**Spark Batch Pipeline**

**Extract Historical Data**

In addition of using the Yelp files from the previous assignment, Hadoop MapReduce, I chose to extract a subset of the Yelp data from MySQL that uses double quotes as a delimiter. I used the ‘INTO OUTFILE’ command and specified double quotes as a delimiter. Doing this allowed me to work with clean data. The code below shows a header being prepended to the data; my final solution used data files that did not include a header (I had to re-extract from MySQL).

(SELECT 'business\_id', 'name', **…** 'stars', 'review\_count', 'is\_open', 'type')

Union

(SELECT business\_id, name, **…** stars, review\_count, is\_open, type

FROM business

LIMIT 700000

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

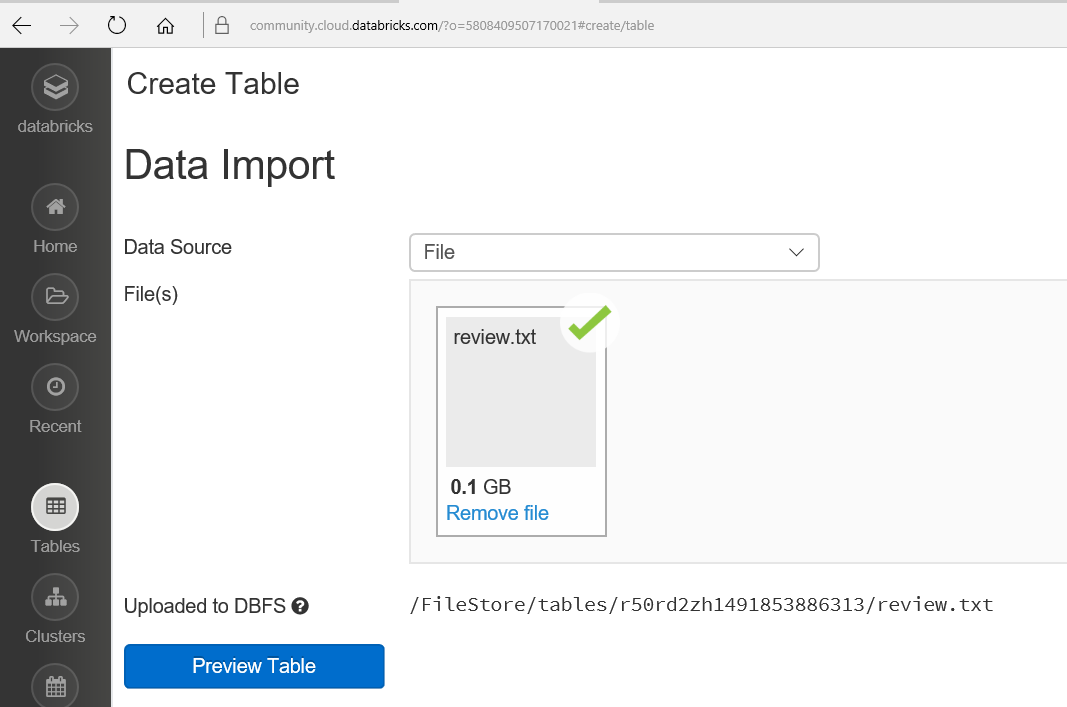
Fields Enclosed By '"' Terminated By '||' Escaped By '"'

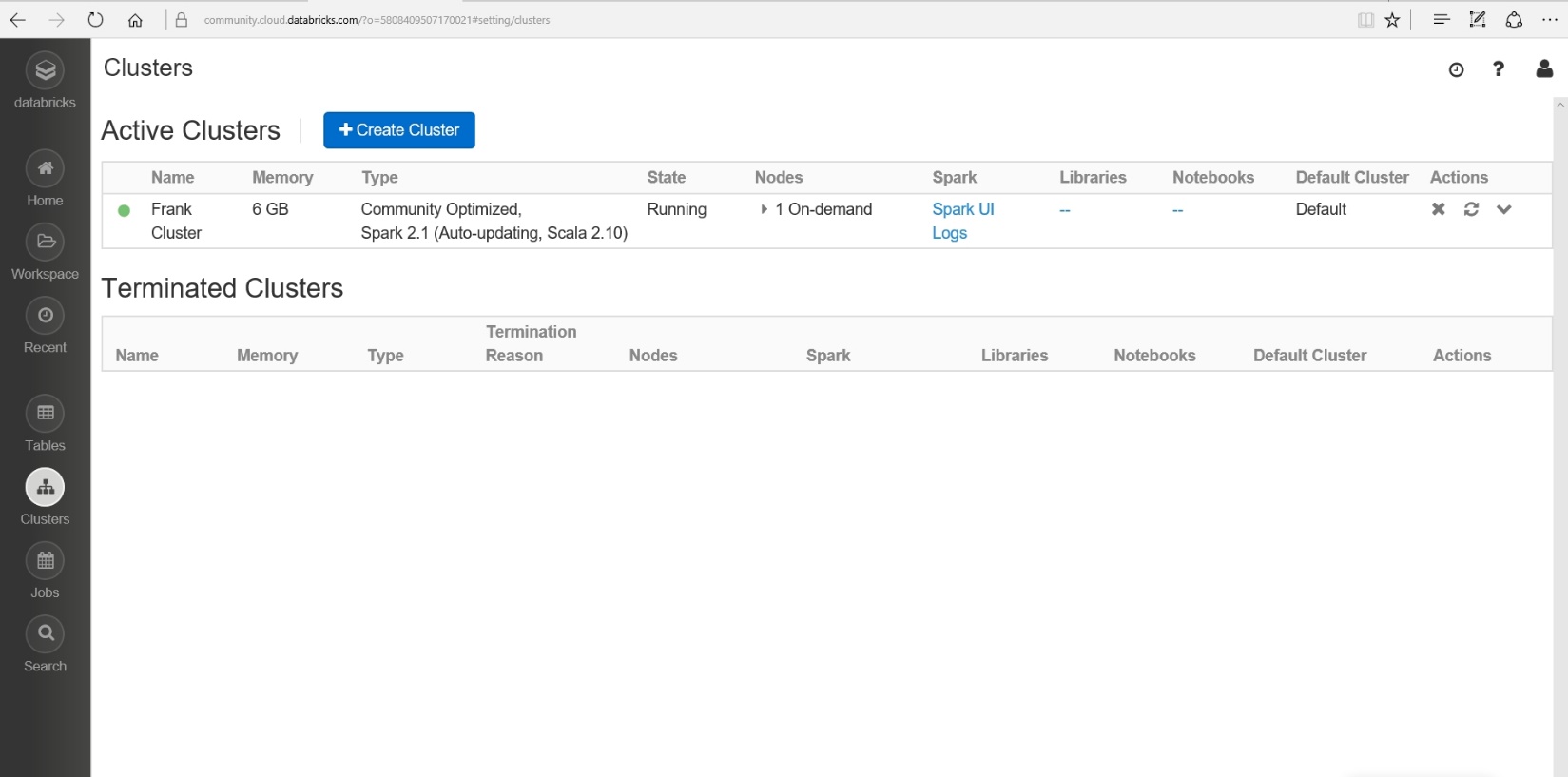
Lines Terminated By '\r\n');

I also extracted some of the Yelp data tables using the same command and specifying a comma as the delimiter. I did this because when I loaded the data into Databricks initially, I thought that you needed to create tables from the files and Databricks required a single character as a delimiter. Later, I learned that creating tables from the uploaded files was only necessary if you’re working with SparkSQL; since I did not work using SparkSQL, these tables were never actually used.

