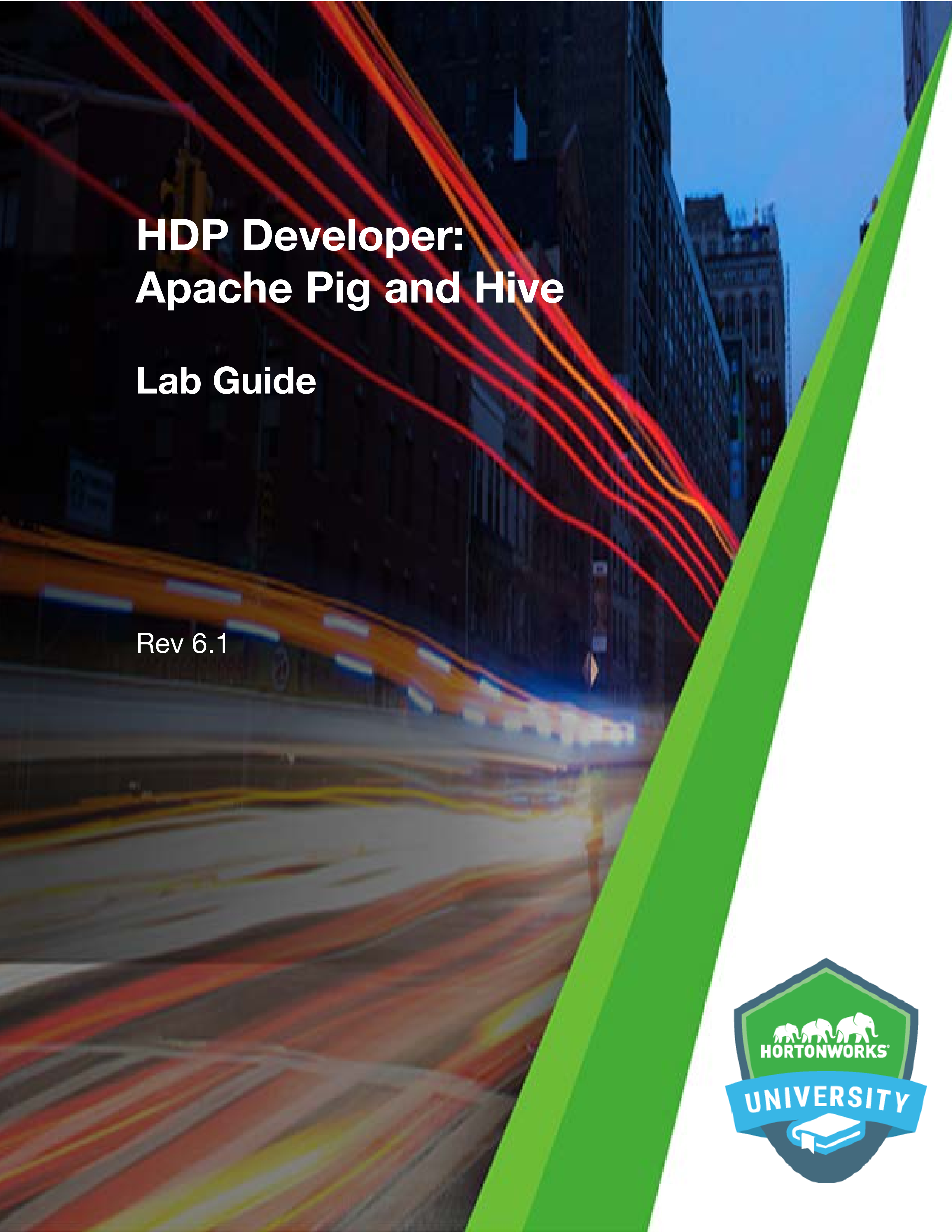


HDP Developer: Apache Pig and Hive

Lab Guide

Rev 6.1

HORTONWORKS UNIVERSITY

This instructional program, including all material provided herein, is supplied without any guarantees from Hortonworks, Inc. Hortonworks, Inc. assumes no liability for damages or legal action arising from the use or misuse of contents or details contained herein.

Become a ***Hortonworks Certified Professional*** and establish your credentials:

- HDP Certified Developer: for Hadoop developers using frameworks like Pig, Hive, Sqoop and Flume.

- HDP Certified Administrator: for Hadoop administrators who deploy and manage Hadoop clusters.

- HDP Certified Developer: Java: for Hadoop developers who design, develop and architect Hadoop-based solutions written in the Java programming language.

- HDP Certified Developer: Spark: for Hadoop developers who write and deploy applications for the Spark framework.

- HDF Certified Professional: for DataFlow Operators responsible for building and deploying HDF workflows.

How to Register: Visit www.examslocal.com and search for “Hortonworks” to register for an exam. The cost of each exam is $250 USD, and you can take the exam anytime, anywhere using your own computer. For more details, including a list of exam objectives and instructions on how to attempt our practice exams, visit http://hortonworks.com/training/certification/

Earn Digital Badges: Hortonworks Certified Professionals receive a digital badge for each certification earned. Display your badges proudly on your résumé, LinkedIn profile, email signature, etc.

Self Paced Learning Library

On Demand Learning

Hortonworks University Self-Paced Learning Library is an on-demand dynamic repository of content that is accessed using a Hortonworks University account. Learners can view lessons anywhere, at any time, and complete lessons at their own pace. Lessons can be stopped and started, as needed, and completion is tracked via the Hortonworks University Learning Management System.

Hortonworks University courses are designed and developed by Hadoop experts and provide an immersive and valuable real world experience. In our scenario-based training courses, we offer unmatched depth and expertise. We prepare you to be an expert with highly valued, practical skills and prepare you to successfully complete Hortonworks Technical Certifications.

Target Audience: Hortonworks University Self-Paced Learning Library is designed for those new to Hadoop, as well as architects, developers, analysts, data scientists, and IT decision makers. It is essentially for anyone who desires to learn more about Apache Hadoop and the Hortonworks Data Platform.

Duration: Access to the Hortonworks University Self-Paced Learning Library is provided for a 12-month period per individual named user. The subscription includes access to over 400 hours of learning lessons.

The online library accelerates time to Hadoop competency. In addition, the content is constantly being expanded with new material, on an ongoing basis.

Visit: http://hortonworks.com/training/class/hortonworks-university-self-paced-learning-library/

Table of Contents

Lab: Starting an HDP 2.3 Cluster

About This Lab

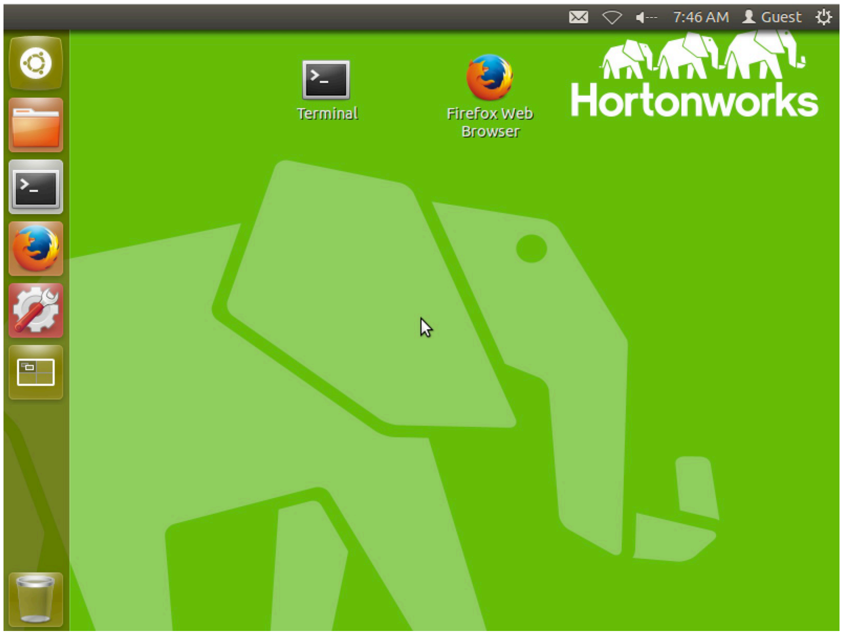
Objective:	To start an HDP cluster in your VM.
File locations:	NA
Successful outcome:	The (local or cloud-hosted) VM will be running HDP 2.3.
Before you begin:	VMWare should be installed on your machine and the classroom VM should be imported, unless you are using a cloud-hosted VM.
Related lesson:	***Understanding Hadoop***

Lab Steps

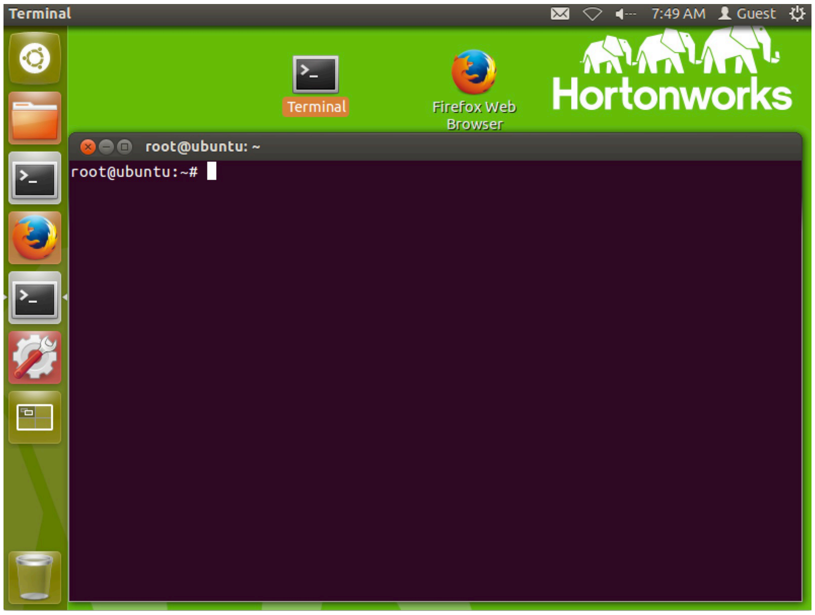
1) Start the VM

a. If applicable, start VMWare Player (or Fusion) on your local machine, select the course VM from the list of virtual machines, then click the Play virtual machine link.

b. You should see the desktop of your local or cloud-hosted VM:



c. Open a Terminal by double-clicking the shortcut on the desktop:



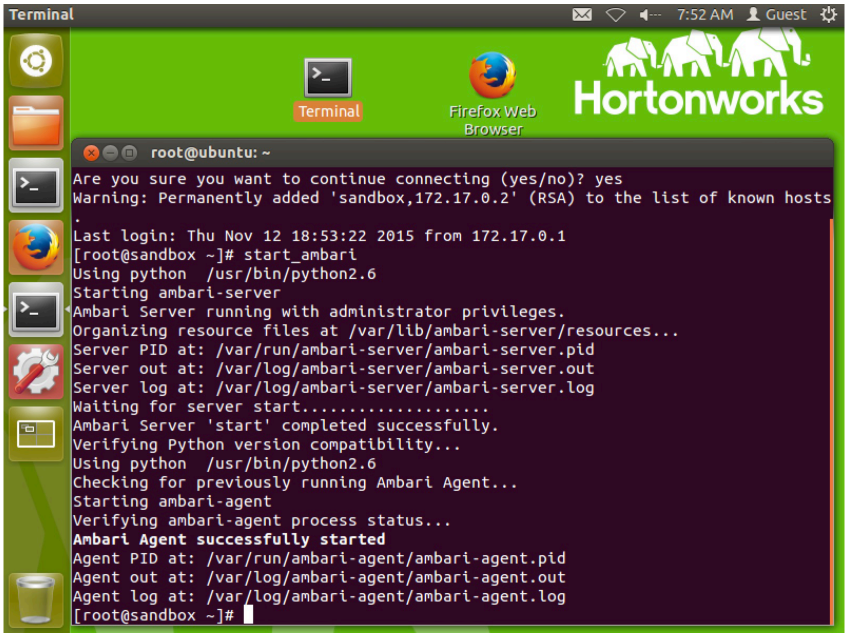
```
root@ubuntu:~#
```

2) Verify that the cluster is running

a) Navigate into the Docker sandbox instance.

Note
Type "yes" if asked "are you sure you want to continue connecting".

```
root@ubuntu:~# ssh sandbox
[root@sandbox ~]#
```



```
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'sandbox,172.17.0.2' (RSA) to the list of known hosts
.
Last login: Thu Nov 12 18:53:22 2015 from 172.17.0.1
[root@sandbox ~]# start_ambari
Using python  /usr/bin/python2.6
Starting ambari-server
Ambari Server running with administrator privileges.
Organizing resource files at /var/lib/ambari-server/resources...
Server PID at: /var/run/ambari-server/ambari-server.pid
Server out at: /var/log/ambari-server/ambari-server.out
Server log at: /var/log/ambari-server/ambari-server.log
Waiting for server start...................
Ambari Server 'start' completed successfully.
Verifying Python version compatibility...
Using python  /usr/bin/python2.6
Checking for previously running Ambari Agent...
Starting ambari-agent
Verifying ambari-agent process status...
Ambari Agent successfully started
Agent PID at: /var/run/ambari-agent/ambari-agent.pid
Agent out at: /var/log/ambari-agent/ambari-agent.out
Agent log at: /var/log/ambari-agent/ambari-agent.log
[root@sandbox ~]#
```

b) From the command line, enter the following command, which displays the usage of the `hdfs dfsadmin` utility:

```
# su -l hdfs -c "hdfs dfsadmin"
```

Note

The “dfs” in `dfsadmin` stands for distributed filesystem, and the `dfsadmin` utility contains administrative commands for communicating with the Hadoop Distributed File System.

c) Notice the `dfsadmin` utility has a `-report` option, which outputs the current health of your cluster. Enter the following command to view this report:

```
# su -l hdfs -c "hdfs dfsadmin -report"
```

d) What is the configured capacity of your distributed filesystem? ___________

Answer: Look for the value of “Configured Capacity” at the start of the output.

e) What is the present capacity? ___________

Answer: Look for the value of “Present Capacity” at the start of the output.

f) How much of your distributed filesystem is used right now? ___________

Answer: Look for the value of “DFS Used.”

g) What do you think an “Under-replicated block” is? ______________________

Answer: Data in HDFS is chunked into blocks and copied to various nodes in the cluster. If a particular block does not have enough copies, it is referred to as “under replicated.”

h) How many available DataNodes does your cluster have? ___________

Answer: 1

3) View the Processes on the Cluster Nodes

Enter the `jps` command, which lists all Java processes running on this machine.

While your specific processes and they order they are presented in may look slightly different than the list below, you should still see the NameNode process running:

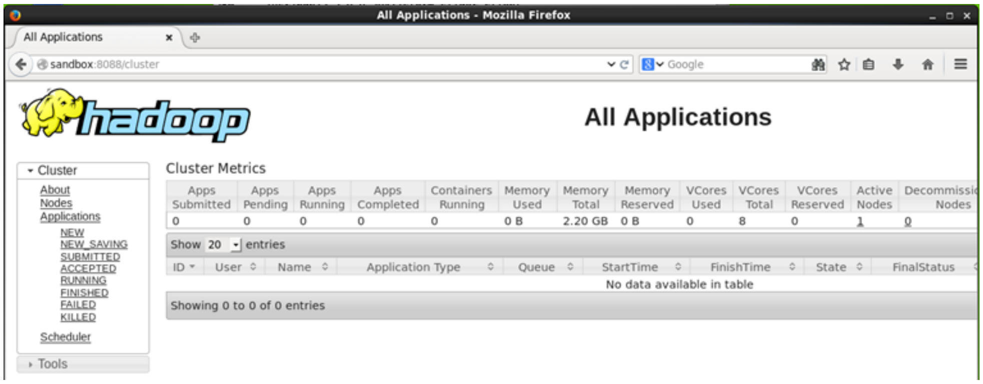
```
# jps
3706 ResourceManager
2988 QuorumPeerMain
3675 RunJar
4032 RunJar
3740 NodeManager
3188 Nfs3
3186 Portmap
3738 JobHistoryServer
2556 DataNode
2557 SecondaryNameNode
2560 NameNode
3712 ApplicationHistoryServer
3511 RunJar
24669 -- process information unavailable
5516 AmbariServer
31813 Jps
3029 Bootstrap
```

4) View the ResourceManager UI

a) Open Firefox in the VM by double-clicking on the Firefox icon.
b) Enter the following URL:

```
http://sandbox:8088/
```

c) Notice that the URL shows the ResourceManager Web UI:



All Applications

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissi Nodes
0	0	0	0	0	0 B	2.20 GB	0 B	0	8	0	1	0

Show 20 entries

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus
No data available in table								

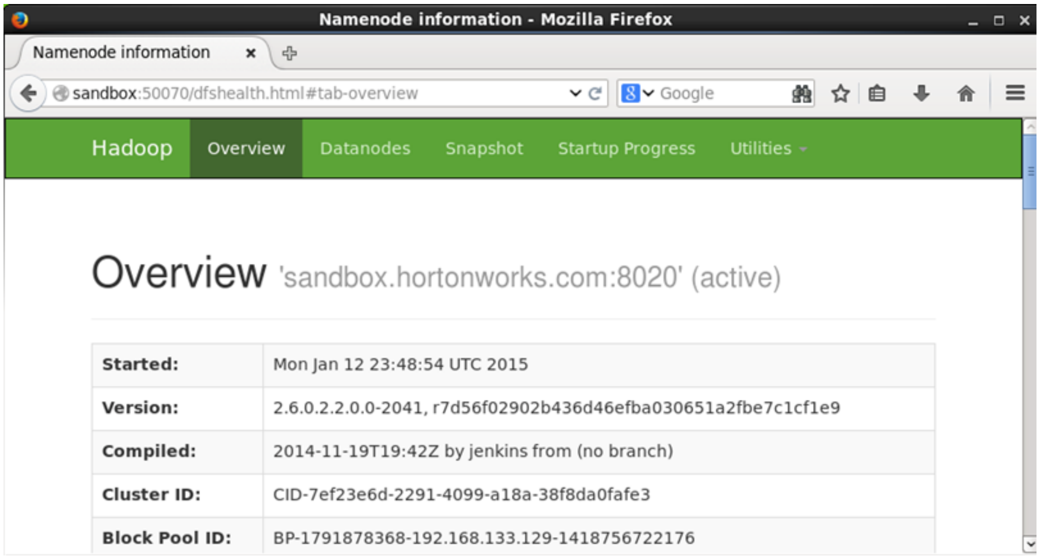
Showing 0 to 0 of 0 entries

The ResourceManager UI displays information about the applications that have been executed on your Hadoop cluster.

5) View the NameNode UI

a) Point your browser to the NameNode UI:

```
http://sandbox:50070/
```



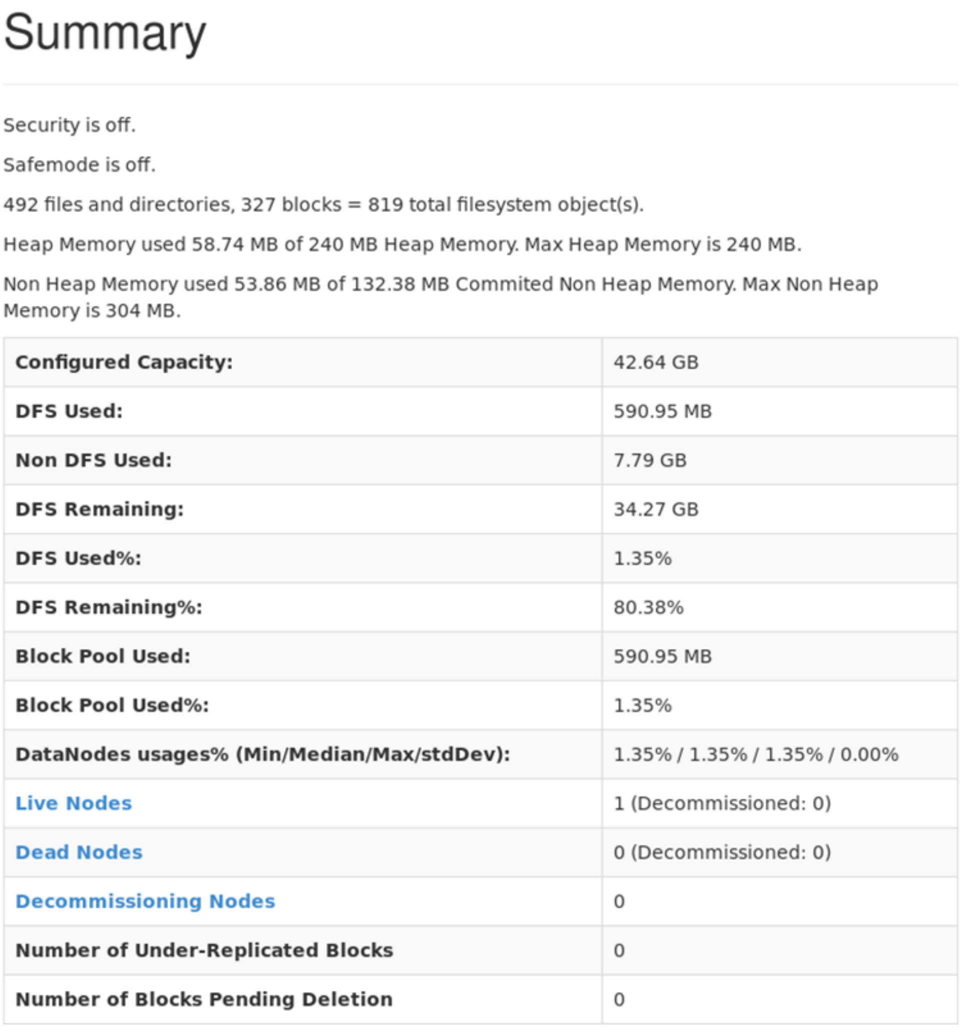
Overview 'sandbox.hortonworks.com:8020' (active)

Started:	Mon Jan 12 23:48:54 UTC 2015
Version:	2.6.0.2.2.0.0-2041, r7d56f02902b436d46efba030651a2fbe7c1cf1e9
Compiled:	2014-11-19T19:42Z by jenkins from (no branch)
Cluster ID:	CID-7ef23e6d-2291-4099-a18a-38f8da0fafe3
Block Pool ID:	BP-1791878368-192.168.133.129-1418756722176

Notice the NameNode UI contains a lot of information about the cluster. The Overview page shows the version of Hadoop and other details.

b) Scroll down to the Summary section on the Overview page.

You will see a table that looks similar to the `hdfs dfsadmin -report` output:



Summary

Security is off.

Safemode is off.

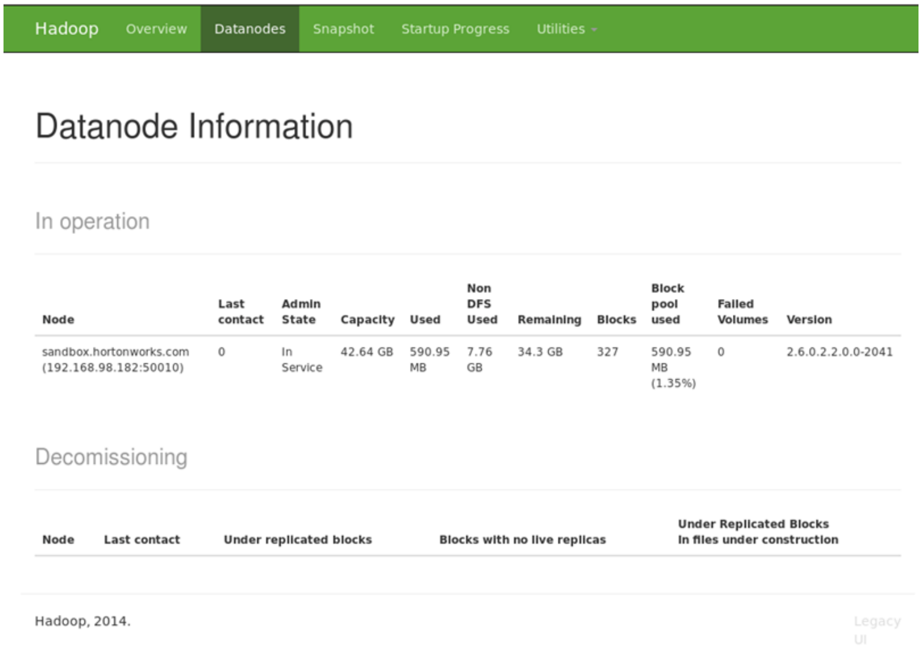
492 files and directories, 327 blocks = 819 total filesystem object(s).

Heap Memory used 58.74 MB of 240 MB Heap Memory. Max Heap Memory is 240 MB.

Non Heap Memory used 53.86 MB of 132.38 MB Commited Non Heap Memory. Max Non Heap Memory is 304 MB.

Configured Capacity:	42.64 GB
DFS Used:	590.95 MB
Non DFS Used:	7.79 GB
DFS Remaining:	34.27 GB
DFS Used%:	1.35%
DFS Remaining%:	80.38%
Block Pool Used:	590.95 MB
Block Pool Used%:	1.35%
DataNodes usages% (Min/Median/Max/stdDev):	1.35% / 1.35% / 1.35% / 0.00%
Live Nodes	1 (Decommissioned: 0)
Dead Nodes	0 (Decommissioned: 0)
Decommissioning Nodes	0
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion	0

c) Click on the Datanodes tab of the NameNode UI:



Datanode Information

In operation

Node	Last contact	Admin State	Capacity	Used	Non DFS Used	Remaining	Blocks	Block pool used	Failed Volumes	Version
sandbox.hortonworks.com (192.168.98.182:50010)	0	In Service	42.64 GB	590.95 MB	7.76 GB	34.3 GB	327	590.95 MB (1.35%)	0	2.6.0.2.2.0.0-2041

Decomissioning

Node	Last contact	Under replicated blocks	Blocks with no live replicas	Under Replicated Blocks In files under construction

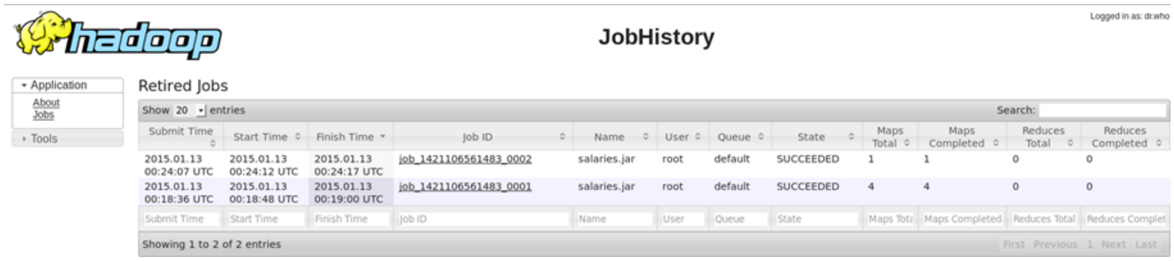
Hadoop, 2014.

You should see one DataNode in your cluster.

