HOL4041: Hands-On Hadoop

JavaOne '14

This hands-on lab will walk you through your first experience with using Hadoop. The target audience for this lab is users with some database and/or programming experience, but no prior experience with Hadoop. This lab covers a basic introduction to several core components of the Hadoop platform. Some exercises, however, include *Additional Exercises* sections that provided less guided exercises for more advanced users.

Exercise 0: Launching Your Virtual Machine

Before starting this lab, you should take a moment to make sure your virtual machine is configured correctly and all software component are working. Please follow these steps:

1. **Install VirtualBox (Not required at JavaOne)**

   If you have not already done so, please download and install the latest version of Oracle's VirtualBox software. The URL to download the software is here:

   http://xxx

   You can find installation instructions here:

   http://xxx

2. **Launch the Virtual Machine (Not required at JavaOne)**

   Open VirtualBox and from the *File* menu select *Import Appliance*.

   Browse to the location of the xxx.vmdk file.

   Select the xxx.vmdk file and click the *OK* button.

   …

3. **Load all the stuff from the Git repo (Not required at JavaOne)**
4. **Open a terminal window**
5. **Open a Firefox browser window**

Exercise 1: Exploring HDFS [15 minutes]

In this lab, you will be interacting with Hadoop through the command line in a terminal. If you do not already have a terminal window open, open one now by following step 4 in Exercise 0 above.

Introduction

As explained in the presentation available here:

http://slideshare.com/...

HDFS is the distributed file system around which Hadoop is built. HDFS allows for the reliable storage of data at massive scales and is optimized for the distributed computations that are the staple of MapReduce programs (typically called *jobs*).

HDFS is separate from the local file system, and the data it stores is spread out across the nodes in the Hadoop cluster. A single file in HDFS may be stored as thousands of *blocks* that are distributed across the entire cluster.

HDFS, like your local file system, has a notion of your user's home directory. In this VM environment, your user name is *cloudera*. Your HDFS home directory is `/user/cloudera`.

Interacting with HDFS is done through the `hadoop` or `hdfs` command. In this lab we will use the `hadoop` command. To view the files in your HDFS home directory, from the terminal window run:

```
hadoop fs -ls
```

You should notice that the command returns with no output. The reason is that your home directory is currently empty. Even though there are files in your local file system's home directory, there are currently none in HDFS.

As with the Linux `ls` command, you can view the files in a specific location by specifying a directory path:

```
hadoop fs -ls /
```

The output above shows you the contents of the HDFS root directory, which includes the `/user` directory where the home directories are stored. Notice that the output from the `hadoop fs -ls` command looks similar to the output from the Linux `ls -l` command. File listings in HDFS are always in the long format, meaning that the file mode, ownership, and size is always displayed.

Data Ingest

In the later sections of this lab, you'll be performing transformations and queries on data in HDFS. Since HDFS is currently empty, you'll need to load some data into HDFS for those later sections. In the `/home/cloudera/handsonhadoop/data` directory you will find a data file called `users.csv`. This file contains the data that will be used through the rest of this lab.

Before you upload the file into HDFS, you should first examine the file a little. Run the following commands:

```
ls -lh handsonhadoop/data/users.csv

wc -l handsonhadoop/data/users.csv

head handsonhadoop/data/users.csv
```

From the output of these commands, you can see that this file is XXXMB and contains XXX

lines of fairly simple comma-delimited data points. You can see from first few rows that the data appears to be some kind of user records.

To upload the file into HDFS, use the `hadoop fs -put` or `hadoop fs -copyFromLocal` command. Both are equivalent.

```
hadoop fs -put handsonhadoop/data/users.csv
```

Now, if you run the command to list the files in your home directory again, you'll find that the `users.csv` file is there. Notice that the file size reported by HDFS is the same as the file size reported by the local file system.

Notice that the file was placed into your home directory, even though the path on the local file system includes a directory path. Because you did not specify where to place the file, HDFS automatically places the file directly into your home directory. The additional directories in the local file path are not created or transferred.

