**ETL (Extract, Transform, Load)**

For this assignment, I decided to export the Yelp data from MySQL since it was relatively clean data. I used the ‘INTO OUTFILE’ command. For example:

**(SELECT 'review\_id', 'user\_id', 'business\_id', 'stars', 'date', 'text', 'useful', 'funny', 'cool', 'type')**

**UNION**

**(SELECT review\_id, user\_id, business\_id, stars, date, text, useful, funny, cool, type**

**FROM review**

**LIMIT 3000000**

**Into Outfile 'C:/ProgramData/MySQL/MySQL Server 5.6/Uploads/review3.txt'**

**Fields Enclosed By '"' Terminated By '|||' Escaped by '"'**

**Lines Terminated By '\r\n');**

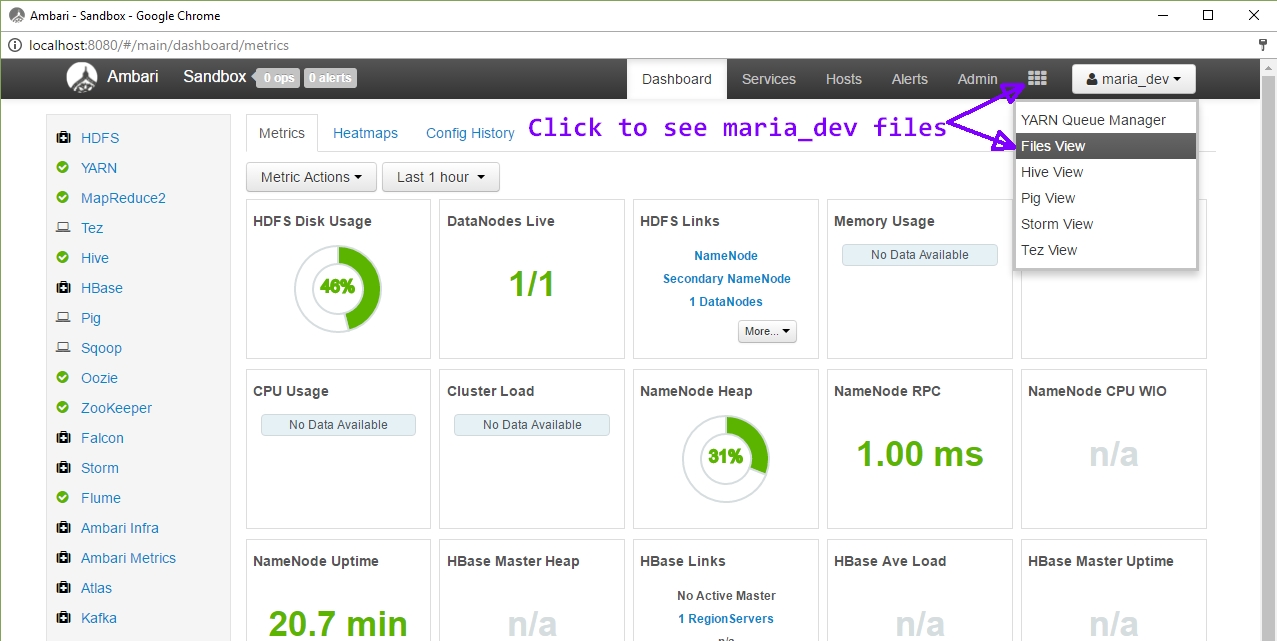
The previous SQL command extracts data from the review table and places it in a text file. The values in the text file are separated by three pipe symbols. I did not use three pipes as a separator for every tables data. For the business table I used a comma as a separator. Some of the values in the review table contained many commas so it made sense to use a separator other than a comma to keep the values separate. I choose to extract a relatively small volume of data – 100,000 records from each table. However, for the review data, I also extracted a much larger chunk of data, 2.1 GB file, so I could run a MapReduce job on data that is larger than the HDFS block size.

After I extracted all the data I needed to setup the Big Data platform, HortonWorks Sandbox. I downloaded a virtual machine monitor called Virtual Box and installed it. Then I downloaded the Sandbox version from the HortonWorks website for Virtual Box. It was a simple matter to setup. I followed the instructions provided by HortonWorks to setup the Sandbox using Virtual Box. The instructions can be found here: <https://hortonworks.com/wp-content/uploads/unversioned/pdfs/InstallingHortonworksSandbox2onWindowsusingVB.pdf>

You must ensure the computer you are planning to use with the Sandbox has Virtualization enabled in the BIOS. I initially did not have it enabled and received an error when starting up HortonWorks. When you start up the Sandbox the virtual machine is a little slow, but be patient. Then you will be prompted to open a web browser and login.

These are the steps to get into the HortonWorks Sandbox once the virtual machine monitor loads the Sandbox virtual machine.

* Enter <http://127.0.0.1:8888> in a web browser.
* In the login screen the presents itself enter **maria\_dev** as the username and password
* You will be presented with the Ambari GUI environment. The first page is called the dashboard.
* To see the files for the user you are logged in under, maria\_dev, you must select the file view, as shown in the image below.