6) View the JobHistory UI

a) The JobHistory UI is at:

```
http://sandbox:19888/
```



JobHistory

Retired Jobs

Show 20 entries Search:

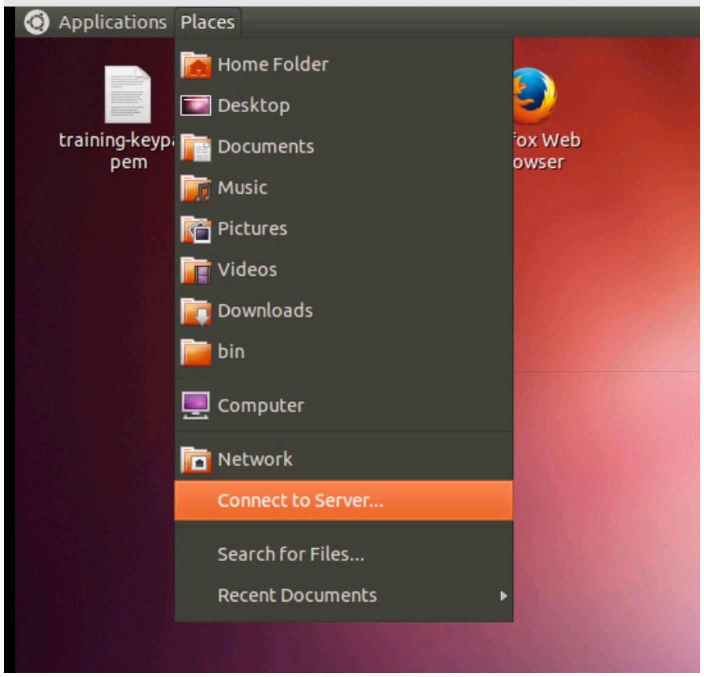
Submit Time	Start Time	Finish Time	Job ID	Name	User	Queue	State	Maps Total	Maps Completed	Reduces Total	Reduces Completed
2015.01.13 00:24:07 UTC	2015.01.13 00:24:12 UTC	2015.01.13 00:24:17 UTC	job_1421106561483_0002	salaries.jar	root	default	SUCCEEDED	1	1	0	0
2015.01.13 00:18:36 UTC	2015.01.13 00:18:48 UTC	2015.01.13 00:19:00 UTC	job_1421106561483_0001	salaries.jar	root	default	SUCCEEDED	4	4	0	0

Showing 1 to 2 of 2 entries

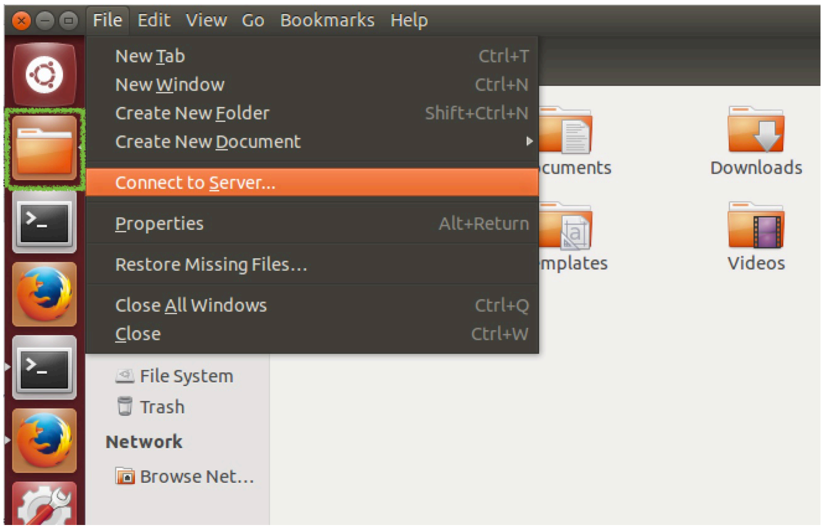
As it sounds, the JobHistory UI shows the jobs that have executed on your cluster. You have not submitted any jobs to your cluster yet, but this page comes in handy as you work on the labs throughout this course.

7) Configure the gedit text editor to be launched on the "ubuntu" parent VM and to allow opening of lab files on the Docker "sandbox" VM.

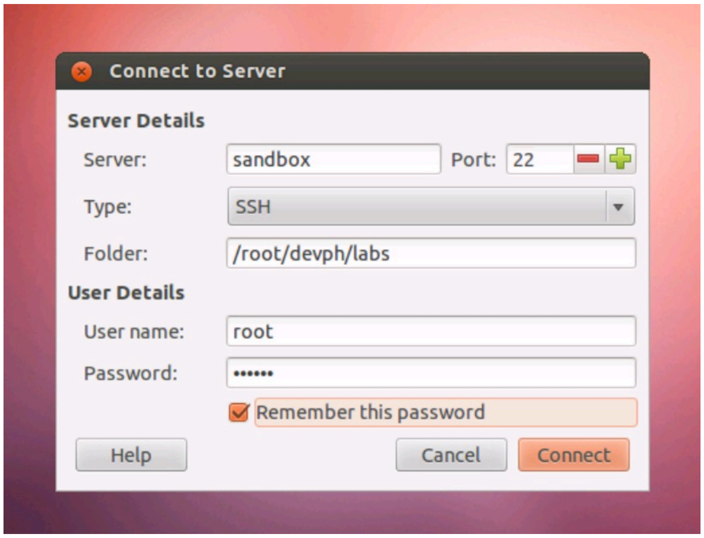
a) ***If using the hosted AWS virtual machine***, click on "Places" in the upper-left corner of the VM and then "Connect to Server…" as shown below.



b) ***If using the downloaded VMware-based virtual machine***, click on "Home Folder" icon on the left nav (highlighted within a green box in the next screenshot) and then select "File" and "Connect to Server…" as shown below.



c) In the new dialogue box that surfaces, change the "Type" pulldown to "SSH" and update the values to look like the following (the password is "hadoop") before clicking on the "Connect" button.



Connect to Server

Server Details

Server: sandbox

Port: 22

Type: SSH

Folder: /root/devph/labs

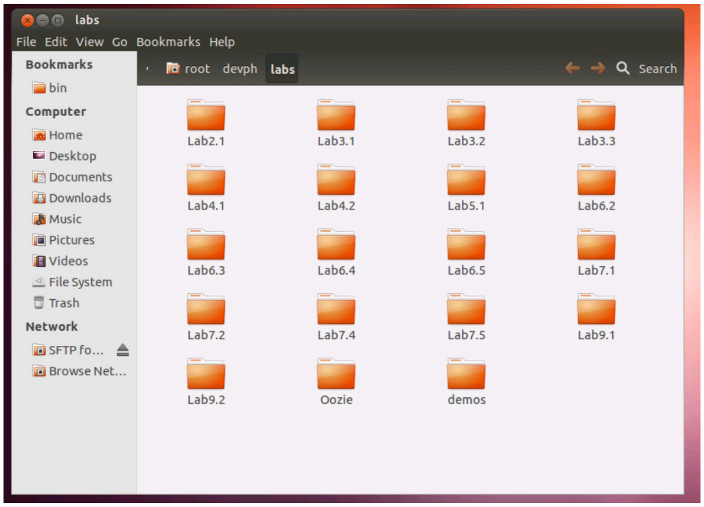
User Details

User name: root

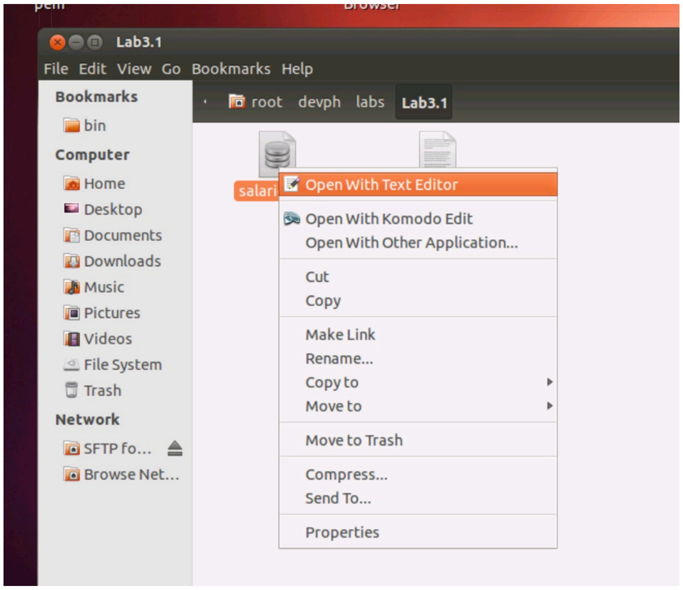
Password: ••••••

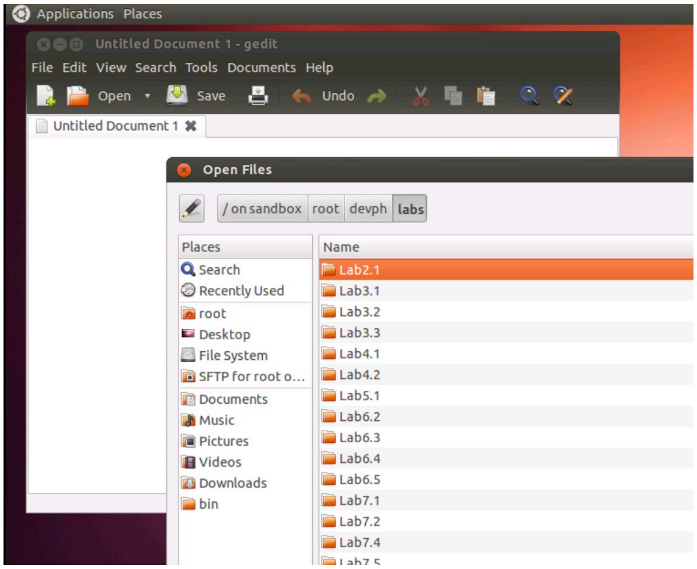
☑ Remember this password

d) Now you will see a UI-based view of the labs folder on the sandbox node.



e) You can now open files directly by right-clicking on them and choosing the option to open with the gedit visual text editor or by using gedit's open/new controls.





Result

You now have a single-node Hortonworks Data Platform 2.3 cluster running in a virtual machine. You will use this cluster to perform the labs in this course.

Demonstration: Understanding Block Storage

About this Demonstration

Objective:	To understand how data is partitioned into blocks and stored in HDFS.
During this Demonstration:	Watch as your instructor performs the following steps.
Related lesson:	***The Hadoop Distributed File System (HDFS)***

Demonstration Steps

1) Put the File into HDFS

a. If not already done, open a Terminal in your VM and type "ssh sandbox".

b. Change directories to `/root/devph/labs/demos`:

```
# cd /root/devph/labs/demos/
```

c. Use `less` to view the contents of the `stocks.csv` file. Press `q` when you are finished to exit `less`.

```
# less stocks.csv
```

d. Try putting the file into HDFS with a block size of 30 bytes:

```
# hdfs dfs -D dfs.blocksize=30 -put stocks.csv
```

Notice that a size of 30 bytes is not a valid blocksize. The blocksize needs to be at least 1,048,576 according to the `dfs.namenode.fs-limits.min-block-size` property:

```
put: Specified block size is less than configured minimum value
(dfs.namenode.fs-limits.min-block-size): 30 < 1048576
```

e. Try the put again, but use a block size of 2,000,000:

```
# hdfs dfs -D dfs.blocksize=2000000 -put stocks.csv
```

Notice that 2,000,000 is not a valid blocksize because it is not a multiple of 512 (the checksum size).

f. Try the put again, but this time use 1,048,576 for the blocksize:

```
# hdfs dfs -D dfs.blocksize=1048576 -put stocks.csv
```

g. This time the put command should have worked. Use ls to verify that the file is in HDFS in the `/user/root` folder:

```
# hdfs dfs -ls
Found 1 items
-rw-r--r--   1 root root    3613198  stocks.csv
```

2) View the Number of Blocks

a. Run the following command to view the number of blocks that were created for `stocks.csv`:

```
# hdfs fsck /user/root/stocks.csv
```

b. Notice there are four blocks. Look for the following line in the output:

```
Total blocks (validated):4 (avg. block size 903299 B)
```

3) Find the Actual Blocks

a. Enter the same `fsck` command as before, but add the `-files` and `-blocks` options:

```
# hdfs fsck /user/root/stocks.csv -files -blocks
```

Notice the output contains the block IDs, which are coincidentally the names of the files on the DataNodes.

b. Run the command again, but this time add the `-locations` flag:

```
# hdfs fsck /user/root/stocks.csv -files -blocks -locations
```

Notice in the output that the IP address of the DataNode appear next to each block.

c. Change directories to the following:

```
# cd /hadoop/hdfs/data/current/BP-xxx/current/finalized/
```

Replace BP-xxx with the actual folder name. To finish this, use the TAB key to complete the filename once you have typed `B`. Then finish typing the rest of the directory path.

d. Try and find the folder that contains the blocks you are looking for and change directories into that folder. The easiest way is to look at the timestamps and find the most recently changed folder. You can use the `stat *` command to view the contents of the directory, then use `ll` to list the contents of that directory.

```
# stat *
# cd <most recently created directory - for example, subdir0>
# ll
```

Important

If the results of the ll command are additional subdirectories rather than block information (as shown in the next lab step), repeat the process above to once again find the newest directory created , change to it, and list its contents.

e. Notice that the actual blocks appear in this folder. Look for files that are exactly 1,048,576 bytes. These are three of the blocks.

   Notice that the fourth block is smaller: 467,470 bytes.

```
-rw-r--r--  1 hdfs hdfs 1048576 blk_1073742090
-rw-r--r--  1 hdfs hdfs    8199 blk_1073742090_1266.meta
-rw-r--r--  1 hdfs hdfs 1048576 blk_1073742091
-rw-r--r--  1 hdfs hdfs    8199 blk_1073742091_1267.meta
-rw-r--r--  1 hdfs hdfs  467470 blk_1073742093
-rw-r--r--  1 hdfs hdfs    3663 blk_1073742093_1269.meta
```

f. You can view the contents of a block (although this is not a typical task in Hadoop). Here is the tail of the second block:

```
# tail blk_1073741905
NYSE,XKK,2007-08-20,9.51,9.64,9.30,9.51,4700,7.17
NYSE,XKK,2007-08-17,9.30,9.99,9.26,9.57,3900,7.21
NYSE,XKK,2007-08-16,9.45,10.00,8.11,9.05,23400,6.82
NYSE,XKK,2007-08-15,9.51,9.51,9.18,9.35,4900,7.04
NYSE,XKK,2007-08-14,9.52,9.52,9.51,9.51,1100,7.17
NYSE,XKK,2007-08-13,9.60,9.60,9.56,9.56,3000,7.20
NYSE,XKK,2007-08-10,9.82,9.82,9.60,9.60,2500,7.23
NYSE,XKK,2007-08-09,9.83,9.87,9.82,9.82,4500,7.40
NYSE,XKK,2007-08-08,9.45,9.90,9.45,9.66,6000,7.28
NYSE,XKK,2007-08-07,9.25,9.50,9.25,9.40
```

Notice the last record in this file is not complete and spills over to the next block, a common occurrence in HDFS.

g. Go back to the home directory.

```
# cd ~
```

Lab: Using HDFS Commands

About this Lab

Objective: To become familiar with how files are added to and removed from HDFS and how to view files in HDFS.

File locations: /root/devph/labs/Lab2.1

Successful outcome: You will have added and deleted several files and folders in HDFS.

Before you begin: Your HDP 2.3 cluster should be up and running within your VM.

Related lesson: ***The Hadoop Distributed File System (HDFS)***

Lab Steps

1) View the hdfs dfs Command

a. Open a Terminal in your VM and type "ssh sandbox".
b. Enter the following command to view the usage of hdfs dfs:

```
# hdfs dfs
```

c. Notice that the usage contains options for performing filesystem tasks in HDFS, like copying files from a local folder into HDFS, retrieving a file from HDFS, copying and moving files around, and making and removing directories. In this lab, you will perform these commands, and many others, to help you become comfortable with working with HDFS.

2) Create a Directory in HDFS

a. Enter the following `-ls` command to view the contents of the user's root directory in HDFS, which is `/user/root`

```
# hdfs dfs -ls
```

You do not have any files in `/user/root` yet, so no output is displayed. Run the `-ls` command again, but this time specify the root HDFS folder:

```
# hdfs dfs -ls /
```

The output should look similar to:

```
Found 10 items
drwxrwxrwx   - yarn   hadoop          0 2014-12-16 19:06 /app-logs
drwxr-xr-x   - hdfs   hdfs            0 2014-12-16 19:13 /apps
drwxr-xr-x   - hdfs   hdfs            0 2014-12-16 19:48 /demo
drwxr-xr-x   - hdfs   hdfs            0 2014-12-16 19:07 /hdp
drwxr-xr-x   - mapred hdfs            0 2014-12-16 19:06 /mapred
drwxr-xr-x   - hdfs   hdfs            0 2014-12-16 19:06 /mr-
history
drwxr-xr-x   - hdfs   hdfs            0 2014-12-16 19:37 /ranger
drwxr-xr-x   - hdfs   hdfs            0 2014-12-16 19:08 /system
drwxrwxrwx   - hdfs   hdfs            0 2014-12-16 19:29 /tmp
drwxr-xr-x   - hdfs   hdfs            0 2015-01-12 05:34 /user
```

Important

Notice how adding the / in the `-ls` command caused the contents of the root folder to display, but leaving off the / showed the contents of `/user/root`, which is the default prefix if you leave off the leading / on any of the hadoop commands (assuming the command is run by the “root” user).

b. Enter the following command to create a directory named test in HDFS:

```
# hdfs dfs -mkdir test
```

c. Verify that the folder was created successfully:

```
# hdfs dfs -ls
Found 1 items
drwxr-xr-x   - root root  0   test
```

d. Create a couple of subdirectories for test:

```
# hdfs dfs -mkdir test/test1
# hdfs dfs -mkdir -p test/test2/test3
```

Notice how the `-p` command can be used to create multiple directories. The second command above will fail if you omit the `-p`.

e. Use the `-ls` command to view the contents of `/user/root`:

```
# hdfs dfs -ls
```

Notice you only see the test directory. To recursively view the contents of a folder, use `-ls -R`:

```
# hdfs dfs -ls -R
```

The output should look like:

```
drwxr-xr-x   - root root  0 test
drwxr-xr-x   - root root  0 test/test1
drwxr-xr-x   - root root  0 test/test2
drwxr-xr-x   - root root  0 test/test2/test3
```

3) Delete a Directory

a. Delete the test2 folder (and recursively, its subcontents) using the `-rm -R` command:

```
# hdfs dfs -rm -R test/test2
```

b. Now run the `-ls -R` command:

```
# hdfs dfs -ls -R
```

The directory structure of the output should look like:

```
.Trash
.Trash/Current
.Trash/Current/user
.Trash/Current/user/root
.Trash/Current/user/root/test
.Trash/Current/user/root/test/test2
.Trash/Current/user/root/test/test2/test3
test
test/test1
```

Note

Notice Hadoop created a .Trash folder for the root user and moved the deleted content there. The .Trash folder empties automatically after a configured amount of time.

4) Upload a File to HDFS

a. Now let's put a file into the test folder. Change directories to `/root/devph/labs/Lab2.1`:

```
# cd /root/devph/labs/Lab2.1/
```

b. Notice this folder contains a file named `data.txt`:

```
# tail data.txt
```

c. Run the following `-put` command to copy data.txt into the test folder in HDFS:

```
# hdfs dfs -put data.txt test/
```

d. Verify that the file is in HDFS by listing the contents of test:

```
# hdfs dfs -ls test
```

The output should look like the following:

```
Found 2 items
-rw-r--r--   1 root root 1529355  test/data.txt
drwxr-xr-x   - root root       0  test/test1
```

5) Copy a File in HDFS

a. Now copy the data.txt file in test to another folder in HDFS using the `-cp` command:

```
# hdfs dfs -cp test/data.txt test/test1/data2.txt
```

b. Verify that the file is in both places by using the `-ls -R` command on test. The output should look like the following:

```
# hdfs dfs -ls -R test
-rw-r--r--   1 root root     1529355 test/data.txt
drwxr-xr-x   - root root           0 test/test1
-rw-r--r--   1 root root     1529355 test/test1/data2.txt
```

c. Now delete the data2.txt file using the `-rm` command:

```
# hdfs dfs -rm test/test1/data2.txt
```

d. Verify that the data2.txt file is in the .Trash folder.

6) View the Contents of a File in HDFS

a. You can use the `-cat` command to view text files in HDFS. Enter the following command to view the contents of data.txt:

```
# hdfs dfs -cat test/data.txt
```

b. You can also use the -tail command to view the end of a file:

```
# hdfs dfs -tail test/data.txt
```

Notice the output this time is only the last 20 rows of data.txt.

7) Getting a File from HDFS

a. See if you can figure out how to use the get command to copy test/data.txt from HDFS into your local `/tmp` folder.

Answer:

```
# hdfs dfs -get test/data.txt /tmp/
# cd /tmp
# ls data*
```

8) The `getmerge` Command

a. Put the file `/root/devph/labs/demos/small_blocks.txt` into the test folder in HDFS. You should now have two files in test: data.txt and small_blocks.txt.

Answer:

```
# hdfs dfs -put /root/devph/labs/demos/small_blocks.txt test/
```

b. Run the following `getmerge` command:

```
# hdfs dfs -getmerge test /tmp/merged.txt
```

c. What did the previous command do? Did you open the file merged.txt to see what happened?

Answer: The two files that were in the test folder in HDFS were merged into a single file and stored on the local file system.

9) Specify the Block Size and Replication Factor

a. Put `/root/devph/labs/Lab2.1/data.txt` into `/user/root` in HDFS, giving it a blocksize of 1,048,576 bytes.

Hint

The blocksize is defined using the dfs.blocksize property on the command line.

Answer:

```
# hdfs dfs -D dfs.blocksize=1048576 -put data.txt data.txt
```

b. Run the following `fsck` command on data.txt:

```
# hdfs fsck /user/root/data.txt
```

c. How many blocks are there for this file?

Answer: The file should be broken down into two blocks.

Result

You should now be comfortable with executing the various HDFS commands, including creating directories, putting files into HDFS, copying files out of HDFS, and deleting files and folders.

Lab: Importing RDBMS Data into HDFS

About this Lab

Objective: Import data from a database into HDFS.

File locations: /root/devph/labs/Lab3.1/

Successful outcome: You will have imported data from MySQL into folders in HDFS.

Before you begin: Your HDP 2.3 cluster should be up and running within your VM.

Related lesson: ***Inputting Data into HDFS***

Lab Steps

1) Create a Table in MySQL

a. If not already done, open a Terminal in your VM and type "ssh sandbox".

b. From the command prompt, change directories to `/root/devph/labs/Lab3.1/`:

```
# cd ~/devph/labs/Lab3.1/
```

c. View the contents of salaries.txt:

```
# tail salaries.txt
```

The comma-separated fields represent a gender, age, salary, and zip code.

d. Notice that there is a `salaries.sql` script that defines a new table in MySQL named salaries. For this script to work, you need to copy salaries.txt to the `/tmp` directory:

```
# cp salaries.txt /tmp
```

e. Now run the `salaries.sql` script using the following command:

```
# mysql test < salaries.sql
```

2) View the Table

a. To verify that the table is populated in MySQL, open the mysql prompt:

```
# mysql
```

b. Switch to the test database, which is where the salaries table was created:

```
mysql> use test;
```

c. Run the show tables command and verify that salaries is defined:

```
mysql> show tables;
+----------------+
| Tables_in_test |
+----------------+
| salaries       |
+----------------+
1 row in set (0.00 sec)
```

d. Select 10 items from the table to verify that it is populated:

```
mysql> select * from salaries limit 10;
+--------+------+--------+---------+----+
| gender | age  | salary | zipcode | id |
+--------+------+--------+---------+----+
| F      |   66 |  41000 |   95103 |  1 |
| M      |   40 |  76000 |   95102 |  2 |
| F      |   58 |  95000 |   95103 |  3 |
| F      |   68 |  60000 |   95105 |  4 |
| M      |   85 |  14000 |   95102 |  5 |
| M      |   14 |      0 |   95105 |  6 |
| M      |   52 |   2000 |   94040 |  7 |
| M      |   67 |  99000 |   94040 |  8 |
| F      |   43 |  11000 |   94041 |  9 |
| F      |   37 |  65000 |   94040 | 10 |
+--------+------+--------+---------+----+
```

e. Exit the mysql prompt:

```
mysql> exit
```

3) Import the Table into HDFS

a. Enter the following Sqoop command (all on a single line), which imports the salaries table in the test database into HDFS:

```
# sqoop import --connect jdbc:mysql://sandbox/test?user=root --
table salaries
```

b. A MapReduce job should start executing, and it may take a couple of minutes for the job to complete.

4) Verify the Import

a. View the contents of your HDFS folder:

```
# hdfs dfs -l
```

b. You should see a new folder named salaries. View its contents:

```
# hdfs dfs -ls salaries
Found 4 items
-rw-r--r--   1 root hdfs   272  salaries/part-m-00000
-rw-r--r--   1 root hdfs   241  salaries/part-m-00001
-rw-r--r--   1 root hdfs   238  salaries/part-m-00002
-rw-r--r--   1 root hdfs   272  salaries/part-m-00003
```

c. Notice there are four new files in the salaries folder named part-m-0000x. Why are there four of these files?

***Answer*:** The MapReduce job that executed the Sqoop command used four `mappers`, so there are four output files (one from each `mapper`).

d. Use the cat command to view the contents of the files. For example:

```
# hdfs dfs -cat salaries/part-m-00000
```

Notice the contents of these files are the rows from the salaries table in MySQL. You have now successfully imported data from a MySQL database into HDFS. Notice that you imported the entire table with all of its columns. Next, you will import only specific columns of a table.

5) Specify Columns to Import

a. Using the --columns argument, write a Sqoop command that imports the salary and age columns (in that order) of the salaries table into a directory in HDFS named salaries2. In addition, set the -m argument to 1 so that the result is a single file.

***Solution*:** The command you enter in the command line will look like this in the terminal window:

```
# sqoop import --connect jdbc:mysql://sandbox/test?user=root --
table salaries
--columns salary,age -m 1 --target-dir salaries2
```

Important
To make it easier to read, following is the same command as above, however we have broken it down into smaller chunks separated by a "\" at the end of the break point in each line. When you see this formatting in the lab, you should type it out as it appears above, and do not enter the \ characters unless specifically instructed to do so.

```
# sqoop import --connect jdbc:mysql://sandbox/test?user=root \
--table salaries \
--columns salary,age \
-m 1 \
--target-dir salaries2
```

b. After the import, verify you only have one part-m file in salaries2:

```
# hdfs dfs -ls salaries2
Found 1 items
-rw-r--r--   1 root hdfs   482  salaries2/part-m-00000
```

c. Verify that the contents of part-m-00000 are only the two columns you specified:

```
# hdfs dfs -cat salaries2/part-m-00000
The last few lines should look like the following:
69000.0,97
91000.0,48
0.0,1
48000.0,45
3000.0,39
14000.0,84
```

6) Importing from a Query

Write a Sqoop import command that imports the rows from salaries in MySQL whose salary column is greater than 90,000.00.

a. Use gender as the `--split-by` value, specify only two mappers, and import the data into the salaries3 folder in HDFS.

Tip
The Sqoop command will look similar to the ones you have been using throughout this lab, except you will use `--query` instead of `--table`. Recall that when you use a `--query` command you must also define a `--split-by` column, or define `-m` to be 1.

Also, do not forget to add `$CONDITIONS` to the `WHERE` clause of your query, as demonstrated earlier in this unit.

