Demonstration: Understanding Block Storage

Objective:	To understand how data is partitioned into blocks and stored in HDFS.
During this demonstration:	Watch as your instructor performs the following steps.

Step 1: Put the File into HDFS

1.1. Review the contents of the file **stocks.csv** located in **labs/demos**.

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

```
# hadoop fs -D dfs.block.size=30 -put stocks.csv stocks.csv
```

1.3. Notice 30 bytes is not a valid blocksize. The blocksize needs to be at least 1048576 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
```

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

```
# hadoop fs -D dfs.block.size=2000000 -put stocks.csv
stocks.csv
```

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

1.6. Try the **put** again, but this time use 1,048,576 for the block size:

```
# hadoop fs -D dfs.block.size=1048576 -put stocks.csv
stocks.csv
```

1.7. This time the **put** command should have worked. Use **ls** to verify the file is in HDFS:

```
# hadoop fs -ls
Found 1 items
-rw-r--r--   3 root root    3613198 2013-08-28 21:55
stocks.csv
```

Step 2: View the Number of Blocks

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

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

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

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

Step 3: Find the Actual Blocks

3.1. 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 coincidentally are the names of the files on the DataNodes.

3.2. Change directories to the following:

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

replacing **BP-xxx** with the actual folder name.

3.3. Notice the actual blocks appear in this folder. List the contents of the folder and look for files that are exactly 1048576. These are 3 of the blocks, and notice the 4th block is smaller: 467470 bytes.

```
-rw-r--r--  1 hdfs hadoop  1048576 Aug 28 21:55
blk_1073741904
-rw-r--r--  1 hdfs hadoop     8199 Aug 28 21:55
blk_1073741904_1086.meta
```

```
-rw-r--r--  1 hdfs hadoop  1048576 Aug 28 21:55
blk_1073741905
-rw-r--r--  1 hdfs hadoop     8199 Aug 28 21:55
blk_1073741905_1087.meta
-rw-r--r--  1 hdfs hadoop  1048576 Aug 28 21:55
blk_1073741906
-rw-r--r--  1 hdfs hadoop     8199 Aug 28 21:55
blk_1073741906_1088.meta
-rw-r--r--  1 hdfs hadoop   467470 Aug 28 21:55
blk_1073741907
-rw-r--r--  1 hdfs hadoop     3663 Aug 28 21:55
blk_1073741907_1089.meta
```

3.4. You can view the contents of a block (although this is not a typical task in Hadoop!). Here is the tail of the 2nd 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.

Lab 2.1: Using HDFS Commands

Objective:	To become familiar with how files are added to and removed from HDFS, and how to view files in HDFS.
Location of Files:	**LABS/Lab2.1**
Successful Outcome:	You will have added and deleted several files and folders in HDFS.
Before You Begin:	SSH into your HDP 2.0 virtual machine.

Step 1: View the **hadoop fs** Command

1.1. From the command line, enter the following command to view the usage of **hadoop fs**:

```
# hadoop fs
```

1.2. Notice the usage contains options for performing file system 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.

Step 2: Create a Directory in HDFS

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

```
# hadoop fs -ls
```

You do not have any files in **/user/root** yet, so no output is displayed.

2.2. Run the **-ls** command again, but this time specify the root HDFS folder:

```
# hadoop fs -ls /
```

The output should look like:

```
Found 6 items
drwxrwxrwt   - yarn   hdfs  0 2013-08-20 13:59 /app-logs
drwxr-xr-x   - hdfs   hdfs  0 2013-08-20 13:53 /apps
drwxr-xr-x   - mapred hdfs  0 2013-08-20 13:57 /mapred
drwxr-xr-x   - hdfs   hdfs  0 2013-08-20 13:58 /mr-history
drwxrwxrwx   - hdfs   hdfs  0 2013-08-28 22:03 /tmp
drwxr-xr-x   - hdfs   hdfs  0 2013-08-28 22:03 /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.

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

```
# hadoop fs -mkdir test
```

2.4. Verify the folder was created successfully:

```
# hadoop fs -ls
Found 1 items
drwxr-xr-x   - root root  0 2013-08-29 03:51 test
```

2.5. Create a couple of subdirectories of **test**:

```
# hadoop fs -mkdir test/test1
# hadoop fs -mkdir test/test2
# hadoop fs -mkdir test/test2/test3
```

2.6. Use the **-ls** command to view the contents of **/user/root**:

```
# hadoop fs -ls
```

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

```
# hadoop fs -ls -R
```

The output should look like:

```
drwxr-xr-x   - root root  0 2013-08-29 03:54 test
drwxr-xr-x   - root root  0 2013-08-29 03:52 test/test1
drwxr-xr-x   - root root  0 2013-08-29 03:54 test/test2
drwxr-xr-x   - root root  0 2013-08-29 03:54
   test/test2/test3
```

Step 3: Delete a Directory

3.1. Delete the **test2** folder (and recursively its subcontents) using the **-rm -R** command:

```
# hadoop fs -rm -R test/test2
```

3.2. Now run the **-ls -R** command:

```
# hadoop fs -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.

Step 4: Upload a File to HDFS

4.1. Now let's put a file into the **test** folder. Change directories to **LABS/Lab2.1**:

```
# cd ~/labs/Lab2.1/
```

4.2. Notice this folder contains a file named **data.txt**:

```
# tail data.txt
```

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

```
# hadoop fs -put data.txt test/
```

4.4. Verify the file is in HDFS by listing the contents of **test**:

```
# hadoop fs -ls test
```

The output should look like the following:

```
Found 2 items
-rw-r--r--   3 root root 1529355 2013-08-29  test/data.txt
drwxr-xr-x   - root root       0 2013-08-29  test/test1
```

Step 5: Copy a File in HDFS

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

```
# hadoop fs -cp test/data.txt test/test1/data2.txt
```

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

```
# hadoop fs -ls -R test
-rw-r--r--   3 root root     1529355 test/data.txt
drwxr-xr-x   - root root           0 test/test1
-rw-r--r--   3 root root     1529355 test/test1/data2.txt
```

5.3. Now delete the **data2.txt** file using the **-rm** command:

```
# hadoop fs -rm test/test1/data2.txt
```

5.4. Verify the **data2.txt** file is in the **.Trash** folder.

Step 6: View the Contents of a File in HDFS

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

```
# hadoop fs -cat test/data.txt
```

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

```
# hadoop fs -tail test/data.txt
```

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

Step 7: Getting a File from HDFS

7.1. See if you can figure out how to use the **get** command to copy **test/data.txt** into your local **/tmp** folder.

Step 8: The getmerge Command

8.1. Put the file **/root/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**.

8.2. Run the following **getmerge** command:

