Working with Hadoop

Introduction to Analytics using Hadoop: Week Two Lecture

Hadoop is a distributed computing architecture that allows us to solve complex problems involving large data sets by breaking the problem into small, functional tasks. These tasks can then be chained together in a series in order to solve the larger problem. To accomplish this, there are two main pieces to the Hadoop picture: data storage and data computation; the former is handled by a distributed file system: HDFS and the latter by a functional application programming interface: MapReduce.

As analysts or developers, our task is particularly to focus on the solution set. Hadoop abstracts low-level tasks like networking and job management in order for us to focus on creating analytical jobs to push to the cluster. However, the cluster itself still requires a user interface, allowing developers to load data and inspect outputs, as well as to start and manage jobs. In principle, this is handled mainly via a command line interface. In addition, there are reporting and status web applications served by the primary daemon processes.

This week, we will learn how to interact with a cluster from the command line in order to develop on our pseudo-distributed "single node clusters". This will allow us to interact with a Hadoop cluster at a high level. At the end of the lecture you should understand how to start and monitor jobs, as well as load and manage data in HDFS. Further, we will introduce Hadoop streaming, and write and execute our first MapReduce jobs.

Table of Contents

Throughout this tutorial there will be several command line interactions demonstrated. I would encourage you to do your best to follow along with the commands while working through the tutorial. If you don't have much experience on the command line, I've provided a brief tutorial about how to work on the command line in the appendix, along with some links to further information. Interactive portions of the tutorial will look as follows:

```
~$ command –flag argument
```

Note: In order to follow along with the commands and details in the following section, please ensure that you have your development virtual environment setup according to the installation instructions. Ensure that all processes in the pseudo-distributed environment are running by typing `jps` on the command line. You should see the 5 daemon processes listed along with "Jps". If you do not see this, use the `/srv/hadoop/bin/start-all.sh` command to start the various daemons in pseudo-distributed mode.

Additionally, the data found in this tutorial can be found either in the Week 1 assignments resources or in the Week 2 assignments resources (unless otherwise specified). If you're unable to locate the data, feel free to use your own, similar data sets, or to request help on the forums.

Working with HDFS

HDFS, which stands for Hadoop Distributed File System is the data keeping and management system implemented in Hadoop. It is comprised principally of three daemon processes: the NameNode, the SecondaryNameNode, and the DataNode. These processes work together in the cluster to abstract the networking details of the cluster and instead provide a hierarchical, file system-like API to the end user.

When working with HDFS, however, keep in mind that the file system is in fact a distributed, remote file system. It is easy to become mislead by the similarity to the POSIX file system, particularly because all requests for file

system lookups are sent to the NameNode who responds very quickly with lookup-type requests. Once you start accessing files, things slow down quickly- and really access should be done through MapReduce jobs. Also keep in mind that because blocks are replicated on HDFS- you'll actually have less disk space available in HDFS than is available from the hardware.

For the most part, interaction with HDFS is performed through a command line interface that will be familiar to those who have used POSIX interfaces on Unix or Linux. Additionally there is an HTTP interface to HDFS, as well as a programmatic interface written in Java, both of which we will discuss later in this section. However since the command line interface is most familiar to developers, this is where we will start.

Basic File System Operations

All of the usual file system operations are available to the user such as creating directories, moving, removing, and copying files, listing directories, and modifying permissions of files on the cluster. To see the available commands in the `fs shell`, type:

```
~$ hadoop fs –help
```

As you can see, many of the familiar commands for interacting with the file system are there, specified as arguments to the `hadoop fs` command as flag arguments in the Java style, e.g. a single – supplied to the command. Secondary flags or options to the command are specified with additional Java style arguments delimited by spaces following the initial command. Be aware that order can matter when specifying such options.

To get started, let's copy some data from the local file system to the remote or distributed file system. To do this, use either the `put` or `copyFromLocal` commands. These commands are identical and write files to the distributed file system without removing the local copy. The `moveFromLocal` command is

similar, but the local copy is deleted after a successful transfer to the distributed file system.

