* tab.



172.16.1.11:28080/main/querymonitor/hostmetrics

Pivotal Greenplum Command Center

FEEDBACK DOCS SUPPORT Server: training Welcome, gpmon

Dashboard
Query Monitor
Host Metrics
Cluster Metrics
History
System

Host Metrics Realtime statistics by server

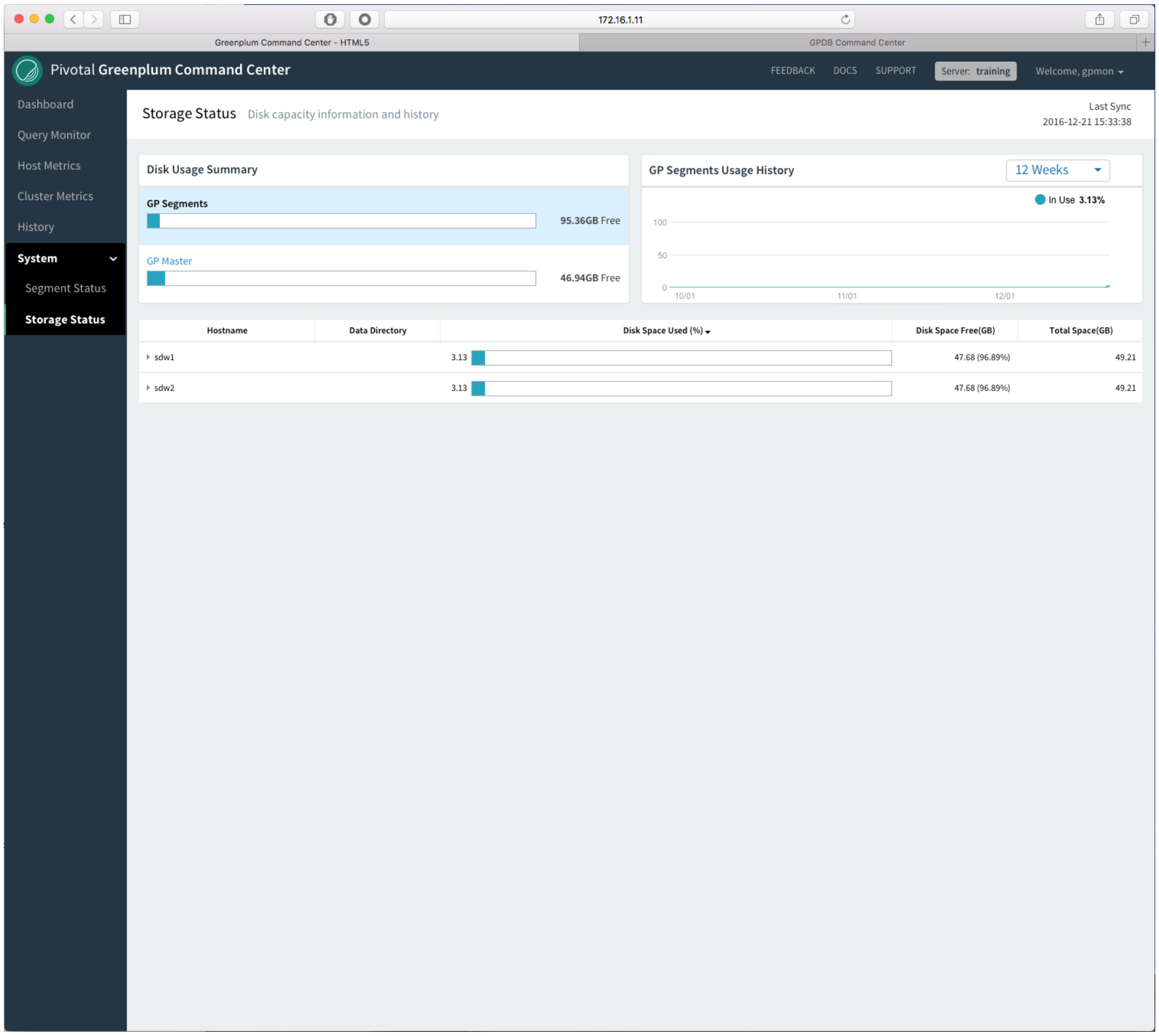
What is this?

Last Sync
2017-02-02 21:24:55

Hostname ▲	CPU Total/Sys/User (%)	Memory In Use (%)	Disk R (MB/s) | Skew	Disk W (MB/s) | Skew	Net R (MB/s) | Skew	Net W (MB/s) | Skew
sdw1	0.03	8.84	0	0	0	0
sdw2	0.03	8.58	0	0	0	0
mdw	0.30	5.82	0	0.01	0	0
smdw	0.10	9.24	0	0	0	0

How would you describe the amount of skew on the segments?

4. From the *System* tab select *Storage Status*



Storage Status Disk capacity information and history

Last Sync
2016-12-21 15:33:38

Disk Usage Summary

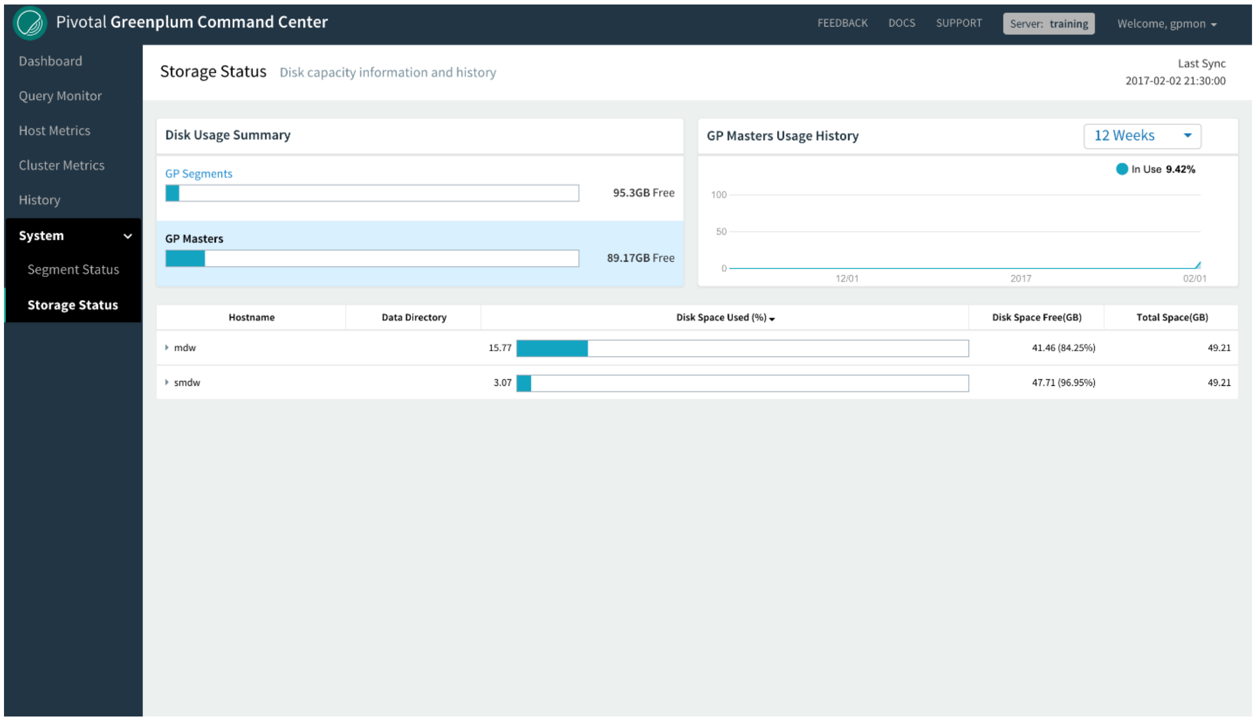
GP Segments 95.36GB Free

GP Master 46.94GB Free

GP Segments Usage History

Hostname	Data Directory	Disk Space Used (%)	Disk Space Free(GB)	Total Space(GB)
sdw1		3.13	47.68 (96.89%)	49.21
sdw2		3.13	47.68 (96.89%)	49.21

By default the Storage Summary for the segments (`sdw1` & `sdw2`) are displayed. To see the Storage Summary for the Segment Servers select **GP Master** in the **Disk Usage Summary** section.



Storage Status Disk capacity information and history

Last Sync
2017-02-02 21:30:00

Disk Usage Summary

GP Segments 95.3GB Free

GP Masters 89.17GB Free

GP Masters Usage History

● In Use 9.42%

Hostname	Data Directory	Disk Space Used (%)	Disk Space Free(GB)	Total Space(GB)
mdw		15.77	41.46 (84.25%)	49.21
smdw		3.07	47.71 (96.95%)	49.21

What is the data directory for the Master (`mdw`) server?

What is the percentage of space used on `mdw`? `sdw1`? `sdw2`?

GPDB Administrator – Lab 6

Data Definition Language

Purpose

In this lab, you will learn how to manage databases and schemas in the Greenplum Database. You will create a database called `faa` and a schema called `faadata`. You will also set the schema search path and learn how to verify which schema you are in.

You will also create some tables in the Greenplum Database and learn how the Greenplum distribution key for a table is chosen.

Create Databases and Schemas

1. Logged in as `gpadmin` on `mdw`, create a new database called `faa` using the `createdb` Greenplum client.

```
createdb faa
```

2. Set the `faa` database to be the default database you connect to when you start a psql session without specifying the database name.

```
export PGDATABASE=faa
```

3. Verify that the `PGDATABASE` environment variable is set correctly:

```
echo $PGDATABASE
```

   It should now be set to `faa`.

4. Start a psql session and connect to the faa database. If the `PGDATABASE` environment variable is set correctly, you do not need to supply the database name.

```
psql
```

5. At the `psql` prompt, list all the databases in the system with the `\l` meta-command.

```
\l
```

Which ones do you see? Is the `faa` database there?

6. At the `psql` prompt, create a schema named `faadata` using the `CREATE SCHEMA` SQL command.

```
CREATE SCHEMA faadata;
```

7. Change the search path on the `faa` database so that the new `faadata` is the default schema.

```
ALTER DATABASE faa SET search_path TO faadata, public, pg_catalog;
```

8. The change to the search path will not become visible until the next client connection. Use the `\c` meta-command to reconnect to the `faa` database:

```
\c faa
```

9. Run the `current_schema()` function to verify that you are indeed in the `faadata` schema.

```
SELECT current_schema();
```

10. Examine the value of the `search_path` configuration parameter to verify it is correct.

```
SHOW search_path;
```

11. Use the `\dn` meta-command to list the schemas in the database.

```
\dn
```

Which ones do you see? ____

Which ones are for system-level objects and which ones are for user-created objects?

Summary

You can create or drop databases using the `CREATE DATABASE` or `DROP DATABASE` commands or the `createdb` or `dropdb` client programs respectively.

By default, every newly created database has a schema named public, which is where objects are created by default.

If you create your own schemas and do not want to use qualified names all the time, you should set the search_path parameter to ensure your schema is first in the search path. You can use the `ALTER DATABASE` command to set the schema search path for the database.

You can also use the `ALTER ROLE` command to set the schema search path for a particular role.

Create Tables

You will create tables in the faa Greenplum database and learn how the Greenplum distribution key for a table is chosen.

For the purpose of this lab exercise, the faa database will have two tables:

- The `test_table` table will act as a fact table.
- The `test_table2` table will act as a dimension table.

1. At the `psql` prompt, create a new table with the following definition.

```
CREATE TABLE test_table
(
id int PRIMARY KEY,
name varchar(30) NOT NULL,
origin varchar(30) NOT NULL,
meaning text
);
```

   Note that the table has a `PRIMARY KEY` and there is no explicit `DISTRIBUTED BY` clause given. The primary key is automatically chosen as the distribution key in this case.

2. Create a table with the following definition and explicitly declare a distribution key using a `DISTRIBUTED BY` clause.

```
CREATE TABLE test_table2 (
id int,
rank int,
year int,
count int)
DISTRIBUTED BY (id, year);
```

3. Why is this distribution key a good choice for this table?

   What would be the distribution key if you left out the `DISTRIBUTED BY` clause?

Would that be a good distribution key for this table? Is that a unique key?

4. Use the `\dt` meta-command to list the tables in the database:

```
\dt
```

   Which ones do you see? Are the test_table and test_table2 tables there?

5. Exit psql:

```
\q
```

Summary

When creating the table, there is an additional clause to declare the Greenplum distribution key column(s).

If a `DISTRIBUTED BY` clause is not supplied, then either the `PRIMARY KEY`, if the table has one, or the first column of the table will be used. This may or may not be the desirable distribution key.

To ensure an even distribution of data, choose a distribution key with high cardinality. If a good choice of distribution columns is not available, choose `DISTRIBUTED RANDOMLY` as the distribution key.

View Indexes and Schemas

1. Start a `psql` session.

```
psql
```

   Verify you are in the faa database. If not, issue the following meta-command: `\c faa`.

2. At the `psql` prompt, create a new view on the test tables:

```
CREATE VIEW test_view AS
SELECT test_table.id
FROM test_table, test_table2
WHERE test_table.id < 11 AND test_table.id = test_table2.id;
```

3. Confirm the view definition is correct.

```
\d test_view
```

4. Create an index on the `id` column of the `test_table2` table:

```
CREATE INDEX test_index ON test_table2 (id);
```

5. Confirm the index was created:

```
\di
```

6. Create a sequence table that you can use to assign unique ids when inserting new records into the `test_table` table. The `id` value in the `test_table` table is currently 1, so the sequence starts at 1.

```
CREATE SEQUENCE test_table_seq START 1;
```

7. Examine the sequence table you just created.

```
SELECT * FROM
  test_table_seq;
```

Exit psql.

```
\q
```

Summary

Views allow you to store frequently used queries and then access them in `SELECT` statements as if they were a regular table. Here, you created a view that performs a JOIN between your two test tables, including a filter on the `id` column.

Indexes are not always the performance enhancer in the Greenplum Database as they are in traditional database management systems. In some cases indexes can improve query performance, and in some cases indexes have no effect or can slightly degrade performance. You have added an index on the rank table id column, since that column will be used often in your queries to join with the names table. When you get to query profiling, you will determine if this index is indeed being utilized.

Sequences are used to generate numbers, helpful for incrementing unique id columns such as the id column of your names table. You can use the sequence when you insert new names into this table to generate a unique id number that won't conflict with the other id numbers already being used.

GPDB Administrator – Lab 7

Data Manipulation Language and Data Query Language

Purpose

In this lab, you will familiarize yourself with the `INSERT`, `UPDATE`, and `DELETE` SQL commands.

Insert, Update, and Delete Records

1. Connect to the `faa` database as the `gpadmin` user, if not already connected.

   ```
   psql faa
   ```

   You can also specify the database using the `-d` option followed by the database name.

2. At the `psql` prompt, add a new record to the `test_table` table.

   ```
   INSERT INTO test_table VALUES
   (nextval('test_table_seq'), 'Esme', 'French', 'to love');
   ```

3. Check to see that the record you created is there.

   ```
   SELECT * from test_table WHERE name='Esme';
   ```

4. Update the record to change the name column to `Sophie` for any instances where the name column is `Esme`.

   ```
   UPDATE test_table SET name='Sophie' WHERE name='Esme';
   ```

5. Check to see that the record was changed:

   ```
   SELECT * from test_table WHERE name='Sophie';
   ```

6. Delete the record. Do not forget the `WHERE` clause or you will delete all of the rows.

   ```
   DELETE from test_table WHERE name='Sophie';
   ```

7. Check to see that the record was deleted.

   ```
   SELECT * from test_table WHERE name='Sophie';
   ```

8. Exit your psql session.

```
\q
```

Creating the `names` Database

Create a database, and some objects within it, for use within the following task.

1. Log into the `template1` database.

```
psql template1
```

2. Create a database called `names`.

```
CREATE DATABASE names;
```

3. Connect to the names DB.

```
\c names
```

4. Using the SQL statement provided, create a schema and tables in the `names` database.

```
CREATE SCHEMA baby;

CREATE TABLE baby.names
(
id INT PRIMARY KEY,
name VARCHAR(30),
origin VARCHAR(30),
meaning TEXT
);

CREATE TABLE baby.rank
(
id INT,
rank INT,
year INT,
gender CHAR(1),
count INT
)
DISTRIBUTED BY (id, gender, year);
```

5. Disconnect from the database.

```
\q
```

Accessing Data to Generate Reports

In this task, you use a variety of statements to generate a report. The report identifies the least popular baby names, across boys and girls.

1. Connect to the names database as the `gpadmin` user, if not already connected.

```
psql names
```

2. Update the `search_path` for the names database so that you can easily access the tables from the names database without needing to specify the schema name.

```
alter database names set search_path to baby, public, pg_catalog;
```

3. Reconnect to the database to access the settings.

```
\c names
```

4. Identify the tables in this database and use `COUNT` to verify the tables have no rows.

```
\dt
  SELECT COUNT(*) FROM names;
  SELECT COUNT(*) FROM rank;
```

5. Using the psql `copy` command, copy data from local files into the `rank` table. The data will be copied from the CSV file, `/home/gp/sql/load_files/boys`, and includes a header line. This command behaves like the SQL `COPY` command which will be covered later in the course. For now, you are populating tables with values to be manipulated and accessed.

```
\copy rank from /home/gp/sql/load_files/boys header delimiter as
',' csv;
```

6. Using `COUNT`, verify data has now been loaded into the rank table.

```
SELECT COUNT(*) FROM rank;
```

7. Using the syntax shown earlier, load data from the files noted to the corresponding tables.

Use the up arrow (↑) to scroll through your PSQL buffer until you find the \copy command and change the line as needed. Use the left (←) and right (→) arrows to move through the command to the required position.

