CS61C Summer 2015 Lab 11: MapReduce and Spark

Goals

- Get hands-on experience running MapReduce and gain a deeper understanding of the MapReduce paradigm.
- Become more familiar with Apache Spark and get hands on experience with running Spark on a local installation.
- Learn how to apply the MapReduce paradigm to Spark by implementing certain problems/algorithms in Spark.

Setup

Copy the contents of `~cs61c/labs/11` to a suitable location in your home directory.

It will be helpful if inexperienced Java programmers partner with experienced Java programmers for this lab.

You will also be working with Spark (in Python!), so you may need to brush up a bit on your Python!

Background Information

In lecture we've exposed you to cluster computing (in particular, the MapReduce framework), how it is set up and executed, but now it's time to get some hands-on experience running programs with a cluster computing framework!

In this lab, we will be introuducing you to two different cluster computing frameworks:

- The first one is Hadoop -- an open source platform which implements the MapReduce framework. For this lab, you will be running the Map and Reduce routines locally (within one process).
- The second framework is Spark! In this lab, we will be converting some of the code write for Hadoop into Python to run in Spark to give us some practice in writing Map and Reduce routines in Spark as well.

Both of these frameworks have their own websites (Hadoop and Spark), so you are free to try to install either onto your local machines, although it may be easier to ssh into the lab computers to complete this lab. Be sure to understand the Spark framework well, as we will be using Spark in the upcoming project!

Avoid Global Variables

When using both Hadoop and Spark, avoid using global variables! This defeats the purpose of having multiple tasks running in parallel and creates a bottleneck when multiple tasks try to access the same global variable. As a result, most algorithms will be implemented without the use of global variables. If necessary, in Hadoop, you can make use of the configuration variables across machines and in Spark, you can make use of broadcast variables, however, we will not need either of these for the lab.

How to run Spark via the command line

For this lab and the project we will be providing you all with a Makefile that will help you run your Spark files, but should you create your own new files (or use Spark outside of this class, which you should!), you will need to know how to run Spark via the command line. For our version of Spark (which is 1.1.0), in order to run your Spark file xxx.py (similar to how you run your Python files with python xxx.py), you just run the following command:

```
$ spark-submit xxx.py # Runs the Spark file xxx.py
```

If your Spark file takes in arguments (much like the Spark files we have provided), the command will be similar, but you will instead add however any arguments that you need, like so:

```
$ spark-submit xxx.py arg1 arg2 # Runs the Spark file xxx.py and passes in arg1 and arg2 to xxx.py
```

Spark also includes this neat interpreter that runs with Python 2.7.3 and will let you test out any of your Spark commands right in the interpreter! The interpreter also takes in files (pass in the file with --py-files flag) and will load your file in the same directory as the executable. If you are looking to just run the interpreter, the command is as follows:

```
$ pyspark # Runs the Spark interpreter. Feel free to test stuff out here!
```

If you want to preload some files (say a.py, b.py, c.py), you can run the following command:

```
$ pyspark --py-files a.py, b.py, c.py # Runs the Spark interpreter and you can now import stuff from a, b, and c!
```

Spark Debugging Quick-tips

If you ever find yourself wondering why your output is strange or something breaks when you run your Spark files, remember these few quick-tips!

- Make use of the take function! The take function can be run on any RDD object (so any object you are trying to parallelize or have run any transformation / action functions on, you will read about this later). This function takes in one argument `num`, which is an integer and it will return back to you the first `num` elements inside of your RDD object. For more information about this, check out the documentation on take.
- You can also test out your functions (map, reduce, etc) inside of the Spark interpreter (pyspark, mentioned above). Simply import the function you want to test out in pyspark (explained above) and you will be able to run this function and check if the output is what you expected! Here is a short example from wordcount.py:

```
$ pyspark --py-files wordcount.py # Run the pyspark interpreter with the wordcount.py file in the executable's directory
>>> from wordcount import flat_map # Import the function you want to test out, in this case, flat_map
>>> file = sc.sequenceFile("/home/ff/cs61c/data/billOfRights.txt.seq") # Load up the sequence file billOfRights.txt.seq
>>> file.take(5) # Returns back to you the first 5 elements in billOfRights.txt.seq
[(<doc_name_1>, <text 1>), (<doc_name_2>, <text 2>), ..., (<doc_name_5>, <text 5>)]
>>> flat_map_output = file.flatMap(flat_map) # Run the imported function flat_map on the file
>>> flat_map_output.take(5) # Return back the first 5 words in your document.
[u'Amendment', u'I', u'Congress', u'shall', u'make']
```

Documentation, and Additional Resources

- The Java API documentation is on the web here. The classes java.util.HashMap, java.util.HashSet and java.util.ArrayList are particularly likely to be useful to you.
- The Hadoop Javadoc is also available here. You mostly shouldn't need this, but it may be handy for `org.apache.hadoop.io.Text`.
- A quickstart programming guide for Spark (click the Python tab to see the Python code) is avaiable here!
- The version of Spark we will be using will be 1.1.0 and the link to the API documentation is available here. For this lab, you should not need to reference too much, but you may have to look at this for the project.