If you have downloaded the shakespeare.txt file from the resources section on the learning management system to your home directory in your virtual environment, you can copy it to HDFS using the following command:

```
 ~$ hadoop fs –copyFromLocal shakespeare.txt
shakespeare.txt
```

Having invoked the Hadoop shell command with the two arguments `<src>` and `<dst>` specified as shakespeare.txt and shakespeare respectively, the command searches your current working directory for the shakespeare.txt file and copies it to the `/user/analyst/shakespeare` file on the local HDFS instance running on the localhost. Relative and absolute paths must be taken into account when specifying paths on both the local file system and the remote file system. The above command is shorthand for:

```
~$ hadoop fs –put /home/analyst/shakespeare.txt \
             hdfs://localhost/user/analyst/shakespeare.txt
```

You'll note that there exists a home directory on HDFS that is similar to the home directory on POSIX systems. Relative paths in reference to the remote file system treat the user's HDFS home directory as the current working directory. In fact, HDFS has a permissions model for files and directories that are very similar to POSIX, but before we get into the specifics of permissions, let's take a look at the file in our home directory using the listing command as follows:

```
~$ hadoop fs –ls .
-rw-r--r-- 1 analyst supergroup 8877968 2013-11-04 17:52
/user/analyst/shakespeare.txt
```

The HDFS file listing command is similar to the Unix `ls –l` command with some HDFS specific features. Specified without any arguments this command

provides a listing of our HDFS home directory. The first column shows the permissions mode of the file. The second column is the replication of the file. Since we're in pseudo-distributed mode, the replication is 1. By default the replication factor is 3 on HDFS. Note that directories are not replicated, so this column is a – in that case. The user and group follow, then the size of the file in bytes (zero for directories). The last modified date and time is up next, with the absolute path of the file appearing last.

Organization is not limited to files simply, but directories can also be created:

```
~$ hadoop fs –mkdir corpora
~$ hadoop fs –ls .
drwxr-xr-x   - analyst supergroup 0 2013-10-23 11:31
/user/analyst/corpora
```

Other basic file operations like `mv`, `cp`, and `rm` will all work as expected on the remote file system. There is, however, no `rmdir` command, instead use `rm –R` to recursively remove a directory with all files in it.

Reading and moving files from the distributed file system to the local file system should be attempted with care, since the distributed file system is maintaining files that are extremely large. However, there are cases when files need to be inspected, particularly output files that are produced as the result of MapReduce jobs. Typically these are not read to the standard output stream but are piped to other programs like `less` or `more`.

To read the contents of a file, you can use the `cat` command; pipe the output of this to `less` in order inspect the contents of the file. A note on `less`: use the arrow keys to navigate the file and type `q` in order to quit.

```
~$ hadoop fs –cat shakespeare.txt | less
```

Alternatively you can use the `tail` command to inspect only the last kilobyte of the file.

```
~$ hadoop fs –tail shakespeare.txt | less
```

There is no similar `head` command to inspect the first kilobyte of the file. It is efficient to `cat` the file and pipe it to `head`, as the `head` command will terminate the stream before the entire file is read. However, using `tail` in this manner would be less efficient, and the `hadoop fs -tail` command instead seeks to the correct position in the file and returns that over the stream[1].

In order to transfer files from the distributed file system to the local file system, use `get` or `copyToLocal`, which are identical commands. Similarly, use the `moveToLocal` command, which also removes the file from the remote file system. Finally, the `getmerge` command merges all files that match a given pattern or directory are copied and merged into a single file on the local host. If these files are large, you may want to pipe them to a compression utility.

```
~$ hadoop fs -get shakespeare.txt ./shakespeare.copied.txt
```

Comparing the original shakespeare.txt file should prove that it is identical to the shakespeare.copied.txt file. It is up to the reader to explore the last commands in the file system shell utility, particularly `test`, `count`, `du`, `stat`, and `text` commands.

File Permissions in HDFS

As mentioned earlier, HDFS has POSIX-like file permissions. There are three types of permissions, read (`r`), write (`w`), and execute (`x`). These permissions define the access levels for the owner, the group, and any other system users. For directories, the execute permission allows access to the contents of the directory, however execute permissions are ignored on HDFS for files. Read and write permissions are self-explanatory.

These permissions are expressed during the directory listing command `ls` with hdfs. Each mode has 10 slots. The first slot is a `d` for directories, otherwise a `-` for files. Each of the following groups of three indicates the `rwx`

[1] Thanks to Chris White for his answer: http://stackoverflow.com/questions/19778137/why-is-there-no-hadoop-fs-head-fs-shell-command

permissions for the owner, group, and other users respectively. There are several hadoop file system shell commands that will allow you to manage the permissions of files and directories, namely the familiar `chmod`, `chgrp`, and `chown` commands.

```
~$ hadoop fs –chmod 664 shakespeare.txt
```

This command will change the permissions of shakespeare.txt to `-rw-rw-r--`. The 664 is an octal representation of the flags to set for the permission triple. Consider 6 in binary, 110 – this means set the read and write flags but not the execute flag. Completely permissible is 7, 111 in binary and read-only is 4, 100 in binary[2]. The `chgrp` and `chown` commands will change the group and owner of the files on the distributed file system.