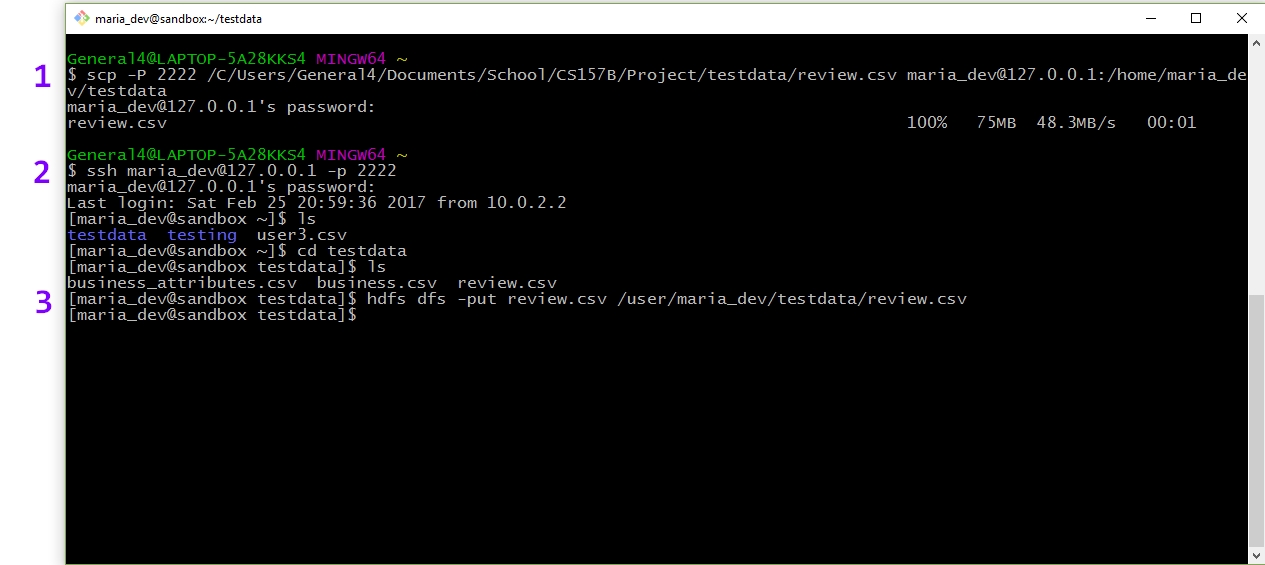


**Loading the Data**

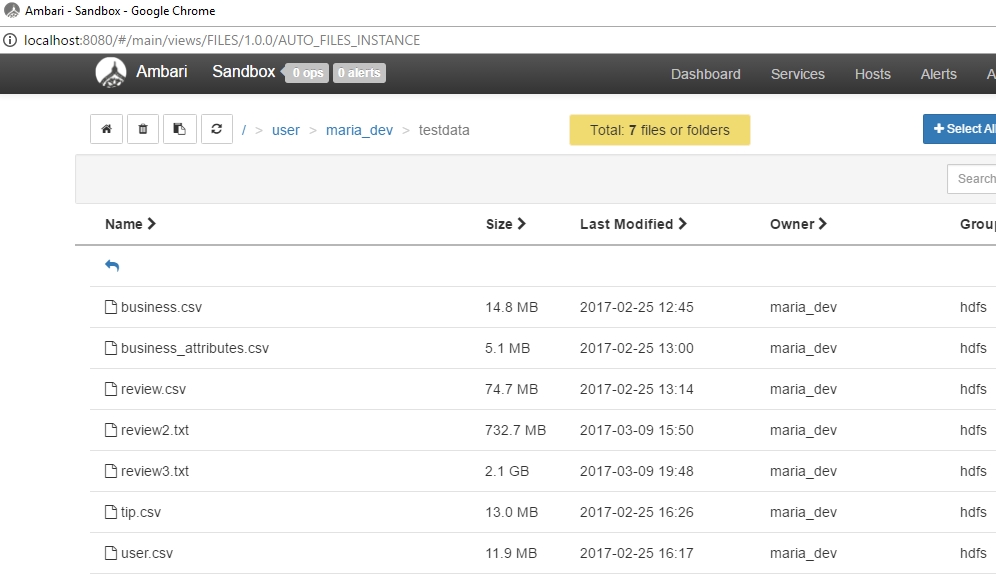
It is possible to use the Ambari GUI to transfer the Yelp data files into the HortonWorks Sandbox, but I choose to use the command line. I decided to do this to avoid any timeout errors and it is typically faster to transfer files without a GUI. Unfortunately, I discovered that it is a three-step process to transfer files using the command line.

**Three steps to transfer files into HortonWorks using a command line:**

1. **scp -P 2222 <local path of file> < user: path on remote HDFS>**
2. **ssh maria\_dev@127.0.0.1 -p 2222     🡨** to login to Hadoop HDFS
3. **hdsf dfs -put <file\_name> <dest. path in SandBox>**



The image above shows the three-step file transfer process



The image above shows my data files in the Sandbox. Notice that there are three review files of varying sizes. I was experimenting running MapReduce jobs with different sizes of input data.

**Query**

Find average review rating for all businesses in a specific zip code

To solve this using MySQL I would write a query like the following:

**SELECT postal\_code, AVG(stars), count(\*) AS business\_count**

**FROM business**

**GROUP BY postal\_code**

**ORDER BY business\_count DESC;**

+-------------+--------------------+----------------+

| postal\_code | AVG(stars) | business\_count |

+-------------+--------------------+----------------+

| 89109 | 3.5757068452380953 | 2688 |

| 85251 | 3.9410935738444195 | 1774 |

| 85281 | 3.67276814386641 | 1557 |

| 89119 | 3.489802855200544 | 1471 |

...