File	**Table**
/home/gp/sql/load_files/girls	rank
/home/gp/sql/load_files/name_ids	names

```
\copy rank from /home/gp/sql/load_files/girls header delimiter as ',' csv;
```

```
\copy names from /home/gp/sql/load_files/name_ids header delimiter as ',' csv;
```

8. Use `COUNT` to verify that the rows have been loaded into the respective tables.

```
SELECT COUNT(*) FROM rank;

SELECT COUNT(*) FROM names;
```

9. The rank table highlights the number of occurrences of babies given a specific name. Each row indicates the ID for the name, the name's ranking or weight, the year the information was collected, and the number of babies given the specific name. The names table associates the ID with the name and provides the origin and local meaning of the name. While the rank can be used to determine the least popular name as defined by the rank, this task will have you use aggregate functions against the count associated with the names.

   Use the `MIN` function to find the lowest count of a name for each year. Remember, `MIN` is an aggregate function and requires the use of `GROUP BY`.

```
SELECT min(count), year FROM rank GROUP BY year;
```

10. Examine the rank table to list the ID(s) that had the lowest count for the year 2004. Use the information returned from the previous example to complete the request.

```
SELECT id FROM rank WHERE year=2004 AND count=164;
```

11. This two-step process can easily be changed to a single step and would allow us to collect the information for all years involved. Use the WITH clause to define a common table expression that allows you to return the ID, year, and count for the lowest assigned name for each year.

```
WITH min_count AS (
SELECT min(count) AS mcount, year
FROM rank
GROUP BY year
)
SELECT rank.id, rank.year, rank.count
FROM min_count, rank
WHERE min_count.year = rank.year
AND min_count.mcount = rank.count;
```

Tip: If you are familiar with the vi editor, you can edit the previous PSQL statement using the `\e` command. It will take you into the editor and allow you to create or modify your statement. When done modifying your statement, use `:wq!` to save and execute the statement. If you prefer Emacs or Vim, exit `psql`, update the EDITOR environment variable to point to your preferred editor, and re-enter PSQL. For example, to set the editor to Emacs in this environment, execute the following in the shell or add it to your `.bash_profile`.

```
export EDITOR=/usr/bin/emacs
```

12. If this will be a frequently accessed statement, it would be best to save it as a view. Create a view called `min_count_vw` that contains the common table expression you defined in the previous step.

```
CREATE VIEW min_count_vw AS
WITH min_count AS
(SELECT min(count) AS mcount, year FROM rank GROUP BY year)
SELECT rank.id, rank.year, rank.count from min_count, rank
 WHERE min_count.year = rank.year AND min_count.mcount = rank.count;
```

13. Using the view you created, list the least assigned baby names for the years collected. Make the report easier to read by sorting the results by year.

```
SELECT name, year
FROM min_count_vw, names
WHERE names.id = min_count_vw.id
ORDER BY year;
```

Summary

If you have many rows to insert, consider using COPY or external tables instead of INSERT. UPDATE is used to change one or more column values of an existing row. DELETE is used to delete rows from a table based on some qualifying condition.

An unqualified delete on a table will delete all rows. A more efficient way to delete all rows from a table is the TRUNCATE command.

A variety of tools is available to help generate information for reports. A portion of this lab exercise focused on using some aggregate functions to collect the information that was needed. Views and common table expressions let you simplify more complex statements and make these statements available for future use.

GPDB Administrator – Lab 8

Roles and Priveleges

Purpose

Learn how to create user-level and group-level roles and assign permissions.

Estimated completion time: 20 minutes

Manage User and Group Roles

User-level roles can log in to a database. Group-level roles are useful for managing permissions and access privileges to database objects. You will learn how to grant privileges on database objects to a group-level role.

Members (user-level roles) of the group inherit the group-level access privileges.

In this step you will:

- Create a group-level role
- Create a user-level role
- Set access permissions

1. If your are not already connected, open a Putty session to `mdw` and log in with the following credentials:

   username: `gpadmin`

   password: `changeme`

2. Start a `psql` session.

```
psql
```

3. If you are not in the `faa` database, issue the following meta-command to connect to the database: `\c faa`.

```
\c faa
```

4. Create a group-level role named `admin` and grant appropriate permissions for an administrator group-level role.

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

5. Create a user-level login role for yourself and give it appropriate permission attributes:

   Replace `student` with your name.

```
CREATE ROLE student WITH LOGIN;
```

6. Add the user-level role you just created as a member of the group-level role `admin` using the `GRANT` command.

   Replace `student` with your name.

```
GRANT admin TO student;
```

7. Confirm that the roles are configured as expected by using the `\du` meta-command in `psql`.

```
\du
```

   You results should look something like this:

```
                       List of roles
 Role name |             Attributes              | Member of
-----------+-------------------------------------+-----------
 admin     | Create role, Create DB, Cannot login |
 gpadmin   | Superuser, Create role, Create DB    |
 student   |                                      | {admin}
```

   This shows a list of all roles in the system.

8. Change to the user-level role you just created using the `SET ROLE` command.

   Replace `student` with your name.

```
SET ROLE TO student;
```

   Note that the prompt has changed to reflect the level of the user you have connected into the database with.

9. Run a query as this role. Select all columns from `test_table`:

```
SELECT * FROM test_table;
```

The command did not succeed. Why not?

10. Exit `psql`:

```
\q
```

11. Set the `PGDATABASE` environment variable to `faa`.

```
export PGDATABASE=faa
```

12. Log in to the database as `gpadmin`.

```
psql
```

13. Grant privileges to the `faa` database to the `admin` role.

```
GRANT ALL ON DATABASE faa TO admin WITH GRANT OPTION;
```

14. Grant privileges to the `faadata` schema to the `admin` role.

```
GRANT ALL ON SCHEMA faadata TO admin WITH GRANT OPTION;
```

15. Grant privileges to `test_table` and `test_table2` to the `admin` role.

```
GRANT ALL ON TABLE test_table, test_table2 TO admin WITH GRANT OPTION;
```

16. Exit `psql`:

```
\q
```

Add a connection record for the role you created to the `pg_hba.conf`. This will allow you to login to the `faa` database.

Replace `student` with your name.

```
echo "local     all     student          trust" >> /data/master/gpseg-1/pg_hba.conf
```

17. Use the `gpstop -u` utility to make new configuration settings active after editing postgresql.conf or pg_hba.conf files.

```
gpstop -u
```

18. Start a `psql` session as the user-level role you created earlier and connect to the `faa` database:

    Replace `student` with your name.

```
psql -U student faa
```

19. Run the `SELECT` query you ran earlier to test access to the tables. You should now have permissions to see this view (and its schema).

```
SELECT * FROM test_table;
```

20. Exit `psql`:

```
\q
```

21. Exit `psql` in each of the three terminal sessions. Close all terminal sessions.

Summary

Greenplum manages database access permissions using the concept of roles. The concept of roles subsumes the concepts of users and groups. A role can be a database user, a group, or both. Roles can own database objects (for example, tables) and can assign privileges on those objects to other roles to control access to the objects. Roles can be members of other roles, thus a member role can inherit the attributes and privileges of its parent role.

Note that if you are using table inheritance and partitioning, table privileges are not passed down from parent tables to child tables. You must explicitly set privileges on each child table. If you use the Greenplum management utilities to partition a table, the utility takes care of passing down the table permissions for you.

When you create a new login role, you must be sure that the `pg_hba.conf` configuration file of the master allows the role to connect to the Greenplum Database. Otherwise the role will be denied access.

GPDB Administrator – Lab 9

Controlling Access

Purpose:

In this exercise, you design and implement the basic security architecture for your data mart. You will create two new users and use groups to control the level of access that users, who will be members of the group, receive. This method makes it easier to administer privileges by working with groups instead of working with individual user accounts.

Estimated completion time: 20 minutes

Implement Basic Security at the Group Level

In this step you will:

- Create user-level roles

1. If you are not already connected, open a Putty session to `mdw` and log in with the following credentials:

   username: `gpadmin`

   password: `changeme`

2. Connect to the `faa` database as `gpadmin`.

```
psql faa
```

3. Create a new user-level role that will be used to verify the security implementation:

```
CREATE ROLE batchuser LOGIN;
```

4. Change to the new `batchuser` role.

```
SET ROLE batchuser;
```

5. Query the `test_table` table.

```
SELECT * FROM faadata.test_table;
```

Did it succeed? If not, why not?

6. Change back to the `gpadmin` role.

```
SET ROLE gpadmin;
```

7. Grant permissions to the `faadata` schema to the `batchuser` role.

```
GRANT USAGE ON SCHEMA faadata TO batchuser;
```

While you have been granted `USAGE` access on the schema, you still do not have access to the existing tables that do not specifically grant access to all.

8. Grant access of our `test_table` to `batchuser`:

```
GRANT select, insert, update, delete ON faadata.test_table TO batchuser;
```

9. Change to the `batchuser` role.

```
SET ROLE batchuser;
```

10. Query the `test_table` table.

```
SELECT * FROM faadata.test_table;
```

You should now have `SELECT`, `INSERT`, `DELETE`, and `UPDATE` access on `test_table` in the `faadata` schema.

11. Create a group-level role and assign privileges indirectly. First, change back to the gpadmin role.

```
        SET ROLE gpadmin;
```

12. Revoke privileges to `faadata.test_table` from `batchuser`.

```
      REVOKE SELECT, INSERT, UPDATE, DELETE ON faadata.test_table from batchuser;
```

13. Revoke privileges to the `faadata` schema from `batchuser`:

```
        REVOKE USAGE ON SCHEMA faadata FROM batchuser;
```

14. Create the group-level role `batch` that you will assign privileges to.

```
CREATE ROLE batch;
```

15. Grant `USAGE` on the `faadata` schema to the `batch` group-level role.

```
GRANT USAGE ON SCHEMA faadata TO batch;
```

16. Grant access on the `test_table` to the `batch` group-level role.

```
GRANT SELECT, INSERT, UPDATE, DELETE ON faadata.test_table TO batch;
```

    Members of the group-level role `batch` now have `SELECT`, `INSERT`, `UPDATE`, and `DELETE` privileges on **faadata.test_table**.

17. Assign the `batchuser` user-level role to the `batch` group-level role.

```
GRANT batch TO batchuser;
```

18. Create a second user-level role, `batchuser2`:

```
CREATE ROLE batchuser2 login;
```

19. Assign the `batchuser2` user-level role to the `batch` group-level role:

```
GRANT batch TO batchuser2;
```

20. Confirm that the `batchuser` and `batchuser2` user-level roles have been added to the `batch` group-level role.

```
\du
```

    Your results should look similar to the following:

```
                        List of roles
 Role name  |              Attributes               | Member of
------------+---------------------------------------+-----------
 admin      | Create role, Create DB, Cannot login |
 batch      | Cannot login                          |
 batchuser  |                                       | {batch}
 batchuser2 |                                       | {batch}
 gpadmin    | Superuser, Create role, Create DB     |
 student    |                                       | {admin}

faa=#
```

21. Verify that both user roles have access the `faadata.test_table` table.

Change to the `batchuser` role:

```
        SET ROLE batchuser;
```

22. Query `test_table`:

```
        SELECT * from faadata.test_table;
```

You should have access to the table because of the role's relationship to the group-level `batch` role.

23. Change to the `batchuser2` role and verify this user has access to the same table:

```
        SET ROLE batchuser2;
```

24. Query `test_table`.

```
        SELECT * from faadata.test_table;
```

Again you should have access to the table because of the role's relationship to the group-level `batch` role.

Exit `psql`

```
        \q
```

Summary

You should work with the user community and the administrators to come up with a security architecture that will fit your access requirements.

Table and schema level access is done through the use of roles.

Row level access is best accomplished with a combination of roles and views against the data that has all of the security logic built into the view. While this may inhibit performance for some queries, it will also ensure that your data is secure and only "need to know" access is given to the end users.

Row level security may also be performed by reporting engines. This may suffice if your data security level is low.

GPDB Administrator – Lab 10

Managing Resources

Purpose

Learn how to manage resource queues

Estimated completion time: 20 minutes

Manage Resource Queues

You will create a resource queue and assign a user role to this resource queue. The username you use is the name you added in the `pg_hba.conf` file. For testing purposes, the resource queue you create will have an *ACTIVE THRESHOLD* of 1, meaning that only one active statement submitted through the resource queue will be allowed to run at any given time.

In this step you will:

- Create a resource queue
- Test resource queue functionality

1. If you are not already connected, open a putty session to `mdw` and log in with the following credentials:

   username: `gpadmin`

   password: `changeme`

2. Start a `psql` session:

   If you are not in the `faa` database, issue the following meta-command to connect to the database: `\c faa`.

```
psql

\c faa
```

3. Create the resource queue, `adhoc`, with *ACTIVE THRESHOLD* equal to 1:

```
CREATE RESOURCE QUEUE adhoc ACTIVE THRESHOLD 1;
```

4. Confirm the resource queue was created correctly by checking the `pg_resqueue` system table:

```
SELECT * from pg_resqueue;
```

The results should look similar to the following:

```
rsqname    | rsqcountlimit | rsqcostlimit | rsqovercommit | rsqignorecostlimit
----------+---------------+--------------+---------------+--------------------
adhoc      |             1 |           -1 | f             |                  0
pg_default|            20 |           -1 | f             |                  0
(2 rows)
```

Note: `pg_default` is the default resource queue to which all roles are assigned if the resource queue is not specified when the role is created.

5. Assign the user-level role that you added to the `adhoc` resource queue:

Replace `<your_name>` with your name.

```
ALTER ROLE <your_name> RESOURCE QUEUE adhoc;
```

6. Confirm that the role was assigned the `adhoc` resource queue by querying the `pg_resqueue` and `pg_roles` system tables.

```
SELECT rolname, rsqname FROM pg_roles AS r, pg_resqueue AS q WHERE
r.rolresqueue=q.oid;
```

The results should look similar to the following:

```
  rolname    |  rsqname
------------+------------
 gpadmin     | pg_default
 admin       | pg_default
 batchuser   | pg_default
 batchuser2  | pg_default
 batch       | pg_default
 drude       | adhoc
(6 rows)
```

7. Exit `psql`:

```
\q
```

8. Start a new `psql` session as the `<your_name>` user-level role:

```
psql -U <your_name>
```

   Replace `<your_name>` with your name.

9. Verify that the resource queue is working.

   To hold a query open, open a cursor within a transaction. This action holds the one active query slot you are allowed in the adhoc resource queue. The cursor will remain open until the transaction is closed, which will give you the time required to test the resource queue's limits:

10. Leave the current Putty session open (**session_1**) and open a second putty session (**session_2**) to `mdw` and log in with the following credentials:

```
BEGIN;
DECLARE rqtest CURSOR FOR SELECT * FROM test_table;
```

   username: `gpadmin`

   password: `changeme`

   To test the resource queue, you must run a query in **session_2** to see if it is allowed to run while the **session_1** is holding open the resource queue's active query slot.

11. Log in to `psql` with `<your_name>`:

   Replace `<your_name>` with your name.

```
psql -U <your name> faa
```

12. Run a query in the **session_2** terminal window:

```
SELECT * FROM test_table2;
```

   Did the query run? What happened?

13. Open a third Putty session (**session_3**) to mdw and login in as `gpadmin`.
14. Connect to the `faa` database as `gpadmin`.

```
psql faa
```

15. Review the state of the resource queues:

```
SELECT * FROM pg_resqueue_status;
```

The results should look similar to the following:

rsqname	**rsqcountlimit**	**rsqcountvalue**	**rsqcostlimit**	**rsqcostvalue**	**rsqwaiters**	**rsqholders**
adhoc	1	1	-1		1	1
pg_default	20	0	-1		0	0

(2 rows)

The `rsqwaiters` column shows the number of statements waiting in a queue. The `rsqholders` column shows the number of queries currently running in a queue.

16. Leaving the **session_2** putty open and visible, return to the **session_1** putty (the one with the open cursor) and end the transaction:

```
END;
```

17. Immediately examine the **session_2** terminal. The waiting query should have executed immediately after the transaction in the **session_1** terminal was ended.

```
SELECT * FROM test_table2;
```

The results should look similar to the following:

```
 id | rank | year | count
----+------+------+-------
(0 rows)
```

18. Exit `psql` in each of the three terminal sessions.

Close all putty sessions.

Summary

Administrators can create resource queues for the various types of workloads in their organization. The administrator would then set limits on the resource queue based on his/her estimate of how resource intensive the queries associated with that workload are likely to be.

Database roles (users) are then assigned to the appropriate resource queue. A resource queue can support multiple roles.

GPDB Administrator – Lab 11

Workload Manager

Purpose:

In this lab you will perform installation and configuration tasks required for the Greenplum Workload Manager software to run.

Estimated completion time: 45 minutes

Install Workload Manager software on the Cluster

This task will install the Workload Manager software on the Master server.

1. If you are not already connected, open a putty session to `mdw` and log in with the following credentials:

   username: `root`

   password: `Piv0tal`

2. Switch user to `gpadmin`.

```
su - gpadmin
```

3. Change to the `/usr/local/greenplum-cc-web` directory

```
cd /usr/local/greenplum-cc-web
```

4. List the contents of `/usr/local/greenplum-cc-web`

```
ls
```

   The `gp-wlm-1.6.0.bin` file is the binary that you will use to install the Greenplum Workload Manager.

5. Launch the Greenplum Workload Manager installer using `bash`. The `--install` option is required. It specifies the directory where the Greenplum Workload Manager will be installed.

```
/bin/bash gp-wlm-1.6.0.bin --install=/home/gpadmin
```

Confirm all Workload Manager services are functioning correctly

This task will ensure that the Workload Manager software was successfully installed and started.

1. Source `/home/gpadmin/gp-wlm/gp-wlm_path.sh` to make Workload Manager commands available.

```
source /home/gpadmin/gp-wlm/gp-wlm_path.sh
```

2. Using the `svc-mgr.sh` utility check to see if all of the Workload Manager services are running.

```
svc-mgr.sh --service=all --action=cluster-status
```

   The response indicates that the `RabbitMQ`, `agent`, `cfgmon`, `rulesengine`, and `svcmon` services are running on the Master, Standby Master, and Segment hosts.

Use the Workload Manager Command Line Interface

This task will introduce the Workload Manager Command Line Interface.