A caveat with file permissions on HDFS: the identity of the client is determined by the username and groups of the process operating across HDFS, which means that remote clients can create arbitrary users on the system. These permissions, therefore, should only be used to prevent accidental data loss and to share file system resources between known users, not as a security mechanism. Hadoop does support Kerberos authentication, but that is beyond the scope of this course.

Other HDFS Interfaces

Hadoop is written in Java, and therefore all interactions with HDFS happen through the Java API. Even the command line utility, the file system shell (`fs`) is a Java application that uses the Java API to interact with HDFS. An intermediate or advanced discussion of Hadoop will then naturally focus on using that API and discuss the various methods for querying and interacting with the file system, however that is beyond the scope of this course.

This also means that there are other existing tools for integrating the underlying file system with Hadoop, e.g. FTP or S3; but most important to this

[2] For more on Unix file permissions see http://mason.gmu.edu/~montecin/UNIXpermiss.htm

course is probably the HTTP interface to HDFS. There are two HTTP interfaces-direct HDFS access through the HDFS daemons that serve HTTP requests, or via proxies, which accesses HDFS on the client's behalf using the API.

The direct HTTP access method is served by the NameNode, which runs on port 50070 by default. With your pseudo-distributed virtual node running, open a browser and type in http://ubuntu.local:50070 to access the NameNode's meta data server. (If you have trouble, it could be the networking settings on your machine, your classmates and I will be happy to help).

You can browse the HDFS file system through the link in the top section; however you may have to edit the URL to once again point to ubuntu.local as opposed to the localhost in order to view the directory. While browsing, the DataNodes themselves stream the file data, by default on port 50075.

Also by default, the direct HTTP interface is read-only. However, new versions of Hadoop support a WebHDFS application that allows all file system operations with authentication via Kerberos. This must be enabled by adding the following property to the `/srv/hadoop/conf/hdfs-site.xml` file:

```
<property>
   <name>dfs.webhdfs.enabled</name>
   <value>true</value>
</property>
```

Keep in mind however that security is important in this mode and it's best to allow systems administrators to enable Kerberos authentication before enabling the WebHDFS system.

MapReduce on the Cluster

Functional programming is a style of computer program architecture that ensures that unit computations are the evaluation of mathematical functions that avoid state. In particular, this means that functions in these types of programs depend only on their inputs and functions are closed together,

sending the output of one function as the input to another, wholly independent function. This style of programming seems to have excellent application to distributed big data computational systems where dependence on the state of the computation would require network traffic or data too large to fit into memory.

MapReduce is a functional pattern and programming model that is inspired by functional programming, particularly `map` and `reduce` canonical functions. It is intended for dealing with large data sets in a parallel, distributed manner, and is implemented in many such systems besides Hadoop like MPI (message passing interface). Hadoop's MapReduce API allows developers to decompose large tasks into smaller analytical pieces and reach final computations by chaining MapReduce jobs together.

1. Map functions take as input a series of key/value pairs and after analysis is expected to output zero or more output key/value pairs. Typically the types of the input keys and values are modified during the course with a different output type.
2. Reduce functions take as input a key and an iterator of values associated with that key and after analysis are expected to output zero or more key value pairs.

In a distributed application like Hadoop, `Mappers` handle large volumes of data in discrete, digestible chunks, then output their analysis to `Reducers` which reduces the variable outputs of many `Mappers` into a final, per-key output, the outputs of which can then be chained to another set of `Mappers` and `Reducers` and so on until a final computation has been reached.

In order to accomplish this with HDFS and MapReduce on Hadoop, the following data flow is defined:

1. The data is sent to an `InputFormat` to be split into manageable blocks that are read in a streaming fashion by a `RecordReader`. The

`RecordReader` then manages the stream into the key/value pairs that are sent to `Mappers`, one for each `RecordReader`.

2. `Mappers` analyze the input key/value pairs one pair at a time and output zero or more key/value pairs as a result of their analysis.
3. Following the map phase, a Partitioning function determines what `Reducer` to send the key/value pairs to. Every `Reducer` is guaranteed to receive all values for the same key. Since some shuffling might be required, this is the phase that will require network traffic.
4. The `Partitioner` then sorts the key/value pairs into the stream so that each `Reducer` operates on a continuous chunk of values for each key. The sorted key/value pairs are passed to the `Reducer`, which then outputs the final key/value pairs.
5. The final key/value pairs are sent to an `OutputFormat` that collects the data and writes it back to the disk.