| 85253 | 3.9275568181818183 | 352 |

| 85248 | 3.690201729106628 | 347 |

| 85051 | 3.3771676300578033 | 346 |

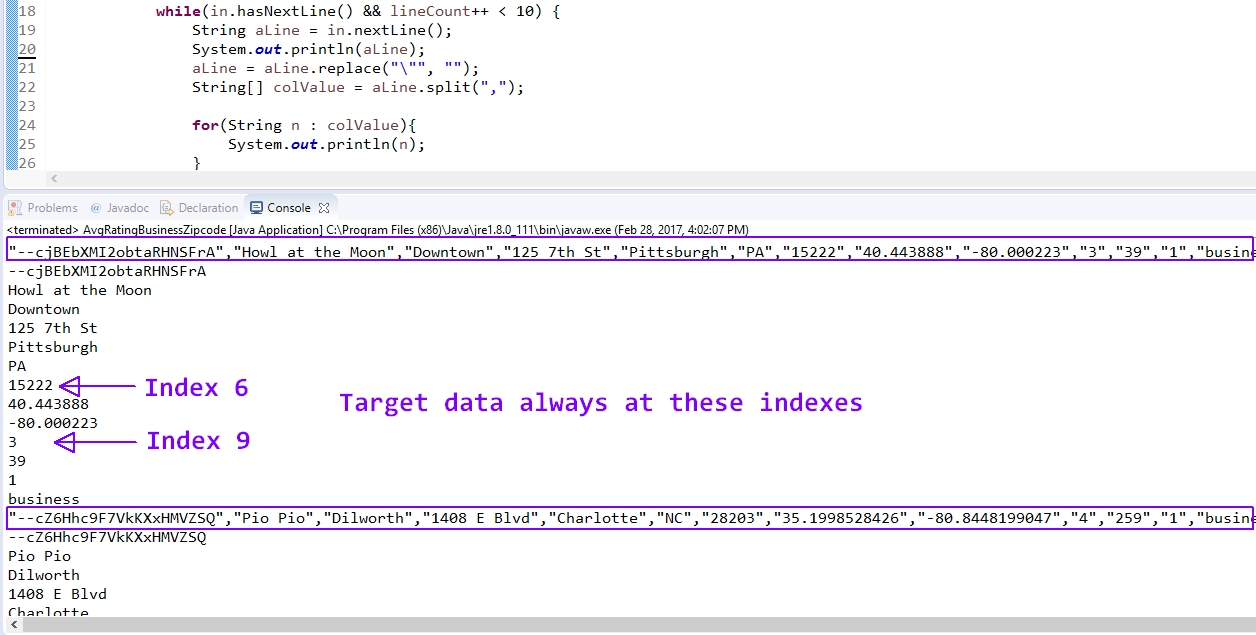
| 89183 | 3.6246376811594203 | 345 |

+-------------+--------------------+----------------+

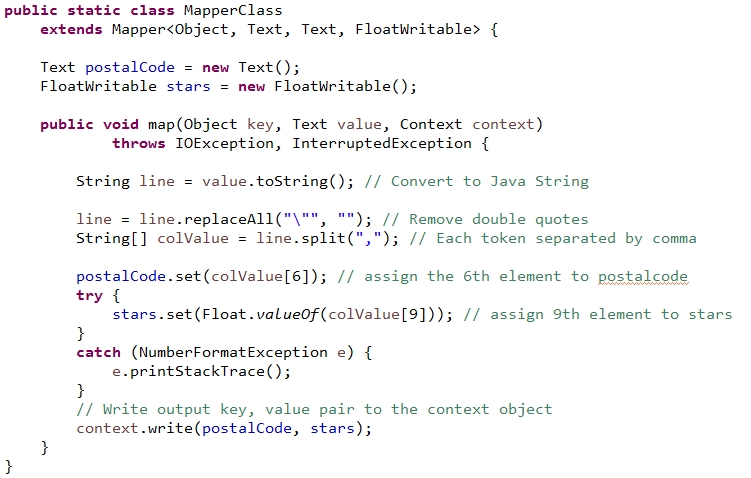
Solving the query using the Hadoop MapReduce API is much different. You must decompose the problem into a map task and a reduce task and then write functions for each of these. The map and reduce functions take key and value as inputs and outputs. The input data is divided into 128 MB chunks, which is the default HDFS block size and each map function works on a node where the data lives. This is called data locality optimization. This allows each map function to run in parallel. For my problem query, the map function needs to capture the postal code and star rating from each business and output a postal code key and star value for each distinct business. The output of the mapper stage is grouped and sorted automatically by Hadoop. This data is partitioned depending on the number of reducers. The default is one reducer task. If the output of the map stage will result in data that is greater than the HDFS block size, Hadoop will create more partitions and hence, more reducers to work on the data. The reduce task will iterate over the star values and compute a sum and then an average. A postal code and the average rating for all the businesses in that postal code will be the output of the reducer function.

The MapReduce job is written in Java using the MapReduce API. You must add many Hadoop JAR files to the project build path to compile the code. The JAR files can be obtained here: http://mvnrepository.com/artifact/org.apache.hadoop

Before trying to write map and reduce functions, I chose to write a regular Java program to zero in on the target data. The picture below shows this.

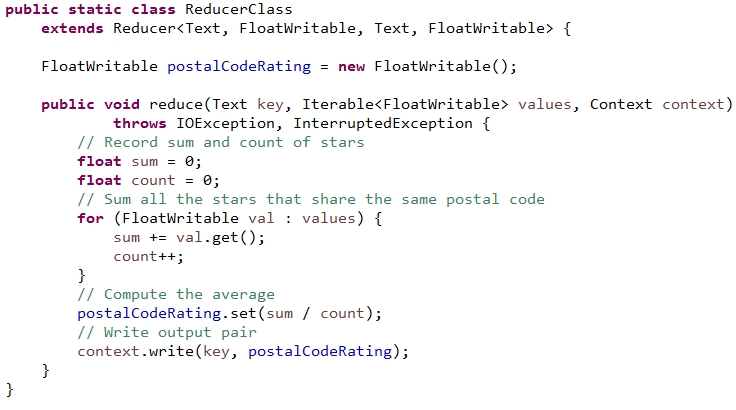


Once I had a better understanding of how to capture the target data, I proceeded to write a MapReduce job using the MapReduce API. To become better familiar with the API, I briefly reviewed the Hadoop MapReduce API documentation. It can be found here: [www.hadoop.apache.org](http://www.hadoop.apache.org) The following image is my map function.



Notice that the key is not even used for the map function. The key, by default, is just the line offset in bytes from the beginning of the file. The map function removes double quotes from the text and tokenizes the line and places it in an array. Then, the values in the 6th and 9th index positions are captured and outputted as key and value.

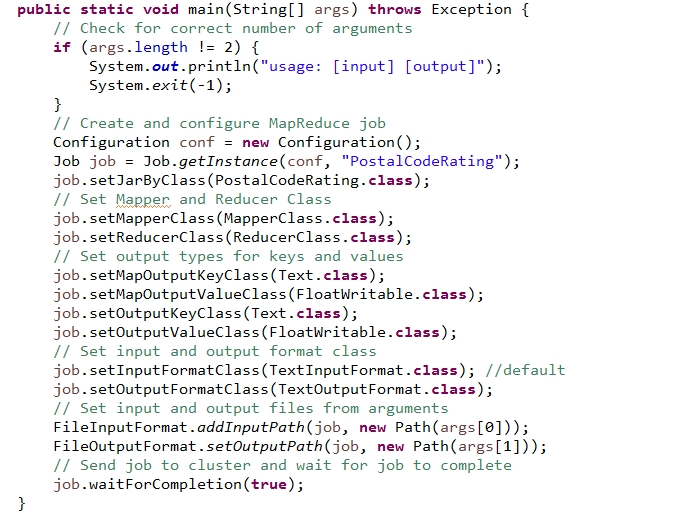
The following is the reduce function.



As mentioned above, the reduce function iterates over the array of values and computes and average. It then outputs the postal code and average star rating for all the businesses in that postal code.

Next, you must write a function that tells Hadoop all about your MapReduce job. This function is called the job runner. You must create an instance of a Job object and use this to configure the MapReduce job.

**Job job = Job.getInstance(conf, “PostalCodeRating”)** This line of code tells Hadoop that the class that contains the map and reduce functions is called PostalCodeRating. Additionally, you tell Hadoop what the input and output types for the map function is using the **setMapOutputKeyClass, setMapOutputValue** There are similar functions for the reducer function. The following is an image that show the job runner function for my MapReduce job.



After the code is written, it must be compiled into class files. Next, these class files need to be packaged up and placed into a JAR file. Transfer this JAR file to Hadoop, because it is used to run the MapReduce job.