1. Use the `gp-wlm` command line to display all of the configured resource queues.

```
gp-wlm --rq-show=all
```

   All configured resource queues and their attributes are displayed.

```
rsqname          resname              ressetting      restypeid
pg_default       active_statements    20              1
pg_default       max_cost             -1              2
pg_default       min_cost             0               3
pg_default       cost_overcommit      0               4
pg_default       priority             medium          5
pg_default       memory_limit         -1              6
```

Use the Workload Manager Shell

This task will introduce the Workload Manager shell.

1. From the Linux command line enter into the `gp-wlm` shell.

   ```
   gp-wlm
   ```

   The command prompt displays the name of the host where `gp-wlm` is running and the name of the Greenplum Database cluster.

2. List all commands available in the Workload Manager shell

   ```
   help
   ```

3. Display all of the configured resource queues

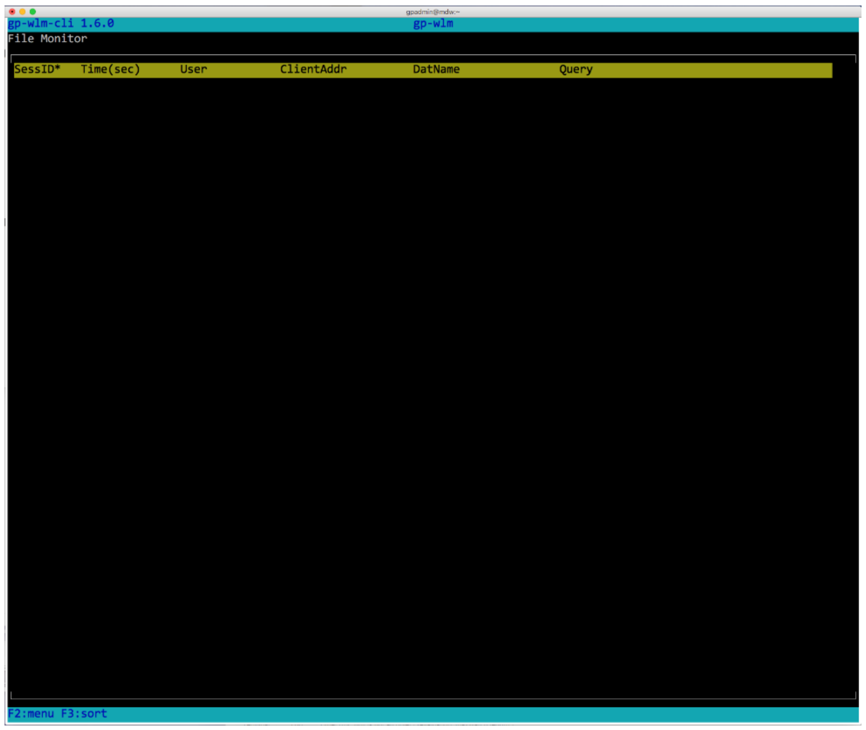
   ```
   rq show all
   ```

Use the Workload Manager Graphical Interface

This task will introduce the Workload Manager Graphical Interface.

1. Enter the Workload Manager Graphical Interface with the `gptop` command. `gptop` can be executed from the Linux command line or as a Workload Manager shell command.

```
gptop
```



```
gp-wlm-cli 1.6.0                                          gp-wlm
File Monitor

 SessID*   Time(sec)     User           ClientAddr         DatName              Query





F2:menu F3:sort
```

The Workload Manager Graphical Interface will be displayed. Enter `q` to exit the `gptop` utility.

GPDB Administrator – Lab 12

Implementing Table Storage Models, Compression, and Tablespaces

Purpose

The purpose of this lab is to create various types of supported Greenplum Database tables. You will create tables of varying types, load data to these tables, and discover which table type offers the best performance under specific conditions.

Creating Temporary Tables

1. Create the `dbstudent` database which will be used to support your tables.

```
createdb dbstudent
```

2. Connect to the `dbstudent` database:

```
psql dbstudent
```

3. Create a new schema called `studentdata`.

```
CREATE SCHEMA studentdata;
```

4. Change the `search_path` parameter on the database `dbstudent` so that the new studentdata schema is the default schema.

```
ALTER database dbstudent SET search_path TO
studentdata,PUBLIC,pg_catalog;
```

Reconnect to the database to re-read the `search_path` settings.

```
\c dbstudent
```

5. Create a temporary table called `dimairline_t` using the following SQL syntax.

```
CREATE TEMP TABLE DimAirline_T
(
AirlineID Smallint,
AirlineName Character varying(95)
)
WITH (OIDS=FALSE)
DISTRIBUTED BY (AirlineID);
```

6. List all the tables in the `dbstudent` database.

```
\dt
```

   Note that the table is listed in a temporary schema. Any other temporary tables created in this session will be saved to the same temporary schema.

7. Reconnect to the `dbstudent` database.

```
\c dbstudent
```

8. List all the tables in the `dbstudent` database.

```
\dt
```

   The temporary schema that was created when you created the temporary table is automatically dropped when the session ends. Any tables in the schema are also dropped. Reconnecting to the database creates another temporary schema which will be used should you create temporary tables in the new session.

Summary

Temporary tables exist in a special schema, so a schema name may not be given when creating a temporary table. When creating temporary tables, the table name must be unique to the session.

Temporary tables are automatically dropped at the end of the session or, optionally, at the end of a transaction with the ON COMMIT clause. This will occur when dropping the session or reconnecting to it, something that occurs often with business intelligence applications and reporting tools.

Temporary tables are a good way of handling complex and intensive SQL statements used for generating reports. They can be used to reduce performance impacts for queries that may have a tendency to generate computational skew.

Creating Compressed Tables

1. If not already connected to the database, connect to the dbstudent database.

```
psql dbstudent
```

2. In this task, you will create two tables: a compressed table and a regular heap table. These tables have an equal number of fields and records.

   Create a regular heap table called `dimairline` as shown below:

```
CREATE TABLE DimAirline
(
AirlineID Smallint,
AirlineName Character varying(95)
)
WITH (OIDS=FALSE)
DISTRIBUTED BY (AirlineID);
```

3. Create a compressed table called `x_airline` as shown below:

```
CREATE TABLE x_airline (LIKE dimairline)
WITH (appendonly=TRUE, oids=FALSE, compresstype=zlib,
compresslevel=5)
DISTRIBUTED BY (airlineid);
```

   Log out of the DB.

```
\q
```

4. Populate the heap table by using `bzcat` to dump the compressed CSV file to stdout, and piping that to the `psql` command shown here, where the `COPY` command specifies `STDIN` as the source of the data to be copied:

```
bzcat /rawdata/FAAData/DimAIRLINES.csv.bz2 | psql dbstudent -c
"COPY dimairline FROM STDIN WITH DELIMITER ',' CSV HEADER"
```

   Log back into the `dbstudent` DB.

```
psql dbstudent
```

5. Populate the compressed table with records from the dimairline table as shown.

```
INSERT INTO x_airline SELECT * FROM dimairline;
```

6. Verify the size of both tables by viewing the `sotdsize` column of the gp_toolkit.gp_size_of_table_disk table. This table stores information on all tables within the database. The size column, `sotdsize`, is displayed in bytes.

```
SELECT sotdsize FROM gp_toolkit.gp_size_of_table_disk
WHERE sotdtablename = 'dimairline';

SELECT sotdsize FROM gp_toolkit.gp_size_of_table_disk
WHERE sotdtablename = 'x_airline';
```

   Note that the listed sizes of the tables differ greatly. The compressed AO table, `x_airline`, is less than 25% of the size of the heap table, `dimairline`.

7. Exit the database once you have completed the steps.

```
\q
```

Summary

Both tables have the same number of fields and records. However, the compressed table is considerably smaller than the uncompressed table. Tables that use compression must be append-optimized (created using the `appendonly=true` specification).

Choosing greater levels of compression reduces storage consumption. Increasing compression requires more CPU cycles to access the data when required, either on reads or writes. For data that is not often accessed, it may be applicable to apply higher compression levels to the table, or partition. This ensures that older data that is not often accessed does not consume large amounts of storage and should balance well against how often it is accessed.

Creating a Column-Oriented Table

1. Create two tables in the `dbstudent` database. The tables will be called `factontimeperformance` and `c_factontimeperformance`. The `factontimeperformance` table is a regular heap table and `c_factontimeperformance` is a column-oriented table. Run the script below to create the heap table.

```
psql -f /rawdata/FAAData/CreateDbstudent_Performance.sql
dbstudent
```

2. The heap table `factontimeperformance` has been created. Connect to the `dbstudent` database and create the column-oriented table, `c_factontimeperformance`, based on `factontimeperformance`, as shown.

```
psql dbstudent

CREATE TABLE c_factontimeperformance (like
factontimeperformance)
WITH (appendonly=TRUE, orientation=COLUMN)
distributed randomly;
```

   Exit the database.

```
\q
```

3. Load the `FactOnTimePerformance` table from the compressed CSV data as you did earlier in this lab, but this time using a `for loop` in the `Bash` shell (this takes about five minutes).

```
for csv in /rawdata/FAAData/On_Time_On_Time_Performance_2008_*.csv.bz2 ; do
bzcat $csv | psql dbstudent -c "COPY FactOnTimePerformance FROM STDIN WITH
DELIMITER ',' CSV HEADER;" ; done
```

   Now, copy the data from this table into the column-oriented table you created earlier.

```
psql dbstudent -c "INSERT INTO c_FactOnTimePerformance SELECT *
FROM FactOnTimePerformance"
```

   - That second INSERT step should have taken only about half the time the previous step took.
   - Upon completion of these steps, each table should contain just over seven million records. This data represents the entire FAA flight data set for 2008.

4. Verify you have the same number of records on both tables.

```
psql dbstudent

SELECT count(*) FROM factontimeperformance;

SELECT count(*) FROM c_factontimeperformance;
```

5. Calculate how many times Southwest Airlines flew to Chicago in the year by 2008 running the query below. Run the query on both tables: `factontimeperformance` and `c_factontimeperformance`. You should see the same results for both tables.

```
SELECT count(*) FROM factontimeperformance WHERE
year = 2008 AND
airlineid = 19393 AND
destcityname = 'Chicago, IL';
```

```
SELECT count(*) FROM c_factontimeperformance WHERE
year = 2008 AND
airlineid = 19393 AND
destcityname = 'Chicago, IL';
```

6. To verify which query is faster, analyze the queries using `explain analyze` command. This command will be discussed in greater detail in later labs. For now, make a note of the `Total runtime` line.

   Run the command for the `factontimeperformance` table which is a row-oriented heap table.

```
EXPLAIN ANALYZE SELECT count(*) FROM factontimeperformance
WHERE year = 2008 AND airlineid = 19393 AND destcityname =
'Chicago,
```

7. Execute the same command for the `c_factontimeperformance` table, which is an append-only column oriented table.

```
EXPLAIN ANALYZE SELECT count(*)
FROM c_factontimeperformance WHERE year = 2008 AND airlineid =
19393 AND destcityname = 'Chicago, IL';
```

   You should observe that the completion time on the column-oriented table is significantly less than the heap table. Since only a few columns are accessed, column-oriented tables offer an advantage in this query.

8. Exit the database.

```
\q
```

Summary

For most general purpose or mixed workloads, row-oriented storage offers the best combination of flexibility and performance. However, there are specific use cases where a column-oriented storage model provides more efficient I/O and storage.

Column-oriented tables offer strong performance gains over row-oriented tables when the table is being used mostly for reads and there are few columns being selected against or aggregated over. For data that is not updated often or is limited to a single column, column-oriented tables may offer a distinct advantage. A negative impact can be seen if you need to select a majority of the columns on a column-oriented table. Understanding the data and how it will be used will help you choose the appropriate storage model for the table.

GPDB Administrator – Lab 13

Data Loading

Purpose

The purpose of this lab is to learn the different techniques for loading data into a Greenplum system.

You will start by using the COPY command. The COPY command is well suited for importing small amounts of data in text or CSV format.

The second technique you will learn for loading data is creating external tables and using an insert statement from the external table into an internal Greenplum table.

The third technique that you will learn for loading data is using the gpfdist utility.

Create a Dimension Table and Load Data Using `COPY`

1. Connect to the `faa` database as the `gpadmin` user:

```
psql faa
```

2. In a separate terminal where you have a `psql` session started, create the first Dimension table `DimAirline`:

```
CREATE TABLE DimAirline
(
AirlineID Smallint,
AirlineName Character varying(95)
)
WITH (OIDS=FALSE)
DISTRIBUTED BY (AirlineID);
```

Note: You can also copy and paste this command from the script `DDLScript.sql` located in `/rawdata/FAAData`.

3. Log out of the faa DB.

```
\q
```

4. Use the `COPY` command to populate the `DimAirline` table using the data from standard input.

```
bzcat /rawdata/FAAData/DimAIRLINES.csv.bz2 | psql faa -c "COPY DimAirline FROM
STDIN CSV HEADER SEGMENT REJECT LIMIT 10 ROWS"
```

   In this example, a maximum of ten row failures is allowed before the copy fails.

5. Verify that the table contains data.

```
psql faa -c "SELECT * FROM DimAirline LIMIT 20"
```

Summary

The COPY command can be used to load data from a file (or from standard input) into a table. COPY can be run as a single command in non-interactive mode or in an interactive psql session. COPY does not load data in parallel, so it is better suited for loading smaller amounts of data. If loading large amounts of data, external tables offer better load performance.

If your data load file contains mixed data, you can use COPY to quickly load the data into a temporary table and then insert it from there through the parent table.

Create a Dimension Table and Load Data Using External Tables

1. Define an external table called `public.airport_external` that points to a CSV file located on another server. The file can be accessed as `file://sdw1/loaddata/DimAIRPORTS.csv`. To create the table, use the `CREATE EXTERNAL TABLE` command.

```
psql faa

CREATE EXTERNAL TABLE public.airport_external
(
AirportID Character(3),
AirportDescription Character varying(85)
)
LOCATION ('file://sdw1/loaddata/DimAIRPORTS.csv')
FORMAT 'CSV' (HEADER)
LOG ERRORS INTO public.airport_err
SEGMENT REJECT LIMIT 10 ROWS;
```

   Note: When using the `file://` protocol, the external data file(s) must reside on a segment host in a location accessible by the Greenplum super user (gpadmin).

2. Verify that you are able to access data from the external table by counting the number of records:

```
SELECT COUNT(*) FROM public.airport_external;
```

   If you receive an error, verify the syntax you used to create the external table, the host name for the file, and the file name. Verify that the file is in the location you specified in the external table syntax. If necessary, drop the external table using `DROP EXTERNAL TABLE public.airport_external` and recreate the external table.

3. Create a heap table to load the external data into.

```
CREATE TABLE faadata.DimAirport
(
AirportID Character(3),
AirportDescription Character varying(85)
)
DISTRIBUTED BY (AirportID);
```

Note: Here, we omitted the "`WITH (OIDS=FALSE)`", because the default behavior is to **not** assign object identifiers (OIDs) to rows. You can also copy and paste this command from the `DDLScript.sql` script.

4. Insert the data into the `DimAirport` from the external table.

```
INSERT INTO faadata.DimAirport
SELECT * FROM public.airport_external;
```

5. Verify that the dimension table now contains data.

```
SELECT * FROM faadata.DimAirport LIMIT 10;
```

Note: The LIMIT clause reduces the size of the result set to the number specified, limiting the amount of memory required by the client. This can be helpful if your query returns a large number of rows but you wish to only view a subset of these rows.

6. Exit the database before proceeding.

```
\q
```

Summary

Both `CREATE EXTERNAL TABLE` and `COPY` operations can be run using the single row error isolation feature. This feature allows you to load good rows while filtering out incorrectly formatted rows. Any such rows can be logged into an error table for further examination. External tables offer additional flexibility over `COPY` because you can use regular `SQL` commands to select and move the data prior to inserting it into it's final destination tables. This is useful for ETL processing common in data warehousing applications.

It is good practice to automatically run `ANALYZE` after any data load so that the query planner has the most up-to-date statistics. If you had any errors during your data loads, it is also a good idea to run `VACUUM` to reclaim any wasted space.

Create a Fact Table and Load Data using gpfdist

1. If not already connected to the master server, open a putty session to the master server and connect as `gpadmin`. `sudo su - gpadmin`
2. Start a `gpfdist` process in the background.

```
nohup gpfdist -d /rawdata/FAAData -p 8081 > gpfdist.log 2>&1
</dev/null &
```

   This starts a parallel file distribution session on port 8081 for files in the directory `/rawdata/FAAData`. These files can now be read into the database from an external table.

3. Verify that `gpfdist` is running.

```
pgrep gpfdist
```

   This should print a process ID for the `gpfdist` process.

4. Examine the contents of the `/rawdata/FAAData/CreateExt-OnTimePerformance.sql` file and go to the end of it. You will not be making any changes to this file.

   This file will create an external table as shown below.

```
CREATE EXTERNAL TABLE public.FactOnTimePerformance_external
(...)
LOCATION (
'gpfdist://mdw:8081/On_Time_On_Time_Performance_2008_1.csv.bz2
',
'gpfdist://mdw:8081/On_Time_On_Time_Performance_2008_2.csv.bz2
'
)
FORMAT 'CSV' (HEADER DELIMITER ',')
LOG ERRORS INTO public.fact_err
SEGMENT REJECT LIMIT 10 ROWS;
```

5. Execute the script as shown.

```
psql -f /rawdata/FAAData/CreateExt-OnTimePerformance.sql -d faa
```

   The output highlights that the fact_err table did not previously exist. It will be created to capture any failures. The last portion of the output highlights that a header is expected within each of the files loaded through this external table.