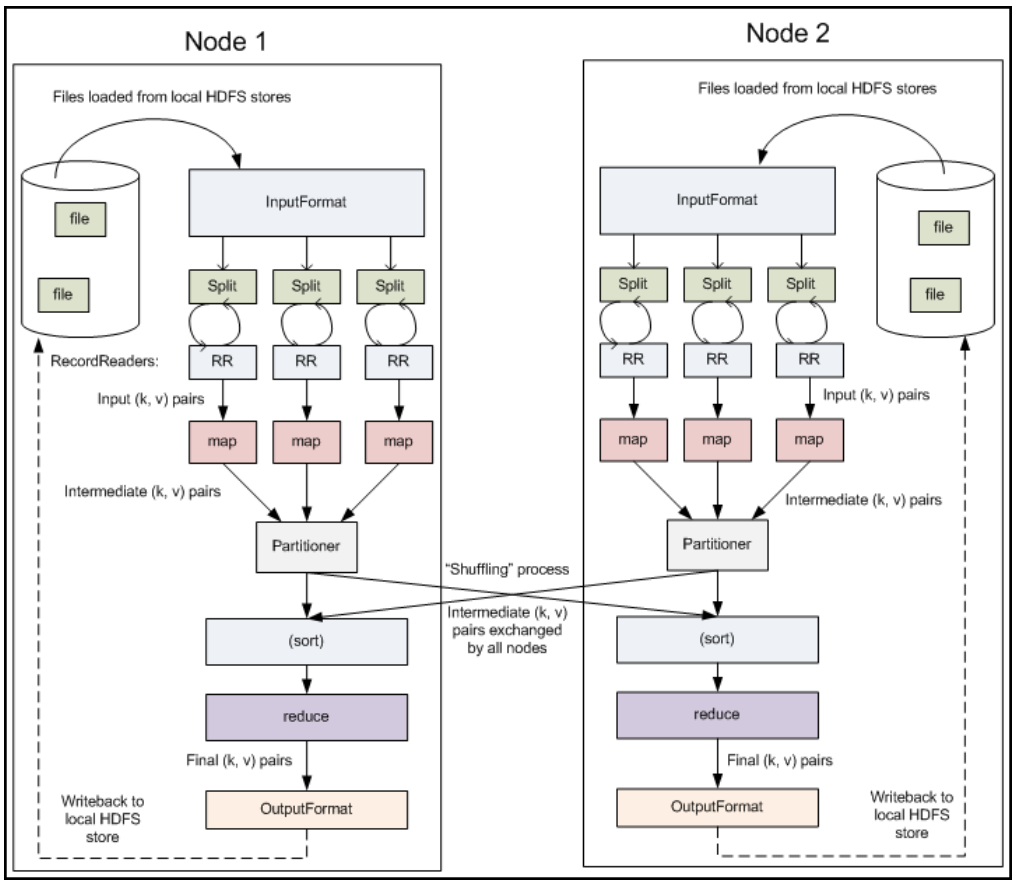

Figure 1: Data flow in MapReduce[3]

The MapReduce programming paradigm allows developers and analysts to focus on writing Mappers and Reducers or chains of them to compute across large data sets without worrying about the network or underlying processing details. They can optimize their jobs by also implementing smarter input and output formats as well as partitioners and record readers, MapReduce is a powerful computational tool.

Running and Managing Jobs

The `hadoop` command line utility does not only allow us to access HDFS, it is also used to run and manage MapReduce jobs. The MapReduce API is written in Java, therefore most MapReduce jobs are compiled JAR files. Hadoop will

[3] Image from http://developer.yahoo.com/hadoop/tutorial/module4.html used under a Creative Commons Unported License.

transmit the JAR files across the network to each node that will run a task (either a Mapper or Reducer) and the individual tasks of the MapReduce job are executed.