**The Command to Run a MapReduce Job**

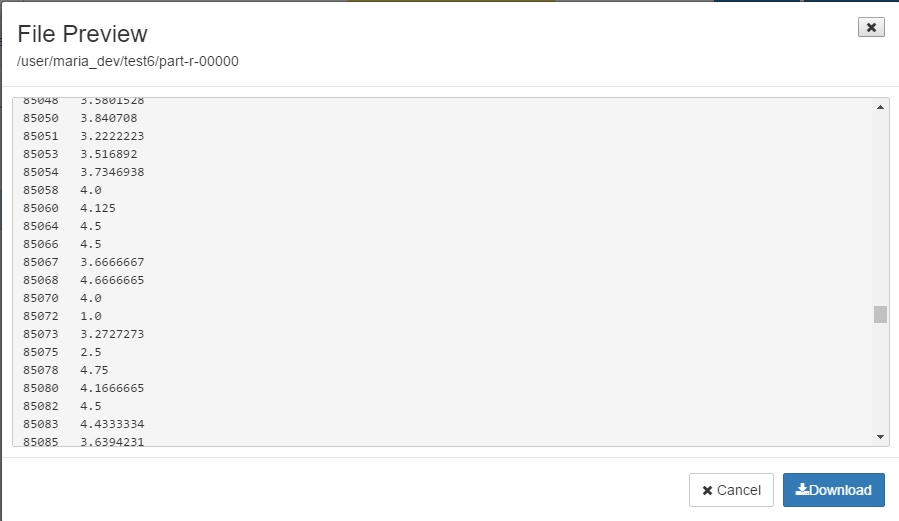
**hadoop jar <jar\_file\_name> [mainClass] <datafile> <path to output directoy>**

**Forexample:**

**hadoop jar MapReduceTest.jar WordCount2 shakespeare.txt /user/maria\_dev/test**

It’s important to note that the output directory, named test in the example above, needs to be unique because Hadoop will abort the MapReduce job if it is not. This is to prevent loss of any previous data analysis.

Depending on the size of the results of the MapReduce job, there will be one or more *part* files created. If the output size is more than the HDFS block size, the results will be in multiple files. For my job only a single output file was created, as seen in the following image.



As you can see, the resulting output data is just like that of the MySQL output data.

**Updates and Maintaining Data Quality**

My MapReduce job worked on clean, historical data. Because of this I had to do very little maintenance work in the mapper function to ensure it worked on the proper type of data. If this job was to be run frequently on newly acquired data, the mapper would have to parse the data and ensure that the values were of the correct type and in the correct form. The use of regular expressions would be required. With a MapReduce job, you cannot set tuple or attribute based checks on new data. The same is true if the data files are updated in the HDFS file system. Since there is no schema imposed on the data or constraint checks, you must program type checking and validation into your mapper function. One strategy may be to preprocess the data before letting the map function work on it. You could write a server-side script that works on newly acquired data, cleaning and filtering it. Updates to data on the HDFS are rare. Hadoop data is typically written once and read many times. This is unlike the RDBMS where updates occur frequently.

**Optimizing Performance**

With MySQL you can use a profiler to measure the performance of your queries. With Hadoop, the default output for a MapReduce job is verbose. The verbose output allows you to compare job outputs after making optimization changes in the Sandbox. I demonstrate this with the next query. Since the above query worked on the business data set which was relatively small, my next query worked on a larger data size.