6. Connect to the `faa` database.

```
psql faa
```

7. Confirm that the contents of the files can now be accessed through the external table created.

```
SELECT * FROM public.factontimeperformance_external LIMIT 10;
```

8. Create a regular heap table based on the external table and populate it with the data from the external table.

```
CREATE TABLE faadata.factontimeperformance
AS
SELECT * FROM public.factontimeperformance_external
DISTRIBUTED RANDOMLY;
```

9. Confirm that the `faadata.factontimeperformance` table has been created and populated.

```
SELECT * FROM faadata.factontimeperformance LIMIT 10;
```

10. Execute the `/rawdata/FAAData/DropTables.sql` script to drop the tables created. This script will drop the dimairline, dimairport, and factontimeperformance tables.

```
\i /rawdata/FAAData/DropTables.sql
```

11. The next two steps will create and populate the tables that will be used in future labs.

    Create the tables by executing the script `DDLScript.sql`.

```
\i /rawdata/FAAData/DDLScript.sql
```

12. Load data for the tables you just created in the faa schema by executing the script, `Load_FAA_Data.sql`. This creates external tables, using the `gpfdist` process you started earlier, and loads approximately 20M rows. It will take 5-10 minutes to complete.

```
\i /rawdata/FAAData/Load_FAA_Data.sql
```

13. Exit the database.

```
\q
```

Summary

`gpfdist`, the parallel file distribution program, can be used in conjunction with `gpload` or other ETL tools to take advantage of GPDB's parallel architecture. It starts a lightweight session on the port specified and can be used to load multiple files from the same directory.

You can start multiple gpfdist processes to maximize the data transfer rate.

Create a Writeable External Table to Load Data into a Different Database using `gpfdist`.

1. If not already connected to the master server, open a putty session to the master server and connect as `gpadmin`.
2. Create a directory called `data` in your home directory that you will use in this lab. This directory will be used by writable tables to create files with data pushed from the database.

```
mkdir data
```

3. Start a `gpfdist` session that will access files in the `~/data` directory you created. Start this `gpfdist` session on port 8082. You must have write permission on the directory you are using for this session.

```
nohup gpfdist -d /home/gpadmin/data -p 8082 > gpfdist_8082.log 2>&1
</dev/null &
```

   This starts a parallel file distribution session on port 8082 for files in the directory `/home/gpadmin/data`.

4. Verify that `gpfdist` is running for the newly created session.

```
ps -ef | grep gpfdist
```

   You may see a second `gpfdist` session running for port 8081 created earlier in the lab.

5. Connect to the `faa` database as the gpadmin user.

```
psql faa
```

6. Create a writable external called `xtable_ext0`. The table will be a copy of the structure of the `dimairport` table. The table should point to the `wtable` file through the `gpfdist` session running on port 8082.

```
CREATE WRITABLE EXTERNAL TABLE xtable_ext0
(LIKE DIMAIRPORT)
LOCATION ('gpfdist://mdw:8082/wtable')
FORMAT 'TEXT' (DELIMITER '|');
```

   The `gpfdist` process is pointing to `/home/gpadmin/data` on the master server, mdw. This writeable table is therefore pointing to the file, `/home/gpadmin/data/wtable`. Column values will be separated with the pipe (|) symbol.

7. Populate the `wtable` file with content from the `dimairport` table. This is known as unloading data. It does not, however, remove data from the `dimairport` table.

```
INSERT INTO xtable_ext0 SELECT * FROM dimairport;
```

8. Verify that the contents of the file correspond with the data contained in the `dimairport` table. Use the escape shell (!) to execute the `less` command for the `/home/gpadmin/data/wtable` file.

```
\! less /home/gpadmin/data/wtable
```

   Hit `q` to exit.

9. Create a new database called `dbbackup`.

```
CREATE database dbbackup;
```

10. Connect to the `dbbackup` database.

```
\c dbbackup
```

11. Create a new schema for the `dbbackup` database called `backupdata`.

```
CREATE SCHEMA backupdata;
```

12. Change the `search_path` parameter on the `dbbackup` database so that the `backupdata` schema is the default schema.

```
ALTER database dbbackup SET search_path TO backupdata, PUBLIC,
pg_catalog;
```

13. Reconnect to the `dbbackup` database. This step is necessary to reload the updated `search_path` parameter.

```
\c
```

14. Create the `dimairport` table in the `dbbackup` database using the following syntax.

```
CREATE TABLE dimairport
(
airportid CHARACTER(3),
airportdescription CHARACTER VARYING(85)
);
```

15. Populate the `dimairport` table with the content from the `/home/gpadmin/data/wtable` file.

```
COPY dimairport
FROM '/home/gpadmin/data/wtable' WITH DELIMITER '|';
```

Summary

Writable external tables are used to select rows from database tables and output the rows to files, named pipes, or to executable programs. You could unload data from a Greenplum Database table and send that data to a text file. Once the text file is created, you can use it to populate a table within another Greenplum Database. This technique, shown in the exercise you just completed, is an example of migrating data from one database to another. If necessary, you can perform transforms on the data as it is being inserted into the database.

Writable external tables only allow `INSERT` operations.

Loading a Compressed File using gpfdist

1. Connect to the `faa` database as `gpadmin`.

```
\c faa
```

2. Create an external table called `public.gztable` based on the DDL of the `factontimeperformance` table. Populate `public.gztable` with the contents from a compressed file without first decompressing it.

```
CREATE EXTERNAL TABLE public.gztable
(LIKE factontimeperformance)
LOCATION('gpfdist://mdw:8081//On_Time_On_Time_Performance_2008_1.c
sv.gz')
FORMAT 'CSV' (HEADER DELIMITER ',');
```

3. Verify the external table has been created listing external tables with the `\dx` command.

```
\dx
```

4. To make the output easier to see, complete the following to change the output mode in PSQL.

Issue the following command to eliminate the header from the output.

```
\t on
```

Issue the following command to change to unaligned output. The default separator for unaligned output is the pipe (|) symbol.

```
\a
```

Issue the following command to change the field separator to a comma.

```
\f
```

5. List two records from this external table as shown.

```
SELECT * FROM gztable limit 2;
```

6. Reset your output by issuing the following PSQL meta commands:

```
\t off
\a
```

7. Create a new table called `faadata.performance` and populate it using the data from the external table, `public.gztable`.

```
CREATE TABLE faadata.performance AS
SELECT * from public.gztable
DISTRIBUTED RANDOMLY;
```

8. Exit the database.

```
\q
```

Summary

Data in files compressed with gzip or bzip2, can be loaded into Greenplum Database tables without first uncompressing the files. This saves storage space on the host serving these files.

GPDB Administrator – Lab 14

Table Partitioning

Purpose

The purpose of this lab is to learn about Greenplum's partitioning capabilities. Partitioning is typically used on large fact tables to improve performance and manageability. Greenplum supports single level range and list partitioning. You will learn how to create and manage partitioned tables.

Create and Manage Table Partitions

1. Connect to the `faa` database as the `gpadmin` user, if not already connected.

```
psql faa
```

2. Create a partitioned version of the fact table. Each month will be stored in a separate partition. The partitioned table will be based on the **faadata.FactOnTimePerformance** table created in an earlier lab.

```
CREATE TABLE faadata.factontimeperformance_parted
(LIKE faadata.FactOnTimePerformance)
DISTRIBUTED BY (AirlineID)
PARTITION BY RANGE (Year)
(
START (2008)
END (2011)
EVERY (1)
);
```

3. List the tables.

```
\dt
```

You should note that the `faadata.factontimeperformance_parted` table was created along with 3 child tables.

4. Check the definition of one of the child tables.

```
        \d+ faadata.factontimeperformance_parted_1_prt_1
```

You should see that the child table has the same structure as the parent table, but also includes a check constraint for the year.

Log out of the DB.

```
        \q
```

5. Load data into the partitioned table to verify that the partition scheme is working.

```
bzcat /rawdata/FAAData/On_Time_On_Time_Performance_2008_1.csv.bz2 | psql faa -c
"COPY faadata.FactOnTimePerformance_parted FROM STDIN WITH DELIMITER ',' CSV
HEADER"
```

Note: You are loading data into the parent table, not the child table. If the partitioning is working correctly, Greenplum will place the data into the appropriate partition when data is inserted into the parent table.

6. Verify that the data was copied into the correct partition.

```
        psql faa -c "SELECT COUNT(*) FROM faadata.FactOnTimePerformance_parted"

        psql faa -c "SELECT COUNT(*) FROM
        faadata.FactOnTimePerformance_parted_1_prt_1"
```

Note: All of the records were placed into the first child partition table. If you query against any of the other child tables, you will find that they do not contain any rows.

7. Load the `January 2011` data into the same table.

```
    bzcat /rawdata/FAAData/On_Time_On_Time_Performance_2011_1.csv.bz2 | psql faa -c
    "COPY faadata.FactOnTimePerformance_parted FROM STDIN WITH DELIMITER ',' CSV
    HEADER"
```

This operation should fail since the partition for 2011 has not been created yet. There is also no default partition to capture any data that does not fall into the range of any defined partitions.

8. Create a new partition for the 2011 data.

```
    psql faa -c "ALTER TABLE faadata.factontimeperformance_parted ADD PARTITION
    Y2011 START (SMALLINT '2011') END (SMALLINT '2012')"
```

9. Load the `January 2011` data again.

```
bzcat /rawdata/FAAData/On_Time_On_Time_Performance_2011_1.csv.bz2 | psql faa -c
"COPY faadata.FactOnTimePerformance_parted FROM STDIN WITH DELIMITER ',' CSV
HEADER"
```

Note: The operation should now succeed.

10. Reconnect to the `faa` DB.

```
psql faa
```

The table and partition names can be long and non-intuitive. Rename the table to `factotperf`.

```
ALTER TABLE factontimeperformance_parted RENAME TO factotperf;
```

11. Verify that the parent table and the child tables have been renamed.

```
\dt
```

12. Rename the 2008 partition so that it has a more intuitive name.

```
ALTER TABLE factotperf RENAME PARTITION for (2008) to Y2008;
```

13. Verify that the partition has been renamed.

```
\dt
```

14. You may need to remove a partition after a period of time to age out data. Drop the 2008 partition with the following command:

```
ALTER TABLE factotperf DROP PARTITION FOR (2008);
```

15. Exchanging a partition allows you to bring data from a table into a partition. Create a table named `fact_temp` that has the same structure as factotperf. This table will contain the data that will eventually be placed into the partitioned table.

```
CREATE TABLE fact_temp (LIKE factotperf);
```

16. Remove any existing data from the `factotoperf` table by truncating the table. This will truncate all partitions within the table.

```
TRUNCATE TABLE factotperf;
```

17. Copy the `January 2011` data into `fact_temp`.

```
\! bzcat /rawdata/FAAData/On_Time_On_Time_Performance_2011_1.csv.bz2 | psql
faa -c "COPY faadata.fact_temp FROM STDIN CSV HEADER"
```

Note: Here, we avoid quitting the `psql` client to run this load by shelling out using the `\!` operator. The reason we can't simply use the `COPY` command here is that all of these large CSV files have been compressed using Bzip2, to save space, but the COPY command does not operate on compressed data. Also, the COPY command in this example does not specify the DELIMITER, since the comma is the default for CSV data.

18. Take the data found in the `factotperf` table and push it to the `Y2011` partition using the `EXCHANGE` clause on the partition. This will also move the data that was in the partition to the `fact_temp` table.

```
ALTER TABLE factotperf EXCHANGE PARTITION Y2011 WITH TABLE
fact_temp;
```

19. Verify that the data is now in the partitioned table and that the `fact_temp` table has no rows.

```
SELECT COUNT(*) FROM factotperf;
SELECT COUNT(*) FROM fact_temp;
```

20. Sometimes a partition may become too large and it might be worthwhile to split the partition. Create a new partitioned table.

```
CREATE TABLE ORDERS
(
ordered INT,
orderDate DATE
)
DISTRIBUTED BY (ordered)
PARTITION BY RANGE (orderDate)
(START ('2008-01-01')
END ('2008-12-31')
EVERY (INTERVAL '1 month')
);
```

21. Split the January partition into two partitions.

```
ALTER TABLE orders
SPLIT PARTITION FOR ('2008-01-01')
AT ('2008-01-16')
INTO (PARTITION jan20081to15, PARTITION jan200816to31);
```

22. Verify that the new partitions have been created.

```
\dt
```

Exit the database before proceeding.

```
\q
```

Summary

Table partitioning addresses the problem of supporting very large tables, such as fact tables, by allowing you to logically divide them into smaller and more manageable pieces. Partitioning is used to improve performance by scanning only the relevant data needed to satisfy a query. It can also facilitate database loading and maintenance. In the Greenplum Database, partitioning is a procedure that creates multiple sub-tables (or child tables) from a single large table (or parent table) and sets exclusion constraints on the child tables. Table data resides in the child tables only; no data actually resides in the parent tables. Note that partitioned tables in the Greenplum Database are also physically distributed across the segment instances just as are non-partitioned tables.

The ALTER TABLE syntax for handling partitions is very flexible. It allows almost as much control over the structure of a table as the CREATE TABLE syntax. With this one command you are given the ability to add partitions, drop partitions, and rename partitions. Remember that, if you drop a partition, this will remove the partition and any of its dependents. Exchanging a partition is an easy way to facilitate tricky operations that, with other databases, would require locking out the users or running late at night so as not to interfere with work. By running a load on a table that is not being used, and then quickly exchanging it once the load is finished, you benefit from greater data security and less downtime. The advantages of exchanging a partition do not end there.

The ALTER TABLE syntax can be used to split a partition if the partition is growing too large. To use this command, you just split your partition on the partitioning key. The value specified will land in the latter of the two partitions created.

GPDB Administrator – Lab 15

Administrative Tasks

Purpose

In this lab, you will start, stop, and restart the Greenplum Database. You will also execute commands that provide information on the state of Greenplum.

Perform System Administration Tasks on the Greenplum Environment

1. If not already connected to mdw, log in as `gpadmin` on your master server and stop the Greenplum database with the following command .

```
gpstop
```

   When prompted, answer `y` to proceed with shutting down the instances.

2. The `gpstart` command is used to start the Greenplum database defined by the $MASTER_DATA_DIRECTORY parameter or as specified with the -d option. To start the Greenplum database in restrictive mode where only the Greenplum superuser can connect, issue the following command in UNIX as `gpadmin`.

```
gpstart -R
```

   When prompted, answer `y` to proceed with starting the instances. You can skip any prompts by including the `-a` option as a part of the command.

   This mode is useful if you need to perform some isolated work in the environment and need to ensure other users cannot connect to the Greenplum Database.

3. Verify the state of the cluster to ensure that the database is running.

```
gpstate
```

4. Verify the student account that you created in earlier labs cannot connect to the database.

```
psql -U <Your name> faa
```

5. There are times when you want to change parameters of the database by editing either `pg_hba.conf` or `postgresql.conf`, but do not want to shut down the database to make it aware of the changes. The `gpstop -u` command lets the master re-read the configuration files without shutting down services.

   Using `vi`, edit the `pg_hba.conf` file and add the following line at the end.

```
host all all 127.0.0.1/32 trust
```

   The new entry allows all users from `127.0.0.1` to connect to all of the databases within the Greenplum Database cluster.

```
vi $MASTER_DATA_DIRECTORY/pg_hba.conf
```

6. Read the new configuration into the database by issuing the following command.

```
gpstop -u
```

   This will reload all the configuration files without shutting the database down.

7. Log in to `psql` and check for skew on the main fact table, `factontimeperformance`.

```
psql faa

SELECT gp_segment_id, count(*) FROM factontimeperformance GROUP
BY gp_segment_id;
```

8. Check for processing skew.

```
SELECT gp_segment_id, count(*) FROM factontimeperformance WHERE
cancelled='false' GROUP BY gp_segment_id;
```

9. Exit the database.

```
\q
```

10. There may be times when you need to perform administrative duties on the segment hosts from the master. You can use the gpssh command from UNIX as gpadmin. The `/rawdata/Binaries/hostfile_exkeys` file contains the names of all hosts in this cluster. You can use this file to query the hosts.

```
gpssh -f /rawdata/Binaries/hostfile_exkeys df
```

11. After a database has been running for a long time, its catalog can become bloated, especially if there are a large number of UPDATEs and DELETEs to the database environment. A bloated system requires regular maintenance consisting of scheduled VACUUMs. These systems may periodically require a full VACUUM ANALYZE to clean up space.

    It is safer to run a regular VACUUM. A full VACUUM FULL can slow down the performance of the system dramatically. It should be executed when in maintenance mode. To run a vacuum on only one catalog table, open a `psql` session and execute the following command while connected to the database.

```
psql faa -c 'vacuum analyze pg_catalog.pg_class;'
```

    The pg_class table keeps the object names in the database. If there are a lot of updates to the table, such as with creating and dropping objects, the table can become bloated and negatively impact the system.

    Note: You do not directly make changes to this table. This table is affected by updates in your environment, such as when you create or delete a table.

12. To run a `VACUUM ANALYZE` on all catalog tables create the following script and execute it.

    Using `vi`, create a new file called `gp_vacuum_analyze` with the following content.

```
#!/bin/bash
export DBNAME="faa"
export VCOMMAND="VACUUM ANALYZE "

psql -tc "select '$VCOMMAND' || ' pg_catalog.' || relname || ';' from
pg_class a,pg_namespace b where a.relnamespace=b.oid and b.nspname=
'pg_catalog' and a.relkind='r'" $DBNAME | psql -a $DBNAME
```

    Save the file and exit `vi`.

    Change the permissions on the file so that it is executable.

```
chmod 755 gp_vacuum_analyze
```

Execute the script by typing the following.

```
./gp_vacuum_analyze
```

Note: Recall that it is recommended that you execute VACUUM on a regular schedule and ANALYZE separately on its own schedule.