***Solution*:**
In the command below, the "\" at the beginning of line 3 just in front of $CONDITIONS" **is** part of the actual command and is required for it to function properly. All other \ symbols in the command should be ignored in the command line.

```
# sqoop import --connect jdbc:mysql://sandbox/test?user=root \
--query "select * from salaries s where s.salary > 90000.00 and \
\$CONDITIONS" \
--split-by gender \
-m 2 \
--target-dir salaries3


--This is how it should appear in the command line

# sqoop import --connect jdbc:mysql://sandbox/test?user=root --
query "select * from salaries s where s.salary > 90000.00 and
\$CONDITIONS" --split-by gender -m 2 --target-dir salaries3
```

b. To verify the result, view the contents of the files in salaries3. You should have only two output files.

```
# hdfs dfs -ls salaries3
```

c. View the contents of part-m-00000 and part-m-00001.

```
# hdfs dfs -cat salaries3/part-m-00000

# hdfs dfs -cat salaries3/part-m-00001
```

Notice that one file contains females, and the other file contains males. Why?

Answer: You used gender as the split-by column, so all records with the same gender are sent to the same mapper.

d. Verify that the output files contain only records whose salary is greater than 90,000.00.

Result

You have imported the data from MySQL to HDFS using the entire table, specific columns, and also using the result of a query.

Lab: Exporting HDFS Data to an RDBMS

About this Lab

Objective: Export data from HDFS into a MySQL table using Sqoop.

File locations: /root/devph/labs/Lab3.2

Successful outcome: The data in salarydata.txt in HDFS will appear in a table in MySQL named salary2.

Before you begin: Your HDP 2.3 cluster should be up and running within your VM.

Related lesson: ***Inputting Data into HDFS***

Lab Steps

1) Put the Data into HDFS

a. If not already done, open a Terminal in your VM and type "ssh sandbox".
b. Change directories to `/root/devph/labs/Lab3.2`:

```
# cd ~/devph/labs/Lab3.2
View the contents of salarydata.txt:

# tail salarydata.txt
M,49,29000,95103
M,44,34000,95102
M,99,25000,94041
F,93,96000,95105
F,75,9000,94040
F,14,0,95102
M,68,1000,94040
F,45,78000,94041
M,40,6000,95103
F,82,5000,95050

Notice the records in this file contain four values separated by
commas, and the values represent a gender, age, salary, and zip
code, respectively.
```

c. **Create a new directory in HDFS named `salarydata`.**

```
# hdfs dfs -mkdir salarydata
```

d. **Put salarydata.txt into the `salarydata` directory in HDFS.**

```
# hdfs dfs -put salarydata.txt salarydata
```

2) Create a Table in the Database

a. **There is a script in the Exporting HDFS Data to an RDBMS lab folder that creates a table in MySQL that matches the records in salarydata.txt. View the SQL script:**

```
# more salaries2.sql
```

b. **Run this script using the following command:**

```
# mysql test < salaries2.sql
```

c. **Verify that the table was created successfully in MySQL:**

```
# mysql
mysql> use test;
mysql> describe salaries2;
+---------+------------+------+-----+---------+-------+
| Field   | Type       | Null | Key | Default | Extra |
+---------+------------+------+-----+---------+-------+
| gender  | varchar(1) | YES  |     | NULL    |       |
| age     | int(11)    | YES  |     | NULL    |       |
| salary  | double     | YES  |     | NULL    |       |
| zipcode | int(11)    | YES  |     | NULL    |       |
+---------+------------+------+-----+---------+-------+
```

d. **Exit the mysql prompt:**

```
mysql> exit
```

3) Export the Data

a. **Run a Sqoop command that exports the `salarydata` folder in HDFS into the salaries2 table in MySQL. At the end of the MapReduce output, you should see a log event stating that 10,000 records were exported.**

```
# sqoop export \
--connect jdbc:mysql://sandbox/test?user=root \
--table salaries2 \
--export-dir salarydata \
--input-fields-terminated-by ","
```

b. Verify it worked by viewing the table’s contents from the mysql prompt. The output should look like the following:

```
# mysql
mysql> use test;
mysql> select * from salaries2 limit 10;
+--------+------+--------+---------+
| gender | age  | salary | zipcode |
+--------+------+--------+---------+
| M      |   57 |  39000 |   95050 |
| F      |   63 |  41000 |   95102 |
| M      |   55 |  99000 |   94040 |
| M      |   51 |  58000 |   95102 |
| M      |   75 |  43000 |   95101 |
| M      |   94 |  11000 |   95051 |
| M      |   28 |   6000 |   94041 |
| M      |   14 |      0 |   95102 |
| M      |    3 |      0 |   95101 |
| M      |   25 |  26000 |   94040 |
+--------+------+--------+---------+
```

c. Exit the mysql prompt.

Result

You have now used Sqoop to export data from HDFS into a database table in MySQL.

Lab: Importing Log Data into HDFS using Flume

About this Lab

Objective: Import data from a log file into HDFS using Flume.

File locations: /root/devph/labs/Lab3.3

Successful outcome: The data in webtraffic.log on sandbox will be streamed into HDFS directory
/user/root/flumedata/.

Before you begin: Your HDP 2.3 cluster should be up and running within your VM.

Related lesson: ***Flume***

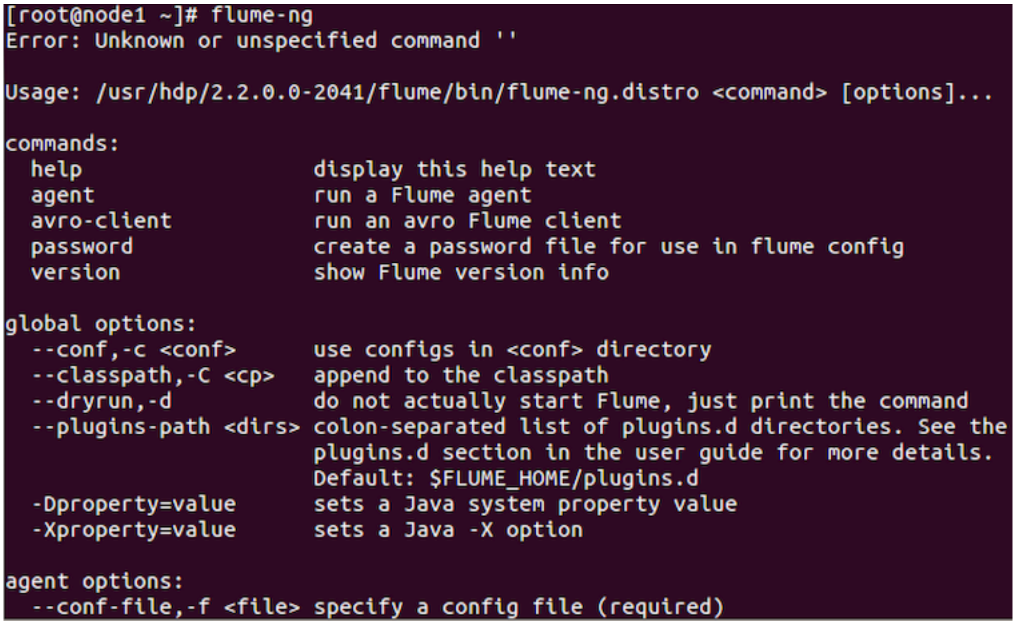
Lab Steps

1) Verify Flume is installed on your cluster.

a. If not already done, open a Terminal in your VM and type "ssh sandbox".
b. Type the following command from prompt:

```
# flume-ng
```

c. The command should return the following usage instructions:



```
[root@node1 ~]# flume-ng
Error: Unknown or unspecified command ''

Usage: /usr/hdp/2.2.0.0-2041/flume/bin/flume-ng.distro <command> [options]...

commands:
  help                      display this help text
  agent                     run a Flume agent
  avro-client               run an avro Flume client
  password                  create a password file for use in flume config
  version                   show Flume version info

global options:
  --conf,-c <conf>          use configs in <conf> directory
  --classpath,-C <cp>       append to the classpath
  --dryrun,-d               do not actually start Flume, just print the command
  --plugins-path <dirs>     colon-separated list of plugins.d directories. See the
                            plugins.d section in the user guide for more details.
                            Default: $FLUME_HOME/plugins.d
  -Dproperty=value          sets a Java system property value
  -Xproperty=value          sets a Java -X option

agent options:
  --conf-file,-f <file>  specify a config file (required)
```

2) View the contents of the webtraffic.log file

a. Change directories to /root/devph/labs/Labs3.3.

```
# cd /root/devph/labs/Lab3.3
```

b. View the contents of the webtraffic.log:

```
# less webtraffic.log
```

3) Press q to quit viewing the log.

a. A partial agent configuration has been written for you, view the file using the less command.

```
# less logagent.conf
```

b. From the VM desktop, start gedit.

c. Click Open and navigate to /root/devph/labs/Lab3.3

d. Modify this file replacing strings starting with "REPLACE-WITH-" with their appropriate value. HINT: Search for "HDFS Sink" at https://flume.apache.org/FlumeUserGuide.html.

```
agent.sources = weblog
agent.channels = memoryChannel
agent.sinks = mycluster

## Sources
#########################################################
agent.sources.weblog.type = exec
agent.sources.weblog.command = tail -F REPLACE-WITH-PATH2-
webtraffic.log-FILE
agent.sources.weblog.batchSize = 1
agent.sources.weblog.channels = REPLACE-WITH-CHANNEL-NAME

## Channels
########################################################
agent.channels.memoryChannel.type = memory
agent.channels.memoryChannel.capacity = 100
agent.channels.memoryChannel.transactionCapacity = 100

## Sinks
###########################################################
agent.sinks.mycluster.type = REPLACE-WITH-CLUSTER-TYPE
agent.sinks.mycluster.hdfs.path=/user/root/flumedata
agent.sinks.mycluster.channel = REPLACE-WITH-CHANNEL-NAME
```

e. Click Save to save the logagent configuration.
f. From the terminal window, start the logagent from /root/devph/labs/Lab3.3.

```
# flume-ng agent -n agent -f logagent.conf &
```

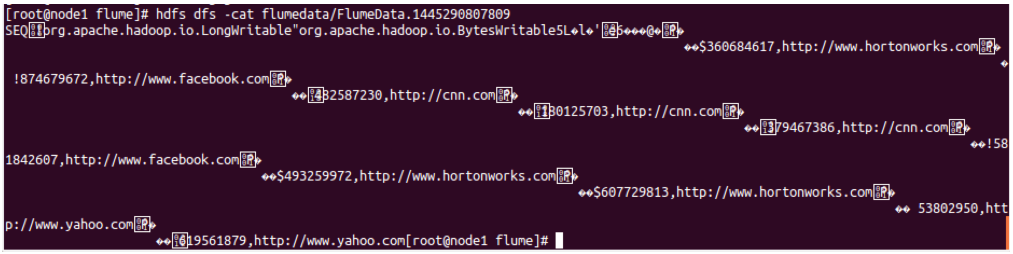
g. Hit Enter to get a command prompt back.
h. The sink for logagent is the HDFS folder /user/root/flumedata. Verify that this folder does now exists.

```
# hdfs dfs -ls flumedata/
```

i. View the contents of the streamed data in HDFS in FlumeData.<########>.

```
# hdfs dfs -cat flumedata/FlumeData.<#######>
```

The flumedata/FlumeData.<sequence> file should display the webtraffic as shown below:



```
[root@node1 flume]# hdfs dfs -cat flumedata/FlumeData.1445290807809
SEQ org.apache.hadoop.io.LongWritable"org.apache.hadoop.io.BytesWritable5L�l�'e5���@��
                                                                    ��$360684617,http://www.hortonworks.com�
                                                                                                          �
 !874679672,http://www.facebook.com�
                                   ��482587230,http://cnn.com�
                                                              ��130125703,http://cnn.com�
                                                                                         ��379467386,http://cnn.com�
                                                                                                                    ��!58
1842607,http://www.facebook.com�
                                ��$493259972,http://www.hortonworks.com�
                                                                        ��$607729813,http://www.hortonworks.com�
                                                                                                                ��  53802950,htt
p://www.yahoo.com�
                  ��619561879,http://www.yahoo.com[root@node1 flume]# 
```

Notice the format difference.

Result

Successful import of data from Flume into HDFS.

Demonstration: Understanding MapReduce

About this Demonstration

Objective:	To understand how MapReduce works.
During this Demonstration:	Watch as your instructor performs the following steps.
Related lesson:	***The MapReduce Framework***

Demonstration Steps

1) Put the File into HDFS

```
# hdfs dfs -put constitution.txt
```

2) Run the WordCount Job

a. The following command runs a wordcount job on the constitution.txt and writes the output to `wordcount_output`:

```
# yarn jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-
mapreduce-examples.jar wordcount constitution.txt wordcount_output
```

b. Notice that a MapReduce job gets submitted to the cluster. Wait for the job to complete.

3) View the Results

a. View the contents of the `wordcount_output` folder:

```
# hdfs dfs -ls wordcount_output
Found 2 items
-rw-r--r-- 1 root root      0 wordcount_output/_SUCCESS
-rw-r--r-- 1 root root  17049 wordcount_output/part-r-00000
```

b. Why is there one part-`r` file in this directory?

____

Answer: The job only used one reducer.

c. What does the "r" in the filename stand for? ___________________________

Answer: The "r" stands for "reducer."

d. View the contents of `part-r-00000`:

```
# hdfs dfs -cat wordcount_output/part-r-00000
```

e. Why are the words sorted alphabetically? ___________________________

   Answer: The key in this MapReduce job is the word, and keys are sorted during the shuffle/sort phase.

f. What was the key output by the WordCount reducer? __________

   Answer: The reducer's output key was each word.

g. What was the value output by the WordCount reducer? ____________

   Answer: The value output by the reducer was the sum of the 1s, which is the number of occurrences of the word in the document.

h. Based on the output of the reducer, what do you think the mapper output would be as `key/value` pairs? ____________

   Answer: The mapper outputs each word as a key and the number 1 as each value.

Lab: Running a MapReduce Job

About this Lab

Objective:	Run a Java MapReduce job.
File locations:	/root/devph/labs/Lab4.1
Successful outcome:	You will see the results of the Inverted Index job in the inverted/output folder in HDFS.
Before you begin:	Your HDP 2.3 cluster should be up and running within your VM.
Related lesson:	***The MapReduce Framework***

Lab Steps

1) Put the Data into HDFS

a. If not already done, open a Terminal in your VM and type "ssh sandbox".

b. The MapReduce job you are going to execute is an Inverted Index application, one of the very first use cases for MapReduce. Open a command prompt and change directories to `/root/labs/Lab4.1`:

```
# cd ~/devph/labs/Lab4.1
```

c. Use `more` to view the contents of the file hortonworks.txt.

```
# more hortonworks.txt
```

Each line looks like:

```
http://hortonworks.com/,hadoop,webinars,articles,download,enterpri
se,team,reliability
```

Each line of text consists of a Web page URL, followed by a comma-separated list of keywords found on that page.

d. Make a new folder in HDFS named inverted/input:

```
# hdfs dfs -mkdir -p inverted/input
```

e. Put hortonworks.txt into HDFS into the `inverted/input/` folder. This file will be the input to the MapReduce job.

```
# hdfs dfs –put hortonworks.txt inverted/input/
```

2) Run the Inverted Index Job

a. From the `/root/devph/labs/Lab4.1` folder, enter the following command (all on a single line):

```
# yarn jar invertedindex.jar inverted.IndexInverterJob
inverted/input inverted/output
```

b. Wait for the MapReduce job to execute. The final output should look like:

```
File Input Format Counters
Bytes Read=1126
File Output Format Counters
Bytes Written=2997
```

3) View the Results

a. List the contents of the inverted/output folder.

```
# hdfs dfs -ls inverted/output
```

How many reducers did this job use?
How can you determine this from the contents of inverted/output?

Answer: The job used one reducer, which you can determine by the existence of only one part-r-n file in the output directory.

b. Use the cat command to view the contents of `inverted/output/part-r-00000`. The file should look like:

```
# hdfs dfs -cat inverted/output/part-r-00000
abouthttp://hortonworks.com/about-us/,
apache
      http://hortonworks.com/products/hortonworksdataplatform/,
http://hortonworks.com/about-us/,
articles
      http://hortonworks.com/community/,http://hortonworks.com/,
...
```

4) Specify the Number of Reducers

a. Try running the job again, but this time specify the number of reducers to be three:

```
# yarn jar invertedindex.jar inverted.IndexInverterJob
-D mapreduce.job.reduces=3 inverted/input inverted/output
```

b. View the contents of inverted/output. Notice there are three `part-r` files:

```
# hdfs dfs -ls inverted/output
Found 3 items
1 root hdfs       1221 inverted/output/part-r-00000
1 root hdfs        977 inverted/output/part-r-00001
1 root hdfs        799 inverted/output/part-r-00002
```

c. View the contents of the three files. How did the MapReduce framework determine which `<key,value>` pair to send to which reducer?

Answer: `<key,value>` pairs are sent to the reducer based on the hashing of the key and using the remainder of dividing by the number of reducers.

Result

You have now executed a Java MapReduce job from the command line that takes an input text file and outputs the inverted indexes of the lines of text. This common task is what Web search engines like Google and Yahoo! use to determine the pages associated with search terms.

Demonstration: Understanding Pig

About this Demonstration

Objective: To understand Pig scripts and relations.

During this Demonstration: Watch as your instructor performs the following steps.

Related lesson: ***Introduction to Pig***

Demonstration Steps

1) Start the Grunt Shell

a. If not already done, open a Terminal in your VM and type "ssh sandbox".

b. Review the contents of the file pigdemo.txt located in /root/devph/labs/demos.

```
# more /root/devph/labs/demos/pigdemo.txt
```

c. Start the Grunt shell:

```
# pig
```

d. Notice that the output includes where the logging for your Pig session will go as well as a statement about connecting to your Hadoop filesystem:

```
[main] INFO  org.apache.pig.Main - Logging error messages to:
/root/devph/labs/demos/pig_1377892197767.log
[main] INFO  org.apache.pig.backend.hadoop.executionengine.
HExecutionEngine - Connecting to hadoop file system at:
hdfs://sandbox.hortonworks.com:8020
```

2) Make a New Directory

a. Notice you can run HDFS commands easily from the Grunt shell. For example, run the `ls` command:

```
grunt> ls
```

b. Make a new directory named demos:

```
grunt> mkdir demos
```

c. Use copyFromLocal to copy the pigdemo.txt file into the demos folder:

```
grunt> copyFromLocal /root/devph/labs/demos/pigdemo.txt demos/
```

d. Verify the file was uploaded successfully:

```
grunt> ls demos
hdfs://sandbox.hortonworks.com:8020/user/root/demos/pigdemo.txt<r
1>89
```

e. Change the present working directory to demos:

```
grunt> cd demos
grunt> pwd
hdfs://sandbox.hortonworks.com:8020/user/root/demos
```

f. View the contents using the cat command:

```
grunt> cat pigdemo.txt
SD      Rich
NV      Barry
CO      George
CA      Ulf
IL      Danielle
OH      Tom
CA      manish
CA      Brian
CO      Mark
```

3) Define a Relation

a. Define the employees relation, using a schema:

```
grunt> employees = LOAD 'pigdemo.txt' AS (state, name);
```

b. Demonstrate the describe command, which describes what a relation looks like:
grunt> describe employees;

```
employees: {state: bytearray,name: bytearray}
```

Note
Fields have a data type, and we will discuss data types later in this unit. Notice that the default data type of a field (if you do not specify one) is bytearray.

c. Let’s view the records in the employees relation:

```
grunt> DUMP employees;
```

Notice this requires a MapReduce job to execute, and the result is a collection of tuples:

```
(SD,Rich)
(NV,Barry)
(CO,George)
(CA,Ulf)
(IL,Danielle)
(OH,Tom)
(CA,manish)
(CA,Brian)
(CO,Mark)
```

4) Filter the Relation by a Field

a. Let’s filter the employees whose state field equals CA:

```
grunt> ca_only = FILTER employees BY (state=='CA');
grunt> DUMP ca_only;
```

b. The output is still tuples, but only the records that match the filter appear:

```
(CA,Ulf)
(CA,manish)
(CA,Brian)
```

5) Create a Group

a. Define a relation that groups the employees by the state field:

```
grunt> emp_group = GROUP employees BY state;
```

b. Bags represent groups in Pig. A bag is an unordered collection of tuples:

```
grunt> describe emp_group;
emp_group: {group: bytearray,employees: {(state: bytearray,name: bytearray)}}
```

c. All records with the same state will be grouped together, as shown by the output of the `emp_group` relation:

```
grunt> DUMP emp_group;
```

The output is:

```
(CA,{(CA,Ulf),(CA,manish),(CA,Brian)})
(CO,{(CO,George),(CO,Mark)})
(IL,{(IL,Danielle)})
(NV,{(NV,Barry)})
(OH,{(OH,Tom)})
(SD,{(SD,Rich)})
```

Note
Tuples are displayed in parentheses. Curly braces represent bags.

6) The `STORE` Command

a. The `DUMP` command dumps the contents of a relation to the console. The `STORE` command sends the output to a folder in HDFS. For example:

```
grunt> STORE emp_group INTO 'emp_group';
```

Notice at the end of the MapReduce job that no records are output to the console.

b. Verify that a new folder is created:

```
grunt> ls
hdfs://sandbox.hortonworks.com:8020/user/root/demos/emp_group<dir>
hdfs://sandbox.hortonworks.com:8020/user/root/demos/pigdemo.txt<r
1>89
```

c. View the contents of the output file:

```
grunt> cat emp_group/part-r-00000
CA      {(CA,Ulf),(CA,manish),(CA,Brian)}
CO      {(CO,George),(CO,Mark)}
IL      {(IL,Danielle)}
NV      {(NV,Barry)}
OH      {(OH,Tom)}
SD      {(SD,Rich)}
```

Notice that the fields of the records (which in this case is the state field followed by a bag) are separated by a tab character, which is the default delimiter in Pig. Use the PigStorage object to specify a different delimiter:

```
grunt> STORE emp_group INTO 'emp_group_csv' USING PigStorage(',');

To view the results:

grunt > ls

grunt > cat emp_group_csv/part-r-00000
```

7) Show All Aliases

a. The aliases command shows a list of currently defined aliases:

```
grunt> aliases;
aliases: [ca_only, emp_group, employees]
```

There will be a couple of additional numeric aliases created by the system for internal use. Please ignore them.

8) Monitor the Pig Jobs

a. Point your browser to the JobHistory UI at `http://sandbox:19888/`.

b. View the list of jobs, which should contain the MapReduce jobs that were executed from your Pig Latin code in the Grunt shell.

c. Notice you can view the log files of the ApplicationMaster and also each map and reduce task.

Note

Three commands trigger a logical plan to be converted to a physical plan and execute as a MapReduce job: `STORE, DUMP`, and `ILLUSTRATE`.

Lab: Getting Started with Pig

About this Lab

Objective: Use Pig to navigate through HDFS and explore a dataset.

File locations: /root/devph/labs/Lab5.1

Successful outcome: You will have a couple of Pig programs that load the White House visitors' data, with and without a schema, and store the output of a relation into a folder in HDFS.

Before you begin: Your HDP 2.3 cluster should be up and running within your VM.

Related lesson: ***Introduction to Pig***

Lab Steps

1) View the Raw Data

a. If not already done, open a Terminal in your VM and type "ssh sandbox".

b. Change directories to the `/root/devph/labs/Lab5.1` folder:

```
# cd ~/devph/labs/Lab5.1
```

c. Unzip the archive in the `/root/devph/labs/Lab5.1` folder, which contains a file named `whitehouse_visits.txt` that is quite large:

```
# unzip whitehouse_visits.zip
```

d. View the contents of this file:

```
# tail whitehouse_visits.txt
```

This publicly available data contains records of visitors to the White House in Washington, D.C.

2) Load the Data into HDFS

a. Start the Grunt shell:

```
# pig
```

b. From the Grunt shell, make a new directory in HDFS named `whitehouse`:

```
grunt> mkdir whitehouse
```

c. Use the `copyFromLocal` command in the Grunt shell to copy the `whitehouse_visits.txt` file to the whitehouse folder in HDFS, renaming the file visits.txt. (Be sure to enter this command on a single line):

```
grunt> copyFromLocal /root/devph/labs/Lab5.1/whitehouse_visits.txt
whitehouse/visits.txt
```

d. Use the `ls` command to verify that the file was uploaded successfully:

```
grunt> ls whitehouse
hdfs://sandbox.hortonworks.com:8020/user/root/whitehouse/visits.tx
t<r 1>183292235
```

3) Define a Relation

a. You will use the TextLoader to load the `visits.txt` file.

Note

TextLoader simply creates a tuple for each line of text, and it uses a single chararray field that contains the entire line. It allows you to load lines of text and not worry about the format or schema yet.

Define the following `LOAD` relation:

```
grunt> A = LOAD '/user/root/whitehouse/' USING TextLoader();
```

b. Use `DESCRIBE` to notice that `A` does not have a schema:

```
grunt> DESCRIBE A;
Schema for A unknown.
```

c. We want to get a sense of what this data looks like. Use the `LIMIT` operator to define a new relation named `A_limit` that is limited to 10 records of `A`.

```
grunt> A_limit = LIMIT A 10;
```

d. Use the `DUMP` operator to view the `A_limit` relation.

   Each row in the output will look similar to the following and should be 10 arbitrary rows from visits.txt:

```
grunt> DUMP A_limit;

(WHITLEY,KRISTY,J,U45880,,VA,,,,,10/7/2010 5:51,10/9/2010
10:30,10/9/2010 23:59,,294,B3,WIN,10/7/2010
5:51,B3,OFFICE,VISITORS,WH,RES,OFFICE,VISITORS,GROUP TOUR
,1/28/2011,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,)
```

4) Define a Schema

a. Load the White House data again, but this time use the PigStorage loader and also define a partial schema:

```
grunt> B = LOAD '/user/root/whitehouse/visits.txt' USING
PigStorage(',') AS (
   lname:chararray,
   fname:chararray,
   mname:chararray,
   id:chararray,
   status:chararray,
   state:chararray,
   arrival:chararray
);
```

b. Use the DESCRIBE command to view the schema:

```
grunt> describe B;
B: {lname: chararray,fname: chararray,mname: chararray,id:
chararray,status: chararray,state: chararray,arrival: chararray}
```

5) The `STORE` Command

a. Enter the following `STORE` command, which stores the `B` relation into a folder named `whouse_tab` and separates the fields of each record with tabs:

```
grunt> store B into 'whouse_tab' using PigStorage('\t');
```

b. Verify that the `whouse_tab` folder was created:

```
grunt> ls whouse_tab
```

You should see two map output files.

c. View one of the output files to verify they contain the `B` relation in a tab-delimited format:

```
grunt> fs -tail whouse_tab/part-m-00000
```

d. Each record should contain seven fields. What happened to the rest of the fields from the raw data that was loaded from `whitehouse/visits.txt`?

Answer: They were simply ignored when each record was read in from HDFS.

6) Use a Different Storer

a. In the previous step, you stored a relation using PigStorage with a tab delimiter. Enter the following command, which stores the same relation but in a `JSON` format:

```
grunt> store B into 'whouse_json' using JsonStorage();
```

b. Verify that the `whouse_json` folder was created:

```
grunt> ls whouse_json
```

c. View one of the output files:

```
grunt> fs -tail whouse_json/part-m-00000
```

Notice that the schema you defined for the B relation was used to create the format of each `JSON` entry:

Result

You have now seen how to execute some basic Pig commands, load data into a relation, and store a relation into a folder in HDFS using different formats.

