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 and replicated 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/javaone14_handsonhadoop/data` directory you will find a data file called `users.csv`. This file contains 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 javaone14_handsonhadoop/data/users.csv

wc -l javaone14_handsonhadoop/data/users.csv

head javaone14_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 javaone14_handsonhadoop/data/users.csv
```

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.

NOTE: 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.

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

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

In the `/home/cloudera/javaone14_handsonhadoop/data` directory you will find a second data file called `logins.log` to be uploaded into HDFS. Run the following commands:

```
wc -l javaone14_handsonhadoop/data/logins.log

head javaone14_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 javaone14_handsonhadoop/data/logins.log 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 are, however, some other common ways to access the data in HDFS that belongs in this exercise.

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 javaone14_handsonhadoop/data/* javaone.csv
```

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

NOTE: 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. Write a command to count the total number of lines in both data files.
2. Write a command to count the number of users named Daniel.
3. Write a command to print the first and last names of all users in California.
4. Write a command to print the list of unique first names.

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 [30 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" therefore generally means data that has structure but 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 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, and vice versa.

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 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. If you do not specify a location, Hive will create a directory for the table under `/user/hive/warehouse`.

   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 as 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_set(id) as ids 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.

    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 `ids`.

Non-tabular data

The previous steps worked 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
```

   This command will place a second copy of the `logins.log` file into the `javaone/logins` directory and call it `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;
   ```

   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 or more 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 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 can, 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. Compare the processing time with the processing time for the same query in the previous exercise.

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.

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 does not support Hive SerDes. This difference is one of the main cases where portability across Hive and Impala is not seamless. To get around the issue, you should use 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 using UDFs or SerDes 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 [30 minutes]

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 [20 minutes]

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.

Introduction

As explained in the presentation available here:

http://slideshare.com/...

Hadoop is composed of two core components: HDFS and MapReduce. As you saw in exercise 1, HDFS is a distributed file system for storing data. MapReduce is the execution engine that underlies Hive and Pig and is the component that did all the actual work in

exercises 2 and 4. (Impala does not rely on MapReduce.) In addition to using an abstraction layer like Hive or Pig, you can also use MapReduce directly.

MapReduce jobs have two main components: map tasks and a reduce tasks. To understand these components, imagine you have three friends, and you'd like the to sort a shuffled deck of cards for you so that all the cards of each suit are together, and all the cards within a suit are sorted from 2 to 10, jack, queen, king, ace. How would you do it most efficiently? One approach you might take is to divide the deck into three chunks and hand one chunk to each friend. This is analogous to storing a data set in HDFS where it is broken into chunks that are distributed across the cluster. Each of your friends would then take her cards and sort them into four piles, one for each suit. This is analogous to the map phase. When everyone had finished sorting, you'd then take all of the hearts and give them to one friend. You give the clubs to another, and so on. (Note that this means one friend will have the cards for two suits.) Each of your friends would then sort the cards in each of her suits. This is analogous to the reduce phase. Finally, when all the sorting is done, you'd collect the cards and stack them up to produce a sorted deck of cards. This is analogous to a getmerge from HDFS.

In more concrete terms, MapReduce will read an input data set from HDFS and feed it, record by record, to a set of independent map tasks. Each map task will produce from each record some amount of intermediate data that has a key associated with it. When all of the input data has been processed by the map tasks, MapReduce will gather up all the intermediate data, group it by keys, and pass it to a set of independent reduce tasks. Each reduce task is guaranteed to get all of the data for each key that it receives. Each reduce task will read the intermediate data, key by key, and produces some amount of final data that is output back into HDFS. Can you see now how sorting cards fits into this model?

There are a couple of methods for using MapReduce, primarily MapReduce jobs written in Java and MapReduce jobs executed through the Hadoop streaming tool. In this exercise we'll focus on Hadoop streaming. Hadoop streaming actually has nothing to do with data streaming. It is instead a tool for running MapReduce jobs not written in Java. (There is also a separate tool for writing jobs in C++ called Hadoop pipes.) Hadoop streaming can be used to run jobs written in any language, including Python, Perl, C, Java, and even Linux command-line utilities.

In this exercise, you'll use Hadoop streaming to operate on the data you uploaded into HDFS in exercise 1.

Hadoop Streaming

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

   ```
   export STREAMING=/usr/lib/hadoop-mapreduce/hadoop-streaming.jar
   ```

   This command will add an environment variable that you'll use in the following steps.

2. To get familiar with Hadoop streaming, you'll start with the simplest job that does something useful: translate the `users.csv` file from comma-delimited to tab-delimited.

   In the terminal window, run the following command:

   ```
   hadoop jar $STREAMING -mapper "tr , '\t'" -numReduceTasks 0 \
   -input javaone/users -output javaone/users.tsv
   ```

   The `-mapper` argument tells Hadoop streaming what command to run as the map task. The `-numReducerTasks` sets the number of reducers, in this case none. The `-input` and `-output` arguments tell Hadoop streaming where to get the data and put the

results.

When the job has finished, the output will show you many lines of data about the job execution. You can see, for example, the number of records and the number of bytes of data that were processed in each phase. Notice that there is no information about the reduce phase because this was a “map-only” job.

NOTE: When you run a Hadoop job, the output directory must not already exist. If you are attempting to rerun a job, even if the job failed on the previous attempt, you must always remove the output directory first. The command to remove the output directory for the above job is:

```
hadoop fs -rm -R javaone/users.tsv
```

The `-R` works just like it does in the Linux `rm` command – it tells Hadoop to delete the directory and everything in it.

---

3. To see the results produced by the job, run the following commands:

   ```
   hadoop fs -ls javaone/users.tsv
   ```

   Notice that the file listing shows that the `users.tsv` “file” is really a directory that contains a number of “part” files, each produced by one map task. Typically the part files are produced by the reduce tasks, but since this was a map-only job, the map tasks produced the output.

   ```
   hadoop fs -cat 'javaone/users.tsv/\*' | head
   ```

   You can see that all of the commas in the file have now been replaced by tabs.

4. To try something a little more complicated, run the following commands:

   ```
   hadoop jar $STREAMING -mapper "grep ,CA," -reducer "wc -l" \
   -numReduceTasks 1 -input javaone/users -output javaone/cacount
   ```

   Here you've set `grep` as the map task, looking for records that have “CA” as a field, and you've set `wc -l` as the reduce task. Note that you've set the number of reduce tasks to 1 explicitly. Why only one?

   ```
   hadoop fs -cat 'javaone/cacount/*'
   ```

   The results should agree with the count you found earlier.

5. To get the counts for all the states, run the following commands:

   ```
   hadoop jar $STREAMING -mapper "awk -F, '{print $6}'" -reducer "uniq -c" \
   -numReduceTasks 1 -input javaone/users -output javaone/count
   ```

   ```
   hadoop fs -cat 'javaone/count/*' | head
   ```

   The results should again agree with the counts you found earlier.

   Recall that `uniq -c` requires that the data be in sorted order. You can see from this command, then, that between the map and reduce phases, MapReduce is collecting and sorting the intermediate data, a phase know as the shuffle and sort phase.

6. Now, to understand Hadoop streaming a little deeper, you'll write a short MapReduce job in Python. Create a file in your home directory called `map.py` with the following contents:

```
import sys

# Read records from stdin
for line in sys.stdin:
    # Split them on commas
    fields = line.strip().split(',')

    # Output the state followed by the full name
    print "%s\t%s %s" % (fields[6], fields[1], fields[2])
```

(You can either use vi or emacs from the command line, or you can open a text editor by selecting *Applications → Accessories → gEdit Text Editor* from the desktop menu bar.)

The most important thing to note that this script is that it is reading input records from STDIN and writing the intermediate data out to STDOUT. This input/output contract is the only requirement placed on the commands executed by Hadoop streaming. By default, the tab character (`\t`) is taken as the delimiter between the key and the value in the output from a task, but as long as both the map and reduce phases are commands begin run through Hadoop streaming, the key/value delimiter makes very little difference. (It is possible to include Java MapReduce tasks in a Hadoop streaming job, in which case use of the key/value delimiter matters.)

Create a second file in your home directory called `reduce.py` with the following contents:

```
import sys

# Track the last key seen
last = None

# Read records from stdin
for line in sys.stdin:
    # Split the records on tabs
    fields = line.strip.split('\t')

    # If this is a new key...
    if last != fields[0]:
        # If there was a previous key...
        if last != None:
            # Print out the cached data
            print "%s,%s" % (last, str(users))

        # Store the new key and reset the cache
        last = fields[0]
        users = list()

    # Add this name to the cache
    users.append(fields[1])

# If there was a last key, print out the cached data
if last != None:
    print "%s,%s" % (last, str(users))
```

Notice that in Hadoop streaming, it's up to you to manage the keys. In this script, you explicitly track the keys so that when the next key appears you can output the data for the previous key. (Remember that the intermediate data comes to the reduce tasks already sorted by key.)

To run this job, run the following commands:

```
hadoop jar $STREAMING -mapper "python map.py" -reducer "python reduce.py" \
-input javaone/users -output javaone/stateusers \
-file ~/map.py -file ~/reduce.py
```

This time you have given your scripts as the map and reduce tasks. Note that you did not specify the number of reducer tasks, which means that you will get the default number of reducers as specified by the cluster config. (How many is that?) Note also the `-file` arguments. In order to run your scripts, a copy must be available from each of the nodes where the tasks will run. Each `-file` argument tells Hadoop streaming to place a copy of the specified file into the working directory of every task on every node where a task will run. This process is called "file staging."

```
hadoop fs -cat 'javaone/stateusers/*' | head
```

The output should list each state followed by the names of the users in that state.

Summary

In this quick tour of Hadoop streaming, you've seen how to use Hadoop streaming to parallelize Linux command-line tools to operate on large data sets. You've also seen how Hadoop streaming let's you write Hadoop jobs using scripts and programs that adhere to a simple input/output contract. Because the input/output contract is the only requirement, Hadoop streaming is a very flexible tool, accommodating jobs written in any language that the worker nodes can run.

To read more about Hadoop streaming, these resources are recommended:

- Hadoop: the Definitive Guide
- Hadoop Streaming Manual: http://hadoop.apache.org/docs/r2.5.0/hadoop-mapreduce-client/hadoop-mapreduce-client-core/HadoopStreaming.html

Additional Exercises

1. Write a Hadoop streaming job that counts the number of unique users who have logged in in the data in the `javaone/logins` directory in HDFS.
2. Write a Hadoop streaming job that counts the number of unique logins in the data in the `javaone/logins` directory in HDFS.
3. Write a Hadoop streaming job that parses the data in the `javaone/logins` directory in HDFS and outputs the user id, state, time, and date, all separated by commas, for each login.
4. Write a Hadoop streaming job that parses the data in the `javaone/logins` directory in HDFS and outputs the each user id and a list of the states from which that user has logged in.

Exercise 6: Java MapReduce [Bonus]

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.

Introduction

As discussed in the previous exercise, Hadoop offers different ways to run jobs. In the previous exercise you looked at Hadoop streaming, which is convenient in many cases, but is not the most common way to run jobs. The most common approach is to write Java MapReduce jobs. In this bonus exercise, you'll examine the anatomy of a Java MapReduce job and then run it.

Job Code

The following code is a complete Java MapReduce job that will produce for each user the list of states from which that user has logged in that are not the user's home state. In the sections following the code you will find a full description of how the code works.