```
# hadoop fs -getmerge test /tmp/merged.txt
```

8.3. What did the previous command do? Open the file **merged.txt** to see what happened?

Step 9: Specify the Block Size and Replication Factor

9.1. Put **LABS/Lab2.1/data.txt** into **/user/root** in HDFS, giving it a blocksize of 1048576 bytes. **HINT**: The blocksize is defined using the **dfs.block.size** property on the command line.

9.2. Run the following **fsck** command on **data.txt**:

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

9.3. How many blocks are there for this file? _____

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 3.1: Importing RDBMS Data into HDFS

Objective:	Import data from a database into HDFS.
Location of Files:	n/a
Successful Outcome:	You will have imported data from MySQL into folders in HDFS.
Before You Begin:	SSH into your HDP 2.0 virtual machine.

Perform the following steps:

Step 1: Create a Table in MySQL

1.1. From the command prompt, change directories to **Lab3.1**:

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

1.2. View the contents of **salaries.txt**:

```
# tail salaries.txt
```

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

1.3. Notice 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** into the publicly-available **/tmp** folder:

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

1.4. Now run the **salaries.sql** script using the following command:

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

Step 2: View the Table

2.1. To verify the table is populated in MySQL, open the **mysql** prompt:

```
# mysql
```

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

```
mysql> use test;
```

2.3. Run the **show tables** command and verify **salaries** is defined:

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

2.4. Select 10 items from the table to verify 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 |
+--------+------+--------+---------+----+
```

2.5. Exit the **mysql** prompt:

```
mysql> exit
```

Step 3: Import the Table into HDFS

3.1. 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://localhost/test
        --table salaries
```

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

Step 4: Verify the Import

4.1. View the contents of your HDFS folder:

```
# hadoop fs -ls
```

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

```
# hadoop fs -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
```

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

_____

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

```
# hadoop fs -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 you imported the entire table with all of its columns. In the next step, you will import only specific columns of a table.

Step 5: Specify Columns to Import

5.1. 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.

5.2. After the import, verify you only have one **part-m** fie in **salaries2**:

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

5.3. Verify the contents of **part-m-00000** are only the 2 columns you specified:

```
# hadoop fs -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
```

Step 6: Importing from a Query

6.1. Write a Sqoop import command that imports the rows from **salaries** in MySQL whose **salary** column is greater than 90,000.00. Use **gender** as the **--split-by** value, specify only 2 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.

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

6.3. View the contents of **part-m-00000** and **part-m-00001**. Notice one file contains females, and the other file contains males. Why? _____

_____

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

Lab 3.2: Exporting HDFS Data to a RDBMS

Objective:	Export data from HDFS into a MySQL table using Sqoop.
Location of Files:	**LABS/Lab3.2**
Successful Outcome:	The data in **salarydata.txt** in HDFS will appear in a table in MySQL named **salary2**.
Before You Begin:	SSH into your HDP 2.0 virtual machine.

Perform the following steps:

Step 1: Put the Data into HDFS

1.1. Change directories to **LABS/Lab3.2**:

```
# cd ~/labs/Lab3.2
```

1.2. 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 4 values separated by commas, and the values represent a **gender**, **age**, **salary** and **zip code**, respectively.

1.3. Create a new directory in HDFS named **salarydata**.

1.4. Put **salarydata.txt** into the **salarydata** directory in HDFS.

Step 2: Create a Table in the Database

2.1. There is a script in the Lab3.2 folder that creates a table in MySQL that matches the records in salarydata.txt. View the SQL script:

```
# more salaries2.sql
```

2.2. Run this script using the following command:

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

2.3. Verify 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    |       |
+---------+------------+------+-----+---------+-------+
```

2.4. Exit the mysql prompt:

```
mysql> exit
```

Step 3: Export the Data

3.1. 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.

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

```
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 |
+--------+------+--------+---------+
```

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

Demonstration: Understanding MapReduce

Objective:	To understand how MapReduce works.
During this demonstration:	Watch as your instructor performs the following steps.

Step 1: Put the File into HDFS

1.1. Change directories to the **demos** folder:

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

1.2. Notice a file named **constitution.txt**:

```
# more constitution.txt
```

1.3. Put the file into HDFS:

```
# hadoop fs -put constitution.txt constitution.txt
```

Step 2: Run the WordCount Job

2.1. The following command runs the WordCount job on the **constitution.txt** and writes the output to **wordcount_output**:

```
# hadoop jar wordcount.jar wordcount.WordCountJob
constitution.txt wordcount_output
```

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

Step 3: View the Results

3.1. View the contents of the **wordcount_output** folder:

```
# hadoop fs -ls wordcount_output
```

You should see a single file named **part-r-00000**:

```
Found 1 items
-rw-r--r--   1 root hdfs   17054 2013-08-29 21:57
      wordcount_output/part-r-00000
```

3.2. Why is there one file in this directory? _____

3.3. What does the "**r**" in the filename stand for? _____

3.4. View the contents of **part-r-00000**:

```
# hadoop fs -cat wordcount_output/part-r-00000
```

3.5. Why are the words sorted alphabetically? _____

3.6. What was the key output by the WordCount reducer? _____

3.7. What was the value output by the WordCount reducer? _____

3.8. Based on the output of the reducer, what do you think the mapper output as key/value pairs? _____

_____

Lab 4.1: Running a MapReduce Job

Objective:	Run a Java MapReduce job.
Location of Files:	**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:	SSH into your HDP 2.0 virtual machine.

Step 1: Put the Data in HDFS

1.1. 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 **LABS/Lab4.1**:

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

1.2. View the contents of the file **hortonworks.txt**. Each line looks like:

```
http://hortonworks.com/,hadoop,webinars,articles,download,e
nterprise,team,reliability
```

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

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

```
# hadoop fs -mkdir inverted
# hadoop fs -mkdir inverted/input
```

1.4. Put **hortonworks.txt** into HDFS in the **inverted/input/** folder. This file will be the input to the MapReduce job.

Step 2: Run the Inverted Index Job

2.1. From the **LABS/Lab4.1** folder, enter the following command (all on a single line):

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

2.2. 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
```

Step 3: View the Results

3.1. List the contents of the **inverted/output** folder. How many reducers did this job use? _____ How can you determine this from the contents of **inverted/output**?

_____

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

```
# # hadoop fs -cat inverted/output/part-r-00000
about  http://hortonworks.com/about-us/,
apache
 http://hortonworks.com/products/hortonworksdataplatform/,h
ttp://hortonworks.com/about-us/,
articles
 http://hortonworks.com/community/,http://hortonworks.com/,
...
```

Step 4: Specify the Number of Reducers

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

```
# hadoop jar invertedindex.jar inverted.IndexInverterJob
-D mapred.reduce.tasks=3 inverted/input inverted/output
```

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

```
# # hadoop fs -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
```

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

_____

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

Objective:	To understand Pig scripts and relations.
During this demonstration:	Watch as your instructor performs the following steps.