Lab: Exploring Data with Pig

About this Lab

Objective: Use Pig to navigate through HDFS and explore a dataset.

File locations: whitehouse/visits.txt in HDFS

Successful outcome: You will have written several Pig scripts that analyze and query the White House visitors' data, including a list of people who visited the President.

Before you begin: At a minimum, complete steps 1 and 2 of the Getting Started with Pig
lab.

Related lesson: ***Introduction to Pig***

Lab Steps

1) Load the White House Visitor Data

a. If not already done, open a Terminal in your VM and type "ssh sandbox".
b. You will use the TextLoader to load the visits.txt file. From the Pig Grunt shell, define the following `LOAD` relation:

```
# pig

grunt> A = LOAD '/user/root/whitehouse/' USING TextLoader();
```

2) Count the Number of Lines

a. Define a new relation named `B` that is a group of all the records in `A`:

```
grunt> B = GROUP A ALL;
```

b. Use `DESCRIBE` to view the schema of `B`.

```
grunt> DESCRIBE B;
```

What is the datatype of the group field? ____________________
Where did this datatype come from? _______________________

Answer: The group field is a chararray because it is just the string "all" and is a result of performing a `GROUP ALL`.

Why does the `A` field of `B` contain no schema? ____________________

Answer: The `A` field of `B` contains no schema because the `A` relation has no schema.

How many groups are in the relation `B`? ____________

Answer: The `B` relation can only contain one group because it a grouping of every single record. Note that the `A` field is a bag, and `A` will contain any number of tuples.

c. The `A` field of the `B` tuple is a bag of all of the records in visits.txt. Use the `COUNT` function on this bag to determine how many lines of text are in visits.txt:

```
grunt> A_count = FOREACH B GENERATE 'rowcount', COUNT(A);
```

Note
The ‘rowcount’ string in the `FOREACH` statement is simply to demonstrate that you can have constant values in a `GENERATE` clause. It is certainly not necessary; it just makes the output nicer to read.

d. Use DUMP on A_count to view the result. The output should look like:

```
grunt> DUMP A_count;

(rowcount,447598)
```

We can now conclude that there are 447,598 rows of text in visits.txt.

3) Analyze the Data’s Contents

a. We now know how many records are in the data, but we still do not have a clear picture of what the records look like. Let’s start by looking at the fields of each record. Load the data using `PigStorage(‘,’)` instead of `TextLoader()`:

```
grunt> visits = LOAD '/user/root/whitehouse/' USING
PigStorage(',');
```

This will split up the fields by comma.

b. Use a `FOREACH...GENERATE` command to define a relation that is a projection of the first 10 fields of the visits relation.

```
grunt> firstten = FOREACH visits GENERATE $0..$9;
```

c. Use `LIMIT` to display only 50 records then `DUMP` the result.
The output should be 50 tuples that represent the first 10 fields of visits:

```
grunt> firstten_limit = LIMIT firstten 50;
grunt> DUMP firstten_limit;

(PARK,ANNE,C,U51510,0,VA,10/24/2010 14:53,B0402,,)
(PARK,RYAN,C,U51510,0,VA,10/24/2010 14:53,B0402,,)
(PARK,MAGGIE,E,U51510,0,VA,10/24/2010 14:53,B0402,,)
(PARK,SIDNEY,R,U51510,0,VA,10/24/2010 14:53,B0402,,)
(RYAN,MARGUERITE,,U82926,0,VA,2/13/2011 17:14,B0402,,)
(WILE,DAVID,J,U44328,,VA,,,,)
(YANG,EILENE,D,U82921,,VA,,,,)
(ADAMS,SCHUYLER,N,U51772,,VA,,,,)
(ADAMS,CHRISTINE,M,U51772,,VA,,,,)
(BERRY,STACEY,,U49494,79029,VA,10/15/2010 12:24,D0101,10/15/2010
14:06,D1S)
```

Note
Because `LIMIT` uses an arbitrary sample of the data, your output will be different names but the format should look similar.
Notice from the output that the first three fields are the person’s name. The next seven fields are a unique ID, badge number, access type, time of arrival, post of arrival, time of departure, and post of departure.

4) Locate the POTUS (President of the United States of America)

a. There are 26 fields in each record, and one of them represents the visitee (the person being visited in the White House). Your goal now is to locate this column and determine who has visited the President of the United States. Define a relation that is a projection of the last seven fields ($19 to $25) of visits. Use LIMIT to only output 500 records. The output should look like:

```
grunt> lastfields = FOREACH visits GENERATE $19..$25;
grunt> lastfields_limit = LIMIT lastfields 500;
grunt> DUMP lastfields_limit;

(OFFICE,VISITORS,WH,RESIDENCE,OFFICE,VISITORS,HOLIDAY OPEN HOUSE/)
(OFFICE,VISITORS,WH,RESIDENCE,OFFICE,VISITORS,HOLIDAY OPEN
HOUSES/)
(OFFICE,VISITORS,WH,RESIDENCE,OFFICE,VISITORS,HOLIDAY OPEN HOUSE/)
(CARNEY,FRANCIS,WH,WW,ALAM,SYED,WW TOUR)
(CARNEY,FRANCIS,WH,WW,ALAM,SYED,WW TOUR)
(CARNEY,FRANCIS,WH,WW,ALAM,SYED,WW TOUR)
(CHANDLER,DANIEL,NEOB,6104,AGCAOILI,KARL,)
```

b. It is not necessarily obvious from the output, but field $19 in the visits relation represents the visitee. Even though you selected 500 records in the previous step, you may or may not see POTUS in the output above. (The White House has thousands of visitors each day, but only a few meet the President.)

c. Use `FILTER` to define a relation that only contains records of visits where field $19 matches POTUS. Limit the output to 500 records.

   The output should include only visitors who met with the President. For example:

```
grunt> potus = FILTER visits BY $19 MATCHES 'POTUS';
grunt> potus_limit = LIMIT potus 500;
grunt> DUMP potus_limit;

(ARGOW,KEITH,A,U83268,,VA,,,,,2/14/2011 18:42,2/16/2011
16:00,2/16/2011 23:59,,154,LC,WIN,2/14/2011
18:42,LC,POTUS,,WH,EAST ROOM,THOMPSON,MARGRETTE,,AMERICA'S GREAT
OUTDOORS ROLLOUT EVENT
,5/27/2011,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,)
(AYERS,JOHNATHAN,T,U84307,,VA,,,,,2/18/2011 19:11,2/25/2011
17:00,2/25/2011 23:59,,619,SL,WIN,2/18/2011
19:11,SL,POTUS,,WH,STATE FLOO,GALLAGHER,CLARE,,RECEPTION
,5/27/2011,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,)
```

5) Count the POTUS Visitors

a. Let's discover how many people have visited the President. To do this, we need to count the number of records in visits where field $19 matches `POTUS`. See if you can write a Pig script to accomplish this. Use the `potus` relation from the previous step as a starting point. You will need to use `GROUP ALL` and then a `FOREACH` projection that uses the `COUNT` function.

   If successful, you should get 21,819 as the number of visitors to the White House who visited the President.

b. ***Solution***:

```
grunt> potus = FILTER visits BY $19 MATCHES 'POTUS';
grunt> potus_group = GROUP potus ALL;
grunt> potus_count = FOREACH potus_group GENERATE COUNT(potus);
grunt> DUMP potus_count;
```

6) Finding People Who Visited the President

a. So far you have used `DUMP` to view the results of your Pig scripts. In this step, you will save the output to a file using the `STORE` command.

b. Now `FILTER` the relation by visitors who met with the President:

```
grunt> potus = FILTER visits BY $19 MATCHES 'POTUS';
```

c. Define a projection of the `potus` relationship that contains the name and time of arrival of the visitor:

```
grunt> potus_details = FOREACH potus GENERATE
(chararray) $0 AS lname:chararray,
(chararray) $1 AS fname:chararray,
(chararray) $6 AS arrival_time:chararray,
(chararray) $19 AS visitee:chararray;
```

d. Order the `potus_details` projection by last name:

```
grunt> potus_details_ordered = ORDER potus_details BY lname ASC;
```

e. Store the records of `potus_details_ordered` into a folder named `potus` and using a comma delimiter:

```
grunt> STORE potus_details_ordered INTO 'potus' USING
PigStorage(',');
```

f. View the contents of the `potus` folder:

```
grunt> ls potus
hdfs://sandbox.hortonworks.com:8020/user/root/potus/_SUCCESS<r 1>0
hdfs://sandbox.hortonworks.com:8020/user/root/potus/part-r-00000<r
1>501378
```

g. Notice that there is a single output file, so the Pig job was executed with one reducer. View the contents of the output file using `cat`:

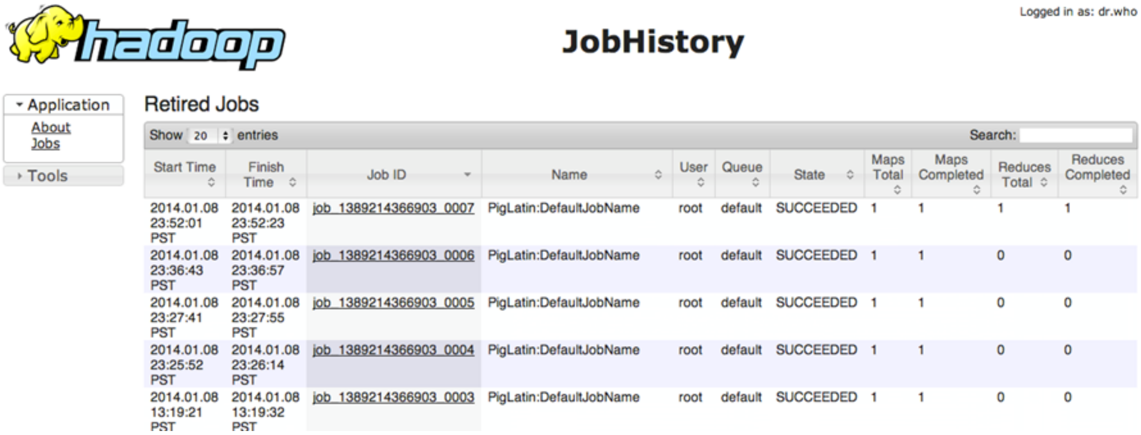
```
grunt> cat potus/part-r-00000
```

The output should be in a comma-delimited format and should contain the last name, first name, time of arrival (if available), and the string `POTUS`:

```
CLINTON,WILLIAM,,POTUS
CLINTON,HILLARY,,POTUS
CLINTON,HILLARY,,POTUS
CLINTON,HILLARY,,POTUS
CLONAN,JEANETTE,,POTUS
CLOOBECK,STEPHEN,,POTUS
CLOOBECK,CHANTAL,,POTUS
CLOOBECK,STEPHEN,,POTUS
CLOONEY,GEORGE,10/12/2010 14:47,POTUS
```

7) View the Pig Log Files

a. Each time you executed a `DUMP` or `STORE` command, a MapReduce job is executed on your cluster.
You can view the log files of these jobs in the JobHistory UI. Point your browser to `http://sandbox:19888/`:



Retired Jobs

Show 20 entries Search:

Start Time	Finish Time	Job ID	Name	User	Queue	State	Maps Total	Maps Completed	Reduces Total	Reduces Completed
2014.01.08 23:52:01 PST	2014.01.08 23:52:23 PST	job_1389214366903_0007	PigLatin:DefaultJobName	root	default	SUCCEEDED	1	1	1	1
2014.01.08 23:36:43 PST	2014.01.08 23:36:57 PST	job_1389214366903_0006	PigLatin:DefaultJobName	root	default	SUCCEEDED	1	1	0	0
2014.01.08 23:27:41 PST	2014.01.08 23:27:55 PST	job_1389214366903_0005	PigLatin:DefaultJobName	root	default	SUCCEEDED	1	1	0	0
2014.01.08 23:25:52 PST	2014.01.08 23:26:14 PST	job_1389214366903_0004	PigLatin:DefaultJobName	root	default	SUCCEEDED	1	1	0	0
2014.01.08 13:19:21 PST	2014.01.08 13:19:32 PST	job_1389214366903_0003	PigLatin:DefaultJobName	root	default	SUCCEEDED	1	1	0	0

b. Click on the job's ID to view the details of the job and its log files.

Result

You have written several Pig scripts to analyze and query the data in the White House visitors' log. You should now be comfortable with writing Pig scripts with the Grunt shell and using common Pig commands like `LOAD`, `GROUP`, `FOREACH`, `FILTER`, `LIMIT`, `DUMP`, and `STORE`.

Lab: Splitting a Dataset

About this Lab

Objective:	Research the White House visitor data and look for members of Congress.
File locations:	n/a
Successful outcome:	Two folders in HDFS, congress and not_congress, containing a split of the White House visitor data.
Before you begin:	You should have the White House visitor data in HDFS in /user/root/whitehouse/visits.txt.
Related lesson:	***Advanced Pig Programming***

Lab Steps

1) Explore the Comments Field

a. If not already done, open a Terminal in your VM and type "ssh sandbox".

b. In this step, you will explore the comments field of the White House visitor data. From the Pig Grunt shell, start by loading visits.txt:

```
# pig

grunt> cd whitehouse
grunt> visits = LOAD 'visits.txt' USING PigStorage(',');
```

c. Field $25 is the comments. Filter out all records where field $25 is null:

```
grunt> not_null_25 = FILTER visits BY ($25 IS NOT NULL);
```

d. Now define a new relation that is a projection of only column $25:

```
grunt> comments = FOREACH not_null_25 GENERATE $25 AS comment;
```

e. View the schema of comments and make sure you understand how this relation ended up as a tuple with one field:

```
grunt> describe comments;
comments: {comment: bytearray}
```

2) Test the Relation

a. A common Pig task is to test a relation to make sure it is consistent with what you are intending it to be. But using `DUMP` on a big data relation might take too long or not be practical, so define a `SAMPLE` of comments:

```
grunt> comments_sample = SAMPLE comments 0.001;
```

b. Now `DUMP` the `comments_sample` relation. The output should be non-null comments about visitors to the White House, similar to:

```
grunt> DUMP comments_sample;

(ATTENDEES VISITING FOR A MEETING)
(FORUM ON IT MANAGEMENT REFORM/)
(FORUM ON IT MANAGEMENT REFORM/)
(HEALTH REFORM MEETING)
(DRIVER TO REMAIN WITH VEHICLE)
```

3) Count the Number of Comments

a. The comments relation represents all non-null comments from visits.txt. Write Pig statements that output the number of records in the comments relation. The correct result is 222,839 records.

***Solution*:**

```
comments_all = GROUP comments ALL;
comments_count = FOREACH comments_all GENERATE
COUNT(comments);
DUMP comments_count;
```

4) Split the Dataset

Note

Our end goal is find visitors to the White House who are also members of Congress. We could run our MapReduce job on the entire visits.txt dataset, but it is common in Hadoop to split data into smaller input files for specific tasks, which can greatly improve the performance of your MapReduce applications. In this step, you will split visits.txt into two separate datasets.

a. In this step, you will split visits.txt into two datasets: those that contain "`CONGRESS`" in the comments field, and those that do not.

b. Use the `SPLIT` command to split the visits relation into two new relations named congress and `not_congress`:

```
grunt> SPLIT visits INTO congress IF($25 MATCHES
'.* CONGRESS .*'), not_congress IF (NOT($25 MATCHES
'.* CONGRESS .*'));
```

c. Store the congress relation into a folder named ‘`congress`’.

```
grunt> STORE congress INTO 'congress';
```

d. Similarly, `STORE` the `not_congress` relation in a folder named ‘`not_congress`’.

```
grunt> STORE not_congress INTO 'not_congress';
```

e. View the output folders using `ls`. The file sizes should be equivalent to the following:

```
grunt> ls congress
hdfs://sandbox.hortonworks.com:8020/user/root/whitehouse/congress/
_SUCCESS<r 1>0
hdfs://sandbox.hortonworks.com:8020/user/root/whitehouse/congress/
part-m-00000<r 1>45618
hdfs://sandbox.hortonworks.com:8020/user/root/whitehouse/congress/
part-m-00001<r 1>0
grunt> ls not_congress
hdfs://sandbox.hortonworks.com:8020/user/root/whitehouse/not_congr
ess/_SUCCESS<r 1>0
hdfs://sandbox.hortonworks.com:8020/user/root/whitehouse/not_congr
ess/part-m-00000<r 1>90741587
hdfs://sandbox.hortonworks.com:8020/user/root/whitehouse/not_congr
ess/part-m-00001<r 1>272381
```

f. View one of the output files in congress and make sure the string “`CONGRESS`” appears in the comment field:

```
grunt> cat congress/part-m-00000
```

5) Count the Results

a. Write Pig statements that output the number of records in the congress relation. This will tell us how many visitors to the White House have “`CONGRESS`” in the comments of their visit log. The correct result is 102.

Note
You now have two datasets: one in ‘`congress`,’ with 102 records, and the remaining records in the ‘`not_congress`’ folder. These records are still in their original, raw format.

***Solution*:**

```
grunt> congress_grp = GROUP congress ALL;
grunt> congress_count = FOREACH congress_grp GENERATE
COUNT(congress);
grunt> DUMP congress_count;
```

Result

You have just split ‘visits.txt’ into two datasets, and you have also discovered that 102 visitors to the White House had the word “`CONGRESS`” in their comments field. We will further explore these visitors in the next lab as we perform a join with a dataset containing the names of members of Congress.

Lab: Joining Datasets with Pig

About this Lab

Objective:	Join two datasets in Pig.
File locations:	/root/devph/labs/Lab6.2
Successful outcome:	A file of members of Congress who have visited the White House.
Before you begin:	If you are in the Grunt shell, exit it using the `quit` command. In this lab, you will write a Pig script in a text file.
Related lesson:	Advanced Pig Programming

Lab Steps

1) Upload the Congress Data

a. If not already done, open a Terminal in your VM and type "ssh sandbox".

b. Put the file `/root/devph/labs/Lab6.2/congress.txt` into the `whitehouse` directory in HDFS.

```
# hdfs dfs -put /root/devph/labs/Lab6.2/congress.txt whitehouse/
```

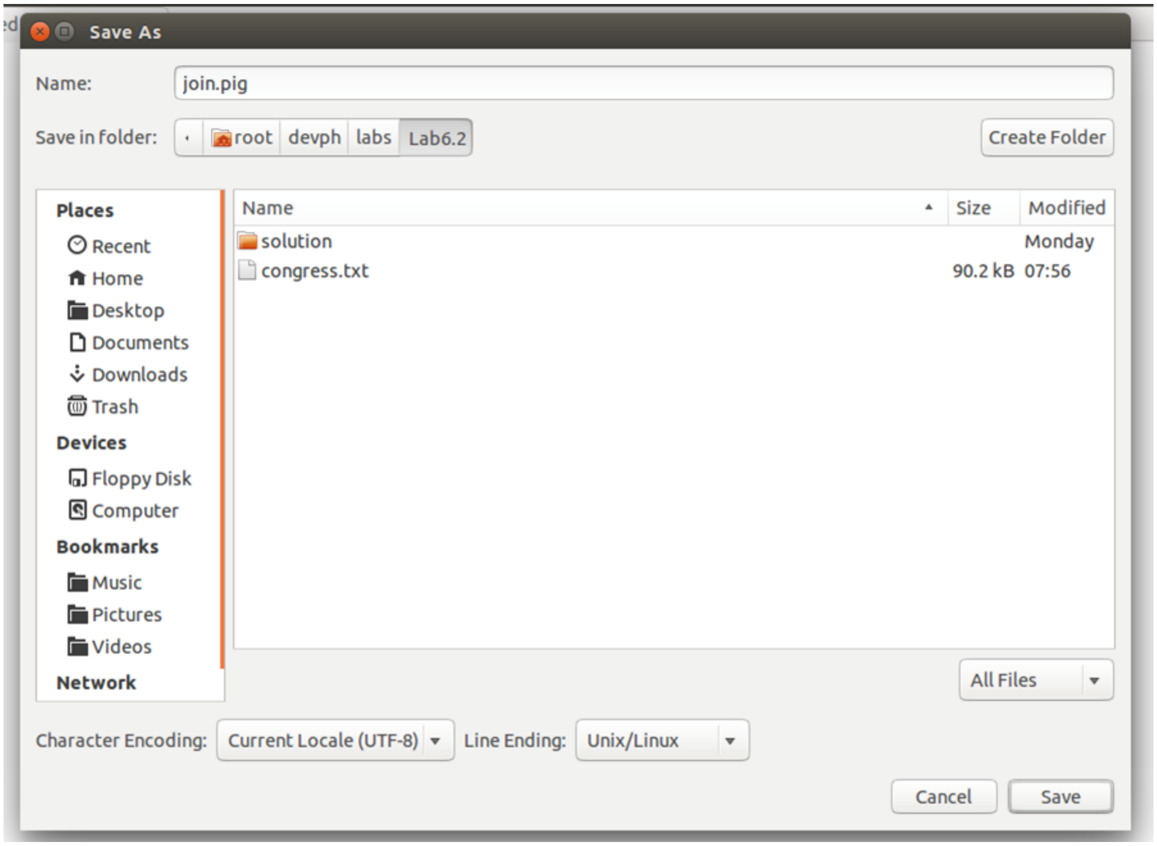
c. Use the `hdfs dfs -ls` command to verify that the congress.txt file is in `whitehouse`, and use `hdfs dfs -cat` to view its contents. The file contains the names of and other information about the members of the U.S. Congress.

```
# hdfs dfs -ls whitehouse

# hdfs dfs -cat whitehouse/congress.txt
```

2) Create a Pig Script File

a. In this lab, you will not use the Grunt shell to enter commands. Instead, you will enter your script in a text file. Start by opening the `gedit` text editor using the shortcut provided on the left-hand toolbar of your VM.

b. Click the Save button and save the new, empty file as `join.pig` in the `devph/labs/Lab6.2` folder:



Name: join.pig

Name	Size	Modified
solution		Monday
congress.txt	90.2 kB	07:56

Character Encoding: Current Locale (UTF-8)

Line Ending: Unix/Linux

c. At the top of the file, add a comment:

```
--join.pig: joins congress.txt and visits.txt
```

3) Load the White House Visitors

a. Define the following visitors relations, which will contain the first and last names of all White House visitors:

```
visitors = LOAD 'whitehouse/visits.txt' USING PigStorage(',') AS
(lname:chararray, fname:chararray);
```

That is the only data we are going to use from visits.txt.

4) Define a Projection of the Congress Data

a. Add the following load command that loads the ‘congress.txt’ file into a relation named `congress`. The data is tab-delimited, so no special Pig loader is needed:

```
congress = LOAD 'whitehouse/congress.txt' AS (
  full_title:chararray,
  district:chararray,
  title:chararray,
  fname:chararray,
  lname:chararray,
  party:chararray
);
```

b. The names in visits.txt are all uppercase, but the names in congress.txt are not. Define a projection of the `congress` relation that consists of the following fields:

```
congress_data = FOREACH congress GENERATE
  district,
  UPPER(lname) AS lname,
  UPPER(fname) AS fname,
  party;
```

5) Join the Two Datasets

a. Define a new relation named `join_contact_congress` that is a JOIN of visitors and `congress_data`. Perform the join on both the first and last names.

b. Use the `STORE` command to store the result of `join_contact_congress` into a directory named ‘`joinresult`’.

Solution:

```
join_contact_congress = JOIN visitors BY (lname,fname),
        congress_data BY (lname,fname);
STORE join_contact_congress INTO 'joinresult';
```

6) Run the Pig Script

a. Save your changes to `join.pig`.

b. Open a Terminal window and change directories to the Joining Datasets lab folder:

```
# cd ~/devph/labs/Lab6.2
```

c. Run the script using the following command:

```
# pig join.pig
```

d. Wait for the MapReduce job to execute. When it is finished, write down the number of seconds it took for the job to complete (by subtracting the `StartedAt` time from the `FinishedAt` time) and write down the result.

e. The type of join used is also output in the job statistics. Notice the statistics output has “`HASH_JOIN`” underneath the “`Features`” column, which means a hash join was used to join the two datasets.

7) View the Results

a. The output will be in the `joinresult` folder in HDFS. Verify that the folder was created:

```
# hdfs dfs -ls -R joinresult
-rw-r--r--   1 root root          0 joinresult/_SUCCESS
-rw-r--r--   1 root root      40892 joinresult/part-r-00000
```

b. View the resulting file:

```
# hdfs dfs -cat joinresult/part-r-00000
```

The output should look like the following:

```
DUFFY SEAN WI07 DUFFY SEAN Republican
JONES WALTER NC03 JONESWALTER Republican
SMITH ADAM WA09 SMITH ADAM Democrat
CAMPBELL JOHN CA45 CAMPBELL JOHN Republican
CAMPBELL JOHN CA45 CAMPBELL JOHN Republican
SMITH ADAM WA09 SMITH ADAM Democrat
```

8) Try Using Replicated on the Join

a. Delete the `joinresult` directory in HDFS:

```
# hdfs dfs -rm -R joinresult
```

b. Modify your `JOIN` statement in `join.pig` so that is uses replication. It should look like this:

```
join_contact_congress = JOIN visitors BY (lname,fname),
  congress_data BY (lname,fname) USING 'replicated';
```

c. Save your changes to `join.pig` and run the script again.

```
# pig join.pig

```

d. Notice this time that the statistics output shows Pig used a “`REPLICATED_JOIN`” instead of a “`HASH_JOIN`”.

e. Compare the execution time of the `REPLICATED_JOIN` vs. the `HASH_JOIN`. Did you have any improvement or decrease in performance?

Note

Using replicated does not necessarily increase the join time. There are way too many factors involved, and this example is using small datasets. The point is that you should try both techniques (if one dataset is small enough to fit in memory) and determine which join algorithm is faster for your particular dataset and use case.

9) Count the Results

a. In `join.pig`, comment out the `STORE` command:

```
--STORE join_contact_congress INTO 'joinresult';
```

You have already saved the output of the `JOIN`, so there is no need to perform the `STORE` command again.

b. Notice in the output of your `join.pig` script that we know which party the visitor belongs to: Democrat, Republican, or Independent. Using the `join_contact_congress` relation as a starting point, see if you can figure out how to output the number of Democrat, Republican, and Independent members of Congress that visited the White House. Name the relation counters and use the `DUMP` command to output the results:

```
join_group = GROUP join_contact_congress
```

BY congress_data::party;

```
counters = FOREACH join_group GENERATE group,
```

COUNT(join_contact_congress);

```
DUMP counters;
```

Tip
When you group the `join_contact_congress` relation, group it by the party field of `congress_data`. You will need to use the `::` operator in the `BY` clause. It will look like: `congress_data::party`

The correct results are shown here:

```
(Democrat,637)
(Republican,351)
(Independent,2)
```

10) Use the `EXPLAIN` Command

a. At the end of `join.pig`, add the following statement:

```
EXPLAIN counters;
```

If you do not have a counters relation, then use `join_contact_congress` instead.

b. Run the script again. The Logical, Physical, and MapReduce plans should display at the end of the output.

c. How many MapReduce jobs did it take to run this job? ____________

Answer**:** Two MapReduce jobs: the first job only requires a `map` phase, and the second job has a `map` , a combine and a `reduce` phase.