Let's revisit the Word Count example- the "Hello World" of MapReduce. Download the WordCount.zip from the Resources section of the class. This folder contains the following Java files:

- `WordCount.java`: a MapReduce driver class
- `WordMapper.java`: a Mapper class for the job
- `SumReducer.java`: A reducer class for the job

Without leaving the directory execute the following commands:

```
$ javac –classpath /srv/hadoop/hadoop-core-1.2.1.jar *.java
```

This command compiles the java files in the directory. You must include the classpath for the Hadoop Java API, note that `/srv/hadoop` is the location of the hadoop installation if you were following the installation instructions. This will be different on CDH4 or if you installed it differently. It may be worth checking if there is a `$HADOOP_HOME` or `$HADOOP_PREFIX` environment variable that specifies this location. Also note that this core api jar also includes the hadoop version.

```
$ jar cvf wc.jar *.class
```

The compiling process has created a set of `.class` files in the local directory. Archive these together into a JAR using the `jar` command. Hadoop requires jobs submitted in JAR files. You can then submit the job to the cluster:

```
$ hadoop jar wc.jar WordCount shakespeare.txt wordcounts
```

The `hadoop jar` command submits the wc.jar file to the cluster. You have to specify the class whose `main` method should be invoked, in this case `WordCount` as well as the input and output HDFS paths,

`/user/analyst/shakespeare.txt` and `/user/analysts/wordcounts` respectively.

Watch as the job executes, specifying the status of the mappers and the reducers, and whose final output gives you statistics for the completion of the job!

Notes: If you are receiving compilation errors, this may be because the version of Hadoop you are using has the old-style API code. If this is the case, please download the java files that implement the new-style API from the resource section of the class. If you are receiving job errors- insure that you have uploaded the shakespeare.txt file to HDFS and that there is no directory called wordcounts already in your home directory. Hadoop will not override existing files on the file system and will fail if the output directory already exists.

You can view the contents of the job output:

```
$ hadoop fs –ls wordcounts
```

You should see output files named similar to `part-00000`, there will be one for each reducer, but since we're only using one reducer, there should only be one part. You'll also see a `_SUCCESS` file and a `_logs` directory. In order to read the contents of the output do the following:

```
$ hadoop fs –cat wordcounts/part-00000 | less
```

You can now page through the various word counts from the Shakespeare corpus!

If something goes wrong in your MapReduce job, you'll need to be able to stop it (consider if you've accidentally added an infinite loop or some memory intensive process to your Mapper or Reducer!). However, typing `Ctrl+C` (issuing a keyboard interrupt on Unix) will only kill the process displaying the progress, it won't actually stop the job!

In order to see the running jobs enter the following on the terminal (you may want to start another WordCount job and then open another terminal window to issue the command):

```
$ hadoop job -list
```

You can then kill a job by copying the job id and running the `-kill` command.

```
$ hadoop job -kill jobid
```

You can also view the status of jobs and interact with their log files and report from the web interface of the JobTracker daemon, which by default is served on port 50030. To view the JobTracker daemon open a browser and navigate to http://ubuntu.local:50030/. You can also track individual tasks using port 50060 along with the UI served by the TaskTracker daemons. Navigate to http://ubuntu.local:50060 to see running, successful, and failed jobs.

If you're having problems with the mdns (ubuntu.local) you may also navigate using the IP address of the virtual environment. Use the `ifconfig` command in the virtual environment and replace ubuntu.local with the IP address.

Hadoop Streaming

In the former example, we issued a job to the cluster using a jar file. The Hadoop MapReduce API is written in Java, and therefore is the natural choice for writing expressive jobs. However, Hadoop also provides an API that allows you to write MapReduce jobs in languages other than Java called Hadoop Streaming. As you'll see in the next section, MapReduce can be emulated using Unix pipes. Similarly, Hadoop Streaming utilizes the Unix standard streams in order to interface with non-Java programs. Any programming language that can read from `stdin` and write to `stdout` can leverage Hadoop Streaming.

In particular, we will leverage Hadoop Streaming to write MapReduce jobs in Python, Ruby, and R- and there are several projects dedicated to supporting writing MapReduce jobs in these languages, notably Dumbo for Python, Mandy for Ruby, and RHadoop for R. In the supplementary reading, you will find discussions on how to implement MapReduce jobs for the various languages you are working with.

Hadoop Streaming sends a stream of text lines to the mapper function via standard input, making Streaming well suited for text based processing where rows are separated by lines. Mappers are expected to write key/value pairs as tab delimited lines sent to standard output. Reducers receive the tab-delimited lines on standard input and also write their tab-delimited results to standard output.

To execute a Hadoop Streaming job issue the following command:

```
$ hadoop jar \
   /srv/hadoop/contrib/streaming/hadoop-*streaming*.jar \
   -file /path/to/mapper -file /path/to/reducer \
   -mapper /path/to/mapper -reducer /path/to/reducer \
   -input /hdfs/input -output /hdfs/output
```

The JAR file will be the hadoop-streaming.jar, which is found in the contrib directory of the Hadoop installation directory. You must specify all the files that need to be sent to every node using the `-file` flag, including the mapper and reducer file. The `-mapper` and `-reducer` flags indicate which files do what, and finally you specify the `-input` and `-output` as flags rather than arguments.