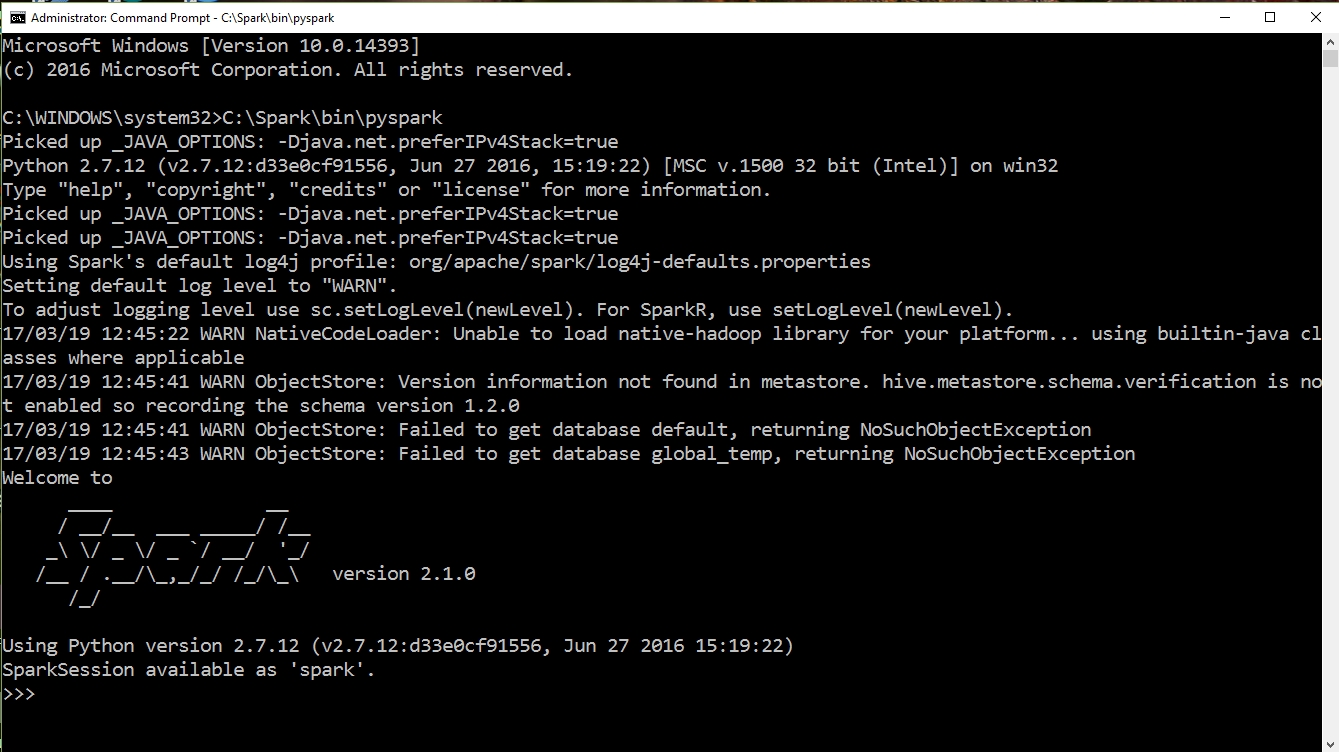
**Load Data**

Initially, I up-loaded the comma separated files to Databricks . However, after learning that it was not necessary to create tables after importing files into Databricks, I imported the double-pipe delimited files I exported from MySQL. These files were saved with the .txt file suffix. Importing files into Databricks is easy. It is accomplished using a GUI interface. Also, the uploads were relatively fast. Since I also installed Spark on my laptop I placed the same set of files I used for Databricks in a directory named data under my Spark directory on my laptop. It's important to note that before I could upload any data It was necessary to create a Spark cluster and notebook. This was a simple matter of making selections and clicking buttons. It was very straight forward. The screen shot below shows the what the Databricks user interface looked like after uploading review.txt. Each time I uploaded a file, I copied the file path to a file for reference.





Before I move on to the next section, I want to mention that I also installed Spark on my laptop computer. I am running Windows 10. It was not very straight forward or easy. It took a lot of time since the Documentation Spark provides, in my opinion, is not very friendly to a rookie programmer. I finally found a more detailed step-by-step guide that helped me. There are some warnings that appear when I run Spark, but so far, all the pySpark programs I have run work well.

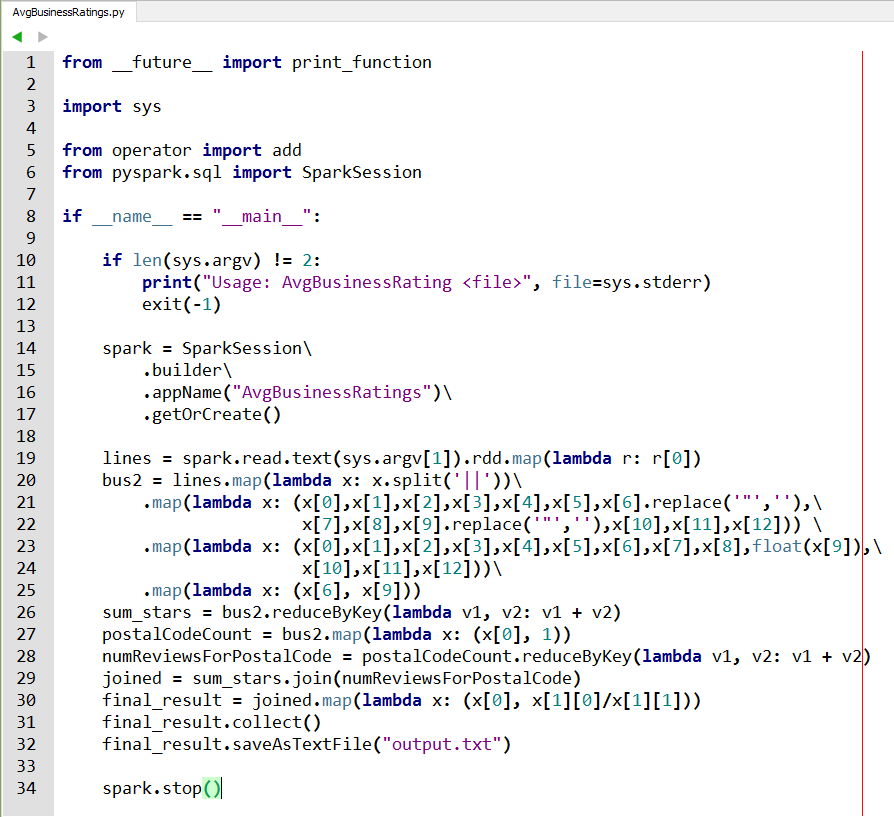


**Query**

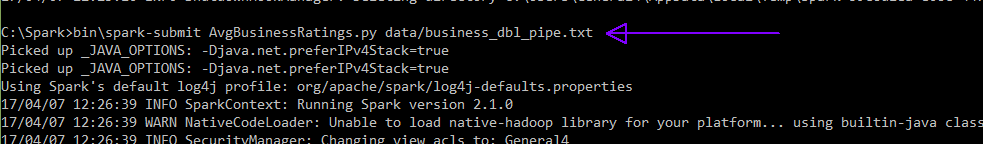
Since I am familiar with Python, I decided to use PySpark to solve my query. To begin, I followed Databricks tutorials and ran some of the transformation and action commands in the Databricks notebook to get a feel for Spark. I became better acquainted with some of the functional programming by running many, many examples. There was lots of trial and error involved during this learning process. I worked in both Databricks and on my local Spark installation. I started this process over Spring Break, so luckily, I had extra time to dedicate to school work. Finally, I felt more confident and attempted (successfully) to write a pySpark program to answer the query I used in my Hadoop MapReduce project.