13. Open a PSQL session and log in as `gpadmin` to the `faa` database.

```
psql faa
```

14. Check the size of the database using the following command.

```
SELECT * from gp_toolkit.gp_size_of_database;
```

15. Retrieve the size of each relational table in schemas of the user database.

```
SELECT * from gp_toolkit.gp_size_of_table_disk;
```

16. Retrieve the size of all schemas in the user database.

```
SELECT * from gp_toolkit.gp_size_of_schema_disk;
```

17. Retrieve the size of all indexes in the user database.

```
SELECT * from gp_toolkit.gp_size_of_index;
```

18. Retrieve the cumulative size of all indexes in a table list this total size for each table in the user database.

```
SELECT * from gp_toolkit.gp_size_of_all_table_indexes;
```

19. Retrieve the uncompressed table size.

```
SELECT * from gp_toolkit.gp_size_of_table_uncompressed;
```

20. Retrieve the total free disk space in Kbytes for each segment server and the file systems included as part of the Greenplum Database.

```
SELECT * from gp_toolkit.gp_disk_free;
```

21. Exit the database.

```
\q
```

22. Restart the database in regular mode so that all users can connect to it. The `-r` option is used to restart the database.

```
gpstop -ar
```

Summary

Checking table skew after loading is one of the most important ways to validate the efficiency of your distribution keys. Check the skew often on large tables that are frequently loaded, as data demographics may have changed. This is important on all Greenplum systems regardless of size. On smaller systems, there are fewer segments involved so the distribution is not over as many units of parallelism. On larger systems the hashing algorithm, particularly on non-unique distribution keys, may hash many more rows to a single segment.

Check the size of databases and regularly VACUUM the system to reduce the chances of bloat.

GPDB Administrator – Lab 16

Backups and Restores

Purpose

In this lab, you will schedule a parallel backup operation with the `gpcrondump` command to dump the faa database. This will create a backup file of the master instance and each active segment instance in the Greenplum Database system.

By default, the dump files are created in the data directory of their respective segment instance or master instance. In this exercise, you will redirect the dump files to one location to make it easier to collect and analyze them.

You will then restore the database using the `gprestoredb` command.

Backups

To create backups you will need a place to keep them. Sometimes it is a network share to a backup device. Normally all segments will need to have this space. You can perform backups as parallel or non-parallel backups using the following.

1. Start an ad-hoc backup from the UNIX prompt as `gpadmin`:

```
gpcrondump faa
```

2. You will be prompted to continue. Type `y` and press Enter to proceed.
3. The backup files will be in the `data/db_dumps` directory on the master and all primary segments.

```
ls -R $MASTER_DATA_DIRECTORY/db_dumps
```

All backup files are stored in a timestamped directory for the day that the backup was started. In this example, the directory, 20150330, is created when the first backup is executed on March 30th, 2015. Any subsequent backups performed on that day are stored in the same directory.

4. Verify the corresponding backup files exist for the segments.

```
gpssh -h sdw1 -h sdw2 \ ls -R /data/primary/gpseg*/db_dumps
```

   Over the next few steps, you will configure the environment so that `gpcrondump` is automated through cron.

   The crontab utility determines whether or not a user has appropriate permission to run a program at a particular point in time by checking the file `/etc/cron.allow`. A user must be explicitly included to this file to be able to use the crontab.

5. As root, add the `gpadmin` account to the `/etc/cron.allow` file in a line by itself. In this example, the cat command is used to add a line to the end of the file. If the file does not exist, it will be created. If it does exist, the double greater than symbols (>>) lets you append to the file. To finish modifying the file with the cat command, hit CTRL-D.
6. Exit from the root account, back to the `gpadmin` account.

```
exit
```

7. Ensure that the permissions of `.bash_profile` file for the gpadmin user includes the execute permission so that cron can properly access and execute the login script:

```
chmod +x /home/gpadmin/.bash_profile
```

8. Using vi, add the following lines to the file `/home/gpadmin/cronbackup.sh`:

```
source /home/gpadmin/.bash_profile
gpcrondump -x faa -c -g -G -a -q >> /tmp/gpcrondump.log
```

9. Change the permissions on `cronbackup.sh` to `755` so that it is readable and executable by cron.

```
chmod 755 cronbackup.sh
```

10. Update the `EDITOR` environment variable to `vi` and export it. When modifying your cron jobs, the editor defined by the EDITOR variable will be used. If you are more familiar with and prefer to use Emacs, you can replace vi shown here with emacs:

```
export EDITOR=vi
```

11. You will set cron to execute the script you created, cronbackup.sh, five (5) minutes from the time you record in this step. Use the date command to obtain the current time:

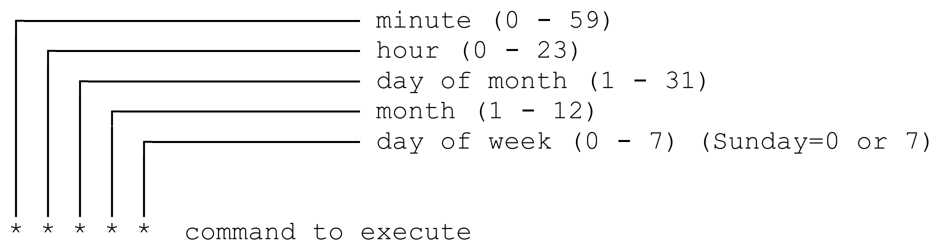
```
date
```

12. Edit the crontab and add a line to execute the script you created, `cronbackup.sh`, five minutes from the time you recorded earlier.

```
crontab -e
```

The syntax below is used to show how each crontab line is defined:

```
17 6 * * * /home/gpadmin/cronbackup.sh
```


minute (0 - 59)
hour (0 - 23)
day of month (1 - 31)
month (1 - 12)
day of week (0 - 7) (Sunday=0 or 7)
* * * * * command to execute

13. Save your changes and exit the editor. Once you have modified your crontab, you should receive a message that a new crontab is being installed.
14. Wait for the time to pass for the job to execute and look for the backup files. Execute the following commands where you should see a second set of backups with a new timestamp.

```
ls -a $MASTER_DATA_DIRECTORY/db_dumps/`
gpssh -h sdw1 -h sdw2 \
ls -aR /data/primary/gpseg*/db_dumps
```

   - If the backup did not execute because time passed before you saved the crontab, you can execute the `/home/gpadmin/cronbackup.sh` script manually with the command, `/bin/bash /home/gpadmin/cronbackup.sh`.

- `Do not move to the next step until the backup process has completed`

Restores

Over the next few steps, you will recover the database from the backup you created.

1. Connect to the gpadmin database as the gpadmin user, if not already connected:

```
psql gpadmin
```

2. Rename the `faa` database:

```
gpadmin=# ALTER DATABASE faa rename to faa1;
```

3. Recreate the `faa` database. You will use this database to recover the data stored in the backup you created.

```
gpadmin=# CREATE DATABASE faa;
```

4. Connect to the `faa` database you created and list the tables in the

```
gpadmin=# \c faa
faa=# \dt
```

5. Exit your PSQL session.
6. Restore the `faa` database from the last backup taken. By not including the key identifier for the backupset, `gpdbrestore` will use the last available backup created to perform the restore operation. Execute the `gpdbrestore` command to restore the faa database:

```
gpdbrestore -s faa
```

7. When prompted, type `y` and press Enter (the procedure may take a few minutes to complete).

- `Note: gpdbrestore -s <DATABASE_NAME>` option looks for the latest set of dump files for the given database name in the segment data's `db_dumps` directory on the Greenplum Database array of hosts.

1. Verify the database has been restored by listing tables from the `faa` database.

```
psql faa
faa=# ALTER DATABASE faa SET search_path TO faadata, public,pg_catalog;
faa=# \c faa
faa=# \dt
```

2. Exit the database before proceeding.

```
\q
```

Incremental Backup and Restore

1. Execute the command below to create two tables in the `dbbackup` database. The first will be a regular table called dimairline, while the second is an append-only table called dimairline_image.

```
psql -f /rawdata/FAAData/CreateDbBackupTables.sql dbbackup
```

   The error is displayed only if the table did not previously exist. The table will be created thereafter.

2. Start a psql session by connecting to the `dbbackup` database.

```
psql dbbackup
```

3. List the available tables on this database by using the command \dt as shown:

```
dbbackup=# \dt dimairline
```

4. Verify how many records the tables have by running the commands below.

```
dbbackup=# select count(*) from dimairline;
dbbackup=# select count(*) from dimairline_image;
```

   - `Note`: Both tables contain 1,540 records.

5. Exit the database before proceeding.

```
\q
```

6. Run a full backup using the following command:

```
gpcrondump -x dbbackup
```

7. Respond with `y` when prompted to continue.
8. Connect to the dbbackup database as gpadmin and populate the append-only `dimairline_image` table, as shown:

```
psql dbbackup
dbbackup=# COPY backupdata.DimAirline_image FROM
'/rawdata/FAAData/DimAIRLINES.csv'
WITH DELIMITER ',' CSV HEADER QUOTE '"';
```

9. Verify the number of records both the `dimairline` and `dimairline_image` tables have by executing the following commands:

```
dbbackup=# select count(*) from dimairline;
dbbackup=# select count(*) from dimairline_image;
```

- `Note:` The table dimairline_image now contains 3,080 records.

10. Exit the database and start an incremental backup for the dbbackup database.

```
dbbackup=# \q
gpcrondump -x dbbackup --incremental
```

11. Respond with `y` when prompted to continue.

- `Note:` The incremental backup provides a `Dump key` as shown that should be used to restore the incremental backup. Record the dump key here:

12. Connect to the `dbbackup` database and truncate the `dimairline` and `dimairline_image` tables.

```
psql dbbackup
dbbackup=# truncate table dimairline;
dbbackup=# truncate table dimairline_image;
```

13. Verify the number of records in the tables:

```
dbbackup=# select count(*) from dimairline;
dbbackup=# select count(*) from dimairline_image;
```

14. Exit the database and execute the `gpdbrestore` command to recover the tables.

```
dbbackup=# \q
gpdbrestore -t 20150330150138
```

    - Replace the dump key shown here with the dump key you recorded earlier.

15. Respond with `y` when prompted to continue.
16. Access the dbbackup database and verify the number of records the tables contain by executing the following commands:

```
psql dbbackup
dbbackup=# select count(*) from dimairline;
dbbackup=# select count(*) from dimairline_image;
```

17. Exit the database before proceeding.

```
\q
```

Summary

Backups are typically automated with `gpcrondump`, which is a wrapper for `gp_dump` and `pg_dumpall`.

The `gpcrondump` utility dumps the contents of a Greenplum Database into SQL utility files, which can then be used to restore the database schema and user data at a later time using `gpdbrestore`.

Keep in mind that a database in the Greenplum Database is actually comprised of several PostgreSQL instances (the master and all active segments), each of which must be dumped individually. The `gpcrondump` utility takes care of dumping all of the individual instances across the system.

Note that the 14 digit timestamp is the number that uniquely identifies the backup job, and is part of the filename for each dump file created by a `gp_dump` operation. This timestamp must be passed to the `gpdbrestore` utility when restoring a Greenplum Database.

Incremental backups let you backup append-only tables if a change has been made to the table or its contents. The `--incremental` option on the `gpcrondump` command lets you take advantage of the space-saving features that come with performing incremental backups on your tables. Restoring from an incremental backup requires that you have all backups from the last full backup.

Recover from a Failed Master (OPTIONAL)

Your standby server has been installed and configured during the installation of the Greenplum software.

You will perform the following steps:

- Failover to the standby server
- Verify the Greenplum state operating with standby server
- Failback to the master server

1. Before proceeding, verify that the standby server is properly configured. If not already connected, login as root and switch user to `gpadmin` on your master server.
2. Verify the file `/etc/hosts` on ` the master and standby servers contain the same content.

```
    cat /etc/hosts
    gpssh -h smdw -e 'cat /etc/hosts'
```

3. Verify the contents of the `.bash_profile` file are the same as the `.bash_profile` file on the master server for the `gpadmin` user.

```
gpssh -h mdw -h smdw -e 'cat ~/.bash_profile'
```

   - As Command Center has not been configured to run on the standby server, you do not have to make changes to the /home/gpadmin/.bash_profile file to include it. All other changes reflecting the Greenplum Database must be the same.

4. Verify that you can ssh to both segment servers from the standby server. Open a new `PuTTY` connection to the standby server, `smdw`. Connect to the standby server first. Login as root and switch to the `gpadmin` user account.
5. From the PuTTY session where you have connected to the standby server, connect to the first segment server, `sdw1`.

```
ssh sdw1
```

6. Exit from the first segment server and connect to the second segment server, sdw2.

```
exit
ssh sdw2
```

7. Exit from the second segment server, sdw2.

```
exit
```

8. Before initiating a failover, verify the state of the master to standby server to ensure that the database is synchronized. To verify the state, execute the following command on the master server, `mdw`.

```
gpstate -f
```

9. To safeguard against incidents that may occur in your lab environment, create a backup of all databases in the environment. You will create a backup of all databases except `template0`, `template1`, and `postgres`.
10. First, obtain the list of databases in the environment. The highlighted databases will be backed up.

```
psql -l
```

11. Create a backup of all databases highlighted in the previous step using the `gpcrondump` command. You can specify multiple databases using the `-x` option with a comma separated list of databases. You will also copy the `postgresql.conf` and `pg_hba.conf` file as part of the backups with the `-g` option. The `-a` option will execute the command in non-interactive mode. Backup files will be saved to the `/home/gpadmin/db_backup` directory on the master and segment servers with the `-u` option. You will need the configuration files after you complete the failover process.

```
gpcrondump -x dbbackup -x dbstudent -x faa -x faa1 -x gpadmin -x
gpperfmon -x names -u ~/db_backup -g -a
```

12. From the master server, switch to the root user and issue the reboot command to reboot the master server. The database will not start automatically as there are no startup scripts in place for the database start up.

```
su -
reboot
```

The purpose of this step is to simulate unavailability of the master server. You can then force the standby server to become the new primary master server. Do not proceed until this step has been completed.

13. Confirm that your master server is down, by pinging it as shown from the standby master server.

```
ping mdw
```

14. From your standby server, `smdw`, promote the standby master to be the primary master. You will need to specify the port to use for the database activation. You will continue to use port 5432.

```
export PGPORT=5432
gpactivatestandby -d /data/master/gpseg-1
```

15. Respond with `y` when asked to continue.

It may take a few minutes for the process to complete.

Note that the `postgresql.conf` and `pg_hba.conf` files are not synchronized as part of the master replication process. Therefore, custom settings preserved on the master are not available here. This therefore required that you set the `PGPORT` environment variable before promoting the standby server to master.

16. Verify the state of your database by running the `gpstate` command with the `-s` option to obtain detailed information. Search the output for strings that contain the word, `master`.

```
gpstate -s | grep -i master
```

Note that there is no standby running at this point.

17. Open a PSQL session to the `faa1` database to verify the database has been recovered.

```
psql faa1
```

18. Display the user tables for the `faa1` database.

```
\dt
```

19. Exit the database.

```
\q
```

20. After activating the standby server as the master server, update the database query statistics on all databases. For each database, execute the `ANALYZE` command to update statistics. Use the following script to perform this step.

```
for db in \ `psql -tc "select datname from
pg_database where datistemplate='f';"`; do
echo -n "$db: "
psql $db -c 'ANALYZE';
done
```

21. After activating a standby master in a recovery scenario and making it your current primary master, you can continue running that instance as your primary master. This assumes that the capabilities and dependability of that host machine are equivalent to the original master host.

- Before restoring the master and standby instances on original hosts, ensure that the conditions that caused the original failure have been fully fixed.

22. Verify the original master server, `mdw`, is back online. Ping `mdw` from `smdw`.

```
ping mdw
```

23. Reconnect your original `PuTTY` session to the original master server, `mdw`. Login as `root` and switch to the `gpadmin` user account.
24. On your original master server, `mdw`, rename the directory `/data/master/gpseg-1` to `/data/master/gpseg-1_orig`. The utility `gpinitstandby` will recreate the directory and requires that it does not exist.

```
mv $MASTER_DATA_DIRECTORY ${MASTER_DATA_DIRECTORY}_orig
```

25. From the standby server, `smdw`, execute the `gpinitstandby` command to promote the original master server, `mdw`, to be the new standby server.

```
gpinitstandby -s mdw
```

26. You will be prompted to continue. Type `y` and press Enter to proceed.
27. Use the `gpstate` command to check the status of the standby master. The output of the `gpstate` command shows that the original standby server, `smdw`, is now the master server. It also shows that the original master server, `mdw`, is now the standby server.

```
gpstate -s | grep -i master
```

You can also obtain the state of the standby and mirrors with the command, `gpstate -f`.

28. Now that the failover has succeeded, reverse the roles of the master and standby servers so that `mdw` and `smdw` are back to their original roles. To perform this task, complete the following steps.

    On the current master server, `smdw`, stop the Greenplum database master instance only using the `-m` option.

```
gpstop -am
```

29. At this point, the database should no longer be running. From the current standby server, `mdw`, promote `mdw` to be the active master server. Use the `gpactivatestandby` utility to perform this task.

```
export PGPORT=5432
gpactivatestandby -d $MASTER_DATA_DIRECTORY -f
```

30. You will be prompted to continue. Type `y` and press Enter to proceed.
31. On `mdw`, execute the `gpstate` utility to determine the state of the active master server. This screen shows that there is no standby server configured and that `mdw` is back to its original role as the active master server.

```
gpstate -s | grep -i master
```

32. On `smdw`, start the process for changing this server back to its original role as the standby server.

    Rename the directory `/data/master/gpseg-1` to `/data/master/gpseg-1_orig`.

```
mv $MASTER_DATA_DIRECTORY ${MASTER_DATA_DIRECTORY}_orig
```

    This step is necessary as the procedure to initialize a standby server into the Greenplum cluster will create the master data directory.

33. From `mdw`, execute the `gpinitstandby` utility to promote the `smdw` server to the standby role.

```
gpinitstandby -s smdw
```

34. You will be prompted to continue. Type `y` and press Enter to proceed.
35. Run the `gpstate` utility to re-verify the state of the master and standby servers. The screen below shows that the original roles for `mdw` and `smdw` (standby) have been restored.

```
gpstate -s | grep -i master
```

36. Recover the `postgresql.conf` and `pg_hba.conf` file from your backups and push them to the `$MASTER_DATA_DIRECTORY` locations. This will overwrite the existing copies but will give you back your customized versions.