The WordCount example has also been written in Python. Download StreamingWordCount.zip from the resources section and unzip it in the home directory of your virtual environment and `cd` into the `StreamingWordCount` directory.. Type the following commands to execute the same job with Python:

```
$ hadoop jar \
```

```
/srv/hadoop/contrib/streaming/hadoop-streaming-1.2.1.jar
-file mapper.py -file reducer.py \
-mapper mapper.py -reducer reducer.py \
-input shakespeare.txt -output wordcounts
```

Note that if the wordcounts directory already exists in HDFS from the previous Java example, the job will fail. If you're having trouble finding the files specified, you may need to provide absolute paths to the files.

Once again, you can inspect the results of this job by issuing the following command:

```
$ hadoop fs –cat wordcounts/part-00000 | less
```

You should see similar results to those found with the WordCount job, although the results might vary due to different techniques in splitting the word boundary in Java and Python.

Writing a MapReduce Application

Now that we've learned how to execute MapReduce jobs and work with HDFS, it's time to put them together to do some simple analysis on a toy data set- the on time performance of U.S. airlines. In the resources section, download the data set called `ontime_flights.zip` and unzip this file both in your local directory and copy it to HDFS.

The On Time Performance data set is one month of U.S. airport flight data from January 2013 collected by the Research and Innovative Technology Administration (RITA) Bureau of Transportation Studies[4]. These data sets can also be downloaded directly from the RITA website, but the data in the resources section has been cleaned up a bit. For a single month, you'll find 509,519 flights with data that comprises 62 MB on disk. Clearly to put together an entire decade's worth of data (approximately 7.5 GB), you'll need some larger computation mechanism like Hadoop!

[4] http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time

Each row in the data is a tab-delimited series of values, for example:

```
1/17/13      9E     Endeavor Air Inc.   DFW    JFK    1038   -7     1451   -14
1/18/13      9E     Endeavor Air Inc.   DFW    JFK    1037   -8     1459   -6
1/19/13      9E     Endeavor Air Inc.   DFW    JFK    1035   -5     1515   17
1/20/13      9E     Endeavor Air Inc.   DFW    JFK    1037   -8     1455   -10
```

Containing the date of the flight, the airline identifier and carrier, the origination and destination airport codes as well as the departure time, the departure delay, the arrival time, and the arrival delay. Negative delays mean early arrivals or departures. There is actually considerably more data in the data set, but the README.md included with the data discusses the column specifics for each data type.

Our first task will be to compute the arrival delays by airport. Our simple MapReduce algorithm will be as follows:

```
def mapper(key, value) {
   value = value.split('\t')
   airport = value[6]
   delay = value[15]
   emit(airport, delay)
}

def reducer(airport, delays) {
   mean = sum(delays) / len(delays)
   emit(airport, mean)
}
```

The mapper simply takes the key/value pair (ignores the key, which is just the line number) and splits the value on tab. It then seeks to the airport column and the delay column and yields the airport/delay key value pair. The reducer function computes the mean delay and emits the airport along with the mean.

We will implement this using Hadoop Streaming with Python for our first iteration as well as Java. If you know Ruby or another scripting language, please contribute examples to the forum- they may be included in the text in the future!

Implementing our MapReduce Job

We will start by implementing our MapReduce job using Hadoop Streaming with Python. You can find an update for this Job written in Java on the course pages. Please contribute your own examples in Ruby or other languages!

mapper.py:

```
#!/usr/bin/env python

import sys

SEP='\t'

def mapper():
    for line in sys.stdin:
        value = line.split('\t')
        airport = value[6]
        delay = value[15]
        print airport + SEP + delay

if __name__ == '__main__':
    mapper()
```

The mapper is fairly straight forward- it reads each line from standard input, and then splits on tab- it fetches the airport and delay and prints them separated by the tab character.

The reducer is slightly more complex and requires a bit more explanation. Keys are streamed to the reducer, but the values are not grouped together in a compact iterable by default as in the Java API, though there are various libraries that will accomplish this for your specific programming language.

reducer.py:

```
#!/usr/bin/env python

import sys

SEP='\t'

def reducer():

    current_airport = None
    current_total = 0
    current_count = 0
```

```
    airport = None

    for line in sys.stdin:
        line = line.strip()

        airport, delay = line.split(SEP, 1)

        delay = float(delay)

        if current_airport == airport:
            current_total += delay
            current_count += 1
        else:
            if current_airport:
                mean = current_total / current_count
                print current_airport + SEP + str(mean)
            current_total = delay
            current_count = 1
            current_airport = airport

    if current_airport == airport:
        mean = current_total / current_count
        print current_airport + SEP + str(mean)


if __name__ == '__main__':
    reducer()
```

The reducer is slightly more complex. The sort operation has taken care of ordering our key so that each line that comes into the reducer will be ordered as a chunk of keys. To get all the values we have to keep track of the key, and if we have the same key, continue the summation and counting so that we can compute the mean, and if the key is new- that is when we calculate the mean and then output the result, resetting our values for the next key chunk.