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

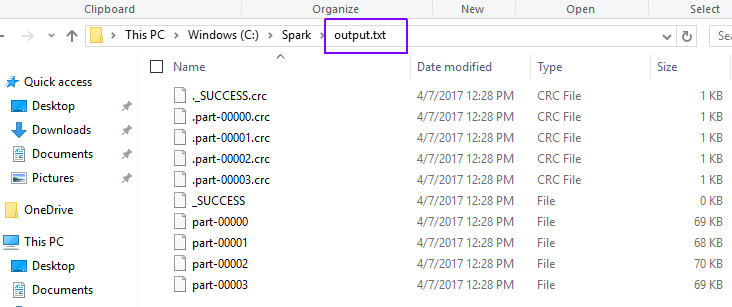
Below is a screenshot of the program I wrote to solve this problem/query.



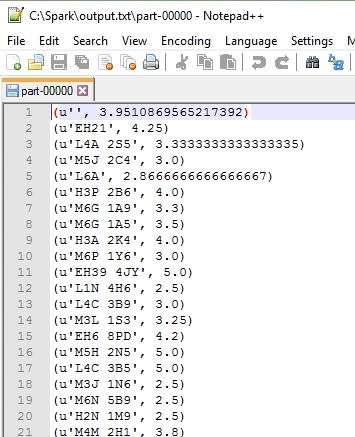
**Running the script on my local Spark installation –**



After running the command, Spark displays a lot of information in the command window. As you can see from the script, the final resulting RDD is saved in a directory named output.txt. The contents of this directory is shown below. The part files (part-00000 – part-00003) contain the output results.



The image below shows a very small sample of the resulting output data.

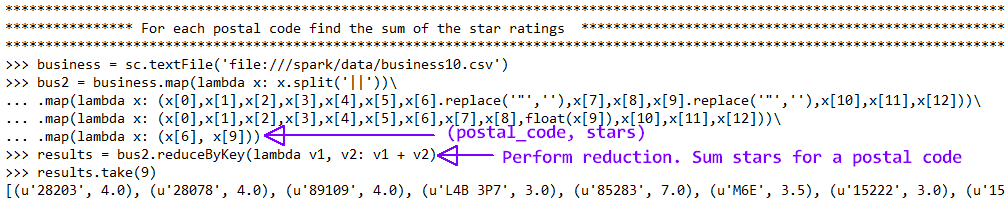


Now I will briefly explain the steps that I undertook to write the Python script shown above. Initially, I choose to work with a very small subset of the business table – only 10 records! When I successfully solved the problem, I would run it against a larger data file. I did this so I knew exactly what my data contained and could compare my expected results with the given results and know for sure whether there was a problem or not. I took baby steps and broke the larger problem into smaller problems.

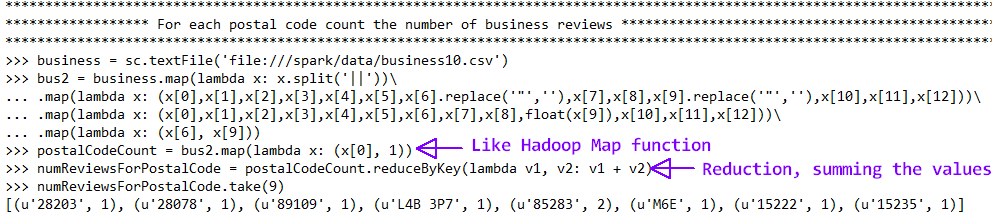
1. For each postal code find the sum of the star ratings
2. For each postal code count the number of business reviews/ratings
3. Join the results from the previous two steps using postal code as a key
4. Compute the average for each zip code

Below are images that show the ‘baby steps’ I undertook. I used the pySpark terminal to run the commands.

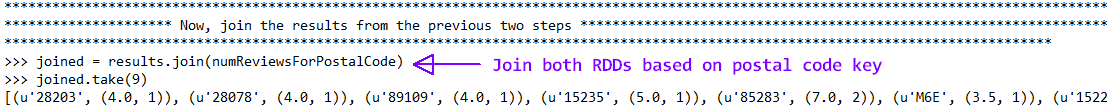
Step 1.



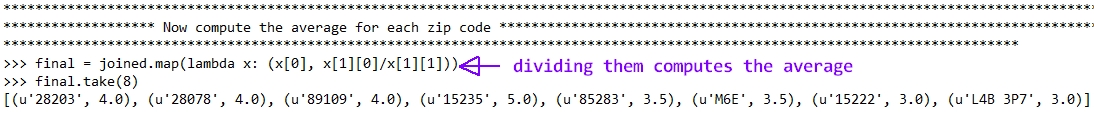
Step 2.



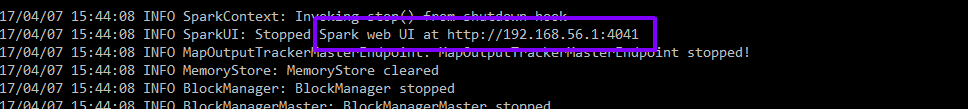
Step 3.



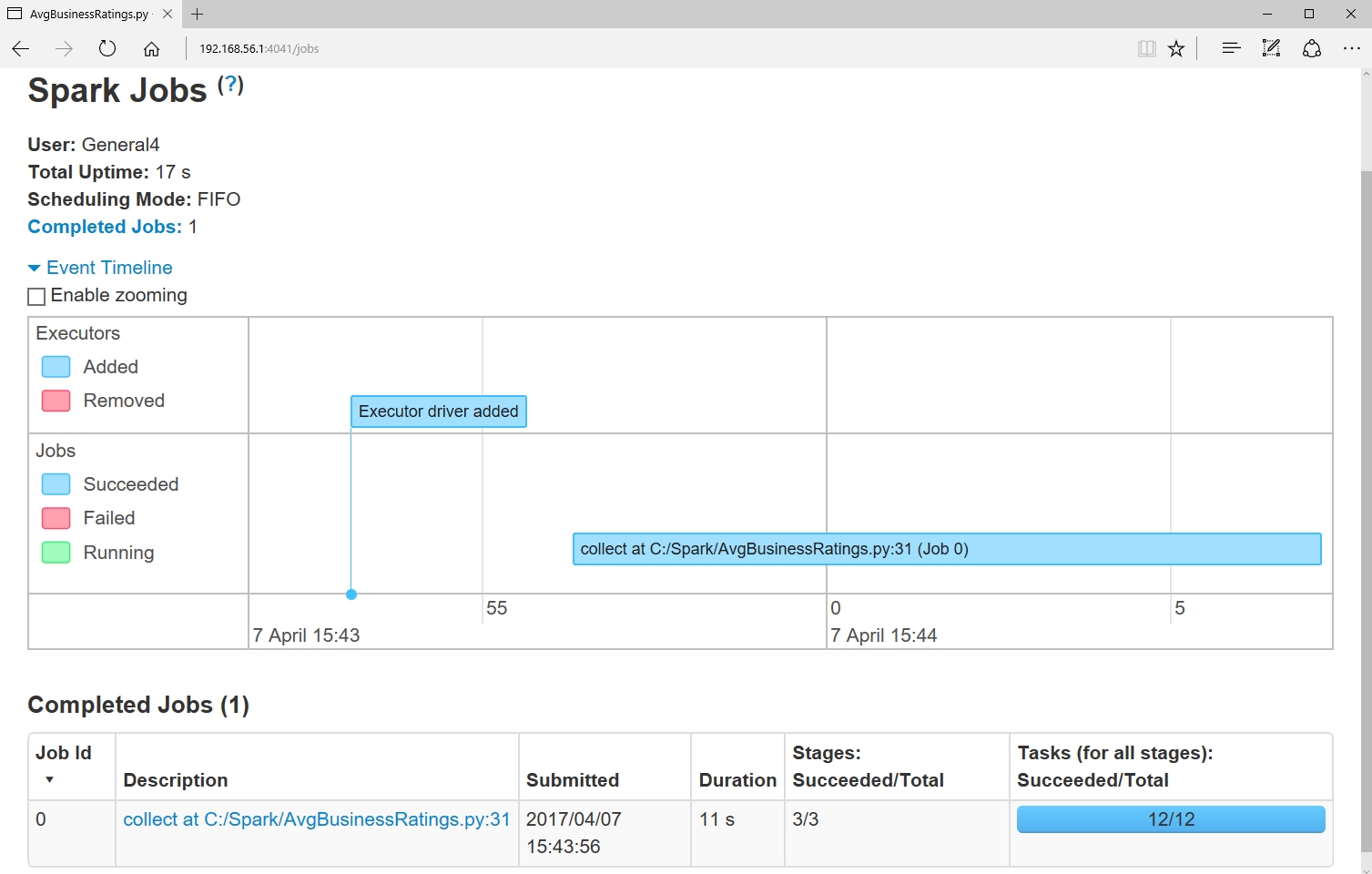
Step 4.