While it's excellent that you now have your data loaded into HDFS, it's generally a good idea to keep your home directory tidy, just as with your local file system. You should create a directory in HDFS for this lab and store the data file there. To create a directory in HDFS, run the following command:

```
hadoop fs -mkdir javaone
```

If you list the files in your home directory now, you will see both the data file and the directory you just created. HDFS has no notion of the current working directory, so whenever you issue an HDFS command, you must either give a fully qualified path or a path that is relative to your home directory.

In the `/home/cloudera/handsonhadoop/data` directory you will find a second data file called `logins.log`. Run the following commands:

```
wc -l handsonhadoop/data/logins.log

head handsonhadoop/data/logins.log
```

From the output, you can see that the `logins.log` file has very different contents from the `users.csv` file.

Upload this second data file into your newly created directory by running:

```
hadoop fs -put handsonhadoop/data/logins.log javaone
```

You can also rename a file during the upload. Run the following commands:

```
hadoop fs -put handsonhadoop/data/logins.log javaone/logins2.log

hadoop fs -ls javaone
```

You can see that there are now two copies of the file in the directory: one named `logins.log` and one named `logins2.log`. Note that they are both the same size.

Now you need to get a copy of the `users.csv` file into the `javaone` directory as well. Rather than uploading the file again, you can instead simply move the file from your HDFS home directory into the new directory. Before you try that, you should first remove the second copy

of the `logins.log` file from the `javaone` directory:

```
hadoop fs -rm javaone/logins2.log
```

You can now move the file from your HDFS home directory into the new directory by running:

```
hadoop fs -mv users.csv javaone
```

Data Access

Now that you have your data in HDFS, you may be wondering what you can do with it. The rest of the exercises in this lab will focus on using the core tools in the Hadoop platform to work with the data you've uploaded. There is, however, one other common way to access the data in HDFS that belongs in this exercise. Once you've processed your data, how do you extract it from HDFS?

The simplest way is using the `hadoop fs -get` command:

```
hadoop fs -get javaone/users.csv
```

After running the above command, you will have a copy of the `users.csv` file in your local filesystem's home directory. As with the `hadoop fs -put` command, if you want to place the file into another directory or rename it during the transfer, you can provide a second argument to the command.

One common pattern in Hadoop is to treat a directory like a single file. HDFS is a write-once file system, meaning that once a file has been written, it cannot be later modified. (In some cases, appending to an existing file is supported.) For this reason (and a couple others), users will typically treat a directory like a file. If you want to modify the “file” represented by the directory, you can add or delete files within the directory. You will see this pattern used in later exercises. To facilitate this pattern, HDFS includes a command to download a directory as a single file, `hadoop fs -getmerge`. To download the `javaone` directory as a single file, run the following command:

```
hadoop fs -getmerge javaone javaone.csv
```

If you now view the line counts of the files in your local filesystem's home directory:

```
wc -l handsonhadoop/data/logins.log handsonhadoop/data/users.csv javaone.csv
```

you will see that the `javaone.csv` file is the size of the `users.csv` file and `login.log` file combined.

The merged file has exactly the contents of all the files in the `javaone` directory merged together. There are no headings to mark file boundaries, and the individual file names are not preserved. The `javaone` directory is treated as a single data set that just happens to be stored in more than one file.

In this case, having both files in the same directory is probably a bad idea since the two files have different schemas. To fix that issue, run the following commands to create two new subdirectories and move the files into them:

```
hadoop fs -mkdir javaone/users
```

```
hadoop fs -mv javaone/users.csv javaone/users

hadoop fs -mkdir javaone/logins

hadoop fs -mv javaone/logins.log javaone/logins

hadoop fs -ls -R javaone
```

From the output of the last command, you can see that you now have a tidy directory structure for your data.

What if your data is so large, that downloading it just to look at it is inconvenient or impossible? Hadoop gives you several methods for access your data in HDFS without downloading it.