Step 1: Start the Grunt Shell

1.1. Review the contents of the file **pigdemo.txt** located in **LABS/demos**.

1.2. Start the Grunt shell:

```
# pig
```

1.3. Notice the output includes where the logging for your Pig session will go, as well as a statement about connecting to your Hadoop file system:

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

Step 2: Make a New Directory

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

```
grunt> ls
```

2.2. Make a new directory named **demos**:

```
grunt> mkdir demos
```

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

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

2.4. Verify the file was uploaded successfully:

```
grunt> ls demos
hdfs://sandbox:8020/user/root/demos/pigdemo.txt<r 1>   87
```

2.5. Change the present working directory to **demos**:

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

2.6. 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
```

Step 3: Define a Relation

3.1. Define the **employees** relation, using a schema:

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

3.2. 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 the default data type of a field (if you do not specify one) is **bytearray**.

3.3. 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)
```

Step 4: Filter the Relation by a Field

4.1. Let's filter the **employees** whose **state** field equals CA:

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

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

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

Step 5: Create a Group

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

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

5.2. Groups in Pig are represented by ***bags***. A bag is an unordered collection of tuples:

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

5.3. 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. *Bags* are represented by curly braces.

Step 6: The STORE Command

6.1. The **DUMP** command dumps the contents of a relation to the console. The **STORE** command sends the output to a file. For example:

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

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

6.2. Verify a new file is created:

```
grunt> ls
hdfs://sandbox:8020/user/root/demos/emp_group.txt    <dir>
hdfs://sandbox:8020/user/root/demos/pigdemo.txt<r 1>   87
```

6.3. View the contents of the output file:

```
grunt> cat emp_group.txt
CA      {(CA,Ulf),(CA,manish),(CA,Brian)}
CO      {(CO,George),(CO,Mark)}
```

```
IL      {(IL,Danielle)}
NV      {(NV,Barry)}
OH      {(OH,Tom)}
SD      {(SD,Rich)}
```

6.4. Notice 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(',');
```

Step 7: Show All Aliases

7.1. The **aliases** command shows a list of currently-defined aliases:

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

Step 8: Monitor the Pig Jobs

8.1. Point your browser to **http://*host*:8000/jobbrowser/jobs/**.

8.2. Enter your username (**root**) in the **Username** text field.

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

8.4. Notice on the **Attempts** tab you can view the log files on a specific container.

8.5. Notice on the **Tasks** tab you can view the results of each map and reduce task.

8.6. The **Counters** tab shows the counter output from the MapReduce job. You saw these same counters output at the command prompt at the end of the job.

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 5.1: Exploring Data with Pig

Objective:	Use Pig to navigate through HDFS and explore a dataset.
Location of Files:	**LABS/Lab5.1**
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:	SSH into your HDP 2.0 virtual machine.

Step 1: View the Raw Data

1.1. Change directories to the Lab5.1 folder:

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

1.2. Unzip the archive in the **Lab5.1** folder, which contains a file named **whitehouse_visits.txt** that is quite large:

```
# unzip whitehouse_visits.zip
```

1.3. 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.

Step 2: Load the Data into HDFS

2.1. Start the Grunt shell:

```
# pig
```

2.2. From the Grunt shell, make a new directory in HDFS named **whitehouse**:

```
grunt> mkdir whitehouse
```

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

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

2.4. Use the **ls** command to verify the file was uploaded successfully:

```
grunt> ls whitehouse
hdfs://sandbox:8020/user/root/whitehouse/visits.txt<r 1>
 175153242
```

Step 3: Define a Relation

3.1. 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/visits.txt' USING
TextLoader();
```

3.2. Use **DESCRIBE** to notice that **A** does not have a schema:

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

3.3. 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**.

3.4. 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**:

```
(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,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,)
```

Step 4: Count the Number of Lines

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

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

4.2. Use **DESCRIBE** to view the schema of **B**. What is the datatype of the **group** field? ______________________ Where did this datatype come from? ______

____

4.3. Why does the **A** field of **B** contain no schema? _______________________

4.4. How many groups are in the relation **B**? _______________

4.5. 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 - just makes the output nicer to read.

4.6. Use **DUMP** on **A_count** to view the results. The output should look like:

```
(rowcount,447598)
```

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

Step 5: Analyze the Data's Contents

5.1. 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/visits.txt'
USING PigStorage(',');
```

This will split up the fields by comma.

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

5.3. 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**:

```
(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 7 fields are a unique ID, badge number, access type, time of arrival, post of arrival, time of departure and post of departure.

Step 6: Locate the POTUS (President of the United States of America)

6.1. 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 7 fields (**$19** to **$25**) of **visits**. Use **LIMIT** to only output 500 records. The output should look like:

```
(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,)
```

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!)

6.2. 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:

```
(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,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,)
```

Step 7: Count the POTUS Visitors

7.1. 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.

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

Step 8: The STORE Command

8.1. 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. Start by loading the data using **PigStorage(',')**, which you may already have defined:

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

8.2. Now **FILTER** the relation by visitors who met with the President:

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

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

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

8.4. View the schema of **potus_details**:

```
grunt> DESCRIBE potus_details;
[main] WARN  org.apache.pig.PigServer - Encountered Warning
IMPLICIT_CAST_TO_CHARARRAY 1 time(s).
potus_details: {lname: chararray,fname:
chararray,arrival_time: chararray,visitee: chararray}
```

8.5. Store the records in **potus_details** in a folder named **'potus'**:

```
grunt> STORE potus_details INTO 'potus' USING
PigStorage('\t');
```

8.6. Change directories into the **potus** folder and execute **ls**:

```
grunt> cd potus
grunt> ls
hdfs://sandbox:8020/user/root/potus/part-m-00000<r 1>
 53771
hdfs://sandbox:8020/user/root/potus/part-m-00001<r 1>
 200549
hdfs://sandbox:8020/user/root/potus/part-m-00002<r 1>
 67025
```

```
hdfs://sandbox:8020/user/root/potus/part-m-00003<r 1>
 98902
hdfs://sandbox:8020/user/root/potus/part-m-00004<r 1>
 81131
hdfs://sandbox:8020/user/root/potus/part-m-00005<r 1>   0
```

8.7. Notice there are six map output files, so the Pig job was executed with six mappers. View the contents of the output files using **cat**:

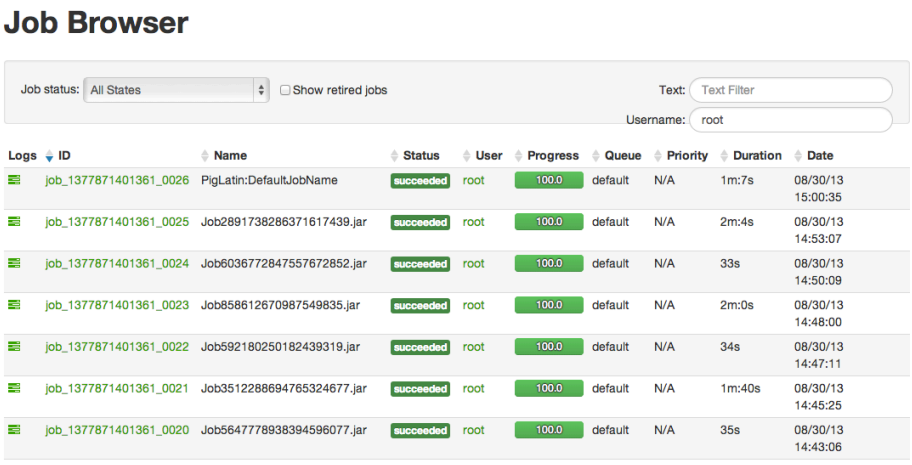
```
grunt> cat part-m-00000
```

The output should contain the last name, first name, time of arrival (if available), and the string 'POTUS':

```
FRIEDMAN     THOMAS      10/12/2010  12:08 POTUS
BASS    EDWIN 10/18/2010  15:01 POTUS
BLAKE  CHARLES     10/18/2010  15:00 POTUS
OGLETREE     CHARLES     10/18/2010  15:01 POTUS
RIVERS EUGENE      10/18/2010  15:01 POTUS
```

Step 9: View the Pig Log Files

9.1. Each time you executed a **DUMP** or **STORE** command, a MapReduce job executed on your cluster. You can view the log files of these jobs in the Sandbox's Job Browser. Point your browser to **http://ip:8000/jobbrowser/jobs/** and type **root** in the Username text field:



9.2. 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

Lab 6.1: Splitting a Dataset

Objective:	Research the White House visitor data and look for members of Congress.
Location of Files:	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**.