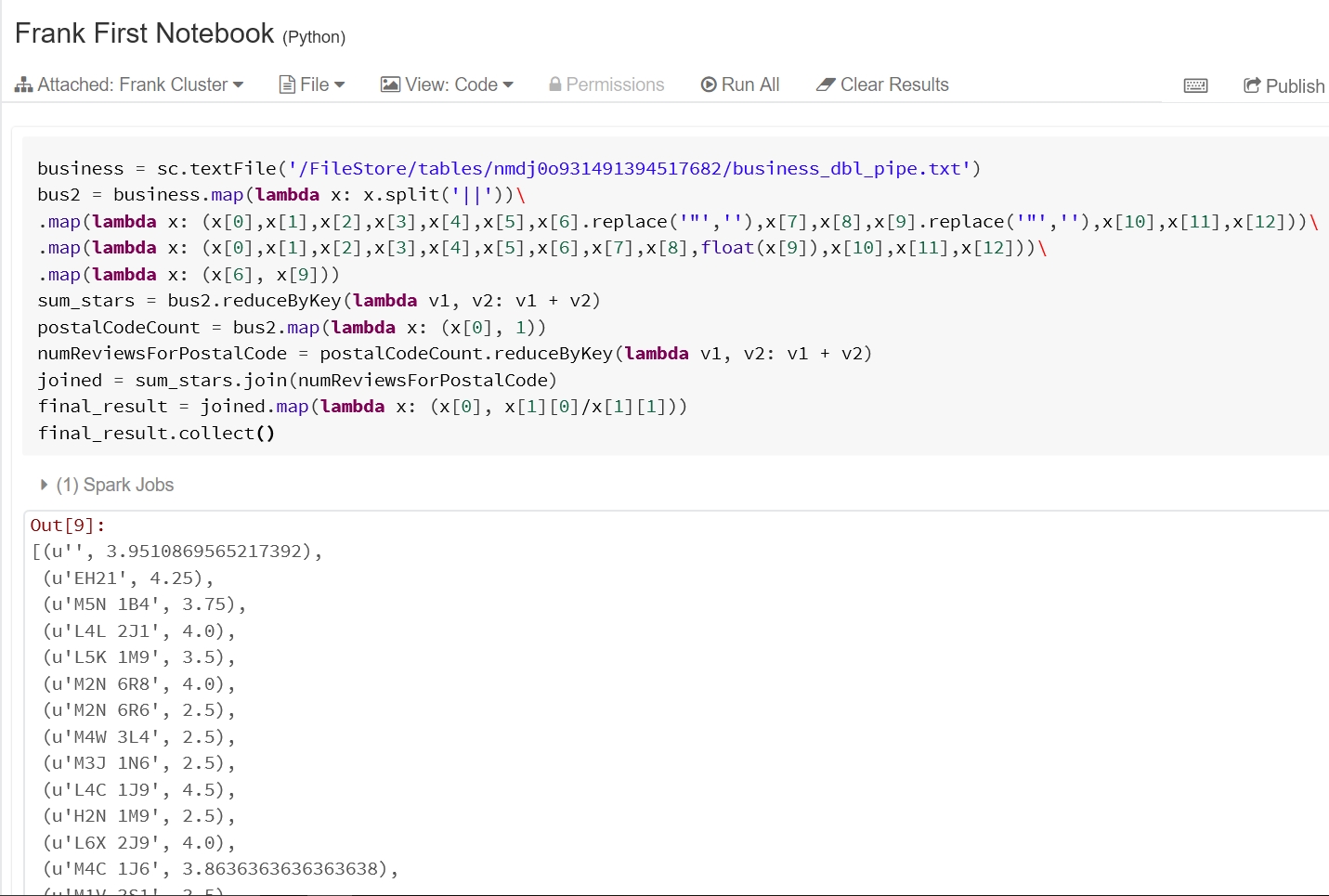
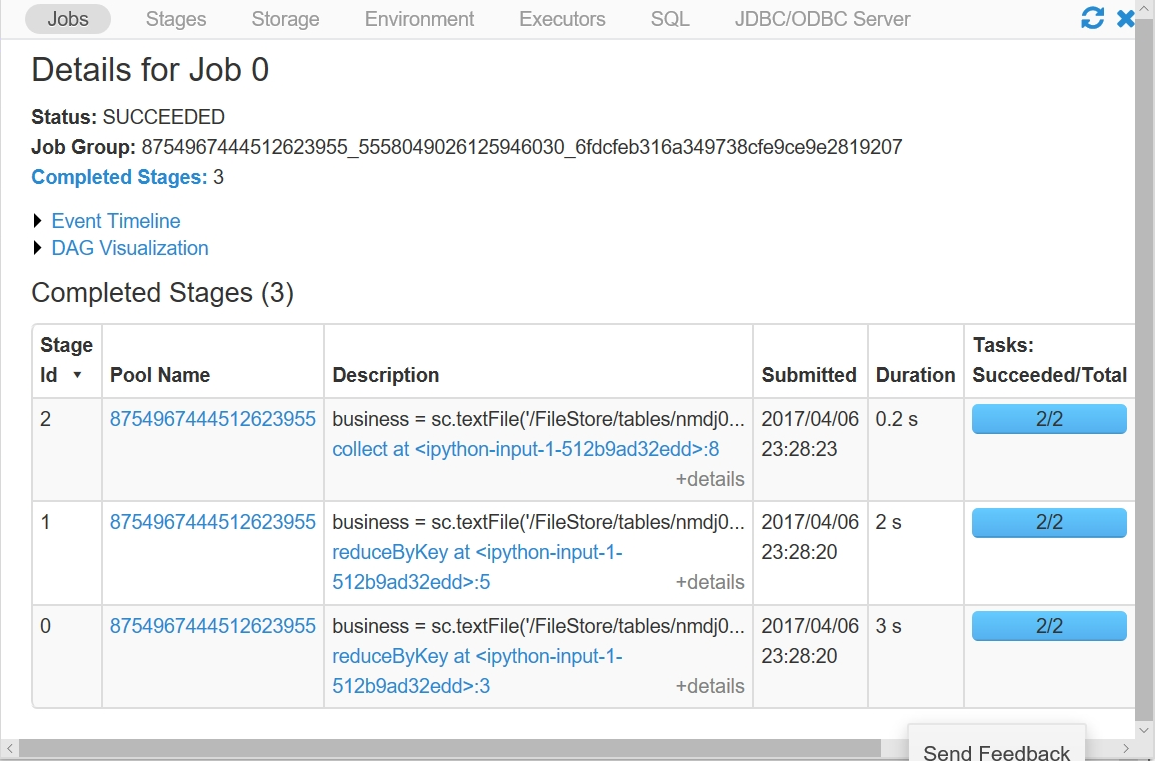
These steps combined are the main part of the python script I presented above. By looking at the output displayed in the terminal window, I determined the IP address of the Spark UI.