Result

You should have a folder in HDFS named `joinresult` that contains a list of members of Congress who have visited the White House (within the timeframe of the historical data in visits.txt).

Lab: Preparing Data for Hive

About this Lab

Objective: Transform and export a dataset for use with Hive.

File locations: /root/devph/labs/Lab6.3

Successful outcome: The resulting Pig script stores a projection of visits.txt in a folder in the Hive warehouse named wh_visits.

Before you begin: You should have visits.txt in a folder named whitehouse in HDFS.

Related lesson: ***Advanced Pig Programming***

Lab Steps

1) Review the Pig Script

a. If not already done, open a Terminal in your VM and type "ssh sandbox".
b. Change directories to the Preparing Data for Hive lab folder:

```
# cd ~/devph/labs/Lab6.3/
```

c. View the contents of `wh_visits.pig`:

```
# more wh_visits.pig
```

d. Notice that all White House visitors who met with the President are the `potus` relation.
e. Notice that the `project_potus` relation is a projection of the last name, first name, time of arrival, location, and comments from the visit.

2) Store the Projection in the Hive Warehouse

a. Open `wh_visits.pig` with the `gedit` text editor.
b. Add the following command at the bottom of the file, which stores the `project_potus` relation into a very specific folder in the Hive warehouse:

```
STORE project_potus INTO '/apps/hive/warehouse/wh_visits/';
```

3) Run the Pig Script

a. Save your changes to `wh_visits.pig`.

b. Run the script from the command line:

```
# pig wh_visits.pig
```

4) View the Results

a. The `wh_visits.pig` script creates a directory in the Hive warehouse named `wh_visits`. Use `ls` to view its contents:

```
# hdfs dfs -ls /apps/hive/warehouse/wh_visits/
-rw-r--r--   1 root hdfs          0
/apps/hive/warehouse/wh_visits/_SUCCESS
-rw-r--r--   1 root hdfs     971339 /apps/hive/warehouse/wh_visits/part-
m-00000
-rw-r--r--   1 root hdfs     142850 /apps/hive/warehouse/wh_visits/part-
m-00001
```

b. View the contents of one of the result files. It should look like the following:

```
hdfs dfs -cat /apps/hive/warehouse/wh_visits/part-m-00000
...
FRIEDMAN THOMAS 10/12/2010 12:08 WH PRIVATE LUNCH
BASS EDWIN 10/18/2010 15:01 WH
BLAKE CHARLES 10/18/2010 15:00 WH
OGLETREE CHARLES 10/18/2010 15:01 WH
RIVERS EUGENE 10/18/2010 15:01 WH
```

Result

You now have a folder in the Hive warehouse named `wh_visits` that contains a projection of the data in visits.txt. We will use this file in an upcoming Hive lab.

Demonstration: Computing PageRank

About this Demonstration

Objective: To understand how to use the PageRank UDF in DataFu.

During this demonstration: Watch as your instructor performs the following steps.

Related lesson: ***Advanced Pig Programming***

Demonstration Steps

1) View the Data

a. If not already done, open a Terminal in your VM and type "ssh sandbox".

b. Review the edges.txt file in the `/root/devph/labs/demos` folder:

```
# cd ~/devph/labs/demos/
# more edges.txt
0     2    3   1.0
0     3    2   1.0
0     4    1   1.0
0     4    2   1.0
0     5    4   1.0
0     5    2   11.0
0     5    6   11.0
0     6    5   11.0
0     6    2   11.0
0     100  2   11.0
0     100  5   11.0
0     101  2   11.0
0     101  5   11.0
0     102  2   11.0
0     102  5   11.0
0     103  5   11.0
0     104  5   11.0
```

c. The first column is the topic, but since we only have a single graph, the topic is 0 for all of the edges.

d. The second and third columns are the source and destination of each edge. For example, there is an edge from 2 to 3 based on the first row.

e. The fourth column is the weight of the edge. Our graph is all evenly weighted.

f. Based on the data above, which pages should be ranked toward the top?

Answer: It looks like 2 and 5 should end up toward the top.

2) Put the Data in HDFS

a) Put edges.txt into HDFS:

```
# hdfs dfs -put edges.txt
```

3) Define the PageRank UDF

a. Install DataFu:

```
yum –y install datafu
```

b. View the contents of `/root/devph/labs/demos/pagerank.pig` using gedit. The first two lines register the DataFu library and define the PageRank function (YOU DO NOT NEED TO TYPE ANYTHING - THE LINES BELOW MERELY DISCUSS WHAT IS ALREADY IN THE SCRIPT):

```
register /usr/hdp/current/pig-client/lib/datafu.jar;
define PageRank datafu.pig.linkanalysis.PageRank();
```

c. The edges are loaded and grouped by topic and source:

```
topic_edges = LOAD '/user/root/edges.txt' as
(topic:INT,source:INT,dest:INT,weight:DOUBLE);
topic_edges_grouped = GROUP topic_edges by (topic, source);
```

d. The data is then prepared for the PageRank function, which is expecting a topic, a source, and its edges:

```
topic_edges_data = FOREACH topic_edges_grouped GENERATE
group.topic as topic,
group.source as source,
topic_edges.(dest,weight) as edges;
```

e. We only have one topic, but the edges still need to be grouped by topic:

```
topic_edges_data_by_topic = GROUP topic_edges_data
BY topic;
```

f. We can now invoke the PageRank function:

```
topic_ranks = FOREACH topic_edges_data_by_topic GENERATE
group as topic,
FLATTEN(PageRank(topic_edges_data.(source,edges)));
```

g. The results are stored in HDFS:

```
store topic_ranks into 'topicranks';
```

4) Run the Script

```
# pig pagerank.pig
```

The job will take a couple of minutes to run.

5) View the Results

a.) View the contents of the topicranks folder in HDFS:

```
# hdfs dfs -ls topicranks
Found 2 items
-rw-r--r--  3 root root          0 topicranks/_SUCCESS
-rw-r--r--  3 root root        181 topicranks/part-r-00000
```

b.) View the contents of the output file

```
# hdfs dfs -cat topicranks/part-r-00000
0     104  0.013636362
0     1    0.02764593
0     103  0.013636362
0     5    0.06821412
0     100  0.013636362
0     102  0.013636362
0     6    0.032963693
0     3    0.2891899
0     2    0.32418048
0     4    0.032963693
0     101  0.013636362
```

6) Analyze the Results

a. Which page should be ranked the highest? ____________
Answer: Page 2

b. Which page should be ranked the lowest? ____________
Answer: Pages 100 to 104 all ranked equally at the bottom.

c. Compare the actual results with your guess from Step 1.

Lab: Analyzing Clickstream Data

About this Lab

Objective: Become familiar with using the DataFu library to sessionize clickstream data.

File locations: /root/devph/labs/Lab6.4

Successful outcome: You will have computed the length of each session along with the average and median values of all session lengths.

Before you begin: Your HDP 2.3 cluster should be up and running within your VM.

Related lesson: ***Advanced Pig Programming***

Lab Steps

1) View the Clickstream Data

a. If not already done, open a Terminal in your VM and type "ssh sandbox".

b. ***If not already done as part of the prior demonstration***, install DataFu:

```
yum –y install datafu
```

c. Open a Terminal and change directories to `~/devph/labs/Lab6.4`.

```
# cd ~/devph/labs/Lab6.4
```

d. View the contents of `clicks.csv`:

```
# more clicks.csv
```

The first column is the user’s ID, the second column is the time of the click stored as a long, and the third column is the URL visited. Enter “`q`” to quit the more command.

e. Put the file in HDFS:

```
# hdfs dfs -put clicks.csv
```

2) Define the Sessionized UDF

a. Using the `gedit` text editor, open the file `/root/devph/labs/Lab6.4/sessions.pig`.

b. Notice two `JAR` files are registered: `datafu.jar` and `piggybank.jar`. The `datafu JAR` contains the `Sessionize` function that you are going to use, and the `piggybank.jar` contains a time-utility function named `UnixToISO`, which is already defined for you in this Pig script.

c. Add the following `DEFINE` statement to define the Sessionize UDF:

```
DEFINE Sessionize datafu.pig.sessions.Sessionize('8m');
```

d. What does the '8m' mean in the constructor?

____

Answer: The '8m' stands for eight minutes, which is the length of the session. You can pick any length of time you want to define your sessions.

3) Sessionize the Clickstream

a. Notice the clicks.csv file is loaded for you in sessions.pig:

```
clicks = LOAD 'clicks.csv' USING PigStorage(',')
     AS (id:int, time:long, url:chararray);
```

b. Notice also that the clicks relation is projected onto `clicks_iso` with the long converted to an ISO time format then grouped by id in the `clicks_group` relation:

```
clicks_iso = FOREACH clicks GENERATE UnixToISO(time)
     AS isotime, time, id;
clicks_group = GROUP clicks_iso BY id;

```

c. Sessionize the clickstream by adding the following nested `FOREACH` loop:

```
clicks_sessionized = FOREACH clicks_group {
         sorted = ORDER clicks_iso BY isotime;
         GENERATE FLATTEN(Sessionize(sorted))
     AS (isotime, time, id, sessionid);
};
```

d. Dump the `sessionized` data:

```
DUMP clicks_sessionized;
```

e. Save your changes to `sessions.pig`.

4) Run the Script

a. Let's verify that the `Sessionized` function is working by running the script:

```
# pig sessions.pig
```

b. Verify that the tail of the output looks similar to the following:

```
(2013-01-10T07:15:20.520Z,1357802120520,2,51d89b38-b14a-4158-8703-
724525d9f787)
(2013-01-10T07:15:39.797Z,1357802139797,2,51d89b38-b14a-4158-8703-
724525d9f787)
(2013-01-10T07:26:30.602Z,1357802790602,2,711525c4-eff6-4697-ade7-
e2ad5ec555e5)
(2013-01-10T07:26:53.357Z,1357802813357,2,711525c4-eff6-4697-ade7-
e2ad5ec555e5)
(2013-01-10T07:26:58.800Z,1357802818800,2,711525c4-eff6-4697-ade7-
e2ad5ec555e5)
(2013-01-10T07:27:05.253Z,1357802825253,2,711525c4-eff6-4697-ade7-
e2ad5ec555e5)
(2013-01-10T07:27:57.844Z,1357802877844,2,711525c4-eff6-4697-ade7-
e2ad5ec555e5)
(2013-01-10T07:28:20.610Z,1357802900610,2,711525c4-eff6-4697-ade7-
e2ad5ec555e5)
(2013-01-10T07:29:01.128Z,1357802941128,2,711525c4-eff6-4697-ade7-
e2ad5ec555e5)
(2013-01-10T07:29:02.190Z,1357802942190,2,711525c4-eff6-4697-ade7-
e2ad5ec555e5)
(2013-01-10T07:29:23.190Z,1357802963190,2,711525c4-eff6-4697-ade7-
e2ad5ec555e5)
(2013-01-10T07:30:04.181Z,1357803004181,2,711525c4-eff6-4697-ade7-
e2ad5ec555e5)
(2013-01-10T07:30:32.455Z,1357803032455,2,711525c4-eff6-4697-ade7-
e2ad5ec555e5)
```

5) Compute the Session Length

a. Comment out the dump statement:

```
--DUMP clicks_sessionized
```

b. Add code to define a projection named sessions that is a projection of only the time and `sessionid` fields of the `clicks_sessionized` relation.
Solution:

```
sessions = FOREACH clicks_sessionized GENERATE time, sessionid;
```

c. Add code to define a relation named `sessions_group` that is the sessions relation grouped by `sessionid`.
Solution:

```
sessions_group = GROUP sessions BY sessionid;
```

d. Add code to define a `session_times` relation using the following projection that computes the length of each session:

```
session_times = FOREACH sessions_group
GENERATE group as sessionid,
(MAX(sessions.time) - MIN(sessions.time)) / 1000.0 / 60
as session_length;
```

e. Dump the `session_times` relation:

```
DUMP session_times;
```

f. Save your changes to `sessions.pig` and run the script. The output should look like the following:

```
(01e5259c-c5a6-45b0-8d04-1be86182d12e,0.16571666666666665)
(164be386-1df2-40dd-9331-563e1b8a7275,4.030883333333334)
(16ab9225-28d3-45f6-9d07-f065223046bb,38.809916666666666)
(18362695-d032-424a-a983-33ab45638700,0.0)
(2699ef77-bd37-4611-a239-ddbd80066043,10.398116666666665)
(3077f9d1-a5d5-4bf9-8212-87ae848b4ed8,3.44485)
(3e732d19-e3ed-4cc4-810f-f05c8534fb28,1.1402833333333333)
(455183ea-c3bb-43fe-9f07-63e0c0199008,14.648516666666666)
(5a65d8dc-1a4e-4355-b86a-f1efc519b084,63.620149999999995)
(5ef45fc4-01df-40d8-805f-a61c60fc421e,0.031733333333333335)
(61e14bcf-1fb4-4f7e-a3b4-2b67b8840756,1.0819833333333333)
(63b53f03-31e9-4a01-8029-6334020080e4,4.48765)
(66f58bc2-7aeb-487d-a28e-21090578cfe2,22.9298)
(812a7fc4-9ea2-4c3b-a3da-17bbd740a49a,0.0061833333333333335)
(84f8c113-d3c9-4590-83a8-5a9edf44c5c5,86.69525)
(85cd8b8c-644b-4fb9-a6c6-3b5082d32f0c,2.509133333333333)
(8e4cfed7-8500-47bb-a5e9-3744de6b1595,0.0)
(a35be8db-de7b-4b55-a230-66389a4e4b5f,0.9713166666666667)
(bcfef9fa-fd71-4962-8a0b-ddcf77ea47a3,0.37246666666666667)
(c092d0c4-3c7d-4cfc-b7f9-078baaa7469f,1.6453333333333333)
(d1d1b88e-b827-4005-b088-233d56c4ea8f,0.6608333333333333)
(e0f48349-1d2a-4cd7-8258-e36b4b6118fc,31.887883333333333)
(e1ccdf96-fc37-4b7e-9a7c-95acb8f52fa7,0.0)
(fd92f410-19fc-4927-917f-0f86b5d7edb2,17.197683333333334)
(fdfcea38-ddf9-477a-bb3e-401e8874e0ac,2.2512333333333334)
(ff70c6b5-abb2-4606-b12f-3054501947a4,0.051183333333333334)
```

g. How long was the longest session? ____________

Answer: The longest session was 86.69525 minutes.

6) Compute the Average Session Length

a. Comment out the dump statement:

```
--DUMP session_times;
```

b. Define a relation named `sessiontimes_all` that is a grouping of all `session_times`.

```
sessiontimes_all = GROUP session_times ALL;
```

c. Define `sessiontimes_avg` using the following nested `FOREACH` statement:

```
sessiontimes_avg = FOREACH sessiontimes_all
GENERATE AVG(session_times.session_length);
```

d. Dump the `sessiontimes_avg` relation:

```
DUMP sessiontimes_avg;
```

e. Save your changes to `sessions.pig` and run the script again.

f. Verify the output, which should be a single value representing the average session time:

```
(11.88608076923077)
```

7) Compute the Median Session Length

a. Using the `sessiontimes_avg` relation as an example, compute the median session time. You will need to define the Median function from the DataFu library, which is named `datafu.pig.stats.Median()`.

***Solution*:** A quick solution for computing the median is to simply add it to the existing nested `FOREACH` statement in the `sessiontimes_avg` definition:

```
--ADD to the top of the file

DEFINE Median datafu.pig.stats.Median();

--MODIFY sessiontimes_avg definition

sessiontimes_avg = FOREACH sessiontimes_all {
    ordered = ORDER session_times BY session_length;
    GENERATE
      AVG(ordered.session_length) AS avg_session,
        Median(ordered.session_length) AS median_session;
};
```

b. Verify that you got the following value for the median session length:

```
(1.9482833333333334)
```

Result

You have taken clickstream data and sessionized it using Pig to determine statistical information about the sessions, like the length of each session and the average and median lengths of all sessions.

Lab: Analyzing Stock Market Data using Quantiles

About this Lab

Objective: Use DataFu to compute quantiles.

File locations: /root/devph/labs/Lab6.5

Successful outcome: You will have computed quartiles for the daily high prices of stocks traded on the New York Stock Exchange.

Before you begin: Your HDP 2.3 cluster should be up and running within your VM and DataFu should have been installed via "yum –y install datafu".

Related lesson: ***Advanced Pig Programming***

Lab Steps

1) Review the Stock Market Data

a. O If not already done, open a Terminal in your VM and type "ssh sandbox".

b. Change directories to the `~/devph/labs/Lab6.5` folder.

```
# cd ~/devph/labs/Lab6.5
```

c. View the contents of the `stocks.csv` file, which contains the historical prices for New York Stock Exchange stocks that begin with the letter “`Y`”:

```
# tail stocks.csv
NYSE,YSI,2004-11-23,17.35,17.48,16.90,17.26,207400,13.26
NYSE,YSI,2004-11-22,17.20,17.43,16.90,17.35,204100,13.33
NYSE,YSI,2004-11-19,17.20,17.45,16.85,17.45,304100,13.41
NYSE,YSI,2004-11-18,17.40,17.45,17.10,17.11,180900,13.14
NYSE,YSI,2004-11-17,17.16,17.77,17.15,17.35,320400,13.33
NYSE,YSI,2004-11-16,17.20,17.33,17.05,17.15,245000,13.18
NYSE,YSI,2004-11-15,16.95,17.20,16.90,17.20,174400,13.21
NYSE,YSI,2004-11-12,17.05,17.14,16.99,17.00,359900,13.06
NYSE,YSI,2004-11-11,16.92,17.04,16.81,17.00,263800,13.06
NYSE,YSI,2004-11-10,16.90,17.05,16.80,17.00,243300,13.06
```

The first column is always "NYSE." The second column is the stock's symbol. The third column is the date that the prices occurred. The next columns are the open, high, low, close, and trading volume.

d. Put stocks.csv into your /user/root folder in HDFS *(if you already have a stocks.csv file in your HDFS home directory from exercising the block storage demonstration,* ***please delete the existing file first as the contents are different)***:

```
# hdfs dfs -put stocks.csv
```

2) Define the Quantile Function

a. Using gedit, create a new text file in the /root/devph/labs/Lab6.5 lab folder named quantile.pig.

b. On the first line of the file, register the datafu JAR file which you installed in the prior lab.

```
register /usr/hdp/current/pig-client/lib/datafu.jar;
```

c. Define the datafu.pig.stats.Quantile function as a quantile, and pass in the values for computing the quartiles of a set of numbers:

```
define Quantile datafu.pig.stats.Quantile(
        '0.0','0.25','0.50','0.75','1.0');
```

3) Load the Stocks

a. Enter the following LOAD command, which loads the first five values of each row:

```
stocks = LOAD 'stocks.csv' USING PigStorage(',') AS
     (nyse:chararray,
     symbol:chararray,
     closingdate:chararray,
     openprice:double,
     highprice:double,
     lowprice:double);
```

4) Filter Null Values

a. The quantile function fails if any of the values passed to it are null. Define a relation named stocks_filter that filters the stocks relation where the highprice is not null.

Solution:

```
stocks_filter = FILTER stocks BY highprice is not null;
```

5) Group the Values

a. We want to compute the quantiles for each individual stock (as opposed to all the stock prices that start with a "Y"), so define a relation named `stocks_group` that groups the `stock_filter` relation by symbol.

Solution:

```
stocks_group = GROUP stocks_filter BY symbol;
```

6) Compute the Quantiles

a. Define the following relation that invokes the Quantile method on the `highprice` values:

```
quantiles = FOREACH stocks_group {
    sorted = ORDER stocks_filter BY highprice;
    GENERATE group AS symbol,
       Quantile(sorted.highprice) AS quant;
};
```

b. How many times will the quantile function be invoked in the nested `FOREACH` statement above?

Answer: The `FOREACH` statement iterates over the `stocks_group`, which is a grouping by symbol. So the quantile function will be invoked once for each unique stock symbol in the `stocks.csv` file.

c. Add a `DUMP` statement that outputs the quantiles relation:

```
DUMP quantiles;
```

7) Run the Script

a. Save your changes to `quantile.pig`.

b. Run the script:

```
# pig quantile.pig
```

c. There is stock information in the input data, so the output will be the quantiles of the high price of these five stocks:

```
(YGE,(3.22,10.97,14.79,19.6,41.5))
(YPF,(9.0,23.62,31.94,41.47,69.98))
(YSI,(1.56,8.04,16.435000000000002,19.93,23.61))
(YUM,(21.9,32.08,37.85,48.91,73.87))
(YZC,(4.41,14.4,20.795,47.13,116.73))
```

8) Compute the Median

a. Now that you have a working Pig script for computing the quantiles of the high prices of stocks, see if you can modify the script (you only have to make a few changes) to compute the median value of the high prices.

Solution:

(1) Change the `define` statement to the following:

```
define Median datafu.pig.stats.Median();
```

(2) Change the `quantiles` definition to a `medians` definition as follows:

```
medians = FOREACH stocks_group {
    sorted = ORDER stocks_filter BY highprice;
    GENERATE group AS symbol,
       Median(sorted.highprice) AS median;
};
```

(3) Change the `DUMP` command to display `medians`:

```
DUMP medians;
```

(4) Save the modified file as `median.pig`

(5) Run the modified script and view the results

```
# pig median.pig
```

(6) The output will be the median values for the same five stocks:

```
(YGE,(14.79))
(YPF,(31.94))
(YSI,(16.435000000000002))
(YUM,(37.85))
(YZC,(20.795))
```

Result

You have used the DataFu library to compute the quantiles of a collection of numbers using Pig.

Lab: Understanding Hive Tables

About this Lab

Objective:	Understand how Hive table data is stored in HDFS.
File locations:	/root/devph/labs/7.1
Successful outcome:	A new Hive table filled with the data from the wh_visits folder.
Before you begin:	Complete the Preparing Data for Hive lab, or put the data from the solution of that lab into HDFS.
Related lesson:	Hive Programming

Lab steps

1) Review the Data

a. If not already done, open a Terminal in your VM and type "ssh sandbox".

b. Use the `hdfs dfs -ls` command to view the contents of the `/apps/hive/warehouse/wh_visits/` folder in HDFS that was created in an earlier lab. You should see six `part-m` files:

```
# hdfs dfs -ls /apps/hive/warehouse/wh_visits/
```

c. Recall that the Pig projection to create these files had the following schema (no typing necessary - this is reprinted below for reference only):

```
project_potus = FOREACH potus GENERATE
   $0 AS lname:chararray,
   $1 AS fname:chararray,
   $6 AS time_of_arrival:chararray,
   $11 AS appt_scheduled_time:chararray,
   $21 AS location:chararray,
   $25 AS comment:chararray ;
```

In this lab, you will define a Hive table that matches these records and contains the exported data from your Pig script.

2) Define a Hive Script

a. In the Understanding `/root/devph/labs/Lab7.1` folder, there is a text file named `wh_visits.hive`. View its contents.

Notice that it defines a Hive table named `wh_visits` with the following schema that matches the data in your `project_potus` folder:

```
# cd ~/devph/labs/Lab7.1
# more wh_visits.hive
create table wh_visits (
   lname string,
   fname string,
   time_of_arrival string,
   appt_scheduled_time string,
   meeting_location string,
   info_comment string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' ;
```

Note

You cannot use comment or location as column names because those are reserved Hive keywords, so we changed them slightly.

b. Run the script with the following command:

```
# hive -f wh_visits.hive
```

If successful, you should see "`OK`" in the output along with the time it took to run the query.

3) Verify the Table Creation

a. Start the Hive Shell:

```
# hive
```

b. From the hive> prompt, enter the "`show tables`" command:

```
hive> show tables;
```

You should see `wh_visits` in the list of tables.

c. Use the describe command to view the details of `wh_visits`:

```
hive> describe wh_visits;
OK
lname               string
fname               string
time_of_arrival     string
appt_scheduled_time string
meeting_location    string
info_comment        string
```

d. Try running a query (even though the table is empty):

```
hive> select * from wh_visits limit 20;
```

You should see 20 rows returned. How is this brand new Hive table already populated with records? ______________________

Answer: In a previous lab, you already populated the `/apps/hive/warehouse/wh_visits` folder with the output of a Pig job.

Why did the previous query not require a Tez or MapReduce job to execute?

Answer: The query selected all columns and did not contain a `WHERE` clause, the query just needs to read in the data from the file and display it.

4) Count the Number of Rows in a Table

a. Enter the following Hive query, which outputs the number of rows in `wh_visits`:

```
hive> select count(*) from wh_visits;
```

How many rows are currently in `wh_visits`? ____________

Answer: 21,819

5) Selecting the Input File Name

a. Hive has two virtual columns that get created automatically for every table: `INPUT__FILE__NAME` and `BLOCK__OFFSET__INSIDE__FILE`.

Note that between each word in the column name there are two underscore characters, not just one. You must make sure you type both of them when using these columns in a hive command.

You can use these column names in your queries just like any other column of the table. To demonstrate, run the following query:

```
hive> select INPUT__FILE__NAME, lname, fname FROM wh_visits WHERE
lname LIKE 'Y%';
```

b. The result of this query is visitors to the White House whose last name starts with "`Y`." Notice that the output also contains the particular file that the record was found in:

```
hdfs://sandbox.hortonworks.com:8020/apps/hive/warehouse/wh_visits/
part-m-00000YOUNGMICHELLE
hdfs://sandbox.hortonworks.com:8020/apps/hive/warehouse/wh_visits/
part-m-00001YOUNGLEDISI
```

6) Drop a Table

a. **Let’s see what happens when a managed table is dropped. Start by defining a simple table called names using the Hive Shell:**

```
hive> create table names (id int, name string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
```

b. **Use the Hive `dfs` command to put `Lab7.1/names.txt` into the table’s warehouse folder:**

```
hive> dfs -put /root/devph/labs/Lab7.1/names.txt
/apps/hive/warehouse/names/;
```

c. **View the contents of the table’s warehouse folder:**

```
hive> dfs -ls /apps/hive/warehouse/names;
Found 1 items
root hdfs      78 /apps/hive/warehouse/names/names.txt
```

d. **From the Hive Shell, run the following query:**

```
hive> select * from names;
OK
0    Rich
1    Barry
2    George
3    Ulf
4    Danielle
5    Tom
6    manish
7    Brian
8    Mark
```

e. **Now drop the names table:**

```
hive> drop table names;
```

f. **View the contents of the table’s warehouse folder again. Notice the names folder is gone:**

```
hive> dfs -ls /apps/hive/warehouse/names;
ls: '/apps/hive/warehouse/names': No such file or directory
```

Important

Be careful when you drop a managed table in Hive. Make sure you either have the data backed up somewhere else or that you no longer want the data.