```
 1 package javaone;
 2
 3 import java.io.IOException;
 4 import java.util.HashSet;
 5 import java.util.Set;
 6 import java.util.regex.Matcher;
 7 import java.util.regex.Pattern;
 8 import org.apache.hadoop.conf.Configuration;
 9 import org.apache.hadoop.conf.Configured;
10 import org.apache.hadoop.fs.Path;
11 import org.apache.hadoop.io.IntWritable;
12 import org.apache.hadoop.io.LongWritable;
13 import org.apache.hadoop.io.Text;
14 import org.apache.hadoop.mapreduce.Job;
15 import org.apache.hadoop.mapreduce.Mapper;
16 import org.apache.hadoop.mapreduce.Reducer;
17 import org.apache.hadoop.mapreduce.lib.input.MultipleInputs;
18 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
19 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
20 import org.apache.hadoop.util.Tool;
21 import org.apache.hadoop.util.ToolRunner;
22
23 public class HandsOnHadoop extends Configured implements Tool {
24   private static final String LOGIN =
25           "(\\d+) in ([A-Z]{2}) at (\\d{2}:\\d{2}:\\d{2}) "
26           + "on (\\d{2}/\\d{2}/\\d{2})";
27
28   public static void main(String[] args) throws Exception {
29     ToolRunner.run(new Configuration(), new HandsOnHadoop(), args);
30   }
31
32   @Override
33   public int run(String[] strings) throws Exception {
34     Job job = Job.getInstance(getConf());
35
36     job.setJarByClass(HandsOnHadoop.class);
37
38     MultipleInputs.addInputPath(job, new Path("javaone/users"),
39             TextInputFormat.class, UsersMap.class);
40     MultipleInputs.addInputPath(job, new Path("javaone/logins"),
41             TextInputFormat.class, LoginsMap.class);
42     FileOutputFormat.setOutputPath(job, new Path("javaone/nohome"));
43
44     job.setReducerClass(Reduce.class);
45     job.setOutputKeyClass(IntWritable.class);
46     job.setOutputValueClass(Text.class);
47
48     return job.waitForCompletion(true) ? 0 : 1;
49   }
50
```