The Spark UI only stays up as long as the batch job is running, so it was necessary for me to refresh the page several times as the job ran to view different stages of the data. The screenshot below shows a job completed stage. Notice that the total time was 17 seconds and the total job duration took 11 seconds. I will talk more about this in the optimization section to come.



I also ran the script in my Databricks notebook. Just as on my local Spark instance, it was a success. The screenshot image below shows the code with some of the resulting output.



Notice that the total job duration was 5.2 seconds. This was about 50% faster than running it on my local machine. Another difference is the number of tasks. 12 tasks were run on my local Spark installation and only 2 tasks were run on Spark in Databricks.

**Update**

Spark batch processing is not meant to perform updates to the data. Since it reads historical data, any new data would normally be captured by spark streaming. If one absolutely needs to update the historical data, they would need to manually change the text file and the batch job would need to be run again. If just appending to the end of a file, and you know the line number of the last line of text, you could run the batch job and just process the new lines and combine them with the previous results. This assumes that you persist or cache the previous results. Spark has many options for persisting RDDs. One option is to use caching which will save the resulting RDD in memory assuming there is enough memory. The batch job processing the new appended data can then be combined with the cached data. As long as the driver program is running, the persisted data remains intact. To combine previously computed RDD results that ran under a different driver with the results of the updated data, you must use Checkpointing. Checkpointing saves data to a file and is an expensive operation. I will not go into the details of checkpointing but instead focus on caching. Caching is simple, just use the cache method: **aRDD.cache()**

Caching is useful for performing optimization in Spark. I will explain why caching is necessary in the Performance Optimization section below.

In addition to the cache method, Spark offers the **persist** method which has more options called storage levels. Some storage levels include persisting to DISK\_ONLY, MEMORY\_AND\_DISK, or MEMORY\_ONLY. The last storage level mentioned is the same as caching the data. The following is an example using the persist method.

**aRDD.persist(storageLevel.MEMORY\_AND\_DISK\_SER\_2)**

**Optimization of Quality**

Ensuring quality data can be accomplished by adding checks and filters in the code you write. For instance, after splitting a line of input using the split function, it would be a good idea to check that the length of the resulting list of objects the RDD contains, is the proper length before performing any work on it. Performing this check will filter out any bad lines of data and help to ensure more accurate data results. To accomplish this, the aptly named **filter** transformation function is used. Spark defines the filter function as such: ***filter(func)*** *- Return a new dataset formed by selecting those elements of the source on which func returns true.*

|  |  |
| --- | --- |
|  |  |

The following snippet of code is how I would use the filter function in my code to ensure quality data:

**...**

bus2 = business.filter(lambda x: len(x) > 0) \

.map(lambda x : x.split(‘||’)) \

.filter(lambda x: len(x) == 13) \

**...**

Since I know there are thirteen fields in the business data file I check that there are thirteen objects in a list after performing a split function. Performing type checking is another thing I could do to ensure data quality. Since I knew I was working with ‘clean’ data, I did not do any type checking before I casted the stars value to a float. However, in a real-world program I would want to do this. The following is an example of how I would do this in my program.

# The RDD is named bus2

**...**

**.**map**(lambda** x**:** **(**x**[**6**],** x**[**9**]))** **<--** second element must be a float

**try:**

afloat **=** float**(**bus2**[**1**]))**

**except** ValueError**:**

# Handle exception

**...**

If the second element of the RDD, bus2, fails the cast, I would not consider that data and move on to the next object list of the RDD. Ensuring that the input data is of the expected form and performing type checking will go a long way in ensuring that the data quality is optimized.

**Optimization of Performance**

In Databricks, there does not appear to be a way to tweak the settings to try and improve performace. This is because we are using a free, Community Edition of Databricks. My focus will be on improving performance of my local Spark installation. As mentioned briefly in the Update section above, caching is one way to perform performance optimization. Whenever an action function is called on an RDD, all previous transformations that were performed to create the RDD, must be performed again. To avoid this redundant work of re-computing an RDD for each action, Spark allows you to cache the RDD. As mentioned above, the Spark API provides the cache function to perform caching.

**aRDD.cache()**

At the very end of my program, I used the collect function which is an action. It returns a list that contains all the elements of an RDD to the driver program. Since my program did not use an action function until the very end, I did not cache any of the RDDs. However, to demonstrate when to use caching, I wrote a small program. I ran the program with and without caching and compared the results. To get a better idea of the power of caching, check out the snipet of code that follows. It caches the RDD so that after the first action function is called, additional action functions called on the RDD will not require the RDD to be re-computed. This will allow the data to be accessed faster.

#numbers is an RDD

...

numbers**.**cache**()** # cache the RDD

numbers**.**count**()** # triggers RDD computation

numbers**.**min**()** # No RDD re-computation required

numbers**.**max**()** # No RDD re-computation required

...

The following code captures the stars values and runs actions on them with and without caching. I compared the results on both my machine and Databricks, for each case the cached version performed better.

data **=** sc**.**textFile**(**'/FileStore/tables/nmdj0o931491394517682/business\_dbl\_pipe.txt'**)**

# Extract only the star values and turn into floats

stars **=** data**.**map**(lambda** x**:** x**.**split**(**'||'**))** \

**.**map**(lambda** x**:** **(**x**[**9**].**replace**(**'"'**,**''**)))** \

**.**map**(lambda** x**:** **(**float**(**x**[**0**])))**

#stars.cache()