Exercises

Note: Different exercises may be solvable or needed to be solved by reconsidering how map(), flat_map() and reduce() are implemented and called and in which order, so keep this in mind when calling whichever you must use

The following exercises use three different sample input files, two of which are provided by the staff and can be found in `~cs61c/data`:

1. `billOfRights.txt.seq` -- the 10 Amendments split into separate documents (a very small input)
2. `complete-works-mark-twain.txt.seq` -- The Complete Works of Mark Twain (a medium-sized input)

Notice the `.seq` extension, which signifies a Hadoop sequence file. These are NOT human-readable. To get a sense of the texts you'll be using, simply drop the `.seq` portion to view the text file (i.e. `~cs61c/data/billOfRights.txt`).

Although an exercise may not explicitly ask you to use it, we recommend testing your code on the billOfRights data set first in order to verify correct behavior and help you debug.

We recommend deleting output directories when you have completed the lab, so you don't run out of your 500MB of disk quota. You can do this by running:

```
$ make destroy-all
```

Please be careful with this command as it will delete all outputs generated in this lab.

Exercise 0: Generating an Input File for Hadoop

For this exercise you will need the Makefile and Importer.java. In this lab, we'll be working heavily with textual data. We have some pre-generated datasets as indicated above, but it's always more fun to use a dataset that you find interesting. This section of the lab will walk you through generating your own dataset using works from Project Gutenberg (a database of public-domain literary works).

Step 1: Head over to Project Gutenberg, pick a work of your choosing, and download the "Plain Text UTF-8" version into your lab directory.

Step 2: Open up the file you downloaded in your favorite text editor and insert "---END.OF.DOCUMENT---" (without the quotes) by itself on a new line wherever you want Hadoop to split the input file into separate (key, value) pairs. The importer we're using will assign an arbitrary key (like "doc_xyz") and the value will be the contents of our input file between two "---END.OF.DOCUMENT---" markers. You'll want to break the work into reasonably-sized chunks, but don't spend too much time on this part (chapters/sections within a single work or individual works in a body of works are good splitting points).

Step 3: Now, we're going to run our Importer to generate a .seq file that we can pass into the Hadoop programs we'll write. The importer is actually a MapReduce program! You can take a look at Importer.java if you want, but the implementation details aren't important for this part of the lab. You can generate your input file like so:

```
$ make generate-input myinput=YOUR_FILE_FROM_STEP_2.txt
```

Your generated .seq file can now be found in the convertedOut directory in your lab11 directory. Throughout the rest of this lab, you'll be able to run the mapreduce programs we write using `make` commands. The `make` commands will be of the form `make PROGRAMNAME-INPUTSIZE`. If you wish to try out the input file you generated here, you can instead run:

```
$ make PROGRAMNAME myinput=YOUR_SEQ_FILE_FROM_STEP_3.txt.seq # Output in wc-out-PROGRAMNAME/ directory
```

Exercise 1: Running Word Count

For this exercise you will need the Makefile and already-completed WordCount.java. You must compile and package the `.java` source file into a `.jar` and then run it on our desired input. Luckily, this is available as a convenient `make` command:

```
$ make wordcount-small
```

This will run WordCount over `billOfRights.txt.seq`. Your output should be visible in `wc-out-wordcount-small/part-r-00000`. If we had used multiple reduces, the output would be split across `part-r-[id.num]`, where Reducer "`id.num`" outputs to the corresponding file. The key-value pair for your Map tasks is a document identifier and the actual document text.

Next, try your code on the larger input file `complete-works-mark-twain.txt.seq`. In general, Hadoop requires that the output directory not exist when a MapReduce job is executed, however our Makefile takes care of this by removing our old output directory. Remember that we DON'T need to rebuild `wc.jar`, separately; the Makefile takes care of all the details.

```
$ make wordcount-medium
```

Your output for this command will be located in the `wc-out-wordcount-medium` directory. The first few lines will be confusing since the words you see there are actually numbers (for example, chapter numbers). Search through the file for a word like "the" to get a better understanding of the output. You may also notice that the Reduce "percentage-complete" moves in strange ways. There's a reason for it -- your code is only the last third of the progress counter. Hadoop treats the distributed shuffle as the first third of the Reduce. The sort is the second third. The actual Reduce code is the last third. Locally, the sort is quick and the shuffle doesn't happen at all. So don't be surprised if progress jumps to 66% and then slows.