If you just want to see what kind of data is in a file, the `hadoop fs -tail` command is useful. Just like the Linux `tail` command, `hadoop fs -tail` shows you the end of the file. (Note that in some versions of Hadoop, the `hadoop fs -tail` command has a bug that causes the beginning of the first line of output to be left off.) Run the following command:

```
hadoop fs -tail 'javaone/*/*'
```

The output shows you the last lines of both data files. In the command you used the wildcard operator (`*`) to tell HDFS to look in all subdirectories for any files that end with `.csv`. Notice that the expression with the wildcards is in single quotes. You must quote (single or double) any wildcards that you want to pass in the command line to HDFS. If you do not, the local shell will interpret them according to the local filesystem, which in most cases not what you want. (Alternately, you can escape wildcards with a backslash (`\`))

A more generally useful way to access your data in HDFS without downloading it is the `hadoop fs -cat` command, which behaves the same as the Linux `cat` command. The `hadoop fs -cat` command is particularly helpful when you want to perform some operation on the file contents, such as using `grep` to look for a particular bit of text or using `wc -l` to count the lines. To count the lines in your data files, run the following commands:

```
hadoop fs -cat 'javaone/users/*' | wc -l

hadoop fs -cat 'javaone/logins/*' | wc -l
```

The output should be the same as what you saw when counting the lines of the files on your local filesystem earlier. Notice that you used two commands instead of one command with a wildcard for the subdirectory. The reason is that if you had used a wildcard for the subdirectory, both data files would have been concatenated together, and the line count would have been the total for both files.

Also notice that in these two command you used the wildcard to effectively treat the subdirectories as files. Were there more than one file in each, you'd have gotten the total line count for all the files in each subdirectory. This use of wildcards is very common in Hadoop, and we'll use it repeatedly in this lab.

Additional Exercises

1.

Summary

You have now explored getting data into and out of HDFS. You understand how to issue HDFS commands, how the HDFS directory structure is similar to the Linux directory structure, and how to use wildcards.

For more information about Hive, these resources are recommended:

- Hadoop: the Definitive Guide
- The Hadoop File System Shell documentation: http://archive.cloudera.com/cdh5/cdh/5/hadoop/hadoop-project-dist/hadoop-common/FileSystemShell.html
- The HDFS User's Guide: http://archive.cloudera.com/cdh5/cdh/5/hadoop/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html

Exercise 2: Working With Hive [25 minutes]

In this lab, you will be interacting with Hadoop through the Hue web user interface using the Firefox browser. If you do not already have a Firefox browser window open, open one now by following step 5 in Exercise 0 above.

Introduction

The presentation available here:

http://slideshare.com/...

explains that Hive is a component of the Hadoop ecosystem that lets users operate on data stored in HDFS using a SQL-like language called HiveQL. HiveQL is almost SQL-92 compliant, so most of HiveQL will be familiar to users who have experience with SQL.

Before going any further, it's worth taking a moment to discuss how Hive can let users query "unstructured" data. If there is no structure, how can it be queried? Except in a few (generally uninteresting) cases, all data has some structure to it. The term "unstructured" generally means data that does not have a fixed schema. A CSV file is unstructured by that definition, as there's nothing enforcing the types or sizes (or number!) of the fields in a CSV file. If a line in a CSV file contains bad data, then that line either requires human intervention to fix, or it just isn't readable. Hive allows users to overlap a schema on top of data and then query that data using the schema. The powerful thing about Hive is that if that schema turns out not to be useful, then the schema can be discarded or replaced without changing the data at all. This concept is often known as "schema on read." The schema is not applied to the data until you try to query it.

Creating Metadata

For this exercise, we're going to use the Hue user interface rather than the terminal. You can use Hive from the command line by running the `hive` command, but the user experience with Hue is much nicer for many purposes.