**Scaling Out – Working on larger input data**

**Query:**

**Find all businesses that have over 1000 reviews and have a star rating greater than 4**

SELECT business\_id, AVG(stars) AS starAVG, COUNT(\*) AS reviewCount

FROM review

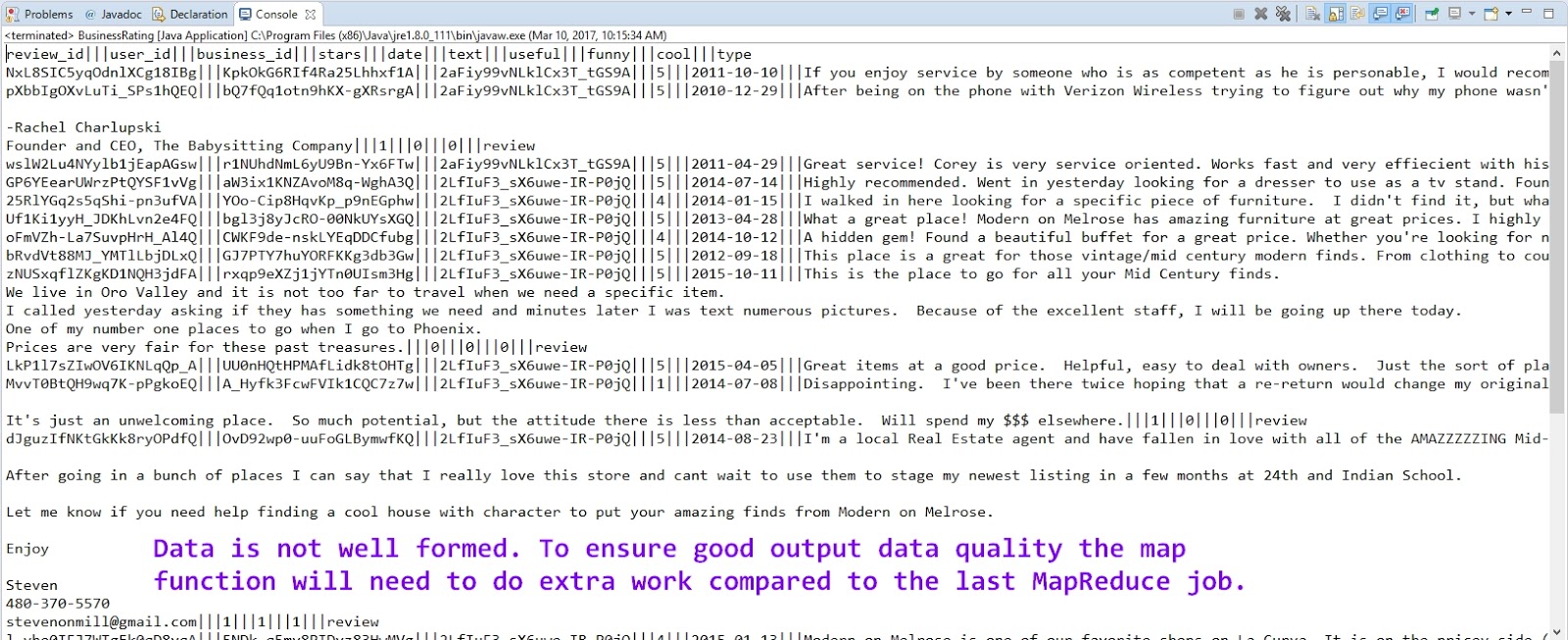
GROUP BY business\_id

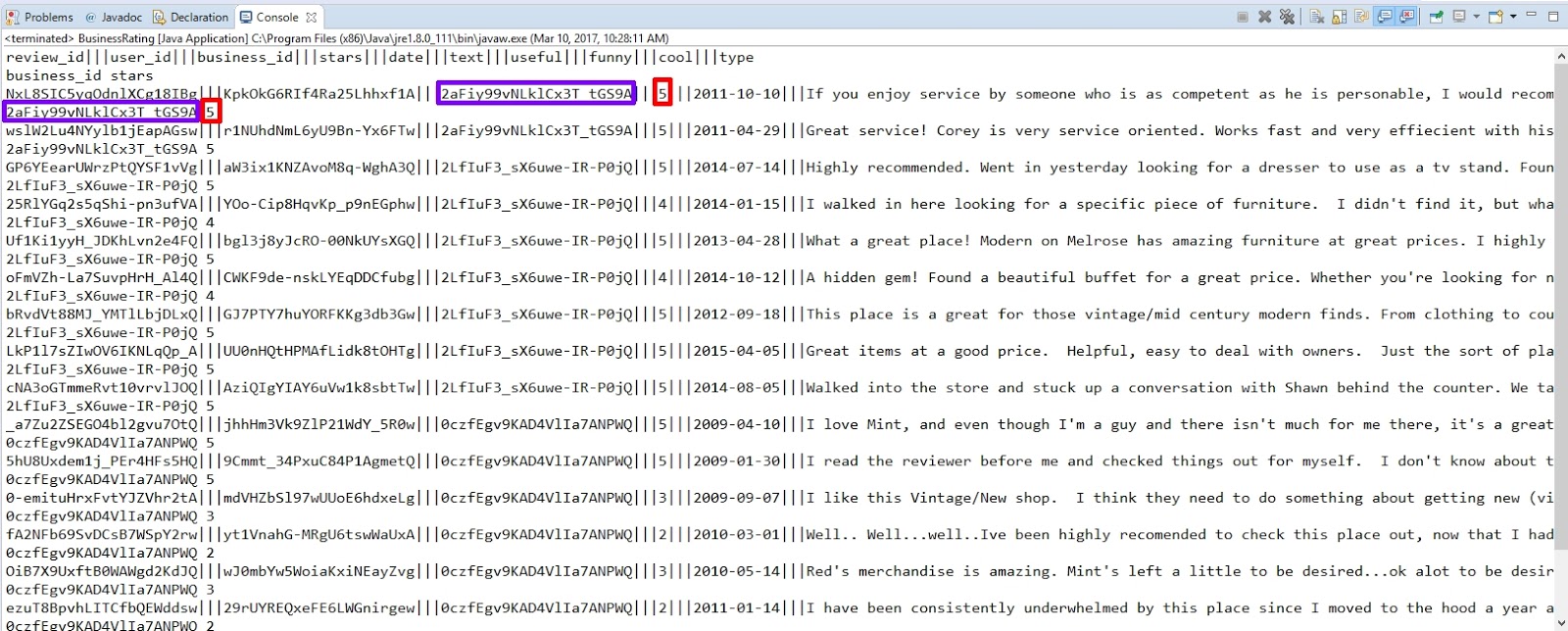
HAVING reviewCount > 1000 AND starAVG > 4;

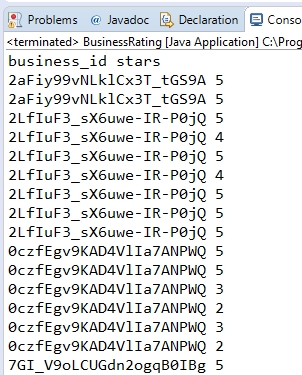
Why this query? I would have liked to try a more complex query that maybe used a join, but time did not allow for this. Instead, I chose to write a similar query as the first to solidify my understanding of how to write a straight-forward MapReduce job.

For this query I will use a file named review3.txt which is 2.1 GB is size. By running a MapReduce job on a larger data size, there will be more map tasks created. Each map task will run in parallel on the node where the data lives. It’s important to note that just because the data input size has increased, it does not necessarily mean that more reducer tasks will also be created. The number of reducers is not based on the input data size. It’s more closely related to the map output data size.

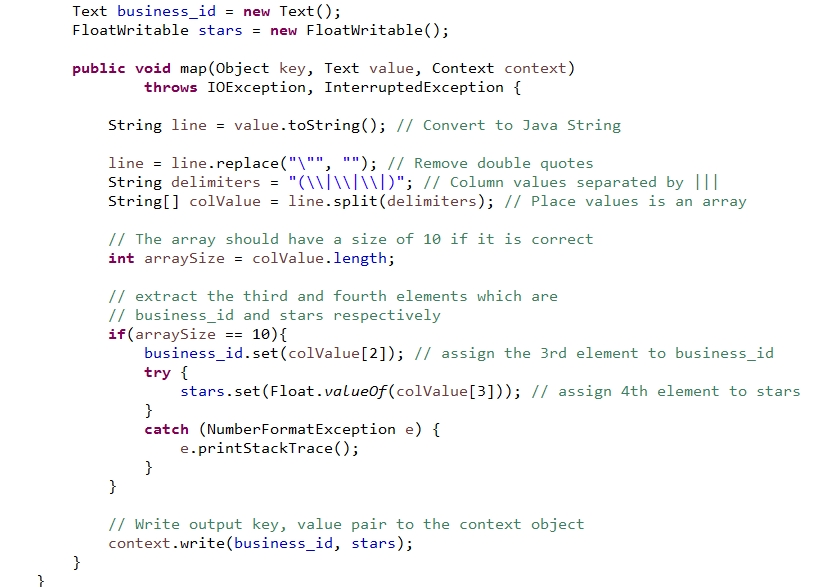
Just as before I wrote a java program to zero in on the target data before trying to write a map and reduce function using the API. The contents of review3.txt had a lot of data the was not well formed. Some data spilled onto multiple lines. Since it was not the case that each line of data was a different record, my map function would have to do extra work to ensure quality data is outputted to the reducer. I needed to make a command decision on how I would ensure only good data left the map function. What I decided to do was throw away all malformed data. How did I do this? The values were separated by three pipe symbols so I used this to tokenize each line and place it in an array. Since, the review data contained 10 columns, only those lines that contained exactly 10 elements (values) where considered.