Exercise 2: Document Word Count

Open DocWordCount.java. Notice that it currently contains the same code as `WordCount.java` (but with modified class names), which you just compiled and tried for yourself. Modify it to **count the number of documents containing each word** rather than the number of times each word occurs in the input.

You should only need to modify the code inside the `map()` function for this part. Each call to `map()` gets a single document, and each document is passed to exactly one `map()`.

You can test DocWordCount using either of the following (for our two data sets):

```
$ make docwordcount-small  # Output in wc-out-docwordcount-small/
```

OR

```
$ make docwordcount-medium # Output in wc-out-wordcount-medium/
```

Check-off

- Explain your modifications to `DocWordCount.java` to your TA.
- Show your output for `billOfRights` (aka the output for running `make docwordcount-small`). In particular, what values did you get for "Amendment", "the", and "arms"? Do these values make sense?

Exercise 3: Working with Spark

Now that you have gained some familiarity with the MapReduce paradigm, we will shift gears into Spark and investigate how to do what we did in the previous exercise in Spark! We have provided a complete wordcount.py, to get you a bit more familiar with how Spark works. To help you with understanding the code, we have added some comments, but feel free to check out transformations and actions on the Spark website for a more detailed explanation on some of the methods that can be used in Spark.

To get you started on implementing DocWordCount.java in Spark, we have provided a skeleton file docwordcount.py. For this lab, we will be using the same .seq files that we used for Hadoop, but Spark also allows us to work with other inputs as well! If you're interested, you can take a look at the Spark website, but you will not need to worry about that for this lab.

In this part, you may find it useful to look at the transformations and actions link provided above, as there are methods that you can use to help sort an output or remove duplicate items. To help with distinguishing when a word appears in a document, you may want to make use of the document ID as well -- this is mentioned in the comments of flat_map. Make sure the output you get in Spark is similar to the output you get in Hadoop.

To test your docwordcount.py, you can run either of the following two commands:

```
$ make sparkdwc-small  # Output in spark-wc-out-docwordcount-small/
```

OR

```
$ make sparkdwc-medium # Output in spark-wc-out-docwordcount-medium/
```

Check-off

- Explain to your TA what you modified in `docwordcount.py`.
- Show your output for `billOfRights` (aka the output for running `make sparkdwc-small`).

Exercise 4: Full Text Index Creation

Open index.py. Notice that the code is similar to `docwordcount.py`. Modify it to output every word and a list of locations (document identifier followed by the word index of EACH time that word appears in that document). Make sure your word indices start at zero. Your output should have lines that look like the following (minor line formatting details don't matter):

```
(word1  document1-id, word# word# ...)
(word1  document2-id, word# word# ...)
. . .
(word2  document1-id, word# word# ...)
(word2  document3-id, word# word# ...)
. . .
```

Notice that there will be a line of output for EACH document in which that word appears and EACH word and document pair should only have ONE list of indices. Remember that you need to also keep track of the document ID as well.

For this exercise, you may not need all the functions we have provided. If a function is not used, feel free to remove the method that is trying to call it. Make sure your output for this is sorted as well (just like in the previous exercise.

You can test index by using either of the following commands (for our two data sets):

```
$ make index-small # Output in spark-wc-out-index-small/
```

OR

```
$ make index-medium # Output in spark-wc-out-index-medium/
```

The output from running `make index-medium` will be a large file. In order to more easily look at its contents, you can use the commands `cat`, `head`, `more`, and `grep`:

```
$ head -25 OUTPUTFILE       # view the first 25 lines of output
$ cat OUTPUTFILE | more     # scroll through output one screen at a time (use Space)
$ cat OUTPUTFILE | grep the # output only lines containing 'the' (case-sensitive)
```

Make sure to verify your output. Open `complete-works-mark-twain.txt` and pick a few words. Manually count a few of their word indices and make sure they all appear in your output file.

Check-off

1. Explain your code in `index.py` to your TA.
2. Show your TA the first page of your output for the word "Mark" in `complete-works-mark-twain.txt.seq` to verify correct output. You can do this by running: `cat spark-wc-out-index-medium/part-00000 | grep Mark | less`