Search the backup directory you created in this task for files with the name `gp_master_config_files_*.tar`.

```
find ~/db_backup -name 'gp_master_config_files_*.tar'
```

37. Extract the contents of the tarred file from the previous step.

```
tar xvf
/home/gpadmin/db_backup/db_dumps/20150331/gp_master_config_files_20150
331103817.tar
```

38. Copy the `pg_hba.conf` file and the `postgresql.conf` file to `$MASTER_DATA_DIRECTORY`.

```
cp data/master/gpseg-1/pg_hba.conf $MASTER_DATA_DIRECTORY
cp data/master/gpseg-1/postgresql.conf $MASTER_DATA_DIRECTORY
```

39. Re-read the configuration files with the `gpstop` command.
40. As a test, verify your non-superuser account, in this case student, can list the tables in the `faa` database.

```
psql faa -c '\dt' -U <Your name>
```

Summary

If the master server fails, the standby server can be used to bring the database back online and accessible to users. If a virtual IP address has been defined for the master and standby server, the virtual IP address can be used by the standby server so that users do not have to use a different IP address or hostname to access the database.

Greenplum Database uses log replication to synchronize data between the master server and the backup server. Committed transactions are synchronized from the master server to the standby server. Should the master become unavailable, the standby can be promoted to act as the master until the master becomes available again.

The replication process is maintained by a WAL process running on the master server and the standby server. You can verify the state of synchronization by using the `gpstate -f` command or selecting against the `pg_stat_replication` view. This view contains the process id (procpid field), the state (state field), and the synchronization state (sync_state field) along with other information on the WAL process.

Note that while transactions are synchronized, the `postgresql.conf` and `pg_hba.conf` files are not. Maintain a backup copy of these files and be prepared to incorporate the changes to those files on the standby server should you need to perform a failover.

Additionally, when promoting the original master server to its original role, retrieve the backup copies of those files and push them to the recovered data directory.

GPDB Administrator – Lab 17

Data Modeling in GPDB

Purpose

In this lab, you create the database objects that will be used for follow-on labs for demonstrating data modeling and design decisions.

Create the datamart Database and Objects

1. Create the `datamart` database using either of the methods discussed.

   From the Linux command prompt in `mdw` as `gpadmin`.

   ```
   createdb datamart
   ```

2. Access the `datamart` database as `gpadmin`.

   ```
   psql datamart
   ```

3. Create the facts and dimensions schema for the database.

   ```
   CREATE SCHEMA dimensions;
   CREATE SCHEMA facts;
   ```

4. Set the default search path for the `datamart` database to include the `facts` and `dimensions` schemas. You may wish to include `public` and `pg_catalog` as well.

   ```
   ALTER DATABASE datamart SET search_path = dimensions, facts, public,
   pg_catalog;
   ```

GPDB Administrator – Lab 18

Physical Design Decisions

Purpose

In this lab, you create table objects based on business requirements provided in the lab. The choice of data types, constraints, distribution keys, and partitioning is up to you.

The logical data model to support the business model is provided below.

Create the Store Dimension

1. The business requirements for the Store dimension are as follows.

   Each store is given a numeric identifier. This identifier is created in the source system (OLTP) and passed to the data warehouse. The Store id is unique for each store and may not be null.

   All address attributes are required.

   The phone number is stored as (XXX)XXX-XXXX formatted character string.

   The store name must allow for upper and lower case data and may contain single or double quotes in the name. The store name may not be null.

   The `Has` columns are passed as either 0 (false) or 1 (true) to the data warehouse. The default value for these columns in the `OLTP` system is 0.

   The country code is three characters (e.g., USA, CAN). Currently the business only operates in the United States and Canada.

   There are currently around thirty five stores in the US and Canada.

   Based on these requirements, implement the Store dimension.

2. Determine the table name for the Store entity.
3. Determine the column name, data type, and any constraints for each attribute of the Store entity.

Attribute Name	Column Name	Constraint	Data Type
Store ID			
Store Name			
Address			
City			
State			
Zip Code			
Zip Code Plus Four			
Phone			
Country Code			
Has Pharmacy			
Has Grocery			
Has Deli			
Has Butcher			
Has Bakery			

4. What will be the DISTRIBUTION KEY? Does it have sufficiently high cardinality?
5. Create an SQL file with the `CREATE TABLE` statement.

   Be sure to include any column and table constraints.

   There is an example (non-optimized) file called `create_store.sql` in the `/home/gp/sql/load_files/adv_sql_files` that you can use as a guide.

   ```
   cd /home/gp/sql/load_files/adv_sql_files
   vi create_store.sql
   ```

   You will be accessing the directory `/home/gp/sql/load_files` directory throughout this lab. To save some time, create an environment variable that points to that directory.

   ```
   export LF=/home/gp/sql/load_files
   ```

6. Execute the script as `gpadmin` by starting a `psql` session and passing in the file name with the `-f` option. Include the `datamart` database as part of the `psql` command.

   ```
   psql datamart -f create_store.sql
   ```

   You should see the words CREATE TABLE immediately under the Linux prompt if there were no errors.

7. Verify the table has been created.

   ```
   psql datamart -c "\dt dimensions.store"
   ```

One row with the table name should be displayed.

Create the Country Dimension

1. The business requirements for the Country dimension are as follows:

   The country code is three characters, such as USA or CAN. Currently, the business only operates in the United States and Canada. This column cannot be NULL.

   The country name is a text column.

   Based on these requirements, implement the Country dimension.

2. Determine the table name for the Country entity from the LDM.
3. Determine the column name, data type and any constraints for each attribute of the Store entity.

Attribute Name	Column Name	Data Type	Constraint
Country Code			
Country Name			

4. What will be the DISTRIBUTION KEY? Is the cardinality of this column high?
5. Create an SQL file with the `CREATE TABLE` statement.

   Be sure to include any column and table constraints.

   There is an example (non-optimized) file called `create_country.sql` in the `/home/gp/sql/load_files/adv_sql_files` that you can use as a guide.

```
vi create_country.sql
```

6. Execute the script as `gpadmin` by starting a `psql` session and passing in the file name with the `-f` option. Include the `datamart` database as part of the `psql` command.

```
psql datamart -f create_country.sql
```

   You should see the words `CREATE TABLE` immediately under the UNIX prompt if there were no errors.

7. Verify the table has been created.

```
psql datamart -c "\dt dimensions.country"
```

One row with the table name should be displayed.

Create the Customer Dimension

1. The business requirements for the Customer dimension are as follows:
   - Each customer is tracked by their phone number as part of their membership rewards program.
   - There is a default customer with a (999)999-9999 customer id that is used when a transaction occurs for a customer that is not a member of the rewards program.
   - The customer name may not be `NULL`.
   - All address attributes are required.
   - The phone number is stored as (XXX)XXX-XXXX formatted character string.
   - The country code is three characters (e.g., USA, CAN). Currently the business only operates in the United States and Canada.
   - There are approximately 10,000 customers signed up for the rewards program.
   - The customer id will have to be generated during the ETL process as it is not a part of the `OLTP` system.

   Based on these requirements, implement the Customer dimension.

2. Determine the table name for the Customer entity from the LDM.
3. Determine the column name, data type and any constraints for each attribute of the Store entity.

Attribute Name	Column Name	Data Type	Constraint
Customer ID			
Customer Name			
Address			
City			
State			
Zip Code			
Zip Code Plus Four			
Phone			
Country Code			

4. What will be the DISTRIBUTION KEY? Does it provide an even distribution?
5. Create an SQL file with the `CREATE TABLE` statement.

   Be sure to include any column and table constraints.

There is an example (non-optimized) file called `create_customer.sql` in the `/home/gp/sql/load_files/adv_sql_files` that you can use as a guide.

```
vi create_customer.sql
```

6. Execute the script as `gpadmin` by starting a `psql` session and passing in the file name with the `-f` option. Include the `datamart` database as part of the `psql` command:

```
psql datamart -f create_customer.sql
```

   You should see the words `CREATE TABLE` immediately under the UNIX prompt if there were no errors.

7. Verify the structure of the table just created.

```
psql datamart -c "\d dimensions.customer"
```

Create the Transaction Fact Table

1. The business requirements for the Transaction fact table are as follows:
   - The transaction represents all transactions involving a customer, at a checkout stand, at a given store, by day. In data warehousing this is called the "shopping basket" type of transaction. It is, in effect, a summary of the all items and includes tax, payment methods, etc.
   - Each transaction is given a numeric identifier. This identifier is created in the source system (`OLTP`) and passed to the data warehouse. The transaction id is unique for each customer transaction and may not be null.
   - If the customer is not a rewards member, the default phone number is (999)999-9999. Otherwise, rewards customers are tracked by their phone numbers.
   - The terminal id may not be null. All terminals are numbered from 1 - `N` in each store. There are no more than 24 terminals in any given store at implementation time.
   - Item count may not be zero or negative.
   - Sales Amount may not be negative.
   - Taxes are represented in the `other amount` attribute.
   - Cash Amount + Debit Amount + Credit Card Amount may not exceed Sales Amount plus Other Amount.
   - There is an average of 100,000 transactions per day across the entire business.
   - Ninety percent of all transactions involve a rewards member customer.
   - The transaction date is a timestamp containing the date with the time. The business only wishes to store the date portion of the transaction date.
   - The historical data that will be loaded only goes back to the beginning of 2008.
2. Determine the table name for Transaction entity.

3. Determine the column name, data type, and any constraints for each attribute of the Transaction entity.

Attribute Name	Column Name	Data Type	Constraint
Transaction ID			
Terminal ID			
Transaction Date			
Store ID			
Customer ID			
Item Count			
Sales Amount			
Discount Amount			
Coupon Amount			
Cash Amount			
Credit Card Amount			
Debit Amount			
Other Amount			

4. What will be the `DISTRIBUTION KEY`? Will it have sufficiently high cardinality that it will result in an even data distribution?

   Create an SQL file with the `CREATE TABLE` statement.

   - Be sure to include any column and table constraints.
   - There is an example (non-optimized) file called `create_ transaction.sql` in the `/home/gp/sql/load_files/adv_sql_files` that you can use as a guide.

```
vi create_transaction.sql
```

5. Execute the script as gpadmin by starting a `psql` session and passing in the file name with the `-f` option. Include the datamart database as part of the psql command.

```
psql datamart -f create_transaction.sql
```

6. Verify the structure of the newly created table.

```
psql datamart -c "\d+ facts.transaction"
```

Load Dimension and Fact Data

1. Connect to the `datamart` database as `gpadmin`.

```
gpadmin@mdw adv_sql_files]$ psql datamart
```

2. Load the `dimensions.country` table using the `COPY` command. Copy the data from `/home/gp/sql/load_files/CountryData.csv` into the `dimensions.country` table.

```
COPY dimensions.country
FROM '/home/gp/sql/load_files/CountryData.csv'
WITH CSV HEADER
LOG ERRORS INTO public.country_err KEEP
SEGMENT REJECT LIMIT 10 ROWS;
```

3. Check the error table, `public.country_err`, for any errors.

```
SELECT * FROM public.country_err;
```

   Were any rows discarded? How many? Why?

4. Load the `dimensions.store` table using the `COPY` command. Copy the data from `/home/gp/sql/load_files/StoreData.csv` into the `dimensions.store` table.

```
COPY dimensions.store
FROM '/home/gp/sql/load_files/StoreData.csv'
WITH CSV HEADER
LOG ERRORS INTO public.store_err KEEP
SEGMENT REJECT LIMIT 10 ROWS;
```

5. Check the error table, `public.store_err` for any errors.

```
SELECT * FROM public.store_err;
```

   Were any rows discarded? How many? Why?

6. Load the `dimensions.customer` table using an external table. Create the external table using the following syntax:

```
CREATE EXTERNAL TABLE public.customer_external
(
custName VARCHAR(50),
address VARCHAR(50),
city VARCHAR(40),
state CHAR(2),
zipcode CHAR(8),
zipPlusFour CHAR(4),
countrycd CHAR(3),
phone CHAR(13)
)
LOCATION ('file://sdw1/home/gp/sql/load_files/CustomerData.csv')
FORMAT 'CSV' (HEADER DELIMITER ',')
LOG ERRORS INTO public.customer_err
SEGMENT REJECT LIMIT 10 ROWS;
```

7. Validate that the table was created correctly.

```
SELECT * FROM public.customer_external;
```

   You can type `q` to get out of that scrolling list.

8. The customers for this business are tracked by their phone numbers. A customer ID must be created for each of the customers. You will use a sequence number to create a unique customer ID for each customer. Create a sequence to support the data load of the customer dimension.

```
CREATE SEQUENCE public.CustomerSequence
INCREMENT BY 1
START WITH 1
NO CYCLE;
```

9. Insert the rows from the external table into the target table. Note that, because you are performing a transformation, adding the CustomerID value from the sequence, you need to specify all the columns in your SELECT statement. Note that the sequence name is all lower case.

```
INSERT INTO dimensions.customer
(CustomerID,
custName,
address,
city,
state,
zipcode,
zipPlusFour,
countrycd,
phone)
SELECT nextval('customersequence'),
custName,
address,
city,
state,
zipcode,
zipPlusFour,
countrycd,
phone
FROM public.customer_external;
```

   Were any rows discarded? How many? Why?

10. Exit the `psql` session.

```
\q
```

11. Load the Transaction data using gpfdist and an external table. You will perform a single transformation during the `INSERT/SELECT` and that is a join to the Customer dimension using the phone column to get the `customerid`.

   First, kill any running `gpfdist` processes.

```
killall gpfdist
```

   Now, start a `gpfdist` process in the background.

```
nohup gpfdist -d /home/gp/sql/load_files -p 8081 > /tmp/gpfdist.log
2>&1 </dev/null &
```

   Verify that `gpfdist` is running.

```
ps -ef | grep gpfdist
```

Note: If you receive an internal error that `gpfdist` cannot create a socket, verify that another `gpfdist` process is not running on the port number you specified. This should not happen since you just killed any running `gpfdist` processes.

12. Connect to the `datamart` database as `gpadmin`.

```
psql datamart
```

13. Create a `gpfdist` based external table to load the Transaction fact table. Use the following DDL.

```
CREATE EXTERNAL TABLE public.transaction_external
(
transId BIGINT,
terminalId INTEGER,
transDate DATE,
storeId SMALLINT,
phone CHAR(14),
itemCnt INTEGER,
salesAmt DECIMAL(9,2),
taxAmt DECIMAL(9,2),
discountAmt DECIMAL(9,2),
couponAmt DECIMAL(9,2),
cashAmt DECIMAL(9,2),
checkAmt DECIMAL(9,2),
ccAmt DECIMAL(9,2),
debitAmt DECIMAL(9,2),
otherAmt DECIMAL(9,2)
)
LOCATION ('gpfdist://mdw:8081/TransactionData001.csv',
'gpfdist://mdw:8081/TransactionData002.csv')
FORMAT 'csv' (HEADER DELIMITER '|')
LOG ERRORS INTO public.transaction_err
SEGMENT REJECT LIMIT 10 ROWS;
```

14. Validate that the table was created correctly.

```
SELECT * FROM public.transaction_external;
```

15. Insert the rows from the external table into the target transaction fact table in `psql`. Join the external transaction table to the customer dimension using the phone number column to get the customer id.

```
INSERT INTO facts.transaction
(transid,
terminalid,
transdate,
storeid,
customerid,
itemcnt,
salesamt,
taxamt,
discountamt,
couponamt,
cashamt,
checkamt,
ccamt,
debitamt,
otheramt)
SELECT t.transid, t.terminalid, t.transdate, t.storeid,
c.customerid, t.itemcnt, t.salesamt, t.taxamt,
t.discountamt, t.couponamt, t.cashamt, t.checkamt, t.ccamt,
t.debitamt, t.otheramt
FROM public.transaction_external t
INNER JOIN dimensions.customer c
ON TRIM(c.phone) = TRIM(t.phone);
```

16. Log out of the DB.

```
\q
```

cd to the `gpadmin` home directory.

17. Were any rows discarded? How many? Why?
18. Why did you need the TRIM function? What are the implications for loading large numbers of rows where we need to apply a function to the join conditions?

GPDB Developer – Lab 19

Joins in GPDB

Purpose

Experiment with the various features of joins in GPDB

Query plan for a join between two tables

1. Connect to the `datamart` database as `gpadmin`.
2. Verify your search_path includes the `dimensions` and `facts` schemas.

```
SHOW search_path;
```

3. So that you will be able to set *GUCs* to influence the behavior of the query planner, we will first ensure PQO is `disabled`.

```
SET optimizer = off;
```

4. Run a query to find the top 20 most profitable zip codes based on total sales.

```
SELECT c.zipcode, SUM(t.salesamt)
FROM customer c, transaction t
WHERE c.customerid = t.customerid
GROUP BY 1 ORDER BY 2 DESC LIMIT 20;
```

   You should have found that the top zip code was *99336* with a total of *30355.83*.

5. To see how the query will be executed, we run an EXPLAIN.

```
EXPLAIN SELECT c.zipcode, SUM(t.salesamt) FROM customer c, transaction t
WHERE c.customerid = t.customerid GROUP BY 1 ORDER BY 2 DESC LIMIT 20;
```

   On the second line from the bottom of the plan, you should see a line indicating the planner chose to broadcast the `customer` table

```
->  Broadcast Motion 2:2  (slice1; segments: 2)  (cost=0.00..6.96
rows=124 width=13)
```

6. Was that reasonable, given the size of the table? How large is this table?

```
SELECT COUNT(*) FROM customer;
```

7. Since we are using the Legacy Planner, we are able to influence its behavior through the use of *GUCs*, or configuration parameters. There is one which is consulted when considering whether to broadcast or redistribute a table. See what its value is.

```
SHOW gp_segments_for_planner;
```

By default, it should have a value of 0, which tells the planner to use the actual number of primary segments when considering which type of motion to use. If you increase this significantly, it will appear that a broadcast of the table is too expensive, so the planner would opt instead to redistribute.

```
SET gp_segments_for_planner = 1000;
```

Now, run that explain plan once again.

```
EXPLAIN SELECT c.zipcode, SUM(t.salesamt)
FROM customer c, transaction t
WHERE c.customerid = t.customerid
GROUP BY 1 ORDER BY 2 DESC LIMIT 20;
```

Notice that, where there had been a broadcast, you now see this

```
->  Hash  (cost=3.24..3.24 rows=62 width=13)
```

followed by a scan of the `transaction` table, then a hash, then a redistribute motion. So, we see that this join can be accomplished two ways.