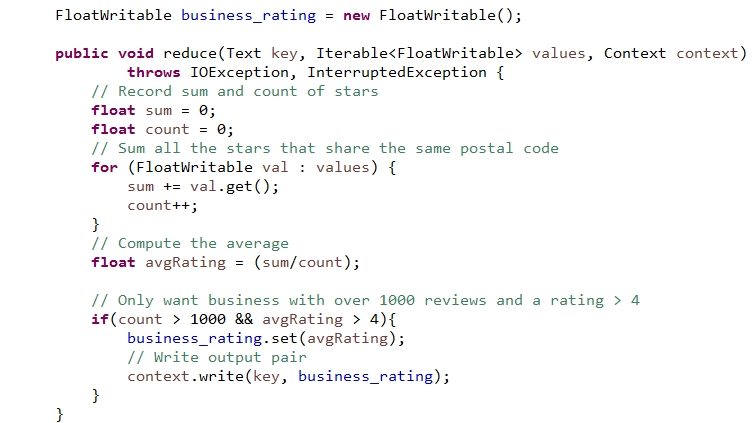




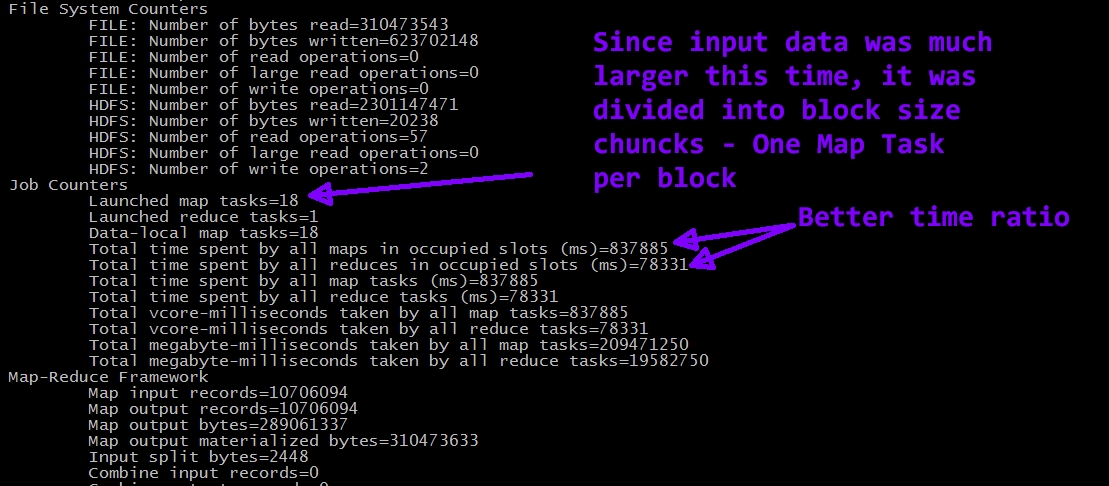
Next, I wrote code using the Hadoop MapReduce API



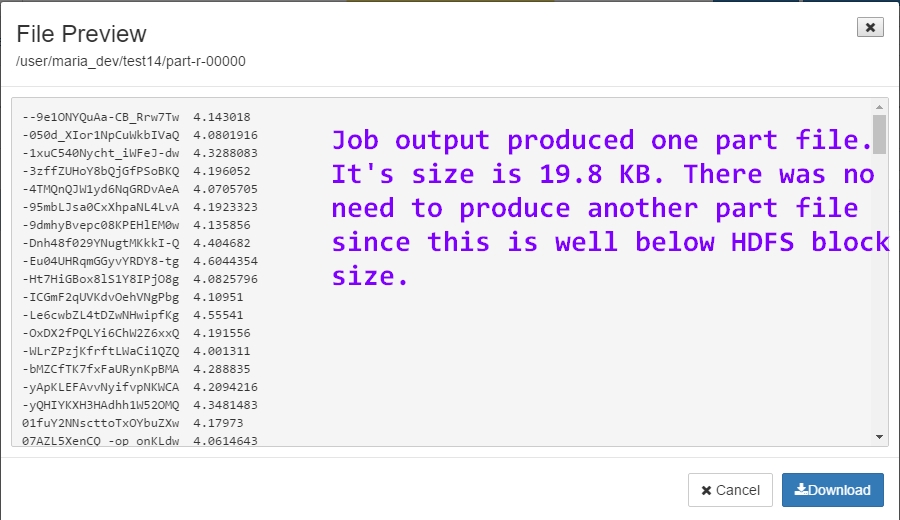
Like the previous map function, each line of text is turned into a Java string and quotes are removed. The string is tokenized using three pipe symbols as a delimiter and placed into an array. If the data is well formed, there would be ten items in the array. The function only considers arrays that have 10 elements. The values at index 2 and 3 are captured. These values correspond to the business\_id and stars. In a real-world program I might also want to check the two extracted elements. For instance the business\_id needs to be 10 characters and the stars should be a numeric and be in the range of 1 to 5. These are checks to ensure data quality.



The reducer function does similar work as before, but this time only writes to the context object if the business has greater than 1000 reviews and a star rating above 4. This shows that each time a reducer works on a key, value pair, it does not necessarily have to output a key, value pair. The Job function was very similar to the last, so I will not show it here. Here is the output from running this job. Notice is has 18 map tasks.

