Assignment 1

Due Friday September 16 2022.

This assignment will introduce you to working with the MapReduce framework and the cluster.

The biggest goal of this assignment is to clear the basic technical hurdles so we can get down to some real work. See the tech instructions for the course, and instructions for our cluster in particular.

Put some files in HDFS

The files that Hadoop jobs use as input (and produce as output) are stored in the cluster's HDFS (Hadoop Distributed File System). There are a few things you need to know to work with it.

The `hdfs dfs command` (which is a synonym for `hadoop fs` commands, which you might also see in docs) is used to interact with the file system. By convention, your home directory is `/user/<USERID>/` and you should keep your files in there. For example, on the cluster, either of these commands will list the files in your home directory:

```
hdfs dfs -ls /user/<USERID>
hdfs dfs -ls
```

Create a directory to hold input files for our first job and copy some files into it. There are some files to work with on the gateway node in `/home/bigdata/`. (These are the texts from the NLTK Gutenberg corpus plus one extra that we'll see later.) It is much easier to separate the files of a data set into their own directory: the easiest way to specify input is "all of the files in this directory."

```
hdfs dfs -mkdir wordcount-2
hdfs dfs -copyFromLocal /home/bigdata/wordcount/* wordcount-2/
hdfs dfs -mkdir wordcount-1
hdfs dfs -cp wordcount-2/a* wordcount-1/
```

This creates a directory `wordcount-2` with the full data set, and `wordcount-1` with a subset that you can run smaller and quicker tasks on while testing. (The full data set here isn't very large, but this is probably a good habit: experiment with small data sets where you can iterate quickly.)

If you'd like to experiment with the same files off the cluster, you can download them: wordcounts-1, wordcounts-2.

Compile a job

The "word count" problem is a common place to start with MapReduce, and we will do that here. The idea is that you have a large collection of text files and would like to count the number of times each word is used in the text.

Start with the provided WordCount code. **Have a look at the code**: it is going to count the number of occurrences of each word in the files (coming to conclusions like "the" appears 2827 times"). Can you see how that's going to happen?

This needs to be compiled to `.class` files and then combined into a `.jar` file that can be submitted to the cluster. See CompilingHadoop.

Build a `.jar` file containing the WordCount class and the two inner classes. Copy the JAR to somewhere in your home directory on the cluster (also described in the instructions above).

Run a job

When we run this job, it takes two arguments on the command line: the directories for input and output files. (Those are handled in the lines above that access `args[0]` and `args[1]`.

The command to **submit the job** to the cluster will be like:

```
yarn jar wordcount.jar WordCount wordcount-1 output-1
```

If all happens to go well, you can **inspect the output** the job created:

```
hdfs dfs -ls output-1
hdfs dfs -cat output-1/part-r-00000 | less
```

And remove it if you want to run again:

```
hdfs dfs -rm -r output-1
```

There was one file created in the output directory because there was one reducer responsible for combining all of the map output (one is the default). We can change the configuration so three reducers run:

```
yarn jar wordcount.jar WordCount -D mapreduce.job.reduces=3 \
    wordcount-1 output-2
```

(or equivalently, could do `job.setNumReduceTasks(3)` in the `run` method.) Re-run the job with three reducers and have a look at the output. [?]

You can also specify zero reducers: in that case, Hadoop will simply dump the output from the mappers. This of this as a chance to debug the intermediate output. Try that:

```
yarn jar wordcount.jar WordCount -D mapreduce.job.reduces=0 \
    wordcount-1 output-3
```

Have a look at this output. Is it what you expected the mapper to be producing? [?]

Development Environment

All of your jobs will eventually run on the cluster, but that's not always the best way to develop code: the turnaround time to upload and submit into the cluster can be uncomfortably long.

Spend some time getting a development environment that you like. See CompilingHadoop for instruction to run MapReduce locally (running as a process on your machine) which is much faster to start and process small data sets than the full cluster.

But, we don't recommend spending a huge amount of time on a Java environment: we will be switching to Python and Spark fairly soon.

Modify WordCount

Copy the example above into a new class `WordCountImproved` (in `WordCountImproved.java`). In this class, we will make a few improvements.

Simplify the code

The original code counts using the Java `Integer` values, which are 32-bit integers. Maybe we wanted to count **lots** of words (this is "big data" after all). **Update the mapper** so it produces `(Text, LongWritable)` pairs, so we are working with 64-bit integers. (You don't have to worry about the reducer: we're just about to throw it away.)

It turns out that the `IntSumReducer` reducer in the original `WordCount` class was instructive but unnecessary. Since this is such a common usage, Hadoop includes an `IntSumReducer` that does exactly the same thing, and org.apache.hadoop.mapreduce.lib.reduce.LongSumReducer that does the same for `Longs`. **Remove the inner class** `IntSumReducer` and use the pre-made `LongSumReducer` instead.

Test your job to make sure it still has the same behaviour. Now we have a bigger problem...

Redefine "words"

Having a closer look at the output, I notice things like this:

```
$ hdfs dfs -cat output-1/part-r-00000 | grep -i "^better"
better  144
better';        1
better, 14
better,"        1
better,'        1
better. 9
better."        3
better.'        1
better; 4
```

All of these are really instances of the word "better", but with some punctuation making them count as different words. This is the fault of the `StringTokenizer` used in the example which just breaks up the line on any whitespace, which apparently isn't quite the right concept of a "word".

And now we open the shockingly-complicated door to international words and characters. First: if you don't have a really good grasp on what the word "Unicode" means, read The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets. Go ahead: we'll wait. [And if you knew those things, try Dark Corners of Unicode.]