With Ruby and Python, there are many libraries to assist with this!

Emulating MapReduce with Unix Pipes

Because Hadoop Streaming operates on `stdin` and `stdout` it is possible to emulate the behavior of Hadoop with Unix pipes and the `head`, `cat`, and `sort` utilities. Let's see how this works with Python:

```
$ head ontime_flights.tsv | mapper.py
```

```
DFW     -14
DFW     -6
DFW     17
DFW     -10
DFW     -19
DFW     -3
DFW     -1
DFW     15
DFW     20
DFW     11
$ head ontime_flights.tsv | mapper.py | sort | reducer.py
DFW     1.0
```

Using this methodology we can quickly test the behavior of our mappers and reducers without computing the entire input or sending the job to HDFS, and it is a great debugging tool! Note: ensure that your mapper.py and reducer.py are executable (`chmod +x`) and the first line is `#!/usr/bin/env python`.

Conclusion

In this lecture you have learned how to interact with HDFS and Hadoop on the command line using the `hadoop` shell command. This command line interface is an extremely useful tool for developers to interact with the cluster both to execute basic file operations and to execute jobs across the cluster. Hadoop also comes with several web-reporting interfaces that are accessible by browsing to the IP address of the daemon node that serves them and using the correct port.

In addition to exploring how to manage data and start and stop Hadoop jobs, we have encountered Hadoop Streaming- a powerful interface to allow programmers to implement MapReduce jobs in languages other than Java, and learned how to execute these types of jobs. We followed this discussion with a lesson writing our first MapReduce job that was sent to the cluster!

In the following week we will continue to explore MapReduce jobs and common MapReduce patterns. From here, please take a look at the discussion questions and dive into the assignments- you'll be writing your first MapReduce jobs!

Assignments

Please post your answers to the following discussion questions. The exercises for the second week are due on Monday.

Supplementary Reading:

Tutorials are a good start for new developers, and in order to maximize programming language coverage, here are a few suggested tutorials on the web for various programming languages.

- Python: http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/
- Ruby: http://oobaloo.co.uk/articles/2010/1/12/mapreduce-with-hadoop-and-ruby.html
- Java: http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html#Example%3A+WordCount+v1.0
- R: https://github.com/RevolutionAnalytics/rmr2/blob/master/docs/tutorial.md

Optional Reading

1. Chapters 2 and 3 from *Hadoop, The Definitive Guide*

Discussion

1. What are some potential pitfalls from performing analysis with the MapReduce functional paradigm? How could poor data flow affect the performance at each stage of the MapReduce implementation? What about other computational resource constraints?

Questions for Thought

1. Why is the output from Hadoop jobs stored in directories with multiple parts? What are the best ways to read or view the output using HDFS commands?

2. POSIX-like permissions exist on HDFS and can be managed with Unix-like commands, what are some of the benefits and potential issues associated with a permissioned distributed file system?
3. Why must keys from the mappers be partitioned to the same reducer?
4. Give an example of a chained MapReduce job for a more extensive computation.

Exercises

There will be more details on the assignments, along with point breakdowns on the assignments page in the course materials.

1. Implement your Shakespeare and Apache Log algorithms for MapReduce
2. Perform some simple analytical computations with MapReduce on the On Time Performance of Flights data set.

Appendix: Working on the Command Line

We humans love metaphor, and what is the Desktop on your computer if not a series of symbolism that allows us to view our machines as extensions of our work lives? How about the Browser? It is a window that let's us explore or browse documents on the Internet. Even the term Windows is used metaphorically in computing environments. There is, however, a lower level than the iconographic metaphors that we're used to. A lower, much more powerful level that uses textual commands to precisely control and modify your computers behavior. We generally refer to this as the command line interface.

With great power comes great responsibility, however. Keep in mind that when using the Command Line, whatever you ask your computer to do, the computer will promptly and without question (most of the time) perform that task, even if that task is binary seppuku.