In Hive, before we can work with our data, we first have to associate a schema with it. The schema is most often referred to as the "metadata." The way to associate metadata with our

data is to create a new Hive table. Unlike in a traditional database, in Hive a table is nothing more than metadata. The data itself is stored independently in HDFS and has a separate life cycle from the metadata, e.g. you can delete the metadata without impacting the data.

1. In the Hue browser window, click on the *Hive* link in the *Query* panel.
2. In the *Query Editor* box, enter:

   ```
   show tables;
   ```

   The output will be displayed when the command completes. What you should see is that there are currently two tables defined in Hive, `sample_07` and `sample_08`. Those are two sample tables that come with the Quickstart VM. You won't be using them in this lab.
3. Click on the *Query Editor* tab in the menu bar to return to the query editor page.
4. In the *Query Editor* box, enter:

   ```
   create external table users (
    id int, fname string, lname string, address string,
    city string, state string, zip int
   )
   location '/user/cloudera/javaone/users';
   ```

   In the `create` statement, the keyword `external` tells Hive that the metadata should be decoupled from the data. If the `external` keyword is left out, deleting the created table will also delete the data from HDFS.

   The `location` keyword tells Hive where to find the data to which this metadata should be applied. The location must be a directory; it cannot be a single file.

   The output will be empty as this SQL statement does not return any results.
5. Click on the *Query Editor* tab in the menu bar to return to the query editor page.
6. In the *Query Editor* box, enter:

   ```
   select * from users;
   ```
7. The output will show all users in the table. Notice that there's something wrong with the data. All columns in all rows are “NULL”. In the previous exercise you verified that the data in HDFS is complete and correct, so the issue is most likely with the metadata, i.e. the table definition.

   You may recall that the data file is comma-delimited. By default, Hive uses 0x01 (or CTRL-A) as the field delimiter, which would explain why all the columns are “NULL”: Hive is using the wrong delimiters to parse the data.
8. To resolve the issue, first you should remove the bad table definition. Recall that you created the table with the external keyword, which means you can delete the table without impacting the data.

   Click on the *Query Editor* tab in the menu bar to return to the query editor page.

   In the *Query Editor* box, enter:

   ```
   drop table users;
   ```

   The output will be empty as this SQL statement does not return any results.
9. Click on the *Query Editor* tab in the menu bar to return to the query editor page.
10. In the *Query Editor* box, enter:

```
create external table users (
 id int, fname string, lname string, address string,
 city string, state string, zip int
)
row format delimited fields terminated by ','
location '/user/cloudera/javaone/users';
```

The `row format` portion of the statement tells Hive to use commas the field delimiter. By default, Hive uses 0x01 (or CTRL-A) as the field delimiter.

The output will be empty as this SQL statement does not return any results.

11. Click on the *Query Editor* tab in the menu bar to return to the query editor page.
12. In the *Query Editor* box, enter:

```
select * from users;
```

The output will show all users. Notice that the data appears to be correct now.

Querying

Now that table queries are working correctly, you can explore the data using some more complex queries.

1. Click on the *Query Editor* tab in the menu bar to return to the query editor page.
2. In the *Query Editor* box, enter:

```
select id, fname, lname from users where state='CA';
```

The output will show all of the users located in California. Notice that the processing time for this query is significantly more than for the previous query. The reason this query is slower to process is that it requires running a MapReduce job to complete. The previous query was the special case of a sequential read of the data, which is handled by reading directly from HDFS.

3. Click on the *Query Editor* tab in the menu bar to return to the query editor page.
4. In the *Query Editor* box, enter:

```
select state, count(*) from users group by state;
```

The output will show each state along with the number of users in that state.

5. Click on the *Query Editor* tab in the menu bar to return to the query editor page.
6. In the *Query Editor* box, enter:

```
select state, collect_list(state) from users group by state;
```

The output will show each state along with a list of the user IDs in that state. One feature of Hive that is not commonly available in traditional databases is the use of complex data types, like lists and maps. Performing this query with a traditional database can be quite challenging.