7) Define an External Table

a. In this step you will see how external tables work in Hive. Start by putting names.txt into HDFS:

```
hive> dfs -put /root/devph/labs/Lab7.1/names.txt names.txt;
```

b. Create a folder in HDFS for the external table to store its data in:

```
hive> dfs -mkdir hivedemo;
```

c. Define the names table as external this time:

```
hive> create external table names (id int, name string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
LOCATION '/user/root/hivedemo';
```

d. Load data into the table:

```
hive> load data inpath '/user/root/names.txt' into table names;
```

e. Verify that the load worked:

```
hive> select * from names;
```

f. Notice the names.txt file has been moved to `/user/root/hivedemo`:

```
hive> dfs -ls hivedemo;
Found 1 items
-rw-r--r--   1 root hdfs       78  hivedemo/names.txt
```

g. Similarly, verify that names.txt is no longer in your `/user/root` folder in HDFS.

```
hive> dfs -ls /user/root/names.txt;
```

Why is it gone? ____________________

Answer: The `LOAD` command moved the file from `/user/root` to `/user/root/names`. The `LOAD` command does not copy files; it moves them.

h. Use the `ls` command to verify that the `/apps/hive/warehouse` folder does not contain a subfolder for the names table.

```
hive> dfs -ls /apps/hive/warehouse;
```

i. Now drop the names table:

```
hive> drop table names;
```

j. View the contents of `/user/root/hivedemo`. Notice that names.txt is still there.

```
hive> dfs -ls /user/root/hivedemo;
```

Result

You have verified that the data for external tables is not deleted when the corresponding table is dropped. Aside from this behavior, managed tables and external tables in Hive are essentially the same. You now have a table in Hive named `wh_visits` that was loaded from the result of a Pig job. You also have an external table called names that stores its data in `/user/root/hivedemo`. At this point, you should have a pretty good understanding of how Hive tables are created and populated.

Demonstration: Understanding Partitions and Skew

About this Demonstration

Objective:	To understand how Hive partitioning and skewed tables work.
During this Demonstration:	Watch as your instructor performs the following steps.
Related lesson:	***Hive Programming***

Demonstration Steps

1) View the Data

   a) If not already done, open a Terminal in your VM and type "ssh sandbox".

   b) Review the `hivedata_<<state>>.txt` files in `/root/devph/labs/demos`. This will be the data added to the table.

2) Define the Table in Hive

```
# hive

hive> show partitions names;
```

3) Load Data into the Table

   a) When you load data into a partitioned table, you specify which partition the data goes into. For example:

```
hive> load data local inpath
'/root/devph/labs/demos/hivedata_ca.txt'
into table names partition (state = 'CA');
```

   b) Load the `CO` and `SD` files also:

```
hive> load data local inpath
'/root/devph/labs/demos/hivedata_co.txt'
into table names partition (state = 'CO');
load data local inpath '/root/devph/labs/demos/hivedata_sd.txt'
into table names partition (state = 'SD');
```

c) Verify that all of the data made it into the names table:

```
hive> select * from names;
OK
1       Ulf     CA
2       Manish  CA
3       Brian   CA
4       George  CO
5       Mark    CO
6       Rich    SD
```

4) View the Directory Structure

a) List the partitions and view the contents of `/apps/hive/warehouse/names`:

```
hive> show partitions names;
OK
state=CA
state=CO
state=SD

hive> dfs -ls -R /apps/hive/warehouse/names/;
0   /apps/hive/warehouse/names/state=CA
24  /apps/hive/warehouse/names/state=CA/hivedata_ca.txt
0   /apps/hive/warehouse/names/state=CO
16  /apps/hive/warehouse/names/state=CO/hivedata_co.txt
0   /apps/hive/warehouse/names/state=SD
6   /apps/hive/warehouse/names/state=SD/hivedata_sd.txt
```

Notice that each partition has its own subfolder for storing its contents.

5) Perform a Query

a) When you specify a where clause that includes a partition, Hive is smart enough to only scan the files in that partition. For example:

```
hive> select * from names where state = 'CA';

OK

1       Ulf     CA

2       Manish  CA

3       Brian   CA
```

b) Notice that a MapReduce job was not executed. Why? ____________________

Answer: The result of the query is exactly the contents of the underlying files, so there is no need to run a MapReduce job. The files can simply be read and displayed.

c) You can select the `partition` field, even though it is not actually in the data file. Hive uses the directory name to retrieve the value:

```
hive> select name, state from names where state = 'CA';
```

d) You can still run queries across the entire dataset. For example, the following query spans multiple partitions. When you are done, use `quit` to exit the Hive shell.:

```
hive> select name, state from names where state = 'CA' or state =
'SD';

hive> quit;
```

6) Create a Skewed Table

a) Verify the existence of the salaries.txt folder in `~/devph/labs/demos/` and then put it into the `/user/root/salarydata/` folder in HDFS.

```
# ls salaries.txt

# hdfs dfs -put salaries.txt /user/root/salarydata/salaries.txt
```

b) View the contents of `demos/skewdemo.hive`, which defines a skewed table named `skew_demo` using the salaries.txt data:

```
# more skewdemo.hive
```

c) Which values are skewing this table? ___________________________

Answer: The skewed values are the 95102 and 94040 zip codes.

7) Run the `skewdemo.hive` script:

```
# hive -f skewdemo.hive
```

a) View the contents of the underlying Hive warehouse folder:

```
# hdfs dfs -ls -R /apps/hive/warehouse/skew_demo
```

b) Select a few records to make sure the table has data behind it:

```
# hive -f show_skewdemo.hive
```

Lab: Analyzing Big Data with Hive

About this Lab

Objective:	Analyze the White House visitor data.
File locations:	/root/devph/labs/Lab7.2
Successful outcome:	You will have discovered several useful pieces of information about the White House visitor data.
Before you begin:	Complete the Understanding Hive Tables Lab.
Related lesson:	***Hive Programming***

Lab Steps

1) Find the First Visit

a. If not already done, open a Terminal in your VM and type "ssh sandbox".

b. Using `gedit`, create a new text file named `whitehouse.hive` and save it in your `~/devph/labs/Lab7.2` folder.

c. In this step, you will instruct the hive script to find the first visitor to the White House (based on our dataset). This will involve some clever handling of timestamps. This will be a long query, so enter it on multiple lines (note the lack of a ";" at the end of this first step). Start by selecting all columns where the `time_of_arrival` is not empty:

```
select * from wh_visits where time_of_arrival != ""
```

d. To find the first visit, we need to sort the result. This requires converting the `time_of_arrival` string into a timestamp. We will use the `unix_timestamp` function to accomplish this. Add the following order by clause (again, no ";" at the end of the line):

```
order by unix_timestamp(time_of_arrival,
   'MM/dd/yyyy hh:mm')
```

e. Since we are only looking for one result, we certainly don’t need to return every row. Let’s limit the result to 10 rows, so we can view the first 10 visitors (this finishes the query, so will end with the ";" character):

```
limit 10;
```

f. Save your changes to `whitehouse.hive`.

g. Execute the script `whitehouse.hive` and wait for the results to be displayed:

```
# cd ~/devph/labs/Lab7.2

# hive -f whitehouse.hive
```

h. The results should be 10 visitors, and the first visit should be in 2009, since that is when the dataset begins. The first visitors are Charles Kahn and Carol Keehan on 3/5/2009.

2) Find the Last Visit

a. This one is easy: just take the previous query and reverse the order by adding `desc` to the order by clause:

```
order by unix_timestamp(time_of_arrival,
   'MM/dd/yyyy hh:mm') desc
```

b. Run the query again, and you should see that the most recent visit was Jackie Walker on 3/18/2011.

```
# hive -f whitehouse.hive
```

3) Find the Most Common Comment

a. In this step, you will explore the `info_comment` field and try to determine the most common comment. You will use some of Hive's aggregate functions to accomplish this. Start by using gedit to create a new text file named `comments.hive` and save it in `~/devph/labs/Lab7.2` folder.

b. You will now create a query that displays the 10 most frequently occurring comments. Start with the following select clause:

```
from wh_visits
select count(*) as comment_count, info_comment
```

This runs the aggregate count function on each group (which you will define later in the query) and names the result `comment_count`. For example, if "`OPEN HOUSE`" occurs five times then `comment_count` will be five for that group.
Notice we are also selecting the `info_comment` column so we can see what the comment is.

c. Group the results of the query by the `info_comment` column:

```
group by info_comment
```

d. Order the results by `comment_count`, because we are only interested in comments that appear most frequently:

```
order by comment_count DESC
```

e. We only want the top results, so limit the result set to 10:

```
limit 10;
```

f. Save your changes to `comments.hive` and execute the script. Wait for the MapReduce job to execute.

```
# hive -f comments.hive
```

g. The output will be 10 comments and should look like:

```
9036
1253 HOLIDAY BALL ATTENDEES/
894  WHO EOP RECEP 2
700  WHO EOP 1 RECEPTION/
601  RESIDENCE STAFF HOLIDAY RECEPTION/
586  PRESS RECEPTION ONE (1)/
580  GENERAL RECEPTION 1
540  HANUKKAH RECEPTION./
540  GEN RECEP 5/
516  GENERAL RECEPTION 3
```

h. It appears that a blank comment is the most frequent comment, followed by the `HOLIDAY BALL`, then a variation of other receptions.

i. Modify the query so that it ignores empty comments. If it works, the comment "`GEN RECEP 6/`" will show up in your output.

Solution:

```
--In comments.hive, insert the following line between your select
and group statements:

where info_comment != ""

Save the changes, then back at the command line, re-run the query:

# hive -f comments.hive
```

4) Least Frequent Comment

a. Run the previous query again, but this time, find the 10 least occurring comments.

```
--Remove DESC from your order statement so that it looks like
this:

order by comment_count

Save the changes, then back at the command line, re-run the query:

# hive -f comments.hive
```

The output should look something like:

```
1   CONGRESSIONAL BALL/
1   CONG BALL/
1   merged to u59031
1   CONGRESSIONAL BALL
1   CONG BALL
1   COMMUNITY COLLEGE SUMMIT
1   48 HOUR WAVE EXCEPTION GRANTED
1   DROP BY VISIT
1   WHO EOP/
1   "POTUS LUNCH WITH WASHINGTON
```

This seems accurate since 1 is the least number of times a comment can appear.

5) Analyze the Data Inconsistencies

a. Analyzing the results of the most- and least-frequent comments, it appears that several variations of `GENERAL RECEPTION` occur. In this step, you will try to determine the number of visits to the POTUS involving a general reception by trying to clean up some of these inconsistencies in the data.

Note
Inconsistencies like these are very common in big data, especially when human input is involved. In this dataset, we likely have different people entering similar comments but using their own abbreviations.

b. Modify the query in `comments.hive`. Instead of searching for empty comments. Search for comments that contain variations of the string "`GEN RECEP`."

```
where info_comment rlike '.*GEN.*\\s+RECEP.*'
```

c. Change the limit clause from 10 to 30:

```
limit 30;
```

d. Run the query again.

```
# hive -f comments.hive
```

e. Notice there are several `GENERAL RECEPTION` entries that only differ by a number at the end or use the `GEN RECEP` abbreviation:

```
580  GENERAL RECEPTION 1
540  GEN RECEP 5/
516  GENERAL RECEPTION 3
498  GEN RECEP 6/
438  GEN RECEP 4
31   GENERAL RECEPTION 2
23   GENERAL RECEPTION 3
20   GENERAL RECEPTION 6
20   GENERAL RECEPTION 5
13   GENERAL RECEPTION 1
```

f. Let’s try one more query to try and narrow `GENERAL RECEPTION` visit. Modify the `WHERE` clause in `comments.hive` to include `“%GEN%”`:

```
where info_comment like  "%RECEP%"
and info_comment like "%GEN%"
```

g. Leave the limit at 30, save your changes, and run the query again.

```
# hive -f comments.hive
```

h. The output this time reveals all the variations of `GEN` and `RECEP`. Next, let’s add up the total number of them by running the following query:

```
from wh_visits
select count(*)
where info_comment like  "%RECEP%"
and info_comment like "%GEN%";

--Then save your changes and run the query again from the command
line:

# hive -f comments.hive
```

i. Notice there are 2,697 visits to the POTUS with `GEN RECEP` in the comment field, which is about 12% of the 21,819 total visits to the POTUS in our dataset.

Note
More importantly, these results show that the conclusion from our first query, where we found that the most likely reason to visit the President was the `HOLIDAY BALL` with 1,253 attendees, is incorrect. This type of analysis is common in big data, and it shows how data analysts need to be creative and thorough when researching their data.

6) Verify the Result

a. **We have 12% of visitors to the POTUS going for a general reception, but there were a lot of statements in the comments that contained `WHO` and `EOP`. Modify the query from the last step and display the top 30 comments that contain "`WHO`" and "`EOP`."**

```
--You should be able to undo changes to comments.hive and restore
it to the state before the last lab. Then make the following two
additional edits:

--Change the where clause to match WHO and EOP

where info_comment like  "%WHO%"
and info_comment like "%EOP%";

--Add the DESC command back to the end of the order statement

order by comment_count DESC

--Finally, double-check select count(*) as comment_count,
info_count
--Make sure the "as..." portion is there

--Then save your changes and run the query again from the command
line:

# hive -f comments.hive
```

The result should look like:

```
894  WHO EOP RECEP 2
700  WHO EOP 1 RECEPTION/
43   WHO EOP RECEP/
20   WHO EOP HOLIDAY RECEP/
13   WHO/EOP #2/
8       WHO EOP RECEPTION
7       WHO EOP RECEP
1       WHO EOP/
1       WHO EOP RECLEAR
```

b. Modify the script again, this time to run a query that counts the number of records with `WHO` and `EOP` in the comments, and run the query:

```
from wh_visits
select count(*)
where info_comment like  "%WHO%"
and info_comment like "%EOP%";

--Run the query from the command line:

# hive -f comments.hive
```

You should get 1,687 visits, or 7.7% of the visitors to the POTUS. So `GENERAL RECEPTION` still appears to be the most frequent comment.

7) Find the Most Visits

a. See if you can write a Hive script that finds the top 20 individuals who visited the POTUS most. Use the Hive command from Step 3 earlier in this lab as a guide.

Tip
Use a grouping by both `fname` and `lname`.

The following script will accomplish the intention of the previous step:

```
from wh_visits
  select count(*) as most_visit, fname, lname
  group by fname, lname
  order by most_visit DESC
  limit 20;
```

To verify that your script worked, here are the top 20 individuals who visited the POTUS along with the number of visits (your output may vary slightly due to randomization of names):

```
16 ALAN PRATHER
15 CHRISTOPHER FRANKE
15 ANNAMARIA MOTTOLA
14 ROBERT  BOGUSLAW
14 CHARLES POWERS
12 SARAH HART
12 JACKIE WALKER
12 JASON FETTIG
12 SHENGTSUNG WANG
12 FERN SATO
12 DIANA FISH
11 JANET BAILEY
11 PETER WILSON
11 GLENN DEWEY
11 MARCIO BOTELHO
11 DONNA WILLINGHAM
10 DAVID AXELROD
10 CLAUDIA CHUDACOFF
10 VALERIE JARRETT
10 MICHAEL COLBURN
```

Result

You have written several Hive queries to analyze the White House visitor data. The goal is for you to become comfortable with working with Hive, so hopefully you now feel like you can tackle a Hive problem and be able to answer questions about your big data stored in Hive.

Demonstration: Computing ngrams

About this Demonstration

Objective:	To understand how to compute ngrams using Hive.
During this demonstration:	Watch as your instructor performs the following steps.
Related lesson:	***Hive Programming***

Demonstration Steps

1) Create a Hive Table for the Data

a. If not already done, open a Terminal in your VM and type "ssh sandbox".

b. This demonstration computes `ngrams` on the U.S. Constitution, which is in a text file in the `/root/devph/labs/demos` folder:

```
# cd ~/devph/labs/demos/
# more constitution.txt

--press q to exit more
```

c. Start the Hive shell and define the following table:

```
# hive

hive> create table constitution (
line string
)
ROW FORMAT DELIMITED;
```

Each line of text in the text file is going to be a record in our Hive table.

2) Load the Hive Table

a. Load constitution.txt into the `constitution` table:

```
hive> load data local inpath
'/root/devph/labs/demos/constitution.txt' into table constitution;
```

b. Verify that the data is loaded:

```
hive> select * from constitution;
```

You should see the contents of constitution.txt again.

3) Compute a Bigram

```
hive> select explode(ngrams(sentences(line),2,15)) as x
from constitution;
```

The result should look similar to:

```
{"ngram":["of","the"],"estfrequency":194.0}
{"ngram":["shall","be"],"estfrequency":100.0}
{"ngram":["the","United"],"estfrequency":76.0}
{"ngram":["United","States"],"estfrequency":76.0}
{"ngram":["to","the"],"estfrequency":57.0}
{"ngram":["shall","have"],"estfrequency":44.0}
{"ngram":["the","President"],"estfrequency":30.0}
{"ngram":["shall","not"],"estfrequency":29.0}
{"ngram":["in","the"],"estfrequency":28.0}
{"ngram":["by","the"],"estfrequency":25.0}
{"ngram":["the","Congress"],"estfrequency":22.0}
{"ngram":["and","the"],"estfrequency":21.0}
{"ngram":["for","the"],"estfrequency":21.0}
{"ngram":["Vice","President"],"estfrequency":21.0}
{"ngram":["the","Senate"],"estfrequency":21.0}
{"ngram":["States","and"],"estfrequency":20.0}
{"ngram":["States","shall"],"estfrequency":19.0}
{"ngram":["any","State"],"estfrequency":18.0}
{"ngram":["Congress","shall"],"estfrequency":18.0}
{"ngram":["on","the"],"estfrequency":17.0}
```

4) Compute a Trigram

```
{"ngram":["the","United","States"],"estfrequency":68.0}
{"ngram":["of","the","United"],"estfrequency":51.0}
{"ngram":["shall","not","be"],"estfrequency":16.0}
{"ngram":["of","the","Senate"],"estfrequency":14.0}
{"ngram":["States","shall","be"],"estfrequency":13.0}
{"ngram":["House","of","Representatives"],"estfrequency":13.0}
{"ngram":["United","States","shall"],"estfrequency":13.0}
{"ngram":["shall","have","been"],"estfrequency":12.0}
{"ngram":["the","several","States"],"estfrequency":12.0}
{"ngram":["President","of","the"],"estfrequency":11.0}
{"ngram":["United","States","and"],"estfrequency":11.0}
{"ngram":["The","Congress","shall"],"estfrequency":10.0}
{"ngram":["the","House","of"],"estfrequency":10.0}
{"ngram":["United","States","or"],"estfrequency":10.0}
{"ngram":["Congress","shall","have"],"estfrequency":10.0}
{"ngram":["the","Vice","President"],"estfrequency":9.0}
{"ngram":["of","the","President"],"estfrequency":8.0}
{"ngram":["Consent","of","the"],"estfrequency":8.0}
{"ngram":["shall","be","the"],"estfrequency":7.0}
{"ngram":["by","the","Congress"],"estfrequency":7.0}
```

5) Compute a Contextual `ngram`

a. **Let’s find the 20 most frequent words that follow “the”:**

```
hive> select explode(context_ngrams(sentences(line),
array("the",null),20)) as result
from constitution;
```

b. The result should look similar to:

```
{"ngram":["United"],"estfrequency":76.0}
{"ngram":["President"],"estfrequency":30.0}
{"ngram":["Congress"],"estfrequency":22.0}
{"ngram":["Senate"],"estfrequency":21.0}
{"ngram":["several"],"estfrequency":15.0}
{"ngram":["Vice"],"estfrequency":12.0}
{"ngram":["State"],"estfrequency":11.0}
{"ngram":["same"],"estfrequency":10.0}
{"ngram":["Constitution"],"estfrequency":10.0}
{"ngram":["States"],"estfrequency":10.0}
{"ngram":["House"],"estfrequency":10.0}
{"ngram":["whole"],"estfrequency":10.0}
{"ngram":["office"],"estfrequency":9.0}
{"ngram":["right"],"estfrequency":8.0}
{"ngram":["Legislature"],"estfrequency":8.0}
{"ngram":["Consent"],"estfrequency":6.0}
{"ngram":["powers"],"estfrequency":6.0}
{"ngram":["supreme"],"estfrequency":6.0}
{"ngram":["people"],"estfrequency":6.0}
{"ngram":["first"],"estfrequency":6.0}
```

Lab: Joining Datasets in Hive

About this Lab

Objective: Perform a join of two datasets in Hive.

File locations: /root/devph/labs/Lab7.4

Successful outcome: A table named stock_aggregates that contains a join of NYSE stock prices with the stock's dividend prices.

Before you begin: Your HDP 2.3 cluster should be up and running within your VM.

Related lesson: ***Hive Programming***

Lab Steps

1) Load the Data into Hive

a. If not already done, open a Terminal in your VM and type "ssh sandbox".

b. View the contents of the file `setup.hive` in `/root/devph/labs/Lab7.4`:

```
# cd ~/devph/labs/Lab7.4/
# more setup.hive
```

c. Notice that this script creates three tables in Hive. The `nyse_data` table is filled with the daily stock prices of stocks that start with the letter K and the dividends table that contains the quarterly dividends of those stocks. The `stock_aggregates` table is going to be used for a join of these two datasets and contain the stock price and dividend amount on the date the dividend was paid.

d. Run the `setup.hive` script from the Joining Datasets in Hive lab folder:

```
# hive -f setup.hive
```

e. To verify that the script worked, enter the Hive Shell and run the following following queries:

```
# hive

hive> select * from nyse_data limit 20;

hive> select * from dividends limit 20;
```

You should see daily stock prices and dividends from stocks that start with the letter K.

f. The `stock_aggregates` table should be empty, but view its schema to verify that it was created successfully, then type `quit` to exit the Hive Shell:

```
hive> describe stock_aggregates;
OK
symbol              string
year                string
high                float
low                 float
average_close       float
total_dividends     float

hive> quit;
```

2) Join the Datasets

a. The join statement is going to be fairly long, so let's create it in a text file. Use `gedit` to create a new text in the `/root/devph/labs/Lab7.4/` folder named `join.hive`.

b. We will break the join statement down into sections. First, the result of the join is going to be put into the `stock_aggregates` table, which requires an insert:

```
insert overwrite table stock_aggregates
```

The overwrite causes any existing data in `stock_aggregates` to be deleted.

c. The data being inserted is going to be the result of a select query that contains various insightful indicators about each stock. The result is going to contain the stock symbol, date traded, maximum high for the stock, minimum low, average close, and the sum of dividends, as shown here:

```
select a.symbol, year(a.trade_date), max(a.high), min(a.low),
avg(a.close), sum(b.dividend)
```

d. The from clause is the `nyse_data` table:

```
from nyse_data a
```

e. The join is going to be a left outer join of the dividends table:

```
left outer join dividends b
```

f. The join is by stock symbol and trade date:

```
on (a.symbol = b.symbol and a.trade_date = b.trade_date)
```

g. Let’s group the result by symbol and trade date:

```
group by a.symbol, year(a.trade_date);
```

h. Save your changes to `join.hive`.

3) Run the Query

a. Run the query and wait for Tez to execute:

```
# hive -f join.hive
```

b. How many total mappers and reducers does it take to perform this query?
Answer: Two mappers and one reducer.

4) Verify the Results

a. Enter the Hive Shell and run a select query to view the contents of `stock_aggregates`:

```
# hive

hive> select * from stock_aggregates;
```

The output should look like:

```
KYO  2004  90.9   66.25  75.79952   0.544
KYO  2005  78.45  62.58  72.042656  0.91999996
KYO  2006  98.01  71.73  85.80327   0.851
KYO  2007  110.01 81.09  3.737686   NULL
KYO  2008  100.78 45.41  79.6098    NULL
KYO  2009  93.2   52.98  77.04389   NULL
KYO  2010  93.83  85.94  90.71      NULL
stock_symbolNULLNULLNULLNULLNULL
```

5) List the contents of the `stock_aggregates` directory in HDFS. The `000000_0` file was created as a result of the join query:

```
hive> dfs -ls -R /apps/hive/warehouse/stock_aggregates/;
-rw-r--r--   3 root hdfs      41109
/apps/hive/warehouse/stock_aggregates/000000_0
```

a. View the contents of the `stock_aggregates` table using the cat command:

```
hive> dfs -cat /apps/hive/warehouse/stock_aggregates/000000_0;
```

Result

The `stock_aggregates` table is a joining of the daily stock prices and the quarterly dividend amounts on the date the dividend was announced, and the data in the table is an aggregate of various statistics like max high, min low, etc.

Lab: Computing ngrams of Emails in Avro Format

About this Lab

Objective: Use Hive to compute ngrams.

File locations: /root/devph/labs/Lab7.5

Successful outcome: A bigram of words found in a collection of Avro-formatted emails.

Before you begin: Your HDP 2.3 cluster should be up and running within your VM.

Related lesson: ***Hive Programming***

Lab Steps

1) View an Avro Schema

a. If not already done, open a Terminal in your VM and type "ssh sandbox".

b. Change directories to the `/root/devph/labs/Lab7.5` folder. Notice this folder contains an Avro file named `sample.avro`.

```
# cd ~/devph/labs/Lab7.5

# ls sample.avro
```

c. Enter the following command to view the schema of the contents of `sample.avro`:

```
# avro cat --print-schema sample.avro
```

d. How many fields do records in `sample.avro` have?

Answer: Four fields

e. Create a schema file for `sample.avro`:

```
# avro cat --print-schema sample.avro > sample.avsc
```

f. Put the schema file in HDFS:

```
# hdfs dfs -put sample.avsc
```

2) Create a Hive Table from an Avro Schema

a. **View the contents of the `CREATE TABLE` query defined in the `create_sample_table.hive` file in your Computing ngrams of Emails in Avro Format lab folder:**

```
# more create_sample_table.hive
```

b. **Make sure the `avro.schema.url` property points to the schema file you created in the previous step:**

```
WITH SERDEPROPERTIES (
   'avro.schema.url'='hdfs:///user/root/sample.avsc')
```

c. **Run the `CREATE TABLE` query:**

```
# hive -f create_sample_table.hive
```

3) Verify the Table

a. **Start the Hive shell.**

```
# hive
```

b. **Run the `show tables` command and verify that you have a table named `sample_table`.**

```
hive> show tables;
```

c. **Run the describe command on `sample_table`. Notice the schema for `sample_table` matches the Avro schema from `sample.avsc`.**

```
hive> describe sample_table;
```

d. **Let's associate some data with `sample_table`. Copy `sample.avro` into the Hive warehouse folder by running the following command (all on a single line):**

```
hive> dfs -put /root/devph/labs/Lab7.5/sample.avro
/apps/hive/warehouse/sample_table;
```

e. **View the contents of `sample_table, then quit the Hive Shell`:**

```
hive> select * from sample_table;
OK
Foo   19   10, Bar Eggs Spam 800

hive> quit;
```

Note that there is only one record in `sample.avro`. You have now seen how to create a Hive table using an Avro schema file. This was a simple example; next you will complete these steps using a large data file that contains emails in an Avro format.

4) Create an Email-User Table

a. There is an Avro file in your /root/devph/labs/Lab7.5 folder named `mbox7.avro`, which represents emails in an Avro format from a Hive mailing list for the month of July. Use the `--print-schema` option of `avro` to view the schema of this file.

```
# avro cat --print-schema mbox7.avro
```

b. How many fields do records in `mbox7.avro` have?