stars**.**count**()** # Actions triggers RDD re-computation

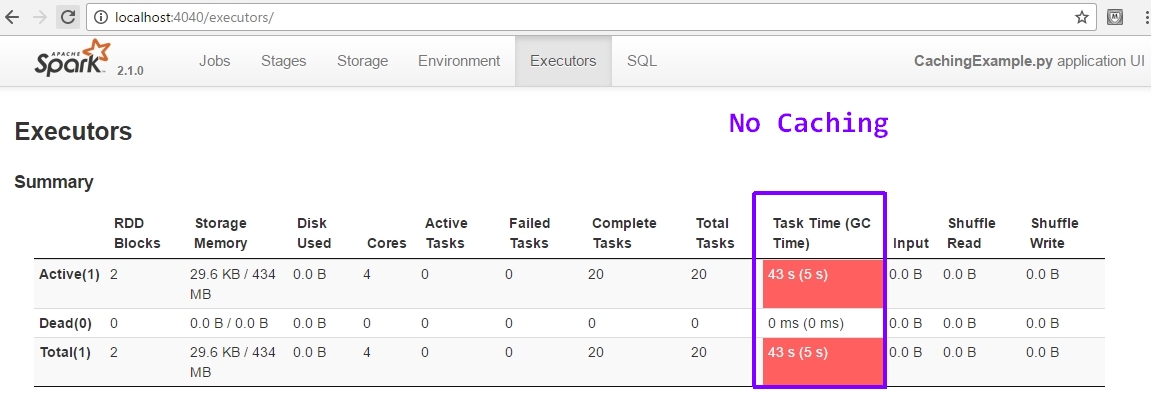
stars**.**min**()**

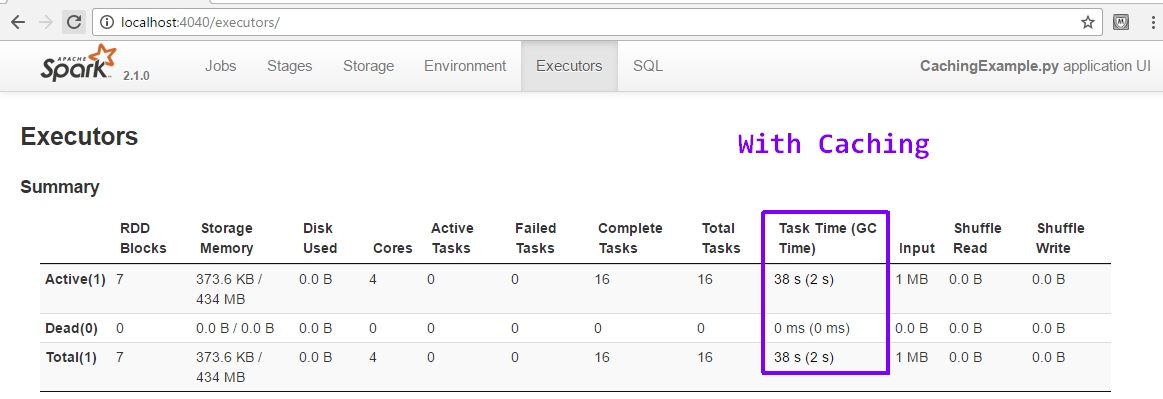
stars**.**max**()**

stars**.**stats**()**

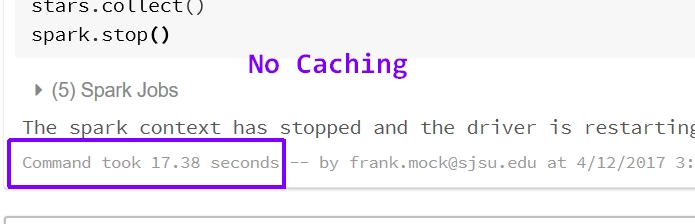
stars**.**collect**()**

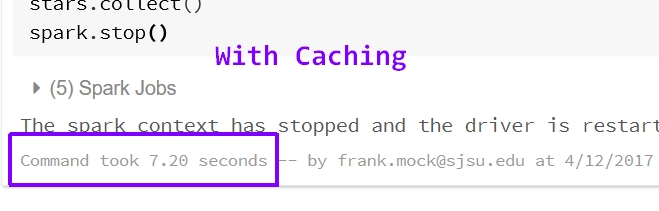
spark**.**stop**()**



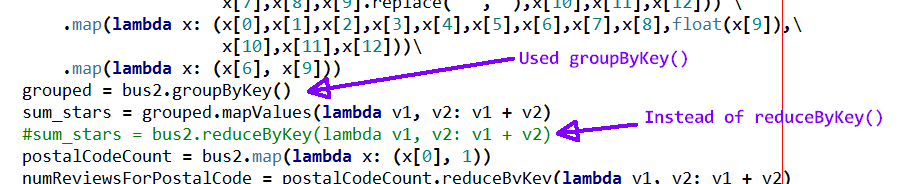


The code was much faster on Databricks, but I got the same results – caching was faster.

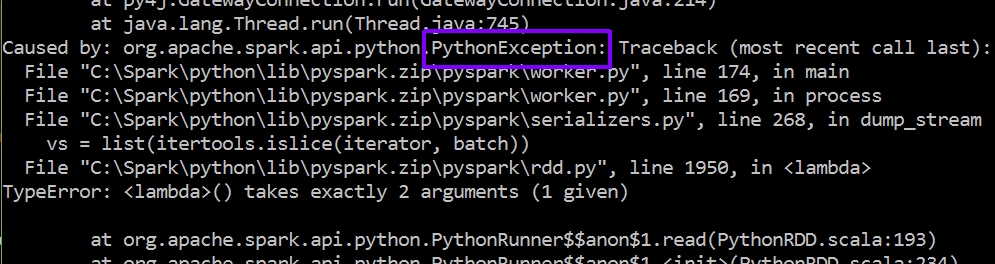




Another way to improve performance in Spark is to ensure you are using the correct reduction transformation function. When performing reduction for the purpose of aggregation operations, it’s better to use the **reduceByKey** or **foldByKey** functions versus the **groupByKey** function. This is because, groupByKey will cause more data to be transferred when the data is shuffled. In contrast, reduceByKey, combines the results of the aggregation operations before the shuffle which results in better performance. I changed my code to include a groupByKey function in place of one of the reduceByKey functions. I wanted to compare the execution time of each to see how much groupByKey would slow down the execution time.

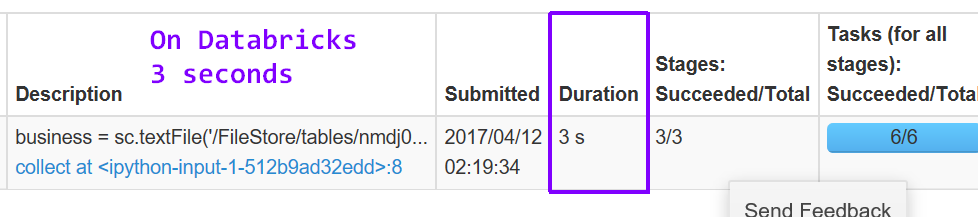


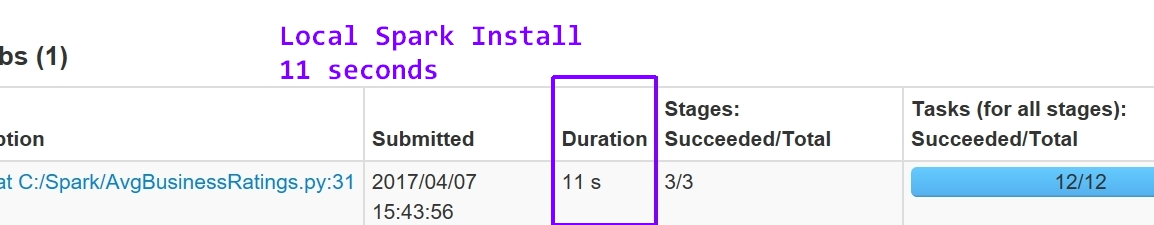
Unfortunately, I continually kept getting errors. The error messages did not directly reference my code as the source of the problem, so I was unsure what I was doing wrong.



At least I learned from reading tutorials and Spark documentation that it is much better to use reduceByKey versus groupByKey to reduce the amount of data being transferred during shuffle operations. I wish I could have seen how using groupBykey would have affected by code, but due to the amount of time this Spark project is taking I had to move on to other school work.

One final thing I would like to mention with regards to performance optimization, is that I noticed when my scripts ran on Databricks they ran faster than on my local machine. They were working with the same data so I found this fact interesting. I tried to determine why they had different execution times, but could not determine why. I can only guess that Databricks is optimized in some way or has more dedicated memory. On my machine the number of tasks was double than that of Databricks. Perhaps this is a clue to why it was slower. Maybe, the increased number of tasks caused more shuffling which resulted in increased job time.