Perform the following steps:

Step 1: Explore the Comments Field

1.1. In this step, you will explore the comments field of the White House visitor data. Start by loading **visits.txt**:

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

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

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

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

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

1.4. 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}
```

Step 2: Test the Relation

2.1. 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**:

```
comments_sample = SAMPLE comments 0.001;
```

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

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

Step 3: Count the Number of Comments

3.1. 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.

Step 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.

4.1. In this step, you will split visits.txt into two datasets: those that contain "CONGRESS" in the comments field, and those that do not. Start by loading the data:

```
visits = LOAD 'visits.txt' USING PigStorage(',');
```

4.2. Define the following two relations, which create the desired split:

```
congress = FILTER visits BY ($25 MATCHES '.* CONGRESS .*');
not_congress = FILTER visits BY
                  (NOT($25 MATCHES '.* CONGRESS .*'));
```

4.3. Store the **congress** relation into a folder named **'congress'**:

```
STORE congress INTO 'congress';
```

4.4. Similarly, **STORE** the **not_congress** relation in a folder named **'not_congress'**.

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

```
grunt> ls congress
whitehouse/congress/part-m-00000<r 1>  1445
whitehouse/congress/part-m-00001<r 1>  21118
whitehouse/congress/part-m-00002<r 1>  23055
whitehouse/congress/part-m-00003<r 1>  0
whitehouse/congress/part-m-00004<r 1>  0
whitehouse/congress/part-m-00005<r 1>  0
grunt> ls not_congress
whitehouse/not_congress/part-m-00000<r 1>    28156101
whitehouse/not_congress/part-m-00001<r 1>    26038125
whitehouse/not_congress/part-m-00002<r 1>    30730780
whitehouse/not_congress/part-m-00003<r 1>    5817011
whitehouse/not_congress/part-m-00004<r 1>    247561
whitehouse/not_congress/part-m-00005<r 1>    24390
```

4.6. View one of the output files in **congress**: and make sure the string "CONGRESS" appears in the comment field:

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

Step 5: Count the Results

5.1. 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. Next, you will create a projection of **'visits.txt'** that you will use in a later Hive lab.

Lab 6.2: Joining Datasets

Objective:	Join two datasets in Pig.
Location of Files:	**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.

Perform the following steps:

Step 1: Upload the Congress Data

1.1. Put the file **LABS/Lab6.2/congress.txt** into the **whitehouse** directory in HDFS.

1.2. Use the **ls** command to verify the **congress.txt** file is in **whitehouse**, and use **cat** to view its contents. The file contains the names and other information about the members of the U.S. Congress.

Step 2: Create a Pig Script File

2.1. In this lab, you will not use the Grunt shell to enter commands. Instead, you will enter your script in a text file. Using a text editor, create a new file named **join.pig** in the **Lab6.2** folder.

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

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

Step 3: Load the White House Visitors

3.1. 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**.

Step 4: Define a Projection of the Congress Data

4.1. 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
);
```

4.2. 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;
```

Step 5: Join the Two Datasets

5.1. 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.

5.2. Use the **STORE** command to store the result of **join_contact_congress** into a directory named **'joinresult'**.

Step 6: Run the Pig Script

6.1. Save your changes to **join.pig**.

6.2. Run the script using the following command:

```
# pig join.pig
```

6.3. 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 **FinishedAt** time from the **StartedAt** time) and write down the result: _____

6.4. 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.

Step 7: View the Results

7.1. The output will be in the 'joinresult' folder in HDFS. Verify the folder was created:

```
# hadoop fs -ls -R joinresult
-rw-r--r--   1 root hdfs       2194 joinresult/part-m-00000
-rw-r--r--   1 root hdfs       6718 joinresult/part-m-00001
-rw-r--r--   1 root hdfs      14930 joinresult/part-m-00002
-rw-r--r--   1 root hdfs       8486 joinresult/part-m-00003
-rw-r--r--   1 root hdfs       8146 joinresult/part-m-00004
-rw-r--r--   1 root hdfs        418 joinresult/part-m-00005
```

7.2. View the resulting file:

```
# hadoop fs -cat joinresult/part-m-00000
```

The output should look like the following:

```
DUFFY  SEAN  WI07  DUFFY SEAN  Republican
JONES  WALTER     NC03  JONES WALTER     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
```

Step 8: Try Using Replicated on the Join

8.1. Delete the **joinresult** directory in HDFS:

```
# hadoop fs -rm -R joinresult
```

8.2. Modify your **JOIN** statement in **join.pig** so that is uses replication.

8.3. Save your changes **to join.pig** and run the script again.

8.4. Notice this time that the statistics output shows Pig used a "**REPLICATED_JOIN**" instead of a "**HASH_JOIN**".

8.5. 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.

Step 9: Count the Results

9.1. 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.

9.2. 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:

```
DUMP counters;
```

> **HINT**: 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`

9.3. The correct results are shown here:

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

Step 10: Use the EXPLAIN Command

10.1. At the end of **join.pig**, add the following statement:

```
EXPLAIN counters;
```

If you do not have a **counters** relations, then use **join_contact_congress** instead.

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

10.3. How many MapReduce jobs did it take to run this job? _____

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 6.3: Preparing Data for Hive

Objective:	Transform and export a dataset for use with Hive.
Location of Files:	**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.

Perform the following steps:

Step 1: Review the Pig Script

1.1. From a command prompt, change directories to **Lab6.3**:

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

1.2. View the contents of **wh_visits.pig**:

```
# more wh_visits.pig
```

1.3. Notice the **potus** relation is all White House visitors who met with the President.

1.4. Notice the **project_potus** relation is a projection of the last name, first name, time of arrival, location and comments from the visit.

Step 2: Store the Projection in the Hive Warehouse

2.1. Open **wh_visits.pig** with a text editor.

2.2. 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/';
```