Answer: Four fields

c. Run the `--print-schema` command again, but this time output the schema to a file named `mbox.avsc`:

```
# avro cat --print-schema mbox7.avro > mbox.avsc
```

d. Put the Avro schema file into `/user/root` in HDFS:

```
# hdfs dfs -put mbox.avsc
```

e. Use `more` to view the contents of the `create_email_table.hive` ;:script in your `/root/devph/labs/Lab7.5` folder. Verify the `avro.schema.url` property is correct.

```
# more create_email_table.hive
```

f. Run the script to create the `hive_user_email` table:

```
# hive -f create_email_table.hive
```

g. Copy `mbox7.avro` into the warehouse directory:

```
# hdfs dfs -put mbox7.avro /apps/hive/warehouse/hive_user_email
```

h. Start the Hive shell and verify the table has data in it:

```
# hive

hive> select * from hive_user_email limit 20;
```

5) Compute a Bigram

a. Use the Hive `ngrams` function to create a bigram of the words in `mbox7.avro`:

```
hive> select
   ngrams(sentences(content),2 ,10)
   from hive_user_email;
```

The output will be kind of a jumbled mess:

```
[{"ngram":["2013","at"],"estfrequency":802.0},{"ngram":["of","the"
],"estfrequency":391.0},{"ngram":["I","am"],"estfrequency":368.0},
{"ngram":["I","have"],"estfrequency":340.0},{"ngram":["J","E9r"],"
estfrequency":306.0},{"ngram":["for","the"],"estfrequency":291.0},
{"ngram":["you","are"],"estfrequency":289.0},{"ngram":["user","hiv
e.apache.org"],"estfrequency":289.0},{"ngram":["to","the"],"estfre
quency":276.0},{"ngram":["E9r","F4me"],"estfrequency":270.0}]
```

b. To clean this up, use the Hive explode function to display the output in a more readable format:

```
hive> select
  explode(ngrams(sentences(content),2 ,10))
  from hive_user_email;
```

You should see a nice, readable list of 10 bigrams:

```
{"ngram":["2013","at"],"estfrequency":802.0}
{"ngram":["of","the"],"estfrequency":391.0}
{"ngram":["I","am"],"estfrequency":368.0}
{"ngram":["I","have"],"estfrequency":340.0}
{"ngram":["J","E9r"],"estfrequency":306.0}
{"ngram":["for","the"],"estfrequency":291.0}
{"ngram":["you","are"],"estfrequency":289.0}
{"ngram":["user","hive.apache.org"],"estfrequency":289.0}
{"ngram":["to","the"],"estfrequency":276.0}
{"ngram":["E9r","F4me"],"estfrequency":270.0}
```

c. Typically when working with word comparison we ignore case. Run the query again, but this time add the Hive `lower` function and compute 20 bigrams:

```
hive> select
  explode(ngrams(sentences(lower(content)),2 ,20))
  from hive_user_email;
```

The output should look like the following:

```
{"ngram":["2013","at"],"estfrequency":802.0}
{"ngram":["i","have"],"estfrequency":409.0}
{"ngram":["of","the"],"estfrequency":391.0}
{"ngram":["i","am"],"estfrequency":372.0}
{"ngram":["if","you"],"estfrequency":347.0}
{"ngram":["in","hive"],"estfrequency":337.0}
{"ngram":["for","the"],"estfrequency":309.0}
{"ngram":["j","e9r"],"estfrequency":306.0}
{"ngram":["you","are"],"estfrequency":289.0}
{"ngram":["user","hive.apache.org"],"estfrequency":289.0}
{"ngram":["to","the"],"estfrequency":276.0}
{"ngram":["outer","join"],"estfrequency":271.0}
{"ngram":["2013","06"],"estfrequency":270.0}
{"ngram":["e9r","f4me"],"estfrequency":270.0}
{"ngram":["left","outer"],"estfrequency":270.0}
{"ngram":["in","the"],"estfrequency":252.0}
{"ngram":["gmail.com","wrote"],"estfrequency":248.0}
{"ngram":["17","16"],"estfrequency":248.0}
{"ngram":["06","17"],"estfrequency":246.0}
{"ngram":["wrote","hi"],"estfrequency":234.0}
```

6) Compute a Context `ngram`

a. From the Hive shell, run the following query, which uses the `context_ngrams` function to find the top 20 terms that follow the word "error":

```
hive> select
explode(context_ngrams(sentences(lower(content)),
array("error", null) ,20))
from hive_user_email;
```

The output should look like the following:

```
{"ngram":["in"],"estfrequency":102.0}
{"ngram":["return"],"estfrequency":97.0}
{"ngram":["org.apache.hadoop.hive.ql.exec.udfargumenttypeexception
"],"estfrequency":49.0}
{"ngram":["failed"],"estfrequency":49.0}
{"ngram":["is"],"estfrequency":41.0}
{"ngram":["message"],"estfrequency":40.0}
{"ngram":["when"],"estfrequency":39.0}
{"ngram":["please"],"estfrequency":36.0}
{"ngram":["while"],"estfrequency":28.0}
{"ngram":["org.apache.thrift.transport.ttransportexception"],"estf
requency":28.0}
{"ngram":["datanucleus.plugin"],"estfrequency":26.0}
{"ngram":["during"],"estfrequency":18.0}
{"ngram":["query"],"estfrequency":16.0}
{"ngram":["hive"],"estfrequency":16.0}
{"ngram":["could"],"estfrequency":16.0}
{"ngram":["java.lang.runtimeexception"],"estfrequency":13.0}
{"ngram":["13"],"estfrequency":12.0}
{"ngram":["error"],"estfrequency":12.0}
{"ngram":["exec.execdriver"],"estfrequency":10.0}
{"ngram":["exec.task"],"estfrequency":10.0}
```

b. What is the most likely word to follow “error” in these emails?

   Answer: “in”

c. Run a Hive query that finds the top 20 results for words in mbox7.avro that follow the phrase “error in.”

   Solution:

```
select
  explode(context_ngrams(sentences(lower(content)),
      array("error", "in", null) ,20))
  from hive_user_email;
```

Result

You have written several Hive queries that computed bigrams based on the data in the `mbox7.avro` file. You should also be familiar with working with Avro files, a popular file format in Hadoop.

Lab: Using HCatalog with Pig

About this Lab

Objective: Use HCatalog to provide the schema for a Pig relation.

File locations: n/a

Successful outcome: You will have written a Pig script that uses HCatLoader to retrieve a schema from an HCatalog table and HCatStorer to write data to a table managed by HCatalog.

Before you begin: Your HDP 2.3 cluster should be up and running within your VM. Complete the Understanding Hive Tables Lab.

Related lesson: ***Using HCatalog***

Lab Steps

1) Start the Grunt Shell

a. If not already done, open a Terminal in your VM and type "ssh sandbox".
b. Start the Grunt shell for use with HCatalog:

```
# pig -useHCatalog
```

2) Load an HCatalog Table

a. Define a relation for the `wh_visits` table in Hive using the `HCatLoader()`:

```
grunt> visits = LOAD 'wh_visits' USING
org.apache.hive.hcatalog.pig.HCatLoader();
```

b. View the schema of the `visits` relation to verify that it matches the schema of the `wh_visits` table:

```
grunt> describe visits;

visits: {lname: chararray,fname: chararray,time_of_arrival:
chararray,appt_scheduled_time: chararray,meeting_location:
chararray,info_comment: chararray}
```

3) Run a Pig Query

a. Let’s execute a query to verify that everything is working. Define the following relation:

```
grunt> joe = FILTER visits BY (fname == 'JOE');
```

b. Dump the relation:

```
grunt> DUMP joe;
```

The output should be visitors from `wh_visits` with the first name “`JOE`.”

4) Create an HCatalog Schema

a. Quit the Grunt shell and start the Hive shell.

```
grunt> quit;

# hive
```

b. An HCatalog schema is essentially just a table in the Hive metastore. To define a schema for use with HCatalog, create a table in Hive:

```
hive> create table joes (fname string, lname string, comments
string);
```

c. Verify that the table was created successfully using ‘`show tables`.’

```
hive> show tables;
```

d. Use the `describe` command to view the schema of joes:

```
hive> describe joes;
OK
fname                string
lname                string
comments             string
```

5) Using HCatStorer

a. Exit the Hive shell and start the Grunt shell again. Be sure to use the `useHCatalog` option:

```
hive> quit;

# pig -useHCatalog
```

b. Define the visits and joe relations again (using the up arrow to browse through the history of Pig commands).

c. In this step, you will use HCatStorer in Pig to input records into the joes table. To do this, you need a relation whose fields match the schema of joes. You can accomplish this using a projection. Define the following relation:

```
grunt> project_joe = FOREACH joe GENERATE fname, lname,
info_comment;
```

d. Store the projection into the HCatalog table using the `STORE` command:

```
grunt> STORE project_joe INTO 'joes' USING
org.apache.hive.hcatalog.pig.HCatStorer();
```

This command failed. Why? ________________

Answer: The initial `STORE` command failed because the field names in the relation you were trying to store did not match the column names of the underlying table's schema.

e. Notice that the projection has fields named `fname, lname`, and `info_comment`, but the joes table in HCatalog has a schema with `fname, lname`, and `comments`. The `fname` and `lname` fields match, but `info_comment` needs to be renamed to comments. Modify your projection by using the `AS` keyword:

```
grunt> project_joe = FOREACH joe GENERATE fname, lname,
info_comment AS comments;
```

f. Now run the `STORE` command again:

```
grunt> STORE project_joe INTO 'joes' USING
org.apache.hive.hcatalog.pig.HCatStorer();
```

This time the command should work and a MapReduce job will execute.

6) Verify that the `STORE` Worked

a. Quit the Grunt shell and start the Hive shell again.

```
grunt> quit;

# hive
```

b. View the contents of the joes table:

```
hive> select * from joes;
```

You should see visitors all named "`JOE`," along with their last name and the comments.

7) View the Files

a. You can also check the file system to see if a `STORE` command worked:

```
hive> dfs -ls /apps/hive/warehouse/joes/;
Found 1 items
-rw-r--r--   1 root hdfs        896 /apps/hive/warehouse/joes/part-m-00000
```

Notice that the file for the joes table is named `part-m-00000`. Where did that name come from?

Answer: The `part-m-00000` file is a result of the Pig MapReduce job that executed when you ran the `STORE` command with HCatStorer.

b. Use the cat command to view the contents of `part-m-00000`:

```
hive> dfs -cat /apps/hive/warehouse/joes/part-m-00000;
```

As you can see, this is the same list of names from the Hive select * query, which should be no surprise at this point in the course.

Result

You have seen how to run a Pig script that uses HCatalog to provide the schema using HCatLoader and HCatStorer.

Lab: Advanced Hive Programming

About this Lab

Objective: To understand how some of the more advanced features of Hive work, including multi-table inserts, views, and windowing.

File locations: /root/devph/labs/Lab9.1

Successful outcome: You will have executed numerous Hive queries on customer order data.

Before you begin: Your HDP 2.3 cluster should be up and running within your VM.

Related lesson: ***Advanced Hive Programming***

Lab Steps

1) Create and Populate a Hive Table

a. If not already done, open a Terminal in your VM and type "ssh sandbox".

b. From the command line, change directories to `/root/devph/labs/Lab9.1` folder:

```
# cd ~/devph/labs/Lab9.1
```

c. View the contents of the `orders.hive` file in that folder:

```
# more orders.hive
```

Notice it defines a Hive table named `orders` that has seven columns.

d. Execute the contents of `orders.hive`:

```
# hive -f orders.hive
```

e. From the Hive shell, verify that the script worked by running the following commands:

```
# hive

hive> describe orders;
hive> select count(*) from orders;
```

Your orders table should contain 99,999 records.

2) Analyze the Customer Data

a. Let’s run a few queries to see what this data looks like. Start by verifying that the username column actually looks like names:

```
hive> SELECT username FROM orders LIMIT 10;
```

You should see 10 first names.

b. The orders table contains orders placed by customers. Run the following query, that shows the 10 lowest-price orders:

```
hive> SELECT username, ordertotal FROM orders ORDER BY ordertotal
LIMIT 10;
```

The smallest orders are each $10, as you can see from the output:

```
Chelsea    10
Samantha   10
Danielle   10
Kimberly   10
Tiffany    10
Megan      10
Maria      10
Megan      10
Melissa    10
Christina  10
```

c. Run the same query, but this time use descending order:

```
hive> SELECT username, ordertotal FROM orders ORDER BY ordertotal
DESC LIMIT 10;
```

The output this time is the 10 highest-priced orders:

```
Brandon     612
Mark        612
Sean        612
Jordan      612
Anthony     612
Paul        611
Jonathan    611
Eric        611
Nathan      611
Jordan      610
```

d. Let’s find out if men or women spent more money:

```
hive> SELECT sum(ordertotal), gender FROM orders GROUP BY gender;
```

Based on the output, which gender has spent more money on purchases?

Answer: Men spent $9,919,847, and women spent $9,787,324.

e. The `orderdate` column is a string with the format yyyy-mm-dd. Use the year function to extract the various parts of the date. For example, run the following query, which computes the sum of all orders for each year:

```
hive> SELECT sum(ordertotal), year(order_date) FROM orders GROUP
BY year(order_date);
```

The output should look like this. **Verify, then quit the Hive shell**:

```
4082780  2009
4404806  2010
4399886  2011
4248950  2012
2570749  2013

hive> quit;
```

3) Multi-File Insert

a. In this step, you will run two completely different queries, but in a single job. The output of the queries will be in two separate directories in HDFS. Start by using `gedit` to create a new text file in the `/root/devph/labs/Lab9.1` folder named `multifile.hive`.

b. Within the text file, enter the following query. Notice there is no semicolon between the two `INSERT` statements:

```
FROM ORDERS o
INSERT OVERWRITE DIRECTORY '2010_orders'
SELECT o.* WHERE year(order_date) = 2010
INSERT OVERWRITE DIRECTORY 'software'
SELECT o.* WHERE itemlist LIKE '%Software%';
```

c. Save your changes to `multifile.hive`.

d. Run the query from the command line:

```
# hive -f multifile.hive
```

e. The above query executes in a single job. Even more interesting, it only requires a `map` phase.

Why did this job not require a `reduce` phase?

Answer: Because the query only does a `SELECT *`, no `reduce` phase was needed.

f. Verify that the two queries executed successfully by viewing the folders in HDFS:

```
# hdfs dfs -ls
```

You should see two new folders: `2010_orders` and `software`.

g. View the output files in these two folders. Verify that the `2010_orders` directory contains orders from only the year 2010, and verify that the software directory contains only orders that included ‘`Software`.’

```
# hdfs dfs -ls 2010_orders

# hdfs dfs -cat 2010_orders/000000_0

# hdfs dfs -ls software

# hdfs dfs -cat software/000000_0
```

4) Define a View

a. Start the Hive shell. Define a view named `2013_orders` that contains the `orderid`, `order_date, username`, and `itemlist` columns of the orders table where the `order_date` was in the year 2013.

Solution: The `2013_orders` view:

```
# hive

hive> CREATE VIEW 2013_orders AS
SELECT orderid, order_date, username, itemlist
FROM orders
WHERE year(order_date) = '2013';
```

b. Run the `show tables` command:

```
hive> show tables;
```

You should see `2013_orders` in the list of tables.

c. To verify your view is defined correctly, run the following query:

```
hive> SELECT COUNT(*) FROM 2013_orders;
```

The `2013_orders` view should contain 13,104 records.

5) Find the Maximum Order of Each Customer

a. Suppose you want to find the maximum order of each customer. This can be done easily enough with the following Hive query. Run this query now:

```
hive> SELECT max(ordertotal), userid
FROM orders GROUP BY userid;
```

b. How many different customers are in the orders table? ____________

Answer: There are 100 unique customers in the orders table.

c. Suppose you want to add the `itemlist` column to the previous query. Try adding it to the `SELECT` clause by the following method and see what happens:

```
hive> SELECT max(ordertotal), userid, itemlist
FROM orders GROUP BY userid;
```

Notice this query is not valid because `itemlist` is not in the `GROUP BY` key.

d. We can join the result set of the max-total query with the orders table to add the `itemlist` to our result. Start by defining a view named `max_ordertotal` for the maximum order of each customer:

```
hive> CREATE VIEW max_ordertotal AS
SELECT max(ordertotal) AS maxtotal, userid
FROM orders GROUP BY userid;
```

e. Now join the orders table with your `max_ordertotal` view:

```
hive> SELECT ordertotal, orders.userid, itemlist
FROM orders
JOIN max_ordertotal ON
max_ordertotal.userid = orders.userid
AND
max_ordertotal.maxtotal = orders.ordertotal
ORDER BY orders.userid;
```

f. What did the Tez job look like for this query? __________

Answer: The query resulted in a map-reduce-reduce Tez job.

The end of your output should look like:

```
600  98
Grill,Freezer,Bedding,Headphones,DVD,Table,Grill,Software,Dishwash
er,DVD,Microwave,Adapter
600  99Washer,Cookware,Vacuum,Freezer,2-Way Radio,Bicycle,Washer &
Dryer,Coffee Maker,Refrigerator,DVD,Boots,DVD
600  100Bicycle,Washer,DVD,Wrench Set,Sweater,2-Way
Radio,Pants,Freezer,Blankets,Grill,Adapter,pillows
```

6) Fixing the `GROUP BY` Key Error

a. Let's compute the sum of all of the orders of all of the customers. Start by entering the following query:

```
hive> SELECT sum(ordertotal), userid FROM orders GROUP BY userid;
```

Notice that the output is the sum of all orders, but displaying just the `userid` is not very exciting.

b. Try to add the `username` column to the `SELECT` clause in the following manner and see what happens:

```
hive> SELECT sum(ordertotal), userid, username
FROM orders
GROUP BY userid;
```

This generates an "Expression not in `GROUP BY` key" error, because the `username` column is not being aggregated but the `ordertotal` is.

c. An easy fix is to aggregate the username values using the `collect_set` function, but output only one of them:

```
hive> SELECT sum(ordertotal), userid, collect_set(username)[0]
FROM orders
GROUP BY userid;
```

You should get the same output as before, but this time the `username` is included.

7) Using the `OVER` Clause

a. Now let's compute the sum of all orders for each customer, but this time use the `OVER` clause to not group the output and to also display the `itemlist` column:

```
hive> SELECT userid, itemlist, sum(ordertotal)
OVER (PARTITION BY userid)
FROM orders;
```

Notice the output contains every order, along with the items they purchased and the sum of all of the orders ever placed from that particular customer.

8) Using the Window Functions

a. It is not difficult to compute the sum of all orders for each day using the `GROUP BY` clause:

```
hive> select order_date, sum(ordertotal)
FROM orders
GROUP BY order_date;
```

Run the query above and the tail of the output should look like:

```
2013-07-28  18362
2013-07-29  3233
2013-07-30  4468
2013-07-31  4714
```

b. Suppose you want to compute the sum for each day that includes each order. This can be done using a window that sums all previous orders along with the current row:

```
hive> SELECT order_date, sum(ordertotal)
OVER
(PARTITION BY order_date ROWS BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW)
FROM orders;
```

To verify that it worked, your tail of your output should look like:

```
2013-07-31  3163
2013-07-31  3415
2013-07-31  3607
2013-07-31  4146
2013-07-31  4470
2013-07-31  4610
2013-07-31  4714
```

9) Using the Hive Analytics Functions

a. Run the following query, which displays the rank of the `ordertotal` by day:

```
hive> SELECT order_date, ordertotal, rank()
OVER
(PARTITION BY order_date ORDER BY ordertotal)
FROM orders;
```

b. To verify it worked, the output of July 31, 2013, should look like:

```
2013-07-31  48   1
2013-07-31  104  2
2013-07-31  119  3
2013-07-31  130  4
2013-07-31  133  5
2013-07-31  135  6
2013-07-31  140  7
2013-07-31  147  8
2013-07-31  156  9
2013-07-31  192  10
2013-07-31  192  10
2013-07-31  196  12
2013-07-31  240  13
2013-07-31  252  14
2013-07-31  296  15
2013-07-31  324  16
2013-07-31  343  17
2013-07-31  500  18
2013-07-31  528  19
2013-07-31  539  20
```

c. As a challenge, see if you can run a query similar to the previous one except compute the rank over months instead of each day.

Solution: The rank query by month:

```
SELECT substr(order_date,0,7), ordertotal, rank()
OVER
(PARTITION BY substr(order_date,0,7) ORDER BY ordertotal)
FROM orders;
```

10) Histograms

a. Run the following Hive query, which uses the `histogram_numeric` function to compute 20 (x,y) pairs of the frequency distribution of the total order amount from customers who purchased a microwave (using the orders table):

```
hive> SELECT explode(histogram_numeric(ordertotal,20)) AS x
FROM orders
WHERE itemlist LIKE "%Microwave%";
```

The output should look like the following:

```
{"x":14.333333333333332,"y":3.0}
{"x":33.87755102040816,"y":441.0}
{"x":62.52577319587637,"y":679.0}
{"x":89.37823834196874,"y":965.0}
{"x":115.1242236024843,"y":1127.0}
{"x":142.6468885672939,"y":1382.0}
{"x":174.07664233576656,"y":1370.0}
{"x":208.06909090909105,"y":1375.0}
{"x":242.55486381322928,"y":1285.0}
{"x":275.8625954198475,"y":1048.0}
{"x":304.71100917431284,"y":872.0}
{"x":333.1514423076924,"y":832.0}
{"x":363.7630208333335,"y":768.0}
{"x":397.51587301587364,"y":756.0}
{"x":430.9072847682117,"y":604.0}
{"x":461.68715083798895,"y":537.0}
{"x":494.1598360655734,"y":488.0}
{"x":528.5816326530613,"y":294.0}
{"x":555.5166666666672,"y":180.0}
{"x":588.7979797979801,"y":198.0}
```

b. Write a similar Hive query to compute 10 frequency-distribution pairs for the `ordertotal` from orders table where `ordertotal` is greater than $200.

```
SELECT explode(histogram_numeric(ordertotal,10)) AS x
FROM orders
WHERE ordertotal > 200;


{"x":218.8195174551819,"y":7419.0}
{"x":254.10237580993478,"y":6945.0}
{"x":293.4231618807192,"y":6338.0}
{"x":334.57302573203015,"y":5635.0}
{"x":379.79714934930786,"y":4841.0}
{"x":428.1165628891644,"y":4015.0}
{"x":473.1484734420741,"y":2391.0}
{"x":511.2576946288467,"y":1657.0}
{"x":549.0106899902812,"y":1029.0}
{"x":589.0761194029857,"y":670.0}
```

Result

You should now be comfortable running Hive queries and using some of the more advanced features of Hive, like views and the window functions.

Lab: Running a YARN Application

About this Lab

Objective:	To run a YARN application.
File locations:	n/a
Successful outcome:	You will have executed the DistributedShell YARN application.
Before you begin:	Your HDP 2.3 cluster should be up and running within your VM.
Related lesson:	***Hadoop 2 and YARN***

Lab Steps

1) Run a `DistributedShell` Application

a. If not already done, open a Terminal in your VM and type "ssh sandbox".

b. In a terminal window change directories to the `/usr/hdp/current/hadoop-yarn-client` folder:

```
# cd /usr/hdp/current/hadoop-yarn-client/
```

c. Run the following command, which runs a sample `YARN` application that ships with HDP 2.x:

```
# yarn jar hadoop-yarn-applications-distributedshell.jar
org.apache.hadoop.yarn.applications.distributedshell.Client -jar
hadoop-yarn-applications-distributedshell.jar -shell_command cal
```

The `DistributedShell` command allows you to run a script or `shell` command on your cluster. The example above runs the Unix “`cal`” command, which displays a text calendar.

d. Wait for the YARN job to finish.

2) Verify the Result

a. Enter the following command (all on a single line):

```
# yarn application -list -appStates FINISHED | grep Dist
```

You should see the application ID of the `DistributedShell` command that you just ran:

```
application_1378331467073_0004 DistributedShell    YARN      yarn
default    FINISHED   SUCCEEDED       100%
```

b. Copy and paste the application `ID` of your DistributedShell command and check its status using the following command (but replacing the ID shown here with your actual application ID):

```
# yarn application -status application_1378331467073_0004
```

Notice that the details of the job are displayed. This was a simple application, so there is not a lot of information to analyze:

```
Application Report :
   Application-Id : application_1378331467073_0004
   Application-Name : DistributedShell
   Application-Type : YARN
   User : root
   Queue : default
   Start-Time : 1408060384688
   Finish-Time : 1408060391340
   Progress : 100%
   State : FINISHED
   Final-State : SUCCEEDED
   Tracking-URL : N/A
   RPC Port : -1
   AM Host : sandbox.hortonworks.com/172.16.173.149
   Diagnostics :
```

Note

The YARN application command also has a -kill option (followed by the application's ID) that kills a running YARN job. This is a great tool when you have submitted a job and then need to stop it before it runs to completion.

3) View the Log File

a Enter the following command to view the output for this YARN application that you just executed:

```
# yarn logs -applicationId application_1378331467073_0004
```

Important

The -applicationId must have the correct case for every letter, or else you will see an error message stating that it is missing. The only capital letter in the option is the I. The d, and all other letters, are lower case. Somewhere in the log file, you should see a text calendar of the current month. For example:

```
LogType: stdout
LogLength: 150
Log Contents:
   August 2014
Su Mo Tu We Th Fr Sa
2
4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

4) Optional: Run the Job in Six Containers

a. The DistributedShell application allows you to specify how many containers the ApplicationMaster uses. Add the following arguments to the end of the YARN command from Step 1.2:

```
-num_containers 6 -container_memory 20
```

b. Now find the applicationID and view the aggregated log file:

```
# yarn application -list -appStates FINISHED | grep Dist

# yarn logs -applicationId <applicationId>
```

You should see six calendars this time, one from each container.

c. Notice that this also demonstrates how the log output from multiple containers is aggregated into a single, convenient log file.

Result

In this lab you ran a simple YARN application called the DistributedShell (that ships with HDP 2.x). You also saw how to view the output of the aggregated log file using the YARN logs command.

Lab: Getting Started with Apache Spark

About this Lab

Objective: Read and manipulate HDFS files with Spark.

File locations: /root/devph/labs/Spark

Successful outcome: You will have processed several HDFS file via Spark Core.

Before you begin: Your HDP 2.3 cluster should be up and running within your VM.

Related lesson: ***Programming with Apache Spark***

Lab Steps

1) Execute a WordCount with Spark.

a. If not already done, open a Terminal in your VM and type "ssh sandbox".
b. Copy the constitution.txt file to HDFS.

```
[root@sandbox ~]# cd ~/devph/labs/Spark
[root@sandbox Spark]# hdfs dfs -mkdir spark
[root@sandbox Spark]# hdfs dfs -put
~/devph/labs/demos/constitution.txt spark/
[root@sandbox Spark]# hdfs dfs -ls spark
Found 1 items
-rw-r--r--   1 root hdfs      45489 2015-11-09 09:39
spark/constitution.txt
```

c. **Launch the Python Spark Shell. NOTE: "INFO" comments will be removed from the output in this lab guide going forward.**

```
[root@sandbox ~]# pyspark
Python 2.6.6 (r266:84292, Jul 23 2015, 15:22:56)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-11)] on linux2

... lines removed ...