**Accumulator Example**

An accumulator is an updateable, shared variable that supports parallel processing. A broadcast variable is a read-only shared variable. Both accumulators and broadcast variables are made available to all worker nodes in a cluster. I wrote a very simple example program to demonstrate the use of an accumulator and a broadcast variable. To reduce the amount of filtering and data checking code (review data is dirty), I worked with a very small subset of cleaned review data. It finds the number of reviews that contain the words ‘good’ or ‘great’.

# Accumulator and Broadcast Variable Example

**from** \_\_future\_\_ **import** print\_function

**import** sys

**from** operator **import** add

**from** pyspark**.**sql **import** SparkSession

**from** pyspark **import** SparkContext

sc **=** SparkContext**()**

**if** \_\_name\_\_ **==** "\_\_main\_\_"**:**

**if** len**(**sys**.**argv**)** **!=** 2**:**

**print(**"Usage: AccumulatorExample <file>"**,** file**=**sys**.**stderr**)**

exit**(-**1**)**

spark **=** SparkSession\

**.**builder\

**.**appName**(**"AccumulatorExample"**)**\

**.**getOrCreate**()**

# The words we are looking for

words **=** **[**"good"**,** "great"**]**

# Create a broadcast variable named target\_words

target\_words **=** sc**.**broadcast**(**words**)**

# Initialize accumulator

word\_count **=** sc**.**accumulator**(**0**)**

review\_data **=** spark**.**read**.**text**(**sys**.**argv**[**1**]).**rdd**.**map**(lambda** r**:** r**[**0**])**

review\_words **=** review\_data**.**map**(lambda** x**:** x**.**split**(**"||"**))** \

**.**map**(lambda** x**:** **(**x**[**5**].**replace**(**'"'**,**''**)))** \

**.**flatMap**(lambda** line**:** line**.**split**())** \

**.**map**(lambda** x**:** x**.**lower**())** \

**.**filter**(lambda** x**:** x **in** target\_words**.**value**)**

# Define a function to increment accumulator

**def** inc\_accumulator**(**word**,** word\_count**):**

word\_count **+=** 1

review\_words**.**foreach**(lambda** x**:** inc\_accumulator**(**x**,** word\_count**))**

# Display results

**print(**"Total reviews with good or great is: " **+** str**(**word\_count**.**value**))**

#review\_words.collect()

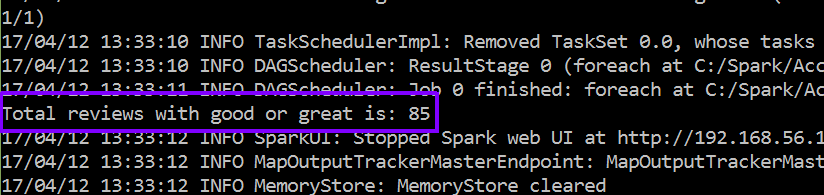
#review\_words.saveAsTextFile("review\_words")

spark**.**stop**()**

I used spark-submit to run the script.



The results of the script



I know this result is correct because I also printed to file and the line count is 85 

**Spark Streaming**

**Extract Data**

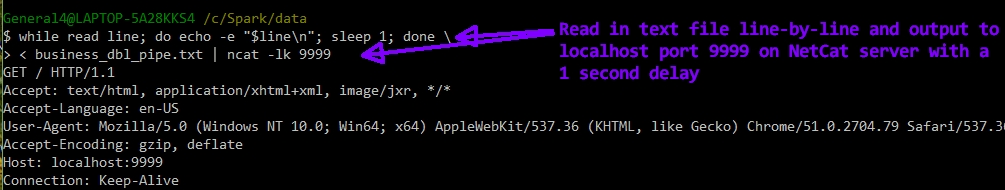
I did not need to extract any additional data to work with Spark Streaming. My existing data was used as an input stream. How I accomplished this is explained in the next section.

**Load Data**

Spark Streaming requires a DStream Source that provides data that updates or continually changes., I chose to use the **socketTextStream** method from the Streaming API to create a DStream from an input TCP source. To provide a streaming input source, I used a simple shell script that reads the lines of a Yelp data file, business\_dbl\_pipe.txt, and every second sends a line to port 9999 of localhost on a NetCat server. The one second interval is the input data’s velocity. Port 9999 will provide input for my Spark Streaming script. This stream of data can be thought of as new user input data being captured from a website.

$ while read line; do echo -e "$line\n"; sleep 1; done \

> < business\_dbl\_pipe.txt | ncat -lk 9999





When doing batch processing with Spark, you do not need to instantiate a SparkContext since it is done automatically for you. This is not true with SparkStreaming; you must explicitly create a StreamingContext. The following lines of code show how my script instantiates a StreamingContext and creates a connection to the data stream. Notice that the StreamingContext constructor has a numeric batch duration argument, which is the number of seconds that streaming data will be split into batches.

**ssc = StreamingContext(sc, 10)** # Instantiate a StreamingContext – uses an existing SparkContex, sc

**lines = ssc.socketTextStream('localhost', 9999)** # Connect to the data stream

**Query**

For Spark Streaming, I used the same query as Spark Batch.

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

**import** re

**from** pyspark**.**streaming **import** StreamingContext

ssc **=** StreamingContext**(**sc**,** 10**)**

lines **=** ssc**.**socketTextStream**(**'localhost'**,** 9999**)**

bus2 **=** lines**.**filter**(lambda** line**:** len**(**line**)** **>** 0**)** \

**.**map**(lambda** x**:** x**.**split**(**'||'**))** \

**.**map**(lambda** x**:** **(**x**[**0**],**x**[**1**],**x**[**2**],**x**[**3**],**x**[**4**],**x**[**5**],**x**[**6**].**replace**(**'"'**,**''**),**x**[**7**],**x**[**8**],** \

x**[**9**].**replace**(**'"'**,**''**),**x**[**10**],**x**[**11**],**x**[**12**]))** \

**.**map**(lambda** x**:** **(**x**[**0**],**x**[**1**],**x**[**2**],**x**[**3**],**x**[**4**],**x**[**5**],**x**[**6**],**x**[**7**],**x**[**8**],**float**(**x**[**9**]),**x**[**10**],**x**[**11**],**x**[**12**]))** \

**.**map**(lambda** x**:** **(**x**[**6**],** x**[**9**]))**

postalCodeCount **=** bus2**.**map**(lambda** x**:** **(**x**[**0**],** 1**))**

numReviewsForPostalCode **=** postalCodeCount**.**reduceByKey**(lambda** v1**,** v2**:** v1 **+** v2**)**

sum\_stars **=** bus2**.**reduceByKey**(lambda** v1**,** v2**:** v1 **+** v2**)**