- Using a broadcast of the smaller table
- Using a redistribution by the join key

Optimize the join through choice of distribution key

1. If this join was between two large fact tables, rather than between a fact table and a dimension table, you might choose to distribute `both` the column frequently used in joins, so the joins can be accomplished locally, within each segment, without data motion. We can try this here pretty easily.

Create a temporary copy of the `facts.transaction` table, altering its distribution key.

```
CREATE TEMP TABLE transaction_tmp AS SELECT * FROM transaction
DISTRIBUTED BY (customerid);
```

2. Run that EXPLAIN again, to see how the query would be run against this redistributed table.

```
EXPLAIN SELECT c.zipcode, SUM(t.salesamt)
FROM customer c, transaction_tmp t
WHERE c.customerid = t.customerid
GROUP BY 1 ORDER BY 2 DESC LIMIT 20;
```

This time, you'll notice that there was `no motion` prior to the

```
->  Hash Join  (cost=4.79..118.44 rows=1973 width=17)
```

line in the query plan. This illustrates a third way to perform this join: locally, within each of the segments, without any data moving across the network.

Summary

In the Greenplum MPP environment, joins ultimately are executed locally, within each of the segments. 0This may happen naturally, if the rows in the tables are co-located due to having been distributed by the column used in the join, or it may occur as a result of data movement. This data movement can take the form of either a *redistribute motion* or a *broadcast motion*, depending on which way the planner/optimizer finds to be the most economical. Having up to date table statistics helps, as does ensuring your distribution key choice is optimal, and that the data types of these columns are the same in each of the tables being joined.

GPDB Administrator – Lab 20

Pivotal Query Optimizer

Purpose

Exercise PQO and observe where its behavior differs from that of the Legacy Planner

Create a partitioned FactOnTimePerformance table

1. Connect to the `faa` database as `gpadmin`.
2. Verify your search_path includes the `faadata` schema, since this is where the existing tables reside (these were created in `lab-11.adoc`).

```
SHOW search_path;
```

3. Create the partitioned fact table.

```
CREATE TABLE FactOnTimePerformancePART
(LIKE FactOnTimePerformance)
WITH (APPENDONLY=true, COMPRESSTYPE=QuickLZ)
DISTRIBUTED RANDOMLY
PARTITION BY RANGE (FlightDate)
(
START ('2010-01-01'::DATE)
END ('2010-02-15'::DATE)
EVERY (INTERVAL '1 DAY')
);
```

4. Load the new partitioned fact table from the existing table, but with only the subset of the data matching the endpoints of the partitioning definition.

```
INSERT INTO FactOnTimePerformancePART
SELECT * FROM factontimeperformance
WHERE FlightDate >= '2010-01-01' AND FlightDate < '2010-02-15';
```

Compare the legacy and PQO query plans

1. Show the default setting for PQO for this DB.

```
SHOW OPTIMIZER;
```

2. Activate PQO for the current database.

```
ALTER DATABASE faa SET OPTIMIZER = ON;
```

Log back in so that the changes take effect.

```
\c
```

Finally, verify that PQO is on now.

```
SHOW OPTIMIZER;
```

3. Turn the optimizer back off.

```
SET OPTIMIZER = OFF;
```

Now, view the plan for a query against this partitioned table.

```
EXPLAIN SELECT
SUBSTRING(d.airlinename FROM 1 FOR 32) "Airline Name",
AVG(f.arrdelay)::NUMERIC(5, 2) "Average Delay (minutes)"
FROM dimairline d, factontimeperformancePART f
WHERE
f.airlineid = d.airlineid
AND flightdate BETWEEN '01-01-2010' AND '02-05-2010'
GROUP BY 1
ORDER BY 2 DESC
LIMIT 10;
```

How many rows does this plan contain?

4. Restore the default optimizer setting (so PQO is working).

```
RESET OPTIMIZER;
```

Again, view the plan for this same query, but using PQO.

```
EXPLAIN SELECT
SUBSTRING(d.airlinename FROM 1 FOR 32) "Airline Name",
AVG(f.arrdelay)::NUMERIC(5, 2) "Average Delay (minutes)"
FROM dimairline d, factontimeperformancePART f
WHERE
f.airlineid = d.airlineid
AND flightdate BETWEEN '01-01-2010' AND '02-05-2010'
GROUP BY 1
ORDER BY 2 DESC
LIMIT 10;
```

Since PQO does not enumerate each partition scan separately the plan should have been significantly more compact.

5. Run the query, which is designed to show the top 10 most delayed airlines during the given time period.

```
SELECT
SUBSTRING(d.airlinename FROM 1 FOR 32) "Airline Name",
AVG(f.arrdelay)::NUMERIC(5, 2) "Average Delay (minutes)"
FROM dimairline d, factontimeperformancePART f
WHERE
f.airlineid = d.airlineid
AND flightdate BETWEEN '01-01-2010' AND '02-05-2010'
GROUP BY 1
ORDER BY 2 DESC
LIMIT 10;
```

Investigate how, with PQO, it's now possible to UPDATE the distribution column of a table

1. Create a simple table for use in this exercise.

```
CREATE TABLE t
(
a INT,
b INT,
c INT
) DISTRIBUTED BY (a);
```

2. Load some data into table `t`.

```
INSERT INTO t (a, b, c) VALUES
(10,15,10),
(11,16,11),
(12,17,12),
(13,18,13),
(14,19,14),
(15,20,15),
(16,21,16),
(17,22,17),
(18,23,18),
(19,24,19),
(20,25,20);
```

3. With the Legacy Planner, try to do an UPDATE to the value in the `DISTRIBUTED BY` clause for table `t`.

```
SET OPTIMIZER = OFF;
EXPLAIN UPDATE t SET a = a + 1 WHERE a > 17;
```

   What was the result? The Legacy Planner did not support this operation. The PQO supports this since it is able to split into two operations, a `DELETE` and an `INSERT`.

4. Restore PQO as the optimizer and try again.

```
RESET OPTIMIZER;
EXPLAIN UPDATE t SET a = a + 1 WHERE a > 17;
```

   You should see a query plan where one operator is the *SPLIT* operator.

5. You can verify this actually works by running through the following three steps:

```
SELECT * FROM t WHERE a > 17 ORDER BY a ASC;
UPDATE t SET a = a + 1 WHERE a > 17;
SELECT * FROM t WHERE a > 17 ORDER BY a ASC;
```

Clean up

1. Drop table `t`.

```
DROP TABLE t;
```

2. Drop the partitioned fact table.

```
DROP TABLE factontimeperformancepart;
```

GPDB Administrator – Lab 21

Query Profiling

Purpose

In this lab, you use a combination of tools, commands, and schemas to analyze the overall performance of your environment. You will examine how the system behaves when retrieving data or updating tables.

Tune the Database and Queries

1. While logged in as gpadmin on your master server, verify your parameters are as shown below.

```
gpconfig -s parameter
```

Verify the following parameters:

```
max_fsm_relations = 1000
max_fsm_pages = 200000
work_mem = 32MB
maintenance_work_mem = 64MB
```

For example, to obtain the value for `max_fsm_relations`, type the following:

```
gpconfig -s max_fsm_relations
```

2. Load data in a table and check performance. Connect to the `faa` database as `gpadmin`.

```
psql faa
```

3. The `psql` timing parameter shows how long it takes to complete a command. The command will be used to compare the time it takes to execute `SELECT` statements on specific tables.

   Turn timing on.

   ```
   \timing on
   ```

4. Create a new table and load as shown. This step will create a new table and load it with more than 20 million records thus it will take some time (**Note** that this takes **10 minutes** in our Vagrant lab environment).

   ```
   CREATE TABLE factontimeperformance2 AS SELECT * FROM
   factontimeperformance DISTRIBUTED RANDOMLY;
   ```

5. After loading data into a table, analyze the table to update the statistics the query planner / optimizer relies upon to create its plans.

   ```
   ANALYZE factontimeperformance2;
   ```

6. Verify the data loaded.

   ```
   SELECT COUNT(*) FROM factontimeperformance2;
   ```

7. Disable `psql` timing. Use the `\timing` command to toggle the setting to off (you could also run `\timing off`).

   ```
   \timing
   ```

8. Greenplum provides an administrative schema called gp_toolkit that you can use to query the system catalogs, log files, and operating environment for system status information. The gp_toolkit schema contains a number of views that you can access using SQL commands. The `gp_toolkit` schema is accessible to all database users, although some objects may require superuser permissions.

   Look for tables that do not have statistics.

   ```
   SELECT * FROM gp_toolkit.gp_stats_missing;
   ```

9. Look for table bloat from lack of vacuuming.

   ```
   SELECT * FROM gp_toolkit.gp_bloat_diag limit 5;
   ```

10. If there is performance issue based on how a table is distributed, the distribution can be changed with an `ALTER TABLE` command. For example (**don't do this**).

    ```
    ALTER TABLE @factontimeperformance@ SET DISTRIBUTED BY (year);
    ```

    Note: the `@` characters are there to ensure you can't run this as written. The **year** column is a really poor candidate for a distribution key, so this action would take a while to complete, only rendering the data set pretty useless if you had more than a handful of segments. To see why, you could run the following query:

    ```
    SELECT year, COUNT(*)
    FROM factontimeperformance
    GROUP BY 1 ORDER BY 1;
    ```

    How many distinct `year` values were there? Had you altered your table in this way, and had you been running on a *real* GPDB instance having dozens, or hundreds, of segments, you would have confined the entire table to only four of these, and performance would suffer significantly.

11. Examine other views such as the `pg_stat_activity`, `pg_locks` and `pg_class` views.

    `pg_stat_activity` has information about current running queries.

    Look for the start time of all current queries in the database.

    ```
    SELECT query_start, procpid FROM pg_stat_activity;
    ```

12. `pg_locks` has data about locks in the database. This syntax shows the locks that are on the tables being accessed.

    ```
    SELECT datname, relname, pid, mode
    FROM pg_locks,pg_database,pg_class
    WHERE pg_locks.database=pg_database.oid AND
    pg_locks.relation = pg_class.oid;
    ```

13. Look for owner and object type in the database.

    ```
    SELECT relname, relowner FROM pg_class LIMIT 5;
    ```

14. Exit your PSQL session.

    ```
    \q
    ```

15. Database logs are another place to look for errors or tuning issues. The following are the database log files that reside on the system:

- `pg_xlog` contains Greenplum Write Ahead Logs (WAL, Greenplum implementation of transaction logging) files. Each file here is typically 16MB in size.
- `pg_clog` contains the commit log files which contain commit status of a transaction. The primary purpose of these files is to perform a database recovery in case of a crash by replaying these logs.
- `pg_log` contains the database instance logs by date. This is where you can look to diagnose errors with the database. Connection information is also logged here.

  Identify the logs that exist and use `gplogfilter` to access the content. Here, `gplogfilter` shows the log messages from the past ten minutes.

```
[gpadmin@mdw ~]$ ls $MASTER_DATA_DIRECTORY/pg_log

[gpadmin@mdw ~]$ gplogfilter -d :10
```

1. Use UNIX operating system commands to see performance from the OS point of view.

- `top` is used to get information on CPU and memory performance.
- `df` provides information on the capacity of the file systems.
- `ps -ef | grep postgres` lets you look for postgres processes currently running.
- `vmstat` is a utility that provides information on virtual memory usage on the system.
- `netstat` is a network status utility.
- `gpstate` is a Greenplum utility that provides information on the state of the cluster.

1. Getting an explain plan out of a query in Greenplum is easy. Prepend the word `EXPLAIN` to your query to see how the query optimizer will execute your query. This output may suggest steps you can take to optimize the execution of the query.

   Open another `psql` session and run the two queries below. Verify the differences in the `EXPLAIN` plan.

```
EXPLAIN SELECT flightnum, dayid
FROM factontimeperformance, dimairline, dimairport
WHERE dimairline.airlinename = 'United Air Lines Inc.: UA'
AND dimairport.airportdescription = 'Denver, CO: Denver International'
AND factontimeperformance.airlineid = dimairline.airlineid
AND dimairport.airportid = factontimeperformance.originairportid;
```

```
EXPLAIN SELECT flightnum, dayid
FROM factontimeperformance, dimairline, dimairport
WHERE dimairport.airportid = factontimeperformance.originairportid
AND dimairline.airlinename = 'United Air Lines Inc.: UA'
AND dimairport.airportdescription = 'Denver, CO: Denver International'
AND factontimeperformance.airlineid = dimairline.airlineid;
```

Note: Both of the queries result in the same execution plan. Based on the statistics Greenplum will optimize the query the same way regardless of the ordering of the `WHERE` clause.

2. Generate execution plans for the following two queries and compare them:

```
EXPLAIN SELECT distinct carrierid, flightnum
FROM factontimeperformance f, dimairline al, dimairport ap
WHERE
f.airlineid = al.airlineid
AND f.originairportid = ap.airportid
AND f.originairportid IN (
SELECT airportid
FROM dimairport
WHERE
airportdescription = 'Denver, CO: Denver International')
AND f.destairportid IN (
SELECT airportid
FROM dimairport
WHERE  airportdescription = 'Boston, MA: Logan International');
```

```
EXPLAIN SELECT distinct carrierid, flightnum
FROM factontimeperformance f, dimairline al, dimairport ap
WHERE
f.airlineid = al.airlineid
AND f.originairportid IN (
SELECT AIRPORTID
FROM dimairport
WHERE airportdescription = 'Denver, CO: Denver International')
AND f.destairportid IN (
SELECT AIRPORTID
FROM dimairport
WHERE airportdescription = 'Boston, MA: Logan International');
```

Note: While both of the queries display the same results, they have very different execution plans. The second query omits one of the join relations and creates a partial Cartesian product which has much slower performance. This can be identified in the execution plan from the additional Materialize step that is required.

3. Exit your PSQL session.

```
\q
```

Summary

The EXPLAIN command allows you to view the query plan for a query. EXPLAIN ANALYZE will actually run the query and show you the plan that was executed but does not return results.

Query plans are read from bottom to top and show a tree plan of nodes. A node represents a database operation, such as a table scan, a join, or a sort. GPDB query plans will also show motion nodes, which are operations that move tuples between the segment instances or from the segment instances to the master. Examining query plans helps uncover areas where performance can be improved.

In addition to query plans, using UNIX-based commands and tools, Greenplum users can obtain detailed information on how the system is behaving. The hardware and network has a very strong impact on performance, so those should always be part of your performance tuning approach.

GPDB Administrator – Lab 22

Query tuning and rewriting

Purpose

In this lab, you will be given a set of queries to analyze. The data that you have previously inserted deliberately does not have statistics collected for it. Do not use the `EXPLAIN` or `ANALYZE` commands until you are instructed to do so. You will want to compare the before and after effects of statistics.

Analyze Queries

Complete the following steps to view the explain plan in `psql`.

1. Access the `datamart` database as `gpadmin`:

```
psql datamart
```

2. Execute an explain plan for a simple query without analyzing the data. This query can be found in the file `/home/gp/sql/load_files/adv_sql_files/lab5query1.sql`.

```
\i /home/gp/sql/load_files/adv_sql_files/lab5query1.sql
```

   Which tables are being scanned? Is there any data motion? Are there any Broadcasts? Which table(s)?

3. How could you reduce the number of tables being scanned?
4. If you add a `WHERE` clause to get transactions for the month of May 2008 only, how does the explain plan change? This SQL can be found in the file `lab5query1a.sql`.

```
AND transaction.transdate BETWEEN '2008-05-01' AND '2008-05-31'
```

5. Execute another explain plan for a simple query without analyzing the data. This query is found in the file `/home/gp/sql/load_files/adv_sql_files/lab5query2.sql`:

```
\i /home/gp/sql/load_files/adv_sql_files/lab5query2.sql
```

Which tables are being scanned? Is there any data motion? Any Broadcasts? Which table(s)? Had you partitioned your tables, how many transaction tables would be scanned? Why?

Summary

You should execute an EXPLAIN statement with any new SQL that accesses large tables, any SQL that is running slowly, or against data objects that are new. This helps you to determine, in advance of going into production, the path that the optimizer will take when executing the query. The EXPLAIN utility can produce a plan for any SQL statement, so take advantage of this functionality as often as possible.

GPDB Administrator – Lab 23

Statistics

Purpose

In this lab you will gather statistics on all of the dimension and fact tables in your data warehouse. You will then do an EXPLAIN ANALYZE step for the two queries from the previous lab.

Gather Statistics and Analyze Queries

1. Connect to the `datamart` database as `gpadmin`.
2. Analyze each of dimensions and facts tables using the `ANALYZE` command.

   Analyze the `dimensions.country` table.

   ```
   analyze dimensions.country;
   ```

   Analyze the `dimensions.customer` table.

   ```
   analyze dimensions.customer;
   ```

   Analyze the `dimensions.store` table.

   ```
   analyze dimensions.store;
   ```

   Analyze the `facts.transaction` table.

   ```
   analyze facts.transaction;
   ```

3. Execute the first query from the previous plan with the *EXPLAIN* command. This query is found in the `/home/gp/sql/load_files/adv_sql_files/lab5query1.sql` file.

   ```
   \i /home/gp/sql/load_files/adv_sql_files/lab5query1.sql
   ```

   Are there any differences between the query plans with statistics?

4. Execute the second explain query plan from the previous lab using the EXPLAIN command. This query is found in the `/home/gp/sql/load_files/adv_sql_files/lab5query2.sql` file.