Step 3: Run the Pig Script

3.1. Save your changes to **wh_visits.pig**.

3.2. Run the script from the command line:

```
# pig wh_visits.pig
```

Step 4: View the Results

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

```
# hadoop fs -ls /apps/hive/warehouse/wh_visits/
Found 6 items
109326 /apps/hive/warehouse/wh_visits/part-m-00000
366061 /apps/hive/warehouse/wh_visits/part-m-00001
111105 /apps/hive/warehouse/wh_visits/part-m-00002
 98920 /apps/hive/warehouse/wh_visits/part-m-00003
 74709 /apps/hive/warehouse/wh_visits/part-m-00004
     0 /apps/hive/warehouse/wh_visits/part-m-00005
```

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

```
# hadoop fs -cat /apps/hive/warhouse/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.

Lab 7.1: Understanding Hive Tables

Objective:	Understand how Hive table data is stored in HDFS.
Location of Files:	**LABS/7.1**
Successful Outcome:	A new Hive table filled with the data from the **wh_visits** folder.
Before You Begin:	Complete Lab 6.3, or put the data from the solution of Lab 6.3 into HDFS.

Perform the following steps:

Step 1: Review the Data

1.1. Use the **ls** command to view the contents of the **wh_visits** folder. You should see six **part-m** files:

```
# hadoop fs -ls /apps/hive/warehouse/wh_visits/
```

1.2. Recall that the Pig projection to create these files had the following schema:

```
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.

Step 2: Define a Hive Script

2.1. In the **Lab7.1** folder, there is a text file named **wh_visits.hive**. View its contents. Notice it defines a Hive table named **wh_visits** with the following schema that matches the data in your **project_potus** folder:

```
# 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.

2.2. Run the script with the following command:

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

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

Step 3: Verify the Table Creation

3.1. Start the Hive Shell:

```
# hive
hive>
```

3.2. From the **hive>** prompt, enter the "**show tables**" command:

```
hive> show tables;
```

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

3.3. Use the **describe** command to view the details of **wh_visits**:

```
hive> describe wh_visits;
OK
lname                   string                  None
```

```
fname                   string                  None
time_of_arrival         string                  None
appt_scheduled_time     string                  None
meeting_location        string                  None
info_comment            string                  None
```

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

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

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

_____

3.5. How come the previous query did not require a MapReduce job to execute?

_____

3.6. Try the following query. Make sure the output looks like first names:

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

This time a MapReduce job executed. Why? _____

Step 4: Count the Number of Rows in a Table

4.1. Enter the following Hive query, which outputs the number of rows in **wh_visits**:

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

4.2. How many rows are currently in **wh_visits**? _____

NOTE: For now, you are done with **wh_visits** table until the next lab when you will analyze the data in more detail.

Step 5: Drop a Table

5.1. 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';
```

5.2. From the command line, put **Lab7.1/names.txt** into the table's warehouse folder:

```
# hadoop fs -put names.txt /apps/hive/warehouse/names/
```

5.3. View the contents of the table's warehouse folder:

```
# hadoop fs -ls /apps/hive/warehouse/names
Found 1 items
root hdfs      78 /apps/hive/warehouse/names/names.txt
```

5.4. 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
```

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

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

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

```
# hadoop fs -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.

Step 6: Define an External Table

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

```
# hadoop fs -put names.txt names.txt
```

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

```
# hadoop fs -mkdir hivedemo
```

6.3. From the Hive Shell, 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';
```

6.4. Load data into the table:

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

6.5. Verify the load worked:

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

6.6. Notice the **names.txt** file has been moved to **/user/root/hivedemo**:

```
# # hadoop fs -ls hivedemo
Found 1 items
-rw-r--r--   1 root hdfs      78  hivedemo/names.txt
```

6.7. Similarly, verify **names.txt** is no longer in your **/user/root** folder in HDFS. Why is it gone? _____

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

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

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

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

RESULT: As you just verified, 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

Objective:	To understand how Hive partitioning works, as well as skewed tables.
During this demonstration:	Watch as your instructor performs the following steps.

Step 1: View the Data

1.1. Review the **hivedata_<<*state*>>.txt** files in **LABS/demos**. This will be the data added to the table.

Step 2: Define the Table in Hive

2.1. View the **create table** statement in **partitiondemo.sql**:

```
# more partitiondemo.sql
create table names (id int, name string)
     partitioned by (state string)
     row format delimited fields terminated by '\t';
```

2.2. Run the query to define the **names** table:

```
# hive -f partitiondemo.sql
```

2.3. Show the partitions (there won't be any yet):

```
hive> show partitions names;
```

Step 3: Load Data into the Table

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

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

3.2. Load the CO and SD files also:

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

3.3. Verify 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
```

Step 4: View the Directory Structure

4.1. From the command line, view the contents of **/apps/hive/warehouse/names**:

```
# hadoop fs -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
```

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

Step 5: Perform a Query

5.1. 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
```

5.2. Notice a MapReduce job was not executed. Why? _____

_____

5.3. Notice 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. For example:

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

5.4. You can still run queries across the entire dataset. For example, the following query spans multiple partitions:

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

Step 6: Create a Skewed Table

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

```
# more skewdemo.hive
```

6.2. Which values is this table being skewed by? _____

6.3. Run the commands in **skewdemo.hive**:

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

6.4. From the Hive shell, verify the table is defined:

```
DESCRIBE skew_demo;
```

6.5. Select a few records to make sure the table had data behind it:

```
select * from skew_demo limit 20;
```

Step 7: View the Files

7.1. View the contents of the underlying Hive warehouse folder:

```
# hadoop fs -ls -R /apps/hive/warehouse/skew_demo
```

Lab 7.2: Analzying Big Data with Hive

Objective:	Analyze the White House visitor data.
Location of Files:	n/a
Successful Outcome:	You will have discovered several useful pieces of information about the White House visitor data.
Before You Begin:	Complete Lab 7.1.

Perform the following steps:

Step 1: Find the First Visit

1.1. Create a new text file named **whitehouse.hive** and save it in your **Lab7.2** folder.

1.2. In this step, you will 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. Start by selecting all columns where the **time_of_arrival** is not empty:

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

1.3. 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:

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

1.4. 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:

```
limit 10;
```

1.5. Save your changes to **whitehouse.hive**.

1.6. Execute the script **whitehouse.hive** and wait for the results to be displayed:

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

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

Step 2: Find the Last Visit

2.1. 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
```

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

Step 3: Find the Most Common Comment

3.1. 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 creating a new text file named **comments.hive**.

3.2. 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 5 times, then **comment_count** will be 5 for that group.

Notice we are also selecting the **info_comment** column so we can see what the comment is.

3.3. Group the results of the query by the **info_comment** column:

```
group by info_comment
```

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

```
order by comment_count DESC
```

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

```
limit 10;
```

3.6. Save your changes to **comments.hive** and execute the script. Wait for the MapReduce job to execute.

3.7. 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
```

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

3.9. OPTIONAL: Modify the query so that it ignores empty comments. If it works, the comment "GEN RECEP 6/" will show up in your output.

Step 4: Least Frequent Comment

4.1. Run the previous query again, but this time find the 10 least occurring comments. The output should look like:

```
1      merged to u59031
1      WHO EOP/
1      WHO EOP RECLEAR
1      WAITING FOR SUPERMAN VISIT
1      ST. PATRICK'S RECEPTION GUESTS
1      SCIENCE FAIR
1      RES PARTY/
1      PRIVATE MEETING
1      PRIVATE LUNCH
1      POTUS PHOTO W/ US ATTORNEYS/
```

This seems accurate since 1 is the least number of times a comment can appear. Plus this query reveals that Superman has visited the President at least once!

Step 5: Analyze the Data Inconsistencies

5.1. 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 like have different people entering similar comments but using their own abbreviations.

5.2. Modify the query in **comments.hive**. Instead of searching for empty comments, search for comments that contain the string "RECEP".