```
51   class UsersMap extends Mapper<LongWritable, Text, IntWritable, Text> {
52     private IntWritable k = new IntWritable();
53     private Text v = new Text();
54
55     @Override
56     protected void map(LongWritable key, Text value, Context context)
57             throws IOException, InterruptedException {
58       String[] parts = value.toString().split(",");
59
60       k.set(Integer.parseInt(parts[0]));
61       v.set("-" + parts[5]);
62
63       context.write(k, v);
64     }
65   }
66
67   class LoginsMap extends Mapper<LongWritable, Text, IntWritable, Text> {
68     private final Pattern p = Pattern.compile(LOGIN);
69     private IntWritable k = new IntWritable();
70     private Text v = new Text();
71
72     @Override
73     protected void map(LongWritable key, Text value, Context context)
74             throws IOException, InterruptedException {
75       Matcher m = p.matcher(value.toString());
76
77       if (m.matches()) {
78         k.set(Integer.parseInt(m.group(1)));
79         v.set(m.group(2));
80
81         context.write(k, v);
82       }
83     }
84   }
85
86   class Reduce extends Reducer<IntWritable, Text, IntWritable, Text> {
87     private Text v = new Text();
88
89     @Override
90     protected void reduce(IntWritable key, Iterable<Text> values, Context context)
91             throws IOException, InterruptedException {
92       Set<String> logins = new HashSet<String>();
93       String home = null;
94
95       for (Text state: values) {
96         if (state.charAt(0) == '-') {
97           home = state.toString().substring(1);
98         } else {
99           logins.add(state.toString());
100        }
101      }
102
103      logins.remove(home);
104      v.set(logins.toString());
105
106      context.write(key, v);
107    }
108  }
109 }
```

Code Explanation

Line 1 is the package statement. This class is in the `javaone` package.

Lines 3-21 are the imports for this class. You can see that MapReduce jobs need to make

use of many different classes from the Hadoop API.

Line 23 is the class declaration. Notice that the class extends `Configured` and implements `Tool`, which is a common idiom in MapReduce jobs. It helps with parsing Hadoop command-line arguments and gives a common structure to your job.

Line 24 is the text for the regular expression that you used in the Hive exercise when creating the `logins` table. You use it later in the reduce task.

Lines 28-30 are the `main` method. For classes that extend `Configured`, the `main` method usually consists only of a call to `ToolRunner.run()` to execute the job. The `ToolRunner.run()` method will look for leading Hadoop arguments in the `args` array, process them, and remove them before passing the `args` array to the job. The `Configuration` object that is created and passed to the job holds all of the information and the job and how it should be executed.

Lines 32-48 are known as the “driver code”. The driver code sets up the job to be executed.

Line 34 creates a `Job` object, which is a helper class that makes it easier to set the required properties in the `Configuration` object.

Line 36 sets up the classpath that will be used by the workers running the job tasks to include the JAR file that contains that job's classes.

Lines 38-41 set up the input paths. Typically there will be a single call to `FileInputFormat.addInputPath()` and `job.setMapperClass()`, but in this case, there are two separate input sources: the user data and the login data, and both need to be processed differently. Instead of the typical approach, this job calls `MulitpleInputs.addInputPath()` twice, each time passing it the path to an input data set and the name of the class that will process it.

Line 42 sets up the output path. There can only be one output path.

Line 44 names the class that will be used as the reducer task.

Lines 45-46 declare the types of the keys and values produced by **both** the map and reduce tasks. The map and reduce tasks will produce integer keys and string values. In Hadoop, the usual Java types are wrapped in instances of `Writable`. `Writable` is Hadoop's equivalent to Java's `Externalizable`, but it is designed to be used at scale. In the case that the map and reduce tasks do not have the same key and value types, then the types for the reduce tasks are declared as they are in lines 45-46, and the types for the map tasks are declared through calls to `job.setMapOutputKeyClass()` and `job.MapOutputValueClass()`. (The reason the types must be declared at all is that type erasure prevents the types declared by the generics from being available at run time.)

Line 48 executes the job and waits for it to complete. If it completes successfully, it returns a 0. If it fails for any reason, it returns a 1.

Lines 51-65 are the first map task class. This class is built to parse the user data.

Line 51 defines the class as a subclass of `Mapper` and declares the types of the input and intermediate keys and values. The input key type for a map task is usually `LongWritable`, and the input value type is usually `Text`. In most cases, the input value will a full line of the input data, and the input key will be the byte offset of that data into the source file. In most cases the input key is ignored. The output key and value types must match the types declared in the driver code.

Lines 52-53 declare `Writable` objects to be used as the key and value for the intermediate

data. In Hadoop, you have to assume that everything you do will be done at massive scale. On a smaller scale, creating a new `Writable` object for every intermediate data record might not be a bad thing. At Hadoop scales, it can create a significant garbage collection overhead. Because the `Writable` class are designed to be reused, it's common practice to reuse the key and value classes.

Lines 55-64 are the `map()` method, which is where the work of the map task is done.

Line 58 retrieves the text data from the input value object and splits it on commas.