Non-tabular data

The previous steps were working with tabular user data. As long as the delimiters and field types are set reasonably, tabular data is quite easy to use in Hive. Fortunately, non-tabular data, such as the `logins.log` file, can also be processed in Hive easily.

1. <!-- XXX Add the Regex JAR XXX -->
2. Click on the *Query Editor* tab in the menu bar to return to the query editor page.
3. In the *Query Editor* box, enter:

```
create external table logins (
 id string, state string, time string, day string
)
row format serde 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
with serdeproperties (
  "input.regex" =
    "(\\d+) in ([A-Z]{2}) at (\\d{2}:\\d{2}:\\d{2}) on
(\\d{2}/\\d{2}/\\d{2})"
)
location '/user/cloudera/javaone/logins';
```

   This time the row format is “SerDe”, which means you'll be using a specified class for serializing and deserializing the data. The SerDe you're using is the regular expression SerDe, with the regular expression used to parse the data set specified in the `input.regex` property.

   The output will be empty as this SQL statement does not return any results.
4. Click on the *Query Editor* tab in the menu bar to return to the query editor page.
5. In the *Query Editor* box, enter:

```
select * from logins;
```

   The output will show all logins in the table. With minimal effort, you have now turned a text log file into a queryable table.

Data and Metadata

To demonstrate the decoupling between data and metadata, you're now going to perform some operations directly on the data behind the `logins` table.

1. In the terminal window, run the following command:

```
hadoop fs -put javaone/logins.log javaone/logins/logins2.log
```

2. In the Hue Hive Query Editor, run the previous query again:

```
select * from logins order by id;
```

   The output will show all logins in the table sorted by `id`. Notice that there are now twice as many entries for each user. By altering the data behind the table, you altered the contents of the table without needing to make any updates in Hive.
3. In the terminal window, run the following command:

```
hadoop fs -rm 'javaone/logins*.log'
```

4. In the Hue Hive Query Editor, run the previous query again:

```
select * from logins order by id;
```

   The output will show no rows because the table now contains no data.
5. In the terminal window, run the following command:

```
hadoop fs -put javaone/logins.log javaone/logins
```

6. In the Hue Hive Query Editor, run the previous query again:

```
select * from logins order by id;
```

7. The output will show that you're now back to where you started.

Joins

One of the most powerful features of SQL is the ability to join two data sets together. Hive extends that capability to function across different data formats. You'll now demonstrate those capabilities by joining together the two tables you just created.

1. Click on the *Query Editor* tab in the menu bar to return to the query editor page.
2. In the *Query Editor* box, enter:

   ```
   select id, collect_set(state) as states from logins group by id;
   ```

   The `collect_set` operation combines values from a column across different rows into an array of unique elements. Hive supports three non-primitive data types: arrays, maps, and structs. Here you've named the collection `states`.

   The output will be every user id along with an array of all the states from which that user logged in.
3. Click on the *Query Editor* tab in the menu bar to return to the query editor page.
4. In the *Query Editor* box, enter:

   ```
   select users.id, users.fname, users.lname, states.states from users
   join (
    select id, collect_set(state) as states from logins group by id
   ) states on users.id = states.id;
   ```

   This query does two things that are interesting. The first is that it performs a join between two tables. The second is that one of the joined tables is a nested query.

   First the nested query is executed. It's the same query you entered in the previous steps. The results of that query are then stored in a temporary table that you've named `states`.

   Second, the `users` table is joined to the `states` table by combining rows where the ID columns are the same. The final results are the ID, names, and login states for all users.

Additional Exercises

1. Write a query to return the list of unique states where users live.
2. Write a query to return the Ids of all users who logged in between 00:00 (midnight) and 01:00.
3. Write a query that lists the names of the users who logged in from each state, with one row per state. For example, if Bob Jones and Sally Jean logged in from CA, then the CA line would contain:

   ```
   CA    ["Bob Jones", "Sally Jean"]
   ```
4. Write a single query statement to calculate the mean number of states from which users have logged in. For example, if Bob logged in from 1 state, and Sally logged in from 2 states, and there are no other users, the mean is (1+2)/2 = 1.5.