```
where info_comment like  "%RECEP%"
```

5.3. Change the limit clause from 10 to 30:

```
limit 30;
```

5.4. Run the query again.

5.5. 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
```

5.6. Let's try one more query to try and narrow GENERAL RECEPTION visist. Modify the WHERE clause in comments.hive to include "%GEN%":

```
where info_comment like  "%RECEP%"
  and info_comment like "%GEN%"
```

5.7. Leave the limit at 30, and run the query again.

5.8. The output this time reveals all the variations of GEN and RECEP. 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%";
```

5.9. 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.

5.10. NOTE: More importantly, these results show that our first query of 1,253 attendees to the HOLIDAY BALL does not mean that the holiday ball is the most likely reason to visit the President. More than twice as many visitors are there for a general reception. This type of analysis is common in big data, and it shows how Data Analysts need to be creative when researching their data.

Step 6: Verify the Result

6.1. 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". 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
```

6.2. Run a query that counts the number of records with WHO and EOP in the comments:

```
from wh_visits
        select count(*)
        where info_comment like  "%WHO%"
          and info_comment like "%EOP%";
```

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.

Step 7: Find the Most Visits

7.1. 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. **HINT**: Use a grouping by both **fname** and **lname**.

7.2. To verify your script worked, here are the top 20 individuals who visited the POTUS, along with the number of visits:

```
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.

Lab 7.3: Understanding MapReduce in Hive

Objective:	To understand better how Hive queries get executed as MapReduce jobs.
Location of Files:	n/a
Successful Outcome:	No specific outcome - you will answer various questions about Hive queries and run a few examples.
Before You Begin:	Start the Hive Shell.

Step 1: The Describe Command

1.1. Run the **describe** command on the **wh_visits** table:

```
hive> describe wh_visits;
OK
lname                   string                  None
fname                   string                  None
time_of_arrival         string                  None
appt_scheduled_time     string                  None
meeting_location        string                  None
info_comment            string                  None
Time taken: 0.677 seconds
```

1.2. Did this query require a MapReduce job? _____

1.3. What is the name of the Hive resource that was accessed to retrieve this schema information? _____

Step 2: A Simple Query

2.1. Run the following query:

```
select * from wh_visits where fname = "JOE";
```

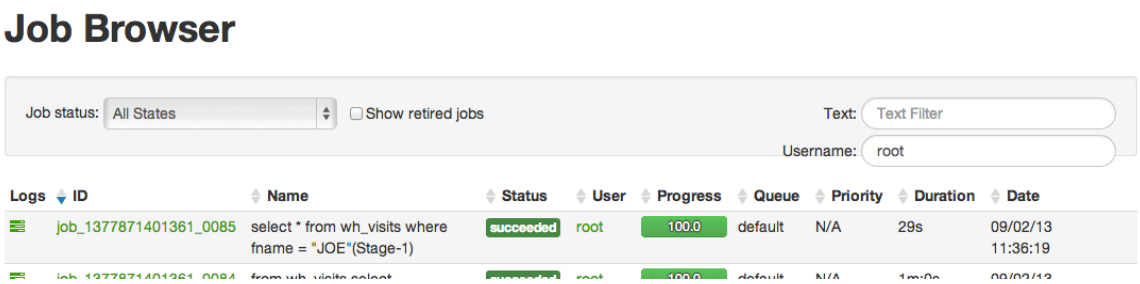
2.2. Does Hive run a MapReduce job to generate the result? _____

2.3. Open your browser and point it to the Job Browser page of the Sandbox:

```
http://<<host>>:8000/jobbrowser/jobs/
```

2.4. Enter **root** in the **Username** field and press **Enter**.

2.5. Notice the most recent MapReduce job executed is your "JOE" query:



2.6. Click in the job id in the **ID** column, which displays detailed information about the job.

2.7. Click on the **Tasks** tab.

2.8. How many map tasks were used to execute this query? _____

2.9. How many reduce tasks were used to execute this query? _____

2.10. Click on the first task in the **Tasks** column.

2.11. How many attempts did it take for this task to succeed? _____

2.12. Click on the **Metadata** tab of the task.

2.13. How long did it take for this query to execute? _____

Step 3: A Sorted Query

3.1. Run the following query:

```
hive> select * from wh_visits where fname = "JOE" sort by
lname;
```

3.2. When the MapReduce job completes, find its job details page from the Job Browser.

3.3. How many map tasks were used to execute this query? _____

3.4. How many reduce tasks were used to execute this query? _____

3.5. Can a query containing a **SORT BY** clause have more than one reducer?

_____

3.6. The map task outputs <key,value> pairs and sends them to the reducer. What do you think this MapReduce job chose as the key for the mapper's output?

_____

Step 4: A Select Query

4.1. Run the following query:

```
hive> select * from wh_visits limit 5;
```

4.2. Does Hive run a MapReduce job to generate the results? _____

4.3. What data is read from HDFS? _____

4.4. Now select a single column from **wh_visits**:

```
hive> select fname from wh_visits limit 5;
```

4.5. Why did this require a MapReduce job but "select *" did not? _____

_____

Step 5: An **ORDER BY** Query

5.1. When you execute a query that contains **ORDER BY**, how many reducers are typically used? _____

5.2. Can you specify the number of reducers to use in an **ORDER BY** query? _____

If so, how? _____

Step 6: Using the **EXPLAIN** Command

6.1. The **EXPLAIN** command shows the execution plan of a query, without actually executing the query. To demonstrate, add **EXPLAIN** to the beginning of the following query that you ran earlier in this lab:

```
hive> explain select * from wh_visits where fname = "JOE"
sort by lname;
```

6.2. Notice the query is executed in two stages. **Stage-0** performs the limit operator. This was the first mapper that executed the query.

6.3. Notice **Stage-1** is a MapReduce job that has one mapper (look for **Map Operator Tree**) and one reducer (under **Reduce Operator Tree**). As you can see from this execution plan, the mapper is doing most of the work.

Step 7: Use EXPLAIN EXTENDED

7.1. Run the previous **EXPLAIN** again, except this time add the **EXTENDED** command:

```
hive> explain extended select * from wh_visits where fname
= "JOE" sort by lname;
```

7.2. Compare the two outputs. Notice the **EXTENDED** command adds a lot of additional information about the underlying execution plan.

Lab 7.4: Joining Datasets in Hive

Objective:	Peform a join of two datasets in Hive.
Location of Files:	**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:	Start the Hive Shell.

Perform the following steps:

Step 1: Load the Data into Hive

1.1. View the contents of the file **setup.hive** in **LABS/Lab7.4**:

```
# more setup.hive
```

1.2. Notice 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.

1.3. Run the **setup.hive** script from the **Lab7.4** folder:

```
# hive -f setup.hive
```

1.4. To verify the script worked, enter the following query from the Hive Shell:

```
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”.