   ```
   \i /home/gp/sql/load_files/adv_sql_files/lab5query2.sql
   ```

   Do you note any differences between this output and the output from the previous lab?

5. Execute these same queries with an `EXPLAIN ANALYZE` statement to compare actual execution demographics with the projected execution demographics. This query is found in the`/home/gp/sql/load_files/adv_sql_files/lab6query1.sql` file:

   ```
   \i /home/gp/sql/load_files/adv_sql_files/lab6query1.sql
   ```

6. What do you note about this output when compared to an `EXPLAIN`?
7. Execute the second query with EXPLAIN ANALYZE as well. This query is found in the `/home/gp/sql/load_files/adv_sql_files/lab6query2.sql` file.

   ```
   \i /home/gp/sql/load_files/adv_sql_files/lab6query2.sql
   ```

8. What do you note about this output when compared to an `EXPLAIN`?

Summary

`ANALYZE` is a SQL command that updates the database statistics used by the query planner. In this use case, a query is run both before and after doing ANALYZE. After statistics are collected, the query planner chooses a HashAggregate operation over a much slower GroupAggregate operation. Without any statistics, the query planner could not estimate how many records might be returned, and therefore could not determine if there was sufficient work memory to do the aggregations in memory. The planner always takes the safe route and does aggregations by reading/writing from disk, which is significantly slower.

GPDB Administrator – Lab 24

Indexing Strategies

Purpose

In this lab, you determine requirements for secondary indexes on the fact table `TRANSACTION`. You will then create the secondary indexes.

Create Indexes to Support Queries

1. Here are some facts that you can use to develop your index strategies:

- Store managers often query the transaction table for their store(s) using the storeid.
- Business Intelligence (BI) tools execute a lot of queries for stores and customers by state for one or more states.
- One of the executive dashboards queries transactions at stores based on the store feature grocery.
- Auditing directly accesses transactions by the transaction id. +

Use the following matrix to determine which indexes you need to build to support the above analysis. A list of the tables that require indexes has been provided:

Table Name	Column(s)	Index Type
transaction		
store		
customer		
store		
transaction		

1. Connect to the `datamart` database as `gpadmin`.
2. Using the `CREATE INDEX` statement syntax, create the above indexes for your database. Note that, when you create an index on a partitioned table, you effectively create an index on each of its child partitions.

- The sample script, `/home/gp/sql/load_files/adv_sql_files/lab7createidx.sql`, can be used for guidance.

Execute the query file.

```
\i /home/gp/sql/load_files/adv_sql_files/lab7createidx.sql
```

1. `ANALYZE` your database tables with new indexes.

   Analyze the `facts.transaction` table.

   ```
   ANALYZE facts.transaction;
   ```

   Analyze the `dimensions.store` table.

   ```
   ANALYZE dimensions.store;
   ```

   Analyze the `dimensions.customer` table:

   ```
   ANALYZE dimensions.customer;
   ```

2. Determine if your indexes are being used by the optimizer by running the `EXPLAIN` utility with some simple queries that should use the index.

   For example:

   ```
   EXPLAIN SELECT * FROM transaction WHERE transid = 100987;
   ```

   If the indexes are not being used, it is likely because of the small size of the data set. Try changing the `enable_seqscan` configuration parameter to `off` if that is the case. Re-execute the EXPLAIN queries and then execute the queries. It is also going to be worthwhile to experiment with the following:

   - SET optimizer = OFF; In this mode, you are using the Legacy Planner, so you are able to influence its behavior using the configuration parameters shown here.
   - SET random_page_cost = 10; This setting influences the Legacy Planner's use of indexes. By reducing the value of the random page cost, you tell the planner that the cost of using an index is lower, so it will be more likely to favor the index. Its default value is 100. If, on the other hand, you were to set it to 10000, the query plan would be far less likely to incorporate your index.

   Note that you can reset all of these configuration values to their default values through the use of this command: `RESET ALL;`

Execute an `EXPLAIN` against a query.

```
EXPLAIN
SELECT *
FROM dimensions.store ds, facts.transaction ft
WHERE ds.storeid = ft.storeid;
```

3. Disable sequential scans.

```
SET enable_seqscan = off;
```

4. Execute `EXPLAIN` against the query again:

```
EXPLAIN
SELECT *
FROM dimensions.store ds, facts.transaction ft
WHERE ds.storeid = ft.storeid;
```

5. Set `enable_seqscan` to on.

```
SET enable_seqscan = on;
```

6. Execute `EXPLAIN` against the query again.

```
EXPLAIN
SELECT *
FROM dimensions.store ds, facts.transaction ft
WHERE ds.storeid = ft.storeid;
```

Summary

In most traditional databases, indexes can greatly reduce data access times. However, in an MPP database such as Greenplum, indexes should be used more sparingly. The Greenplum Database is very fast at sequential scanning. Indexes use a random seek pattern to locate records on disk. Also, unlike a traditional database, the data is distributed across the segments. This means each segment scans a smaller portion of the overall data in order to get the result. If using table partitioning, the total data to scan may be only a fraction of that.

With Greenplum, we recommend that you first try your query workload without adding any additional indexes. Indexes are more likely to improve performance for OLTP type workloads, where the query is returning a single record or a very small data set. Typically, a business intelligence (BI) query workload returns very large data sets, and thus does not make efficient use of indexes.

Note that the Greenplum Database will automatically create PRIMARY KEY indexes for tables with primary keys. If you are experiencing unsatisfactory performance, you may try adding indexes to see if performance improves.

GPDB Developer – Lab 25

Advanced Reporting Using OLAP

Purpose

In this lab, you will be given a set of reporting requirements. You will write the SQL using the various OLAP functions discussed in the module. You may use the examples in the module as the basis for your queries.

Advanced Reporting Using OLAP

1. The reporting requirements for which you will develop queries using OLAP functions are as follows:
   - Create a query that gives a rolling total of the Sales Amount by customer. Give the customer name.
   - Create a query that gives a monthly total of the Sales Amount for all stores with a grocery and a deli by month, by state summarizing at each control break (state and month). (Hint: try one of the grouping sets like ROLLUP or CUBE.)
   - Create a query that lists the running total of each amount column by month, by store. Display the store name with the measures.
   - Create a query that lists the top customer by Sales Amount, by month to support a customer rewards marketing campaign. (Hint: You will likely need 2 "derived tables" or *in-line views* to make this query work. It is tricky!)
   - Create a query that lists the stores, by month, ranked by the most items sold.
   - **Note:** When working with dates, the easiest method to convert a date into month/year is with the following syntax: TO_CHAR(date, 'yyyy-mm').

2. Connect to the `datamart` database as `gpadmin`.

3. The query for the first requirement makes use of the SUM moving window function.

   See the lab example in the, `/home/gp/sql/load_files/adv_sql_files/lab8query1.sql` file if you are having problems writing this SQL. The content is as follows:

```
SELECT c.custname
, t.transdate
, t.SalesAmt
, SUM(t.SalesAmt) OVER (PARTITION BY t.customerid
ORDER BY t.transdate ASC
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS RollingTotalAmt
FROM facts.transaction t
INNER JOIN dimensions.customer c
ON c.customerid = t.customerid;
```

4. The query for the second requirement works really well with the CUBE group function. You can use the ORDER BY clause of the SQL statement to display the results in a more readable format.

   See the lab example in the `/home/gp/sql/load_files/adv_sql_files/lab8query2.sql` file if you need assistance.

```
SELECT s.storename
, TO_CHAR(t.transdate, 'YYYY-MM') AS TransMonth
, SUM(t.SalesAmt) AS MonthlySalesAmt
FROM facts.transaction t
INNER JOIN dimensions.store s
ON s.storeid = t.storeid
WHERE s.grocery = true
AND s.deli = true
GROUP BY CUBE(s.storename, TransMonth)
ORDER BY 1, 2;
```

5. The query for the third requirement is tricky.

   Consider using a derived table or an in-line view to do a summary by month and store of each of the amount columns. Then apply the SUM window function to each of those columns in your table.

See the lab example in the `/home/gp/sql/load_files/adv_sql_files/lab8query3.sql` file if you need a hint.

```
SELECT s.storename
, t.TransMonth
, SUM(t.salesamt) OVER (PARTITION BY s.storename ORDER BY t.TransMonth)
AS TotalSalesAmt
, SUM(t.taxamt) OVER (PARTITION BY s.storename ORDER BY t.TransMonth)
AS TotalTaxAmt
, SUM(t.discountamt) OVER (PARTITION BY s.storename ORDER BY
t.TransMonth) AS TotalDiscountAmt
, SUM(t.couponamt) OVER (PARTITION BY s.storename ORDER BY
t.TransMonth) AS TotalCouponAmt
, SUM(t.cashamt) OVER (PARTITION BY s.storename ORDER BY t.TransMonth)
AS TotalCashAmt
, SUM(t.checkamt) OVER (PARTITION BY s.storename ORDER BY t.TransMonth)
AS TotalCheckAmt
, SUM(t.ccamt) OVER (PARTITION BY s.storename ORDER BY t.TransMonth) AS
TotalCCAmt
, SUM(t.debitamt) OVER (PARTITION BY s.storename ORDER BY t.TransMonth)
AS TotalDebitAmt
, SUM(t.otheramt) OVER (PARTITION BY s.storename ORDER BY t.TransMonth)
AS TotalOtherAmt
FROM (SELECT storeid
, TO_CHAR(transdate,'YYYY-MM') AS TransMonth
, SUM(salesamt) AS SalesAmt
, SUM(taxamt) AS TaxAmt
, SUM(discountamt) AS DiscountAmt
, SUM(couponamt) AS CouponAmt
, SUM(cashamt) AS CashAmt
, SUM(checkamt) AS CheckAmt
, SUM(ccamt) AS CCAmt
, SUM(debitamt) AS DebitAmt
, SUM(otheramt) AS OtherAmt
FROM facts.transaction
GROUP BY 1,2) t
INNER JOIN dimensions.store s
ON s.storeid = t.storeid;
```

6. The query for the fourth requirement is harder. You should consider a derived table or in-line view. However, you will need two:
    - The first one does the summary at your control breaks.
    - The second gets the ranking for you and then executes the final statement against the last derived table to get the results.

See the lab example in the `/home/gp/sql/load_files/adv_sql_files/lab8query4.sql` file for details and hints. The contents of the file are displayed below:

```
SELECT Y.TransMonth
, Y.custname AS CustomerName
, Y.TotalSalesAmt
FROM (SELECT X.TransMonth
, c.custname
, X.TotalSalesAmt
, RANK() OVER (PARTITION BY X.TransMonth
ORDER BY X.TotalSalesAmt DESC) AS CustomerRanking
FROM (SELECT TO_CHAR(t.transdate,'YYYY-MM') AS TransMonth
, t.customerid
, SUM(t.salesamt) AS TotalSalesAmt
FROM facts.transaction t
GROUP BY 1, 2
) X
INNER JOIN dimensions.customer c
ON c.customerid = X.customerid
) Y
WHERE Y.CustomerRanking = 1
ORDER BY 1 ASC;
```

7. The query for the last requirement is similar to the query example in the lesson. Consider using a window partition to solve this query.

   See the lab example in the `/home/gp/sql/load_files/adv_sql_files/lab8query5.sql` file for hints. The content is as follows:

```
SELECT t.TransMonth
, s.StoreName
, t.TotalItemCnt
, RANK() OVER (PARTITION BY t.TransMonth
ORDER BY t.TotalItemCnt DESC) AS ranking
FROM (SELECT storeid
, TO_CHAR(transdate,'YYYY-MM') AS TransMonth
, SUM(itemcnt) AS TotalItemCnt
FROM transaction
GROUP BY 1, 2) t
INNER JOIN store s ON s.storeid = t.storeid
ORDER BY 1 ASC;
```

8. to the faa database to execute queries against larger data sets.

```
\c faa
```

9. The following query lists flights from Denver International airport to Logan International (Boston) airport.

```
SELECT DISTINCT carrierid, flightnum
FROM factontimeperformance f INNER JOIN dimairline al
ON f.airlineid = al.airlineid
INNER JOIN dimairport ap
ON f.originairportid = ap.airportid
WHERE f.originairportid in (
SELECT airportid
FROM dimairport
WHERE airportdescription = 'Denver, CO: Denver International') AND
f.destairportid in (
SELECT airportid
FROM dimairport
WHERE airportdescription = 'Boston, MA: Logan International');
```

10. List the number of flights for each airline carrier leaving New York (originwacid = 22). Provide the information for each carrier per year. Display the grand total number of flights in the year.

    Use the ROLLUP operator to display all the results at once.

```
SELECT year, carrierID, count(*)
FROM factontimeperformance
WHERE originwacid = 22
GROUP BY ROLLUP(year, carrierID)
ORDER BY year, carrierID;
```

11. List the number of flights leaving New York on a carrier and year basis. Display the total number of flights in the year and the total number of flights by a carrier. Use the cube operator to display all the results at once.

```
12.  SELECT year, carrierID, count(*)
13.  FROM factontimeperformance
14.  WHERE originwacid = 22
15.  GROUP BY CUBE(year, carrierID)
16.  ORDER BY year, carrierID;
```

Summary

Use the OLAP grouping functionality whenever you need to display totals and sub-totals based on your group predicates.

Use the OLAP window expressions whenever you want to utilize the values in prior rows for comparison or further aggregation. This precludes having to make multiple scans of the same table in order to accomplish these comparisons.

GPDB Developer – Lab 26

User Defined Functions

Purpose

In this lab, you will create functions to perform simple, repeatable queries and tasks. You will create functions that use the SQL and the PL/pgSQL language. You will be required to create one user defined data type in order to correctly return data.

Note: There is no `GOTO` in PL/PgSQL (this may come up later).

Create SQL Functions

1. Connect to the `datamart` database as `gpadmin`.
2. Create an overloaded SQL function that returns zero or more rows from the store dimension table with a parameter of either `storeid` or `state`. You can create the procedure in the public schema.

   For hints on how to create the function, refer to the lab example in the `/home/gp/sql/load_files/adv_sql_files/lab10query1.sql` file.

   Remember that you want to return a `SETOF` to get more than one row.

```
CREATE OR REPLACE FUNCTION public.GetStore (whichstate CHAR(2))
RETURNS SETOF store AS $$
SELECT *
FROM dimensions.store
WHERE state = $1;
$$
LANGUAGE SQL;

CREATE OR REPLACE FUNCTION public.GetStore (whichid integer)
RETURNS SETOF store AS $$
SELECT *
FROM dimensions.store
WHERE storeid=$1;
$$
LANGUAGE SQL;
```

3. Test your function by selecting for the state of Missouri (MO) or any store id less than 30.

```
  SELECT * FROM public.GetStore('MO');
  SELECT * FROM public.GetStore(11);
```

4. Create a SQL function that takes as a parameter a table name (you can use any of the dimension tables) and returns the appropriate name column in uppercase. Call the function uppername. Create it in the public schema.

   For hints on how to create the function, refer to the lab example in the `/home/gp/sql/load_files/adv_sql_files/lab10query2.sql` file.

```
CREATE FUNCTION public.UpperName (store)
RETURNS text AS $$
SELECT UPPER($1.storename);
$$
LANGUAGE SQL;
```

5. Test the function by selecting the *storeid*, *storename*, *public.Uppername(store)*, *city* columns from *store*.

```
SELECT storeid, storename, public.Uppername(store), city FROM store;
```

Create PL/pgSQL Functions

1. Create a PL/pgSQL function to join the transaction fact table with the store and customer dimensions to return the transaction columns (*transid*, *transdate*, *salesamt*, *taxamt*, *checkamt*, *cashamt*, *ccamt*, *debitamt*), the store columns (*storename*, *city*, *state*) and the customer columns (*custname*, *city*, *state*).

   Parameters are the starting and ending transaction dates.

   HINT: The output will be a user defined data type.

   First, create the user defined data type and call it `ReportTypeA`.

```
CREATE TYPE ReportTypeA AS (
transid BIGINT,
transdate DATE,
salesamt NUMERIC,
taxamt NUMERIC,
checkamt NUMERIC,
cashamt NUMERIC,
ccamt NUMERIC,
debitamt NUMERIC,
storename CHARACTER VARYING,
storecity CHARACTER VARYING,
storestate CHARACTER VARYING,
custname CHARACTER VARYING,
custcity CHARACTER VARYING,
custstate CHARACTER VARYING);
```

2. Create the function, `RunReportA`. Create the function in the public schema.

```
CREATE OR REPLACE FUNCTION public.RunReportA
(IN StartDate DATE, IN EndDate DATE)
RETURNS SETOF ReportTypeA AS
$BODY$
DECLARE r ReportTypeA%ROWTYPE;
BEGIN
FOR r IN SELECT t.transid
,t.transdate
,t.salesamt
,t.taxamt
,t.checkamt
,t.cashamt
,t.ccamt
,t.debitamt
,s.storename
,s.city
,s.state
,c.custname
,c.city
,c.state
FROM facts.transaction t
INNER JOIN dimensions.store s
ON s.storeid = t.storeid
INNER JOIN dimensions.customer c
ON c.customerid = t.customerid
WHERE t.transdate BETWEEN StartDate AND EndDate LOOP
-- Return current row of SELECT
RETURN NEXT r;
END LOOP;
RETURN;
END
$BODY$ LANGUAGE plpgsql;
```

3. Test the function by running a report for the dates, 2008-07-01 to 2008-07-07.

```
SELECT * FROM
public.runreporta ('2008-07-01'::date, '2008-07-07'::date);
```

Summary

You may find that using functions to generate reports is a simple way to store SQL in the database to ensure that any user executing the query cannot change it, so that the query will produce consistent results.

Functions provide a versatile way to perform complex transformation logic. Use functions in conjunction with temporary tables whenever you need to modify rows during ETL processing. Remember that you can also create temporary tables within the PL/pgSQL function.

Use dynamic functions whenever you need to create ad-hoc SQL. Again, this will keep consistent and repeatable SQL stored in the database.