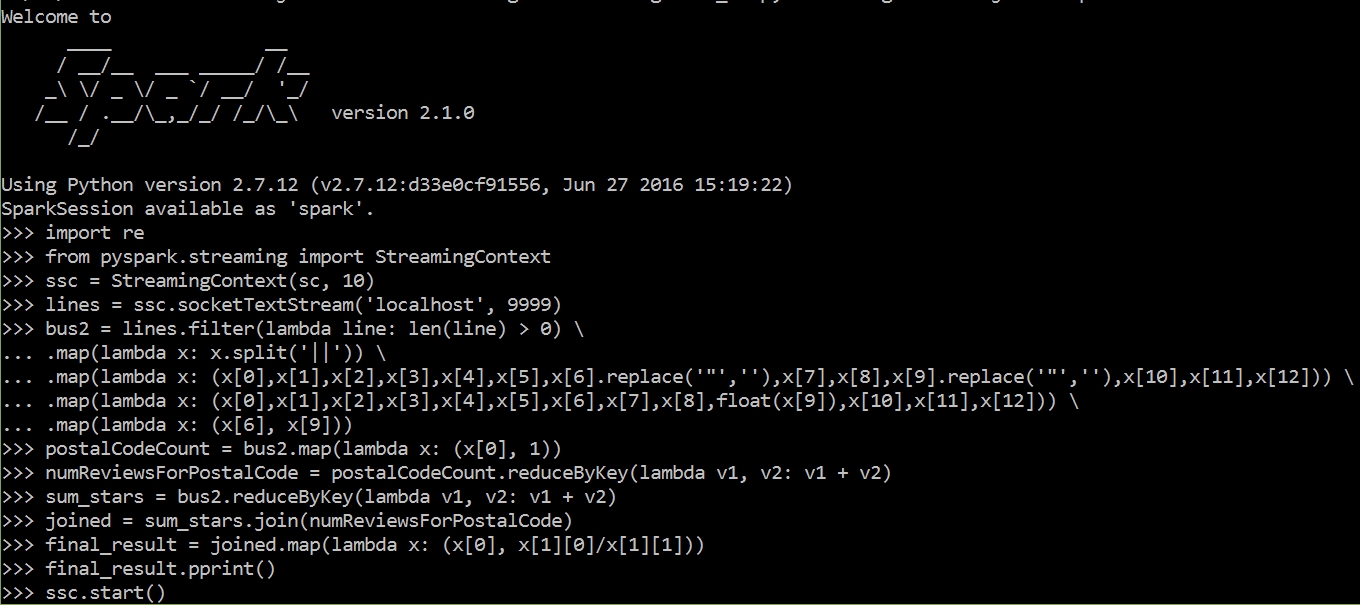
joined **=** sum\_stars**.**join**(**numReviewsForPostalCode**)**

final\_result **=** joined**.**map**(lambda** x**:** **(**x**[**0**],** x**[**1**][**0**]/**x**[**1**][**1**]))**

final\_result**.**pprint**()**

ssc**.**start**()**

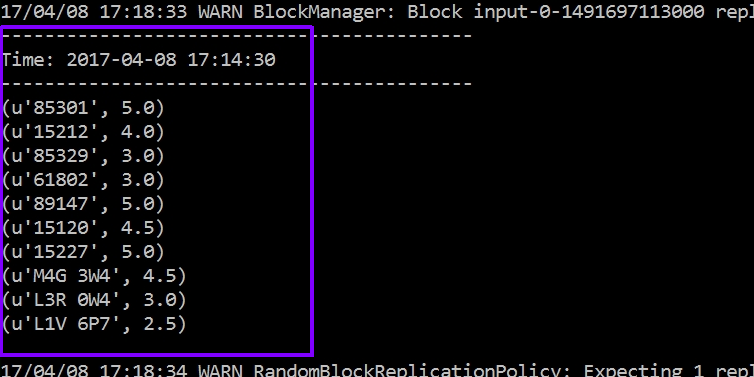
Notice that my solution is almost identical to my batch script. The only difference is the code explicitly creates a StreamingContext and makes a connection to a TCP socket as an input data stream. This is a stateless Spark Streaming solution to my problem query. This means that each batch of a batch interval is processed independently of any other batches with a stream. To make a state DStream, I would use the **updateStateByKey( aFunction )** transformation method. This method returns a new DStream where the state for each key is updated by applying the function that is specified as an argument.

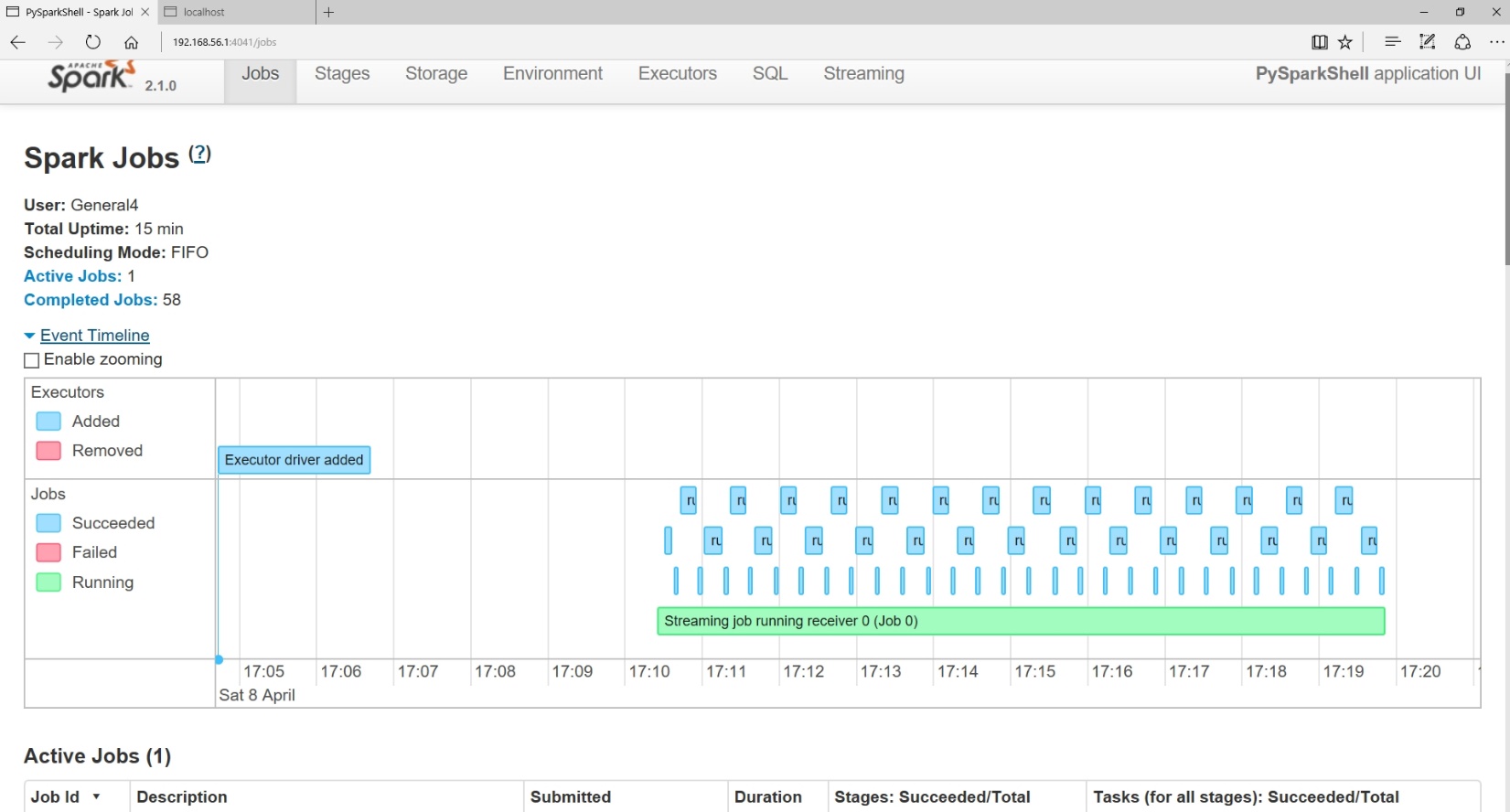


I ran my streaming script from my local Spark installation and not on Databricks. When using local mode, you must specify at least two worker threads. That is what the 2 means in the following command.

**C:\Spark>pyspark --master local[2]**

After entering **ssc.start( )** and pressing enter, I started the flow of input data by opening another command window and entering the script that I mentioned above in the Load Data section. With this data streaming, my script captures lines and performs work on them. The next image is just one of many captured results. As Spark runs the script, the output is very verbose.





**Update**