1.5. The **stock_aggregates** table should be empty, but view its schema to verify it was created successfully:

```
hive> describe stock_aggregates;
OK
symbol                  string                  None
year                    string                  None
high                    float                   None
low                     float                   None
average_close           float                   None
total_dividends         float                   None
```

Step 2: Join the Datasets

2.1. The **join** statement is going to be fairly long, so let’s create it in a text file. Create a new text in the **Lab7.4** folder named **join.hive**, and open the file with a text editor.

2.2. We will break the join statement down into sections. First, the result of the **join** is going to 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.

2.3. 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 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)
```

2.4. The from clause is the nyse_data table:

```
from nyse_data a
```

2.5. The join is going to be a left outer join of the **dividends** table:

```
left outer join dividends b
```

2.6. The join is by stock symbol and trade date:

```
on (a.symbol = b.symbol and a.trade_date = b.trade_date)
```

2.7. Let's group the result by symbol and trade date:

```
group by a.symbol, year(a.trade_date);
```

2.8. Save your changes to join.hive.

Step 3: Run the Query

3.1. Run the query and wait for the MapReduce jobs to execute:

```
# hive -f join.hive
```

3.2. How many MapReduce jobs does it take to perform this query?

_____

Step 4: Verify the Results

4.1. Run a **select** query to view the contents of **stock_aggregates**:

```
 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     81.0  93.737686  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_symbol      NULL  NULL  NULL  NULL  NULL
```

4.2. List the contents of the **stock_aggregates** directory in HDFS. The **000000_0** file was created as a result of the **join** query:

```
# hadoop fs -ls -R /apps/hive/warehouse/stock_aggregates/
  41109 /apps/hive/warehouse/stock_aggregates/000000_0
```

4.3. Notice you can also view the contents of the **stock_aggregates** table by using the **text** command:

```
# hadoop fs -text
/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 table is an aggregate of various statistics like max high, min low, etc.

Lab 8.1: Using HCatalog with Pig

Objective:	Use HCatalog to provide the schema for a Pig relation.
Location of Files:	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:	You will need to have completed the previous lab that created the **wh_visits** table in Hive.

Perform the following steps:

Step 1: Start the Grunt Shell

1.1. SSH into your HDP 2.0 virtual machine..

1.2. Start the Grunt shell for use with HCatalog:

```
# pig -useHCatalog
```

Step 2: Load a HCatalog Table

2.1. Define a relation for the **wh_visits** table in Hive using the **HCatLoader()**:

```
grunt> visits = LOAD 'wh_visits' USING
org.apache.hcatalog.pig.HCatLoader();
```

2.2. View the schema of the **visits** relation to verify it matches the schema of the **wh_visits** table:

```
visits: {lname: chararray,fname: chararray,time_of_arrival:
chararray,appt_scheduled_time: chararray,meeting_location:
chararray,info_comment: chararray}
```

Step 3: Run a Pig Query

3.1. Let's execute a query to verify everything is working. Define the following relation:

```
grunt> joe = FILTER visits BY (fname == 'JOE');
```

3.2. Dump the relation:

```
grunt> DUMP joe;
```

The output should be visitors from **wh_visits** with the firstname "JOE".

Step 4: Create an HCatalog Schema

4.1. Quit the Grunt shell and start the Hive shell.

4.2. 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);
```

4.3. Verify the table was created successfully using **'show tables'**.

4.4. Use the **describe** command to view the schema of **joes**:

```
hive> describe joes;
OK
fname                   string                  None
lname                   string                  None
comments                string                  None
```

Step 5: Using HCatStorer

5.1. Exit the Hive shell and start the Grunt shell again. Be sure to use the useHCatalog option:

```
# pig -useHCatalog
```

5.2. Define the **visits** and **joe** relations again (using the up arrow to browse through the history of Pig commands).

5.3. 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;
```

5.4. Store the projection into the HCatalog table using the **STORE** command:

```
grunt> STORE project_joe INTO 'joes' USING
org.apache.hcatalog.pig.HCatStorer();
```

This command failed. Why? _____

5.5. Notice 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:

```
project_joe = FOREACH joe GENERATE fname, lname,
info_comment AS comments;
```

5.6. Now run the **STORE** command again:

```
grunt> STORE project_joe INTO 'joes' USING
org.apache.hcatalog.pig.HCatStorer();
```

This time the command should work and a MapReduce job will execute.

Step 6: Verify the STORE Worked

6.1. Quit the Grunt shell and start the Hive shell again.

6.2. 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.

Step 7: View the Files

7.1. You can also check the file system to see if a **STORE** command worked. From the command line, view the contents of **/apps/hive/warehouse/joes**:

```
# hadoop fs -ls /apps/hive/warehouse/joes/
Found 1 items
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? _____

7.2. Use the **cat** command to view the contents of **part-m-00000**:

```
# hadoop fs -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.

ANSWERS:

6.4: 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.

8.1: The **part-m-00000** file is a result of the Pig MapReduce job that executed when you ran the **STORE** command with **HCatStorer**.

Lab 9.1: Advanced Hive Programming

Objective:	To understand how some of the more advanced features of Hive work, including multi-table inserts, views and windowing.
Location of Files:	**LABS/Lab9.1**
Successful Outcome:	You will have executed numerous Hive queries on customer order data.
Before You Begin:	SSH into your HDP 2.0 virtual machine.

Step 1: Create and Populate a Hive Table

1.1. From the command line, change directories to the **Lab9.1** folder:

```
# cd ~/labs/Lab9.1
```

1.2. View the contents of the **orders.hive** file in the **Lab9.1** folder:

```
# more orders.hive
```

Notice it defines a Hive table named **orders** that has 7 columns. Notice it also loads the contents of **/tmp/shop.tsv** into the orders table.

1.3. Copy **shop.tsv** into the **/tmp** folder:

```
# copy shop.tsv /tmp/
```

1.4. Execute the contents of **orders.hive**:

```
# hive -f orders.hive
```

1.5. From the Hive shell, verify the script worked by running the following two commands:

```
hive> describe orders;
hive> select count(*) from orders;
```

Your **orders** table should contain 99,999 records.

Step 2: Analyze the Customer Data

2.1. 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 ten first names.

2.2. 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:

```
Jeremy 10
Christina   10
Jasmine     10
Hannah 10
Thomas 10
Michelle    10
Brian  10
Amber  10
Maria  10
Victoria    10
```

2.3. 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:

```
Mark   612
Jordan 612
```

```
Anthony     612
Brandon     612
Sean   612
Paul   611
Nathan 611
Eric   611
Jonathan    611
Andrew 610
```

2.4. 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? _____

2.5. The **orderdate** column is a string with the format **yyyy-mm-dd**. You will need the **substr** 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:

```
SELECT sum(ordertotal), substr(order_date,0,4) FROM orders
GROUP BY substr(order_date,0,4);
```

The output should look like:

```
4082780     2009
4404806     2010
4399886     2011
4248950     2012
2570749     2013
```

Step 3: Multi-File Insert

3.1. In this step, you will run two completely different queries, but in a single MapReduce job. The output of the queries will be in two separate directories in HDFS. Start by creating a new text file in the **Lab9.1** folder named **multifile.hive**.