Okay now... it turns out that not everybody speaks English. You might easily decide that words are things that contain characters A–Z and a–z, but those are words that don't meet that definition: "garçon" "hyvä" "Hyunsev".

Lots of languages have lots of rules for word breaks, and I don't know most of them. We will do our best and make the rule that words are separated by any spaces or any punctuation. The module java.util.regex.Pattern can help us, along with this regular expression and the `.split()` method on it:

```
Pattern word_sep = Pattern.compile("[\\p{Punct}\\s]+");
```

Update your mapper so that it **ignores punctuation** by splitting on the above regular expression. Make it **ignore case** by applying `.toLowerCase()` to each word.

One artifact of this method: it will sometimes emit an empty string as a "word". Make sure your code **ignores any length 0 words** and doesn't count them. (If they are counted, they will be the first line of the output.)

At this point, you should get results more like this:

```
$ hdfs dfs -cat output-4/part* | grep -i "^better"
better  179
```

JSON Input & Reddit Comments

It is quite common for large data sets to be distributed with each record represented as a JSON object, with one object per line in the file(s). That way, input files can be split by line (as the default `TextInputFormat` does for you), and each line is in a well understood and easy to parse format. The input files end up looking like this:

```
{"key1": "value1", "key2": 2}
{"key1": "value3", "key2": 4}
```

For example, this is how the Reddit Comments Corpus is distributed (with each file BZ2 or XZ compressed). For this problem, we will be using (a subset of) it, and determining the **average score in each subreddit**.

Create a `RedditAverage` class for this problem.

You will find small subsets of the full data set on the cluster at `/courses/732/reddit-1` and `/courses/732/reddit-2` (you can just use those as input directories on the cluster: no need to make your own copies) or you can download the same at https://gabaker.ca/732-datasets/.

Parsing JSON

The input to our mapper will be lines (from `TextInputFormat`) of JSON-encoded data. In the mapper, we will need to parse the JSON into actual data we can work with.

We will use the org.json package, which is the reference Java JSON implementation. Start by downloading the JAR file with the classes. Full docs are available, but here is a very quick tutorial on the parts we need:

```
import org.json.JSONObject;

JSONObject record = new JSONObject(input_string);

System.out.println((String) record.get("subreddit"));
System.out.println((Integer) record.get("score"));
```

There are some notes below on how to compile and include this JAR file. But first we also need...

Passing Pairs

Since we want to calculate average score, we will need to pass around *pairs*: the number of comments we have seen, and the sum of their scores. I have written a quick `Writable` implementation `LongPairWritable` to do this for us. Here is how it is used:

```
LongPairWritable pair = new LongPairWritable();
pair.set(2, 9);
System.out.println(pair.get_0()); // 2
System.out.println(pair.get_1()); // 9
```

Your mapper should be writing pairs of `Text` (the subreddit name) and `LongPairWritable` (the comment count and total score).

To compile with the JSON JAR file and `LongPairWritable`, your command will have to be like (assuming the `.jar` file is in the same directory as your code):

```
export HADOOP_CLASSPATH=./json-20180813.jar
${JAVA_HOME}/bin/javac -classpath `${HADOOP_HOME}/bin/hadoop classpath`:LongPairWritable.java Re
${JAVA_HOME}/bin/jar cf al.jar *.class
```

If you want to run on the cluster, you'll need to copy over **both** `.jar` files, and the command will be something like this:

```
export HADOOP_CLASSPATH=./json-20180813.jar
yarn jar al.jar RedditAverage -libjars json-20180813.jar reddit-1 output-1
```

Definitely test at this stage. Remember `-D mapreduce.job.reduces=0` to turn off the reducer and see the map output directly.

Reducing to Averages

Write a reducer that takes the mapper's key/value output (`Text`, `LongPairWritable`) and calculates the average for each subreddit. It should write one `Text`, `DoubleWritable` pair for each subreddit. [?]

The Combiner

At this point, the job should be giving correct output, but the mapper like probably producing lots of pairs like this:

```
canada  (1,1)
canada  (1,9)
canada  (1,8)
canada  (1,1)
```

Those all have to be shuffled to the reducer, but it would be much more efficient to combine them into:

```
canada  (4,19)
```

Remember: the combiner does reducer-like work on chunks of the mapper output, with the goal of minimizing the amount of data that hits the shuffle. The input and output of the combiner must be the same types.

Since our reducer must produce output that is a different type than its input, it doesn't make sense to use it as a combiner. (Go ahead: try it. It should fail complaining about types being wrong.)

Write a combiner (that `extends Reducer` just like a reducer class) that sums the count/score pairs for each subreddit. It should be similar to your reducer, but write long pair values, instead of doubles. [?]

Because you will be using different types as the combiner and reducer output, they must be specified separately with code like:

```
job.setMapOutputKeyClass(???.class);
job.setMapOutputValueClass(???.class);
job.setOutputKeyClass(???.class);
job.setOutputValueClass(???.class);
```

Cleanup

We'll ask you to tidy up after yourself for these assignments. There should be plenty of space on the cluster, but as files get bigger, we'd like to keep as few copies as possible lying around:

```
hdfs dfs -rm -r "output*"
```

Questions

In a file `answers.txt`, answer these questions:

1. Are there any parts of the original `WordCount` that still confuse you? If so, what?
2. How did the output change when you submitted with `-D mapreduce.job.reduces=3`? Why would this be necessary if your job produced large output sets?
3. How was the `-D mapreduce.job.reduces=0` output different?
4. Was there any noticeable difference in the running time of your `RedditAverage` with and without the combiner optimization?

Submission

Submit the files you created to Assignment 1 in CourSys.

Updated Wed Sept. 14 2022, 12:55 by gabaker.