Before we start, you'll want to open a Terminal. If you're on a Mac you can simply open up the Terminal application in your Applications directory. Similarly a Terminal can be found the same way on Linux. If you're on Windows, I recommend using SSH to access the command line on the virtual environment.

Command Syntax

When you come to the terminal, you'll be presented with something like:

```
localhost:~ username$
```

This is called the prompt, and while the details will vary across systems and Terminals, it is communicating some key elements to you, and also prompting you to type in a command. For example, in this case, "`localhost`" tells you the name of the computer that you are controlling (it is possible to command remote computers through the command line), the "`~`" indicates the directory

you're in, in this case `~` is short for "home directory", and the `username` is the name of the logged in user.

Anatomy of a Command

Here is an example command:

```
$ ls -la ~/Desktop
```

This command has several parts:

`$` is the prompt, an indication that you're in the correct place. When we discuss commands I will include the prompt, but you don't have to type it.

`ls` is the command, also called a Utility. These are programs that are stored on your computer that when executed will perform various tasks based on the input (flags and arguments) and output the results in text to you.

`-la` are flags, two flags in particular, the `-l` flag and the `-a` flag. Flags are options or preferences to the command. Flags come in two flavors: single letter flags as in `-l` or whole word flags such as `--version`. In the single letter version you can chain lots of flags together after the dash, however in the whole word flag, you have to space separate the flags. Note that `-la` is the same as `-l -a` generally speaking.

`~/Desktop` is an argument to the Utility. Arguments are used to specify ordered information that the command needs to function. Arguments are usually placed at the end of the command, and there can be any number of space-separated arguments from none to many.

In this particular case, the `ls` utility lists the contents of a directory and the `-l` flag asks for more information, while the `-a` asks to see all files, even hidden ones. `~/Desktop` specifies what directory to show the listing of.

Some Commands to Memorize

Here is a list of the basic utilities on Linux/Unix Terminals that you will use on a regular basis and they are worth committing to memory. Anything with all capital letters that starts with a dollar sign, like $ARG is an argument to give to the command. Replace $ARG with the value you'd like to pass it.

```
man $COMMAND
```

Manual. Get information on how to use the command. Press up and down to scroll through the manual, press Q to quit and go back to the prompt.

```
ls $DIR
```

List the contents of the directory $DIR. If no argument is specified will list the contents of the current working directory.

```
touch $FILE
```

Creates a file at the specified $FILE path or modifies an existing file's meta data concerning it's modified and accessed time.

```
mkdir $DIR
```

Creates a directory in the current working directory.

```
rmdir $DIR
```

Removes an empty directory.

```
cd $DIR
```

Change directory. Changes the current working directory to the $DIR directory argument, allowing you to navigate the file system.

```
pwd
```

Print Working Directory. Lets you know where you are on the computer.

```
less $FILE
```

Displays the contents of a file. Use the up and down arrows to scroll. Press Q to go back to the prompt.

```
cp $FILE $DESTINATION
```

Copy. Copies the file to the destination.

```
mv $FILE $DESTINATION
```

Move. Moves the file to the destination.

```
rm $FILE
```

Remove. Deletes a file permanently - there is NO way to get it back. Be careful when using this!

```
sudo $CMD
```

Super User Do. Append sudo before a command and it will prompt you for your password. When entered, this tells the computer that it should execute the command as though it were the super administrator, and will allow you to do anything. Be careful when using this!

Computer Organization

Computers are organized according to a hierarchical directory file structure. Every computer has a root directory that contains every other folder and directory. In Linux this is simply called /, whereas in Windows it's usually called C:₩. Because of this structure, every file and directory has an associated absolute path, the path from the root directory.

Additionally there is the notion of the relative path - that is, the path to a file or directory from where you currently are in the file system. On the command line you're always in a particular directory and you can go up to its parent folder or down into child folders. To begin with, you start in your home directory.

absolute paths always start with / relative paths never start with /

While navigating the file system, there are two special symbols: `.` and `..` which represent your current working directory and your parent directory respectively. These are often used in relative paths to more clearly specify direction. As you navigate up (towards root) and down (away from root) the file system with the `cd` command, keep in mind your current working directory and use `ls` and `pwd` liberally to keep you on track.

Pro Tips

Here are some shortcuts to make your life on the command line a bit easier (less typing!)

Tab Autocomplete: When typing out a path or a command, you can quickly finish the rest by pressing the tab key after you type two or three letters. If more than one possibility exists, nothing will happen, but if you hit tab twice, it will display all the possibilities. This will make you much, much faster!

Previous Commands: Instead of retyping an entire command that you made a type on, hit the up and down arrows to go through your command history!