3.2. Within the text file, enter the following query. Notice there is no semi-colon between the two **INSERT** statements:

```
FROM ORDERS o
INSERT OVERWRITE DIRECTORY '2010_orders'
   SELECT o.* WHERE substr(order_date,0,4) = '2010'
INSERT OVERWRITE DIRECTORY 'software'
   SELECT o.* WHERE itemlist LIKE '%Software%';
```

3.3. Save your changes to **multifile.hive**.

3.4. Run the query from the command line:

```
# hive -f multifile.hive
```

3.5. The above query executes in a single MapReduce job. Even more interesting, it only requires a map phase. How come this job did not require a reduce phase?

_____

3.6. Verify the two queries executed successfully by viewing the folders in HDFS:

```
# hadoop fs -ls
```

You should see two new folders: **2010_orders** and **software**.

3.7. View the output files in these two folders. Verify the **2010_orders** directory contains orders from only the year 2010, and verify the **software** directory contains only orders that included **'Software'**.

Step 4: Define a View

4.1. 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.

4.2. Run the **show tables** command:

```
hive> show tables;
```

You should see **2013_orders** in the list of tables.

4.3. 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.

Step 5: Find the Maximum Order of Each Customer

5.1. 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;
```

5.2. How many different customers are in the **orders** table? _____

5.3. Suppose you want to add the itemlist column to the previous query. Try adding it to the **SELECT** clause:

```
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.

5.4. 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;
```

5.5. 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;
```

5.6. How many MapReduce jobs did this query need? _____

5.7. The end of your output should look like:

```
600    98
Grill,Freezer,Bedding,Headphones,DVD,Table,Grill,Software,D
ishwasher,DVD,Microwave,Adapter
600    99    Washer,Cookware,Vacuum,Freezer,2-Way
Radio,Bicycle,Washer & Dryer,Coffee
Maker,Refrigerator,DVD,Boots,DVD
600    100   Bicycle,Washer,DVD,Wrench Set,Sweater,2-Way
Radio,Pants,Freezer,Blankets,Grill,Adapter,pillows
```

NOTE: In the next lab, you will optimize this query using a custom Python script to avoid the need for two MapReduce jobs.

Step 6: Fixing the GROUP BY Key Error

6.1. Let's compute the sum of all of the orders of all customers. Start by entering the following query:

```
SELECT sum(ordertotal), userid FROM orders GROUP BY userid;
```

Notice the output is the sum of all orders, but displaying just the **userid** is not very exciting.

6.2. Try to add the **username** column to the **SELECT** clause:

```
SELECT sum(ordertotal), userid, username
FROM orders
GROUP BY userid;
```

This generates the infamous "Expression not in GROUP BY key" error, because **username** column is not being aggregated but the **ordertotal** is.

6.3. An easy fix is to aggregate the **username** values using the **collect_set** function, but output only one of them:

```
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.

Step 7: Using the **OVER** Clause

7.1. 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:

```
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 the orders ever placed from that particular customer.

Step 8: Using the Window Functions

8.1. It is not difficult to compute the sum of all orders for each day using the **GROUP BY** clause:

```
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
```

8.2. 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:

```
SELECT order_date, sum(ordertotal)
OVER
(PARTITION BY order_date ROWS BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW)
FROM orders;
```

To verify 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
```

Step 9: Using the Hive Analytics Functions

9.1. Run the following query, which displays the rank of the **ordertotal** by day:

```
SELECT order_date, ordertotal, rank()
OVER
(PARTITION BY order_date ORDER BY ordertotal)
FROM orders;
```

9.2. 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
```

9.3. 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.

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.

Demonstration: Computing PageRank

Objective:	To understand how to use the PageRank UDF in DataFu.
During this demonstration:	Watch as your instructor performs the following steps.

Step 1: View the Data

1.1. Review the **edges.txt** file in the **LABS/demos** folder:

```
# cd ~/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       1.0
0       5       6       1.0
0       6       5       1.0
0       6       2       1.0
0       100     2       1.0
0       100     5       1.0
0       101     2       1.0
0       101     5       1.0
0       102     2       1.0
0       102     5       1.0
0       103     5       1.0
0       104     5       1.0
```

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

1.3. 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.

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

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

_____

Step 2: Put the Data in HDFS

2.1. Put **edges.txt** into HDFS:

```
# hadoop fs -put edges.txt edges.txt
```

Step 3: Define the PageRank UDF

3.1. View the contents of **demos/pagerank.pig**. The first two lines register the DataFu library and define the **PageRank** function:

```
register /root/labs/Lab10.1/datafu-0.0.10.jar;
define PageRank datafu.pig.linkanalysis.PageRank();
```

3.2. 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);
```

3.3. 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;
```

3.4. 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;
```

3.5. 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)));
```

3.6. The results are stored in HDFS:

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

Step 4: Run the Script

4.1. From the command prompt, enter:

```
# pig pagerank.pig
```

The job will take a couple minutes to run.

Step 5: View the Results

5.1. View the contents of the **topicranks** folder in HDFS:

```
# hadoop fs -ls topicranks
Found 1 items
root hdfs        181   topicranks/part-r-00000
```

5.2. View the contents of the output file:

```
# hadoop fs -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
```

Step 6: Analyze the Results

6.1. Which page should be ranked the highest? _____

6.2. Which page should be ranked the lowest? _____

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

Demonstration: Computing ngrams

Objective:	To understand how to compute ngrams using Hive.
During this demonstration:	Watch as your instructor performs the following steps.

Step 1: Create a Hive Table for the Data

1.1. This demonstration computes ngrams on the U.S. Constitution, which is in a text file in the **LABS/demos** folder:

```
# cd ~/labs/demos/
# more constitution.txt
```

1.2. Start the Hive shell and define the following table:

```
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.

Step 2: Load the Hive Table

2.1. Load **constitution.txt** into the **constitution** table:

```
hive> load data local inpath
'/root/labs/demos/constitution.txt' into table
constitution;
```

2.2. Verify the data is loaded:

```
hive> select * from constitution;
```

You should see the contents of **constitution.txt** again.

Step 3: Compute a Bigram

3.1. Enter the following Hive command, which computes a bigram for the Constitution and shows the top 15 results:

```
hive> select explode(ngrams(sentences(line),2,15)) as x
from constitution;
```

The result should look like:

```
{"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}
```

Step 4: Compute a Trigram

4.1. Run the previous query again, but this time compute a trigram:

```
hive> select explode(ngrams(sentences(line),3,20)) as
result from constitution;
```

4.2. The result should look like:

```
{"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}
```

Step 5: Compute a Contextual ngram

5.1. 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;
```

5.2. The result looks like:

```
{"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}
```