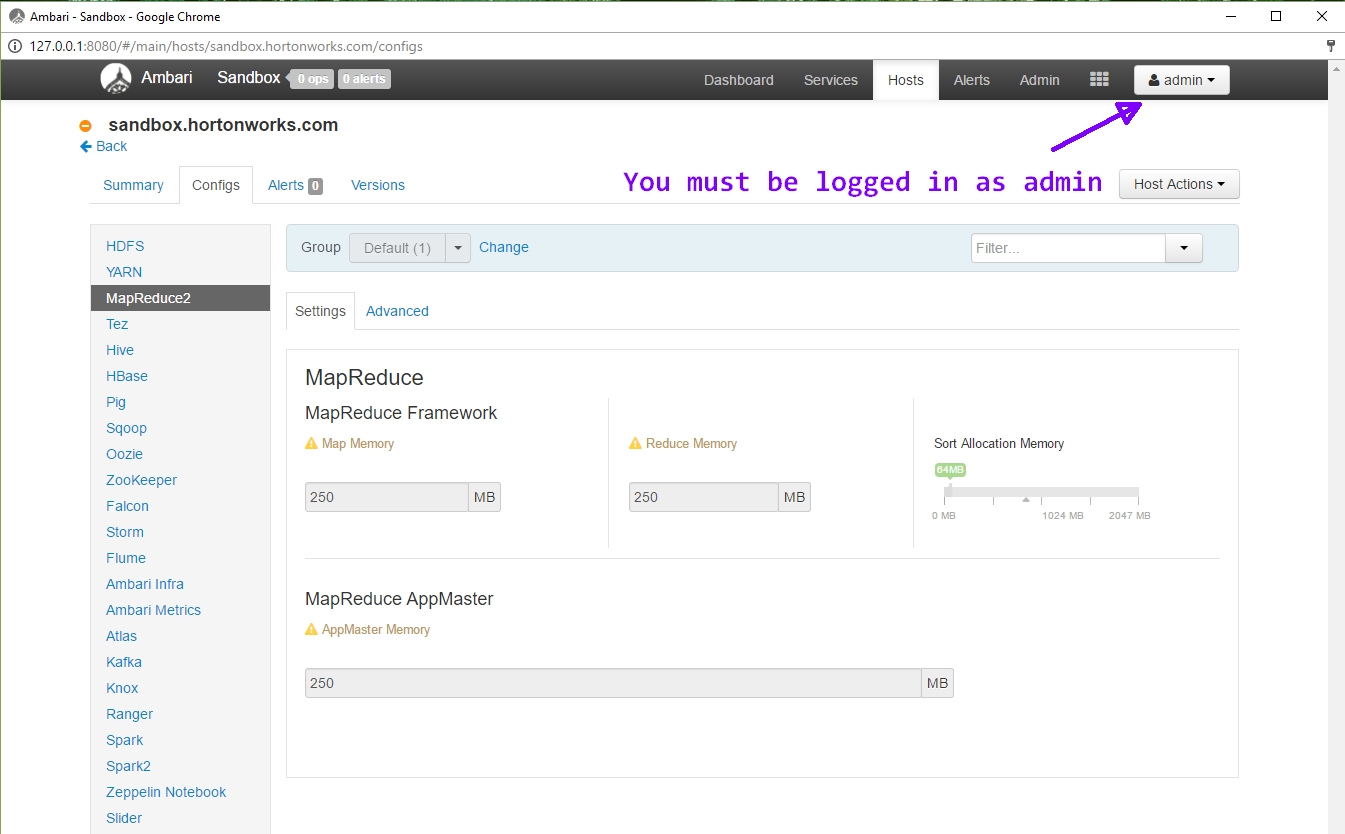


There are 18 map tasks this time since the input data was much larger. The data was divided into HDFS block size chunks. So this time there were 18 map tasks running in parallel. There still is only one reducer because the mapper output only required 1 reducer since the mapper output was relatively small. One part file was produced which was 19.8 KB.

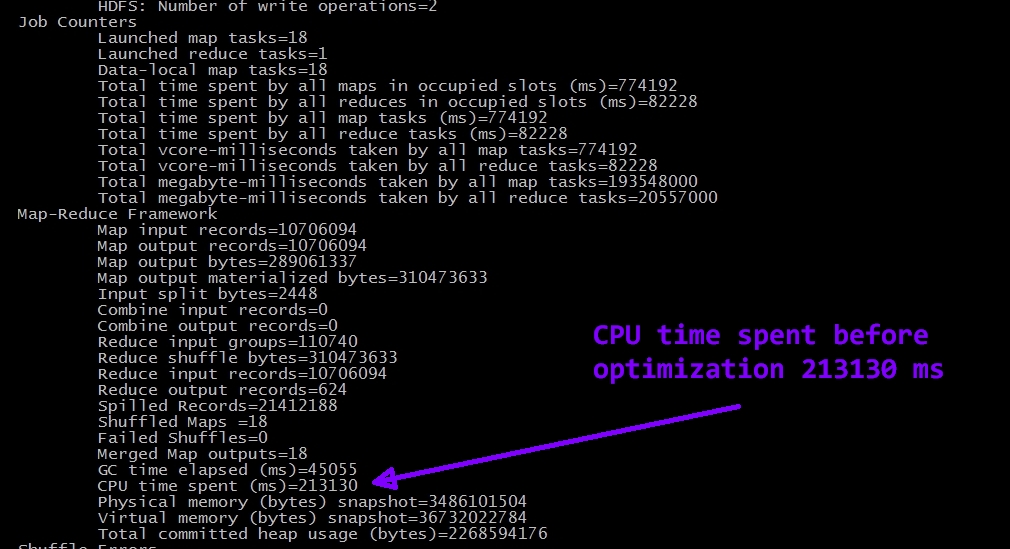


**Performance Optimization**

A good measure of performance is the CPU time spent on a job. On this last job, the CPU time spent was 213130 ms. I want to try and improve this be lowering this value. The HortonWorks Sandbox GUI has some features that allow you to try and improve the performance of your MapReduce jobs. To make any adjustments to the default settings, you must log in as an administrator. Then click on MapReduce2 in the left menu followed by config. After you make any adjustments, a message will be displayed telling you to restart the MapReduce service.