Line 60 sets the intermediate key object to be the first field of the input parsed as an integer.

Line 61 sets the intermediate value object to be the second field of the input with a '`-`' prepended. When there is more than one input data source, it is often useful to flag the data from one or more of the sources so that the reducer can tell them apart.

Line 63 emits the intermediate key and value.

Lines 67-84 are the second map task class. This class is built to parse login data.

Line 67 defines the map class as a subclass of `Mapper` and declares the types for the input and intermediate keys and values, the same as was done in the first map task class.

Line 68 defines regex pattern to use for parsing the login data.

Lines 69-70 define key and value objects to use when emitting intermediate data.

Lines 72-83 are the `map()` method.

Lines 75 and 77 test whether the line of input data matches the regular expression.

If the regular expression matches, then line 78 sets the key to the first matching group parsed as an integer; line 79 sets the value to the second matching group; and line 81 emits the intermediate key and value.

Lines 86-108 are the reduce task class.

Line 86 defines the reduce class as a subclass of `Reducer` and declares the types for the intermediate and output keys and values. The types for the intermediate keys and values must match the types declared by the map task class(es). In this case, because there are two map task classes, both map task classes must have the same types for the intermediate keys and values. In the case of a single job, the output data will be converted to text and written to the output path, so the types of the output keys and values are less important. In the case of “chained” jobs, however, the output from one job will be fed as input to the next job, so the output types may be important.

Line 87 defines an output value object. In this reduce task class the intermediate key output is reused as the output key object.

Lines 89-107 are the `reduce()` method, where the work of the reduce task is done.

Line 92 defines a `Set` in which to keep the login states for the current user.

Line 93 defines a `String` to hold the current user's home state.

Lines 95-101 iterate through the values (states) that are associated with the current key (user ID). There are two important things to know about iterating through the values array in the `reduce()` method. First, you can only iterate through the values once. Any subsequent iterations will see no data. This limitation is why this code keeps a `Set` of states rather than iterating through the values twice. Second, in each iteration through the loop the **same** key

and value object will be reused. The contents of the objects will change with every iteration, but the objects themselves will always be the same objects. This behavior is why the object that stores the home state is a `String` and not a `Text`. Were you to store the `Text` object itself, its contents would change with any subsequent iterations through the values. Both of these sometimes surprising behaviors are done to optimize for scalability.

Line 96 tests for the leading '-' that marks the state as having come from the user data. If the value has a leading '-', then line 97 sets it, minus the leading '-', as the home state. If not, then line 99 adds the value to the set of login states.

Line 103 removes the home state from the login states.

Line 104 sets the output value to list of login states.

Finally, line 106 emits the output key and value, reusing the intermediate key and the output key.

Summary

In this exercise you have walked through a non-trivial Java MapReduce job. You have seen that there are many more moving parts to a Java MapReduce job than with Hadoop streaming or Pig or Hive, but that there is also much power and flexibility. This example only scratched the surface of what's possible from a Java MapReduce job. It did not cover use of the distributed cache, job chaining, secondary sort, custom grouping comparators, custom partitioners, etc.

For a more information about Java MapReduce jobs, see:

- Hadoop: the Definitive Guide
- Udacity's Intro to Hadop and MapReduce: https://www.udacity.com/course/ud617
- Yahoo!'s Hadoop Tutorial: https://developer.yahoo.com/hadoop/tutorial/

Additional Exercises

1. Write a MapReduce job to return the list of unique states where users live.
2. Write a MapReduce job to return the Ids of all users who logged in between 00:00 (midnight) and 01:00.
3. Write a MapReduce job 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 lab, you have seen the core building blocks of Hadoop: HDFS, MapReduce, Hive, Pig, and Impala. Using these tools, you can tackle most big data problems. These tools, however, are only a few of the tools that are available to you as part of the Hadoop ecosystem. What follows is an incomplete list of other tools and technologies you may wish to explore (items with an asterisk are not pre-installed on the Quickstart VM):

- Avro – binary storage format that encapsulates metadata
- Crunch – framework to simplify writing MapReduce code

- DataFu – additional utilities for use with Pig
- Flume – tool for streaming data into HDFS
- HBase – distributed key-value store built on HDFS
- Kafka* – tool for handling streaming events
- Mahout – machine learning library partially built on Hadoop
- MRJob* – Hadoop streaming library for Python
- MRUnit – unit testing for MapReduce jobs
- Oozie – orchestrates the execution of Hadoop workflows
- Oracle NoSQL Database* – distributed key-value store
- Spark – alternative to MapReduce that takes advantage of in-memory caching
    - Spark Streaming – using Spark to processing streaming data
    - MLLib* – machine learning library built on Spark
    - GraphX* – graph processing library built on Spark
- Sqoop – tool to automate the transfer of data between structured data sources (e.g. RDBMS) and HDFS and Hive
- Cloudera Manager – cluster installation and management tool