Summary

In this exercise, you've only just scratched the surface of what's possible with Hive. You have seen how to overlay metadata over existing data to make it queryable, both with tabular and non-tabular data. You've seen how to work with the data behind the tables. You've seen how to use both joins and nested queries to perform complex queries in Hive.

To read more about Hive, these resources are recommended:

- Hadoop: the Definitive Guide
- Hive Language Manual: https://cwiki.apache.org/confluence/display/Hive/LanguageManual

Exercise 3: Working With Impala [10 minutes]

In this lab, you will be interacting with Hadoop through the Hue web user interface using the Firefox browser. If you do not already have a Firefox browser window open, open one now by following step 5 in Exercise 0 above.

Introduction

As explained in the presentation available here:

http://slideshare.com/...

Cloudera Impala is an Apache-licensed open source technology for executing high-performance queries on data in HDFS. Impala is built to be fully compatible with Hive, including support for HiveQL and the sharing of Hive's metadata. The primary difference between Impala and Hive is that Impala is a proper parallel database engine, whereas Hive is a toolkit that translates HiveQL statements into MapReduce jobs. The result is that Impala will typically perform HiveQL queries significantly faster than Hive would, using the same data and same machines. Another notable advantage of Impala is that if you're familiar with using Hive, you're also familiar with using Impala, as the same query language is supported, with only some minor differences.

Querying

Since Impala supports the same syntax as Hive, in the following steps you'll rerun the Hive queries from the previous exercise to compare the performance and results.

1. In the Hue browser window, click on the *Impala* link in the *Query* panel.
2. In the *Query Editor* box, enter:

   ```
   select id, fname, lname from users where state='CA';
   ```

   The output will show all of the users located in California. Notice that the processing time for this query is significantly more than for the previous query. The reason this query is slower to process is that it requires running a MapReduce job to complete. The previous query was the special case of a sequential read of the data, which is handled by reading directly from HDFS.
3. Click on the *Query Editor* tab in the menu bar to return to the query editor page.
4. In the *Query Editor* box, enter:

```
select state, count(*) from users group by state;
```

The output will show each state along with the number of users in that state.

5. Click on the *Query Editor* tab in the menu bar to return to the query editor page.
6. In the *Query Editor* box, enter:

```
select state, collect_list(state) from users group by state;
```

The output will show each state along with a list of the user IDs in that state. One feature of Hive that is not commonly available in traditional databases is the use of complex data types, like lists and maps. Performing this query with a traditional database can be quite challenging.

You may have noticed that you only ran queries against the users table. The reason is that the logins table uses a Hive SerDe to read the data, and Impala requires you to use an Impala SerDe. This is one of the main cases where portability across Hive and Impala is not seamless. To get around the issue, you could you Hive to read the log data and write it into a more traditionally delimited table. This approach is, in fact, a best practice as the performance of queries that have to parse text files is generally poor.

Summary

In this exercise you reran the same queries from the previous exercise. You saw that Impala was able to use the same tables and data as Hive in many cases, and that the query times in Impala are considerably shorter, even on a single-node cluster running in a VM.

In general, the rule of thumb for deciding between Hive and Impala is to use Impala for exploratory or interactive queries against tabular data, and to use Hive for very large scale batch queries or in cases when a particular Hive SerDe or UDF is needed.

Exercise 4: Working With Pig

In this lab, you will be interacting with Hadoop through the command line in a terminal and through the Hue web user interface using the Firefox browser. If you do not already have a terminal window and a Firefox browser window open, open them now by following steps 4 and 5 in Exercise 0 above.

Exercise 5: Working With Hadoop Streaming

In this lab, you will be interacting with Hadoop through the command line in a terminal and through the Hue web user interface using the Firefox browser. If you do not already have a terminal window and a Firefox browser window open, open them now by following steps 4 and 5 in Exercise 0 above.

Summary