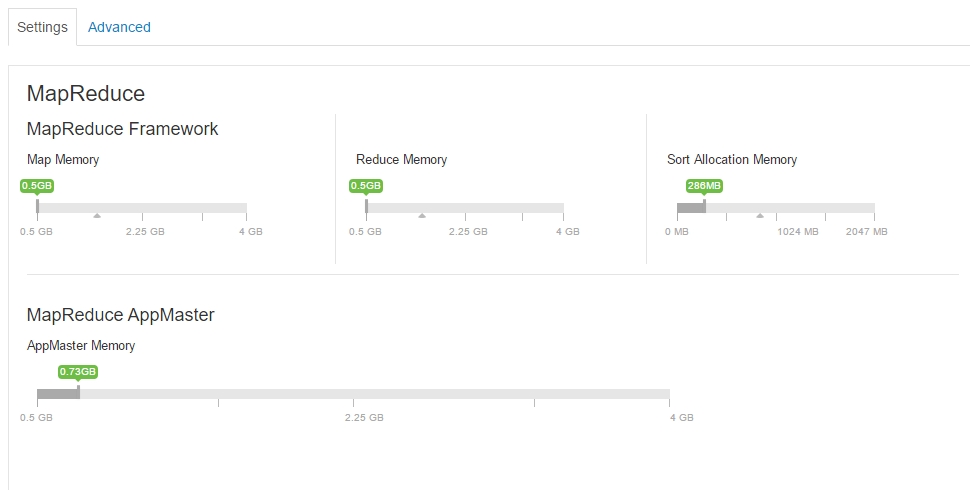


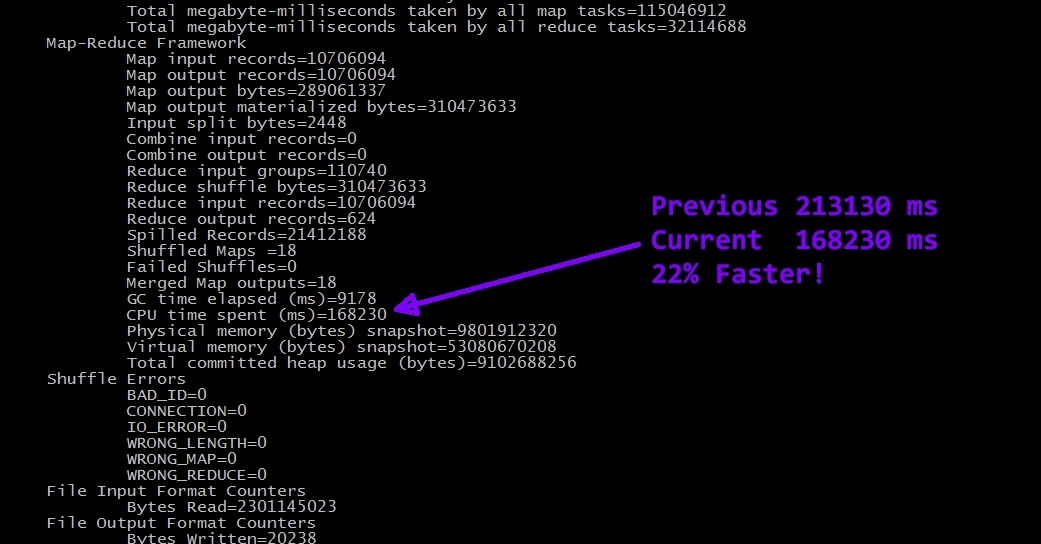
Notice that the virtual memory for a single map and reduce task is flagged in yellow because they are only set at 250 MB each.



Take note of the CPU time spent before optimization in the above image.

Below is an image showing that I increased the virtual memory to .5 GB each. Now they are green because they are not below the recommended amount.

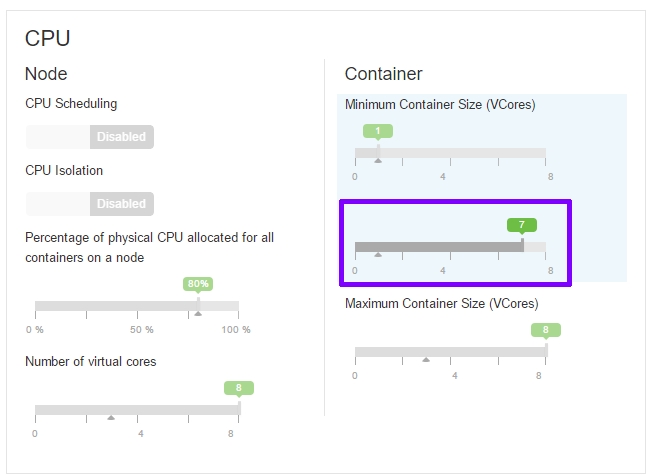


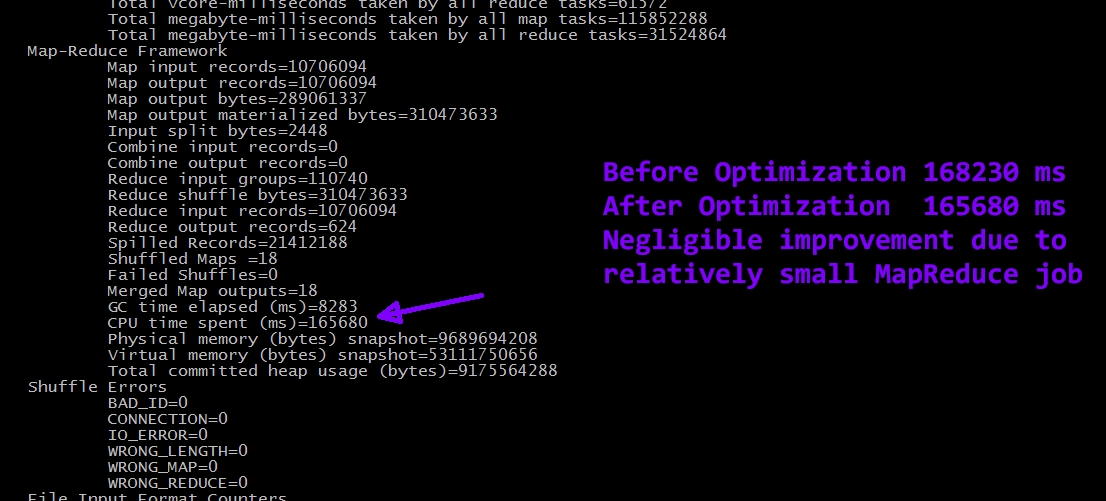


After running the job again, there was a 22% increase in CPU performance!

**Scaling up to boost CPU performance**

Now I am going to try and increase CPU performance by increasing the Virtual Cores. Select the Yarn menu on the left and then select config. Under CPU you can see that the minimum setting for virtual cores is set at 1. I will increase this to force Hadoop to use more than one VCore. Then I will run the job again.





There was a slight improvement, but it is negligible. The MapReduce job that I am running probably isn’t large enough to benefit from using more than one VCore.

**Fine tuning a MapReduce job**

Hadoop works better with a small number of large files versus a large number of small files. If you have too many small files, each of which is smaller than the block size, then a lot of map tasks will be created unnecessarily. This causes a lot of overhead and can easily be avoided. When configuring the job don’t use the **FileInputFormat** when reading in the file. This format will cause splits to occur for each small file which cause many map task to be created. By specifying the **CombineFileInputForma**t small files will be combined and splits will be about the size of the HDFS block. This will cause less map tasks to be created, reducing overhead. Generally, the Map time should be greater than the reducer time. If it is not, then something is not sett up correctly. In my first example, I had a single map task that ran just slightly longer than the reducer. This is because the input file was small and so only a single map task needed to be created. In the second query, I used a much larger input file, so 18 map tasks where created and the ratio of map time to reducer time was improved.

**Data Integrity Checks**

One way to ensure data integrity is to verify that the data has not changed since it was stored. For each file stored in the HDFS, you could store some metadata with it. Part of the metadata could be a checksum. Then, the first thing that should be done when running a MapReduce job is to verify the input data against the checksum. If it is not correct, stop the job and find out why.

As mentioned earlier, checking input data as it is ingested to the HDFS would be wise. One could write a script that processes all input data and writes it to the HDFS along with a corresponding checksum. Then when the mapper function works on the data it must do less work as long as the check sum is correct.

**Combiner Function Optimization**

For map jobs that produce a lot of data using another functions that does internal reduction would make the reducers have to work less and there would be less data transferred to the reducers in the first place. To minimize data transfer to the reducers, write a Combiner function that performs reduction on the mapper output. In the case of my first example, the list of star ratings could be very large if the number of businesses in the postal code was large. If a combiner function summed star ratings so the reducer only needed to compute an average, that would help performance. In the case of my job, not by much. But for big jobs, this would have a much better impact on performance.

Optimized reducer input < 94582, [ 345, 75 ] >

Sum of stars         number of businesses