Welcome to
____              __
/ __/__  ___ _____/ /__
_\ \/ _ \/ _ `/ __/  '_/
/__ / .__/\_,_/_/ /_/\_\   version 1.4.1
/_/

Using Python version 2.6.6 (r266:84292, Jul 23 2015 15:22:56)
SparkContext available as sc, HiveContext available as sqlContext.
>>>
```

d. **Read in the file as a RDD.**

```
>>> baseFile =
sc.textFile("hdfs://sandbox:8020/user/root/spark/constitution.txt"
)
>>> baseFile.take(1)
[u'We the People of the United States, in Order to form a more
perfect ']
```

e. **Break the full lines down into a collection of words.**

```
>>> justWords = baseFile.flatMap(lambda line: line.split(' '))
>>> justWords.take(5)
[u'We', u'the', u'People', u'of', u'the']
```

f. **Map the words with a count of "1" for each.**

```
>>> mappedWords = justWords.map(lambda word: (word, 1))
>>> mappedWords.take(5)
[(u'We', 1), (u'the', 1), (u'People', 1), (u'of', 1), (u'the', 1)]
```

g. **Count up the words; sorting them in reverse.**

```
>>> wordCounts = mappedWords.reduceByKey(lambda a,b:
a+b).sortByKey(ascending=False)
>>> wordCounts.take(10)
[(u'years;', 1), (u'years', 9), (u'year,', 1), (u'year', 1),
(u'written', 6), (u'writs', 1), (u'writing,', 1), (u'would', 2),
(u'work', 1), (u'witnesses', 2)]
```

h. Chain all the method invocations into a single operation, as is more the normal usage pattern. NOTE: Type as a single line without the new line or "\" characters.

```
>>> asOneLine =
sc.textFile("hdfs://sandbox:8020/user/root/spark/constitution.txt"
) \
   .flatMap(lambda line: line.split(' ')) \
   .map(lambda word: (word, 1)) \
   .reduceByKey(lambda a,b: a+b) \
   .sortByKey(ascending=False)
>>> asOneLine.take(10)
[(u'years;', 1), (u'years', 9), (u'year,', 1), (u'year', 1),
(u'written', 6), (u'writs', 1), (u'writing,', 1), (u'would', 2),
(u'work', 1), (u'witnesses', 2)]
```

i. Exit out of the pyspark REPL.

```
>>> quit()
```

2) On a simple customer file, find the top 5 states with the most male customers.

a. Upload customer.csv and explore its format of name, gender, state and duration.

```
[root@sandbox Spark]# hdfs dfs -put customer.csv spark
[root@sandbox Spark]# hdfs dfs -tail spark/customer.csv
celia,F,Maryland,3.97
Evalyn,F,Pennsylvania,2.1
Jeneva,F,Nebraska,9.26
Kelsey,F,Minnesota,8.68
Daine,F,Nebraska,6.34

   ... lines removed ...

Annamae,F,Nebraska,9.11
Racheal,F,Wisconsin,9.65
Ellan,F,Michigan,5.82
```

b. Launch pyspark and read the file.

```
[root@sandbox labs]# pyspark
>>> custFile =
sc.textFile("hdfs://sandbox:8020/user/root/spark/customer.csv")
>>> custFile.take(3)
[u'Irvin,M,Maryland,5.06', u'Owen,M,Illinois,2.01',
u'August,M,Illinois,1.42']
```

c. **Filter out just the male customers.**

```
>>> justMales = custFile.map(lambda line:
line.split(',')).filter(lambda line: line[1] == 'M')
>>> justMales.take(2)
[[u'Irvin', u'M', u'Maryland', u'5.06'], [u'Owen', u'M',
u'Illinois', u'2.01']]
```

d. **Create Key-Value-Pairs (KVP) with state being the key and the number "1" being the value.**

```
>>> mapByStateCode = justMales.map(lambda line: (line[2], 1))
>>> mapByStateCode.take(4)
[(u'Maryland', 1), (u'Illinois', 1), (u'Illinois', 1), (u'New
Jersey', 1)]
```

e. **Count up the number of customers by state.**

```
>>> nbrCustsByState = mapByStateCode.reduceByKey(lambda a,b: a+b)
>>> nbrCustsByState.take(4)
[(u'Wisconsin', 2), (u'New Jersey', 4), (u'Michigan', 8),
(u'Pennsylvania', 2)]
```

f. **Flip the KVP so that the count is first and order that from highest to lowest.**

```
>>> highToLowCountAndState = nbrCustsByState.map(lambda (a,b):
(b,a)).sortByKey(ascending=False)
>>> highToLowCountAndState.take(6)
[(8, u'Michigan'), (8, u'Maryland'), (7, u'Illinois'), (5,
u'Nebraska'), (4, u'New Jersey'), (4, u'Indiana')]
```

g. **Flip the KVP pair back to state and count plus add an index to represent the ordering sequence.**

```
>>> stateCountIndexedHighToLow = highToLowCountAndState.map(lambda
(a,b): (b,a)).zipWithIndex()
>>> stateCountIndexedHighToLow.take(6)
[((u'Michigan', 8), 0), ((u'Maryland', 8), 1), ((u'Illinois', 7),
2), ((u'Nebraska', 5), 3), ((u'New Jersey', 4), 4), ((u'Indiana',
4), 5)]
```

h. **Eliminate all records except for the top 5.**

```
>>> topFive = stateCountIndexedHighToLow.filter(lambda ((a,b),c):
c<5)
>>> topFive.collect()
[((u'Michigan', 8), 0), ((u'Maryland', 8), 1), ((u'Illinois', 7),
2), ((u'Nebraska', 5), 3), ((u'New Jersey', 4), 4)]
```

i. Eliminate the index and the counts to just return the top 5 state names.

```
>>> top5Names = topFive.map(lambda ((a,b),c): a)
>>> top5Names.collect()
[u'Michigan', u'Maryland', u'Illinois', u'Nebraska', u'New
Jersey']
```

j. As before, chain all the method invocations into a single operation as is more the normal usage pattern.

```
>>>
sc.textFile("hdfs://sandbox:8020/user/root/spark/customer.csv") \
   .map(lambda line: line.split(',')) \
   .filter(lambda line: line[1] == 'M') \
   .map(lambda line: (line[2], 1)) \
   .reduceByKey(lambda a,b: a+b) \
   .map(lambda (a,b): (b,a)) \
   .sortByKey(ascending=False) \
   .map(lambda (a,b): (b,a)) \
   .zipWithIndex() \
   .filter(lambda ((a,b),c): c<5) \
   .map(lambda ((a,b),c): a) \
   .collect()
[u'Michigan', u'Maryland', u'Illinois', u'Nebraska', u'New
Jersey']
```

Result:

Successful use of Spark Core and RDD to read files and perform data analysis.

Lab: Exploring Spark SQL

About this Lab

Objective: Create DataFrame structure from Hive tables & HDFS files and utilize both the DataFrame API and SQL to refine returned data.

File locations: /root/devph/labs/Spark

Successful outcome: You will have successfully queries data from multiple DataFrame objects as well as joined them together.

Before you begin: Your HDP 2.3 cluster should be up and running within your VM.

Related lesson: ***Spark SQL and DataFrames***

Lab Steps

1) Run a query on an existing Hive table.

a. Load file into HDFS and create a Hive table mapping to it.

```
[root@sandbox Spark]# hdfs dfs -mkdir spark/cust_fav
[root@sandbox Spark]# hdfs dfs -put cust_fav.csv spark/cust_fav/
[root@sandbox Spark]# hive -f cust_fav.hive
[root@sandbox Spark]# hive -e 'select * from cust_fav limit 5;'
OK
Irvin Riesling
Owen Pinot Noir
August Sauvignon Blanc
Christian Merlot
Arlen Pinot Noir
Time taken: 3.023 seconds, Fetched: 5 row(s)
```

b. Create a DataFrame from querying the Hive table created in the prior step after starting up pyspark again.

```
>>> from pyspark.sql import HiveContext
>>> hc = HiveContext(sc)
>>> custFavDF = hc.sql("SELECT * FROM cust_fav")
>>> custFavDF.show(5)
+---------+---------------+
|cust_name|      wine_type|
+---------+---------------+
|    Irvin|       Riesling|
|     Owen|     Pinot Noir|
|   August|Sauvignon Blanc|
|Christian|         Merlot|
|    Arlen|     Pinot Noir|
+---------+---------------+
```

2) Use customer data from the prior lab to find average length of customers by gender and state.

a. Import the necessary Row definition and then create a RDD from the customer.csv file previously loaded into HDFS.

```
>>> from pyspark.sql import Row
>>> customerRaw =
sc.textFile("hdfs://sandbox:8020/user/root/spark/customer.csv")
>>> customerRaw.take(2)
[u'Irvin,M,Maryland,5.06', u'Owen,M,Illinois,2.01']
```

b. Break each long string representing a row from the input file into discrete customer records.

```
>>> customerRecords = customerRaw.map(lambda line:
line.split(','))
>>> customerRecords.take(2)
[[u'Irvin', u'M', u'Maryland', u'5.06'], [u'Owen', u'M',
u'Illinois', u'2.01']]
```

c. Convert that the RDD to a DataFrame.

```
>>> customerDF = customerRecords.map(lambda c: Row(name=c[0],
gender=c[1], state=c[2], length=float(c[3]))).toDF()
>>> customerDF.show(2)
+------+------+-----+--------+
|gender|length| name|   state|
+------+------+-----+--------+
|     M|  5.06|Irvin|Maryland|
|     M|  2.01| Owen|Illinois|
+------+------+-----+--------+
```

d. Search for the final results with the DataFrame API.

```
>>>
customerDF.select("gender","state","length").groupBy("gender","state").a
vg("length").show()
+------+------------+------------------+
|gender|       state|       AVG(length)|
+------+------------+------------------+
|     F|    Illinois|              7.49|
|     F|  New Jersey|              4.04|
|     F|   Minnesota|             6.204|
|     F|    Michigan| 4.207142857142857|
|     M|     Indiana| 4.369999999999999|
|     M|    Maryland| 5.326250000000001|
|     M|    Nebraska|             6.516|
|     M|    Illinois| 4.858571428571428|
|     M|  New Jersey|4.6499999999999995|
|     M|   Minnesota|            5.4375|
|     F|   Wisconsin|           6.29125|
|     M|    Michigan|5.0337499999999995|
|     F|Pennsylvania|             5.575|
|     F|        Ohio|              5.93|
|     F|        Iowa| 7.430000000000001|
|     M|   Wisconsin|              0.33|
|     M|Pennsylvania|             3.665|
|     M|        Ohio|              4.25|
|     M|        Iowa| 5.753333333333333|
|     F|    Maryland| 5.003749999999999|
+------+------------+------------------+
```

3) Join the prior two DataFrames.

a. Utilize the DataFrame API to perform the join.

```
>>> customerDF.join(custFavDF, customerDF.name ==
custFavDF.cust_name).show(5)
+------+------+---------+----------+---------+---------------+
|gender|length|     name|     state|cust_name|      wine_type|
+------+------+---------+----------+---------+---------------+
|     M|  5.06|    Irvin|  Maryland|    Irvin|       Riesling|
|     M|  2.01|     Owen|  Illinois|     Owen|     Pinot Noir|
|     M|  1.42|   August|  Illinois|   August|Sauvignon Blanc|
|     M|  8.17|Christian|New Jersey|Christian|         Merlot|
|     M|  2.24|    Arlen|   Indiana|    Arlen|     Pinot Noir|
+------+------+---------+----------+---------+---------------+
```

Result

Successful creation of a DataFrame from a Hive tables and a HDFS file; as well as joining these two DataFrames.

Lab: Defining an Oozie Workflow

About this Lab

Objective: Define and run an Oozie workflow.

File locations: /root/devph/labs/Oozie

Successful outcome: You will run an Oozie job that executes a Pig script and a Hive script.

Before you begin: Your HDP 2.3 cluster should be up and running within your VM.

Related lesson: ***Defining Workflow with Oozie***

Lab Steps

1) Store the Job Data in HDFS

a. If not already done, open a Terminal in your VM and type "ssh sandbox".
b. Make sure you have `whitehouse/visits.txt` in HDFS:

```
# hdfs dfs -ls whitehouse
```

If not, the file (zipped within whitehouse_visits.zip) can be found in the `/root/devph/labs/Lab5.1` folder. The Oozie job assumes there is a file named `visits.txt` in a folder named `whitehouse` in HDFS.

2) Deploy the Oozie Workflow

a. Using the `gedit` text editor, open the file `/root/devph/labs/Oozie/workflow.xml`.
b. How many actions are in this workflow? ____________

Answer: Two

Which action will execute first? ________________

Answer: The Pig action named export_congress

If the first action is successful, which action will execute next? ___________

Answer: The Hive action named `define_congress_table`

To deploy this workflow, we need a directory in HDFS:

```
# cd ~/devph/labs/Oozie/
# hdfs dfs -mkdir congress
```

c. Copy `congress_visits.hive` and `whitehouse.pig` from the Oozie folder into the new congress folder in HDFS.

```
# hdfs dfs -put congress_visits.hive congress/congress_visits.hive
# hdfs dfs -put whitehouse.pig congress/whitehouse.pig
```

d. Also, put `workflow.xml` into the congress folder.

```
# hdfs dfs -put workflow.xml congress/workflow.xml
```

e. Verify that you have three files now in your congress folder in HDFS:

```
# hdfs dfs -ls congress
Found 3 items
-rw-r--r--   1 root root  429 congress/congress_visits.hive
-rw-r--r--   1 root root  580 congress/whitehouse.pig
-rw-r--r--   1 root root 1692 congress/workflow.xml
```

3) Deploy the Hive Configuration File

a. If you look at the Hive action in `workflow.xml`, you will notice that it references a file named `hive-site.xml` within the `<job-xml>` tag. This file represents the settings Oozie needs to connect to your Hive instance, and the file needs to be deployed in HDFS (using a relative path to the workflow directory). Make a copy of the original file into the working directory:

```
# cp /etc/hive/conf/hive-site.xml
```

To address an Oozie expression language issue and to ensure the workflow will run in the lab VM, use the gedit text editor to open /root/devph/labs/Oozie/hive-site.xml and delete the marked out stanza below:

~~<property>~~

~~<name>hive.server2.logging.operation.log.location</name>~~

~~<value>${system:java.io.tmpdir}/${system:user.name}/operation_logs~~
~~</value>~~

~~</property>~~

Also change the default execution engine from `tez` to `mr`:

```
<property>
<name>hive.execution.engine</name>
<value>mr</value>
</property>
```

b. **Verify that the differences form the original file and the modified one are correct by returning results similar to those listed below:**

```
# diff /etc/hive/conf/hive-site.xml hive-site.xml
250c250
<       <value>tez</value>
---
<value>mr</value>
544,548d543
<       <name>hive.server2.logging.operation.log.location</name>
<
<value>${system:java.io.tmpdir}/${system:user.name}/operation_logs
</value>
<     </property>
<
<     <property>
783c778
<   </configuration>
\ No newline at end of file
---
</configuration>
```

c. **Copy the modified `hive-site.xml` into the workflow directory:**

```
# hdfs dfs -put hive-site.xml congress/hive-site.xml
```

4) Define the `OOZIE_URL` Environment Variable

a. **Although not required, you can simplify `oozie` commands by defining the `OOZIE_URL` environment variable. From the command line, enter the following command:**

```
# export OOZIE_URL=http://sandbox:11000/oozie
```

5) Run the Workflow

a. **Run the workflow with the following command from the `/root/devph/labs/Oozi`e directory:**

```
# oozie job -config job.properties -run
```

The job.properties file reference above contains the following entries (no need to type this):

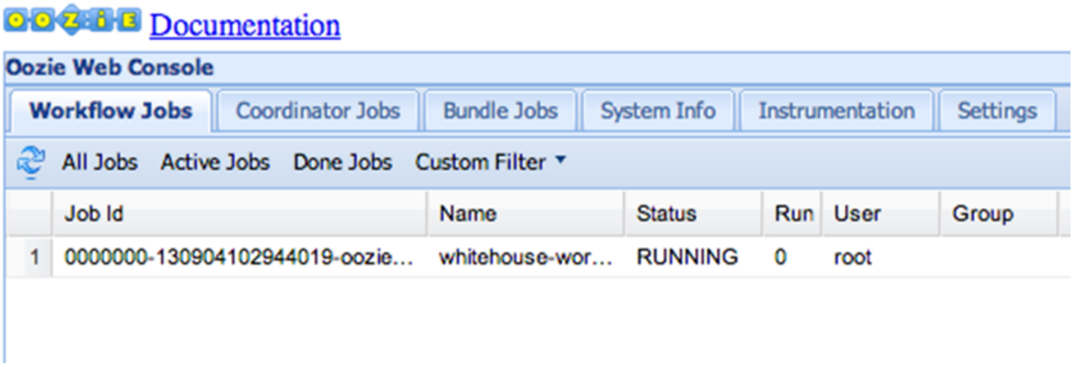
```
oozie.wf.application.path=hdfs://sandbox:8020/user/root/congress

#Hadoop RM
resourceManager=resourcemanager:8050

#Hadoop fs.default.name
nameNode=hdfs://sandbox:8020/

#Hadoop mapred.queue.name
queueName=default

oozie.use.system.libpath=true
```

If successful, the job ID should be displayed at the command prompt.

6) Monitor the Workflow

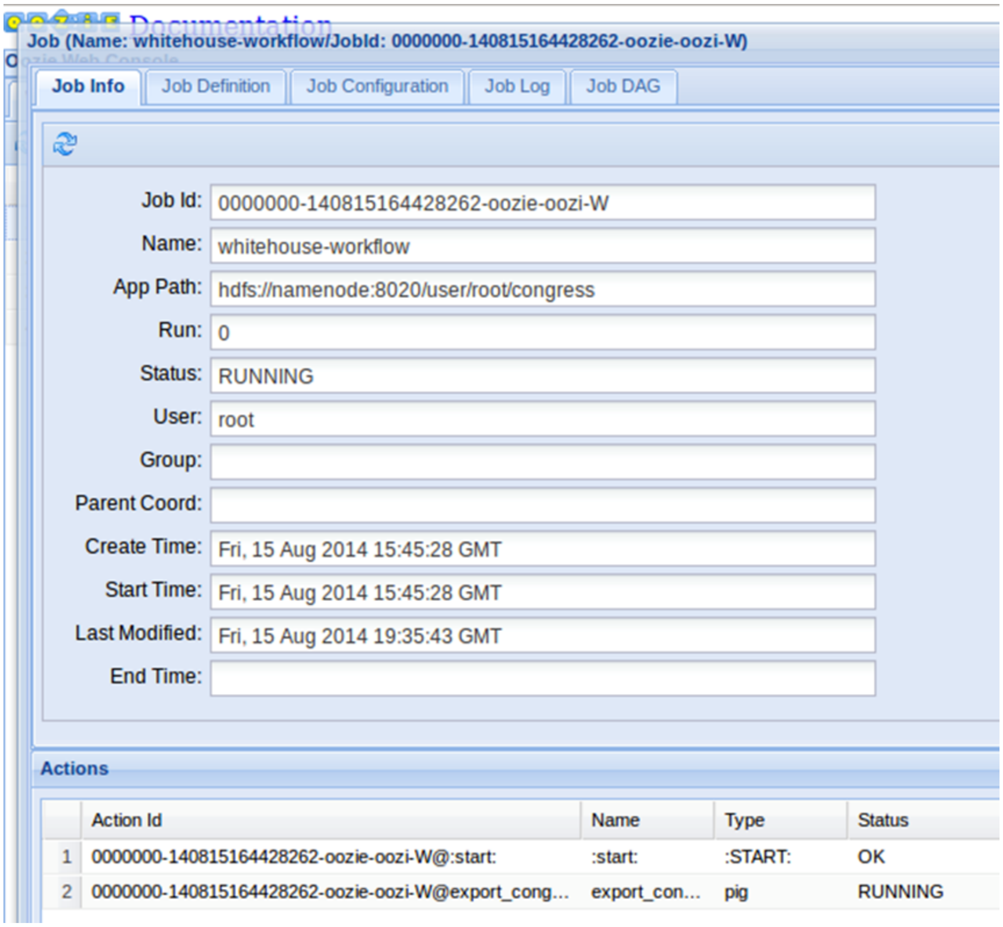
a. Point your Web browser to the Oozie Web Console at the following URL: `http://sandbox:11000/oozie`

You should see your Oozie job in the list of workflow jobs:



	Job Id	Name	Status	Run	User	Group
1	0000000-130904102944019-oozie…	whitehouse-wor…	RUNNING	0	root	

b. Click on the Job ID to the Job Info page:



Job (Name: whitehouse-workflow/JobId: 0000000-140815164428262-oozie-oozi-W)

Job Id:	0000000-140815164428262-oozie-oozi-W
Name:	whitehouse-workflow
App Path:	hdfs://namenode:8020/user/root/congress
Run:	0
Status:	RUNNING
User:	root
Group:	
Parent Coord:	
Create Time:	Fri, 15 Aug 2014 15:45:28 GMT
Start Time:	Fri, 15 Aug 2014 15:45:28 GMT
Last Modified:	Fri, 15 Aug 2014 19:35:43 GMT
End Time:	

	Action Id	Name	Type	Status
1	0000000-140815164428262-oozie-oozi-W@:start:	:start:	:START:	OK
2	0000000-140815164428262-oozie-oozi-W@export_cong...	export_con...	pig	RUNNING

Notice that you can view the status of each Action within the workflow.

7) Verify the Results

a. The Oozie job should take a few minutes to complete. Refresh the status page every so often until the status changes from RUNNING to COMPLETE, or the job no longer appears under Active Jobs tab and shows up under the Done Jobs tab.

   Once the Oozie job is completed successfully, back in the terminal window start the Hive Shell.

```
# hive
```

b. Run a select statement on `congress_visits` and verify that the table is populated:

c.

```
hive> select * from congress_visits;
...
WATERS  MAXINE  12/8/2010 17:00  POTUS  OEOB
   MEMBERS OF CONGRESS AND CONGRESSIONAL STAFF
WATT  MEL  12/8/2010 17:00  POTUS  OEOB
   MEMBERS OF CONGRESS AND CONGRESSIONAL STAFF
WEGNER  DAVID L  12/8/2010 16:46 12/8/2010 17:00 POTUS OEOB
   MEMBERS OF CONGRESS AND CONGRESSIONAL STAFF
WILLOUGHBY JEANNE P 12/8/2010 17:0712/8/2010 17:00 POTUS OEOB
   MEMBERS OF CONGRESS AND CONGRESSIONAL STAFF
WILSON ROLLIE E  12/8/2010 16:49 12/8/2010 17:00 POTUS OEOB
   MEMBERS OF CONGRESS AND CONGRESSIONAL STAFF
YOUNG DON 12/8/2010 17:00 POTUS OEOB MEMBERS OF CONGRESS AND
   CONGRESSIONAL STAFF
MCCONNELL MITCH 12/14/2010 9:00 POTUS WH MEMBER OF CONGRESS
   MEETING WITH POTUS.
Time taken: 1.082 seconds, Fetched: 102 row(s)
```

Result

You have just executed an Oozie workflow that consists of a Pig script followed by a Hive script.

Appendix: Toubleshooting

Quick troubleshooting steps

If the VM was not shut down properly on multiple occasions, or some of daemons are not working properly, please try the following:

1. Execute the following command:

```
service startup_script restart
```

2. If service `startup_script` restart does not work, excute the following:
    a. Run the `jps` command and identify PID for all running processes.
    b. Kill these processes using the `kill -9 <PID>` command
    c. Remove all `*.pid` files from the corresponding `/var/run/<SERVICE>` directories
    d. Run `service startup_script restart`

3. If services still do not come back up, try **stop all**, **start all** from Ambari.

4. If Ambari no longer works, try `service ambari stop` and `service ambari start`.

In most cases, rebooting the VM will fix the problem.

Classes Available Worldwide Through Our Partners

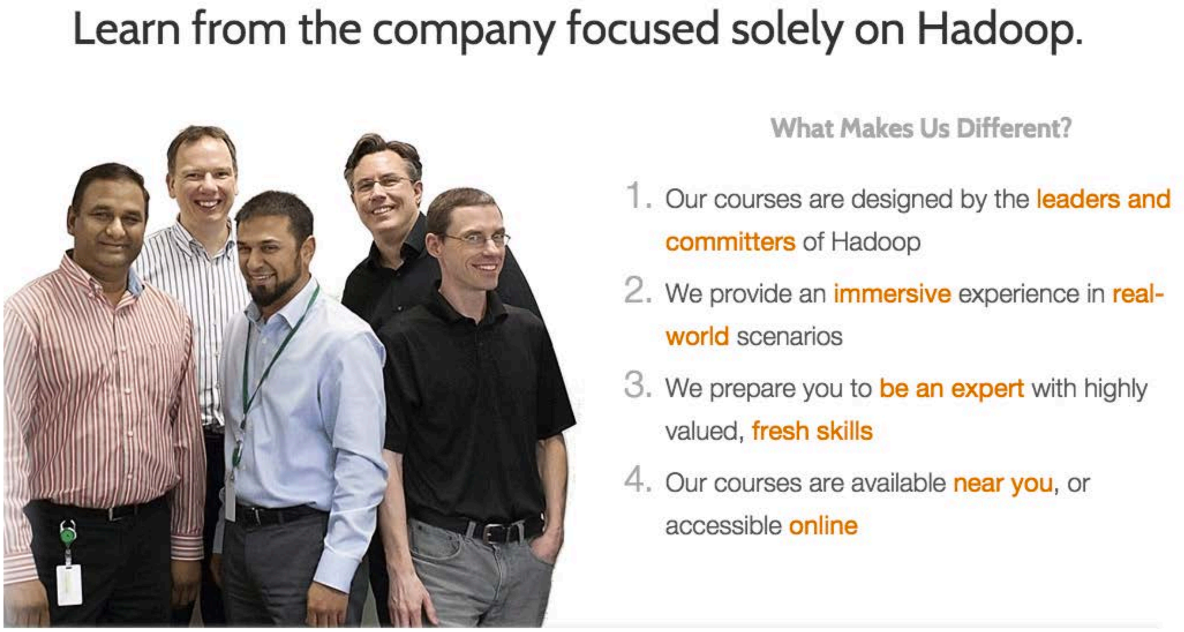
aByres
Absolute Binary Resources
BILGINÇ
IT Academy
codecentric
eCube
the Data Company
JTP
LinuxPolska
OPEN SOURCE COMPANY
netmind
OCTO
Technology
rcs
seed
beyond the obvious
sunset
LEARNING INSTITUTE
TipDM
TutorPro
valtech_
씨이랩 [XIIlab]
experience, idea and insight
/training/etc
CARE TO LEARN
ADVANCEDTRAINING
AVANTUS
BIG DATA
PARTNERSHIP
CENTER FOR PROFESSIONAL DEVELOPMENT ITT
ITT Technical Institute
contexti
CREATE VALUE FROM DATA
CORNERSTONE
DFHz
GURUTEAM
bespoke training solutions
LearnQuest
MHM
Scientia est potentia
obsidian
QA
SIMAFORE
SPRING PEOPLE

Study Options Worldwide

In combination with our partner providers, classes are often available in numerous locations across the world.

Private On-site Training

Hortonworks training in-house covers all of our basic coursework, and provides a more intimate setting for 6 or more students.
Contact us for more details



Learn from the company focused solely on Hadoop.

What Makes Us Different?

1. Our courses are designed by the **leaders and committers** of Hadoop
2. We provide an **immersive** experience in **real-world** scenarios
3. We prepare you to **be an expert** with highly valued, **fresh skills**
4. Our courses are available **near you**, or accessible **online**

Hortonworks University courses are designed by the leaders and committers of Apache Hadoop. We provide immersive, real-world experience in scenario-based training. Courses offer unmatched depth and expertise available in both the classroom or online from anywhere in the world. We prepare you to be an expert with highly valued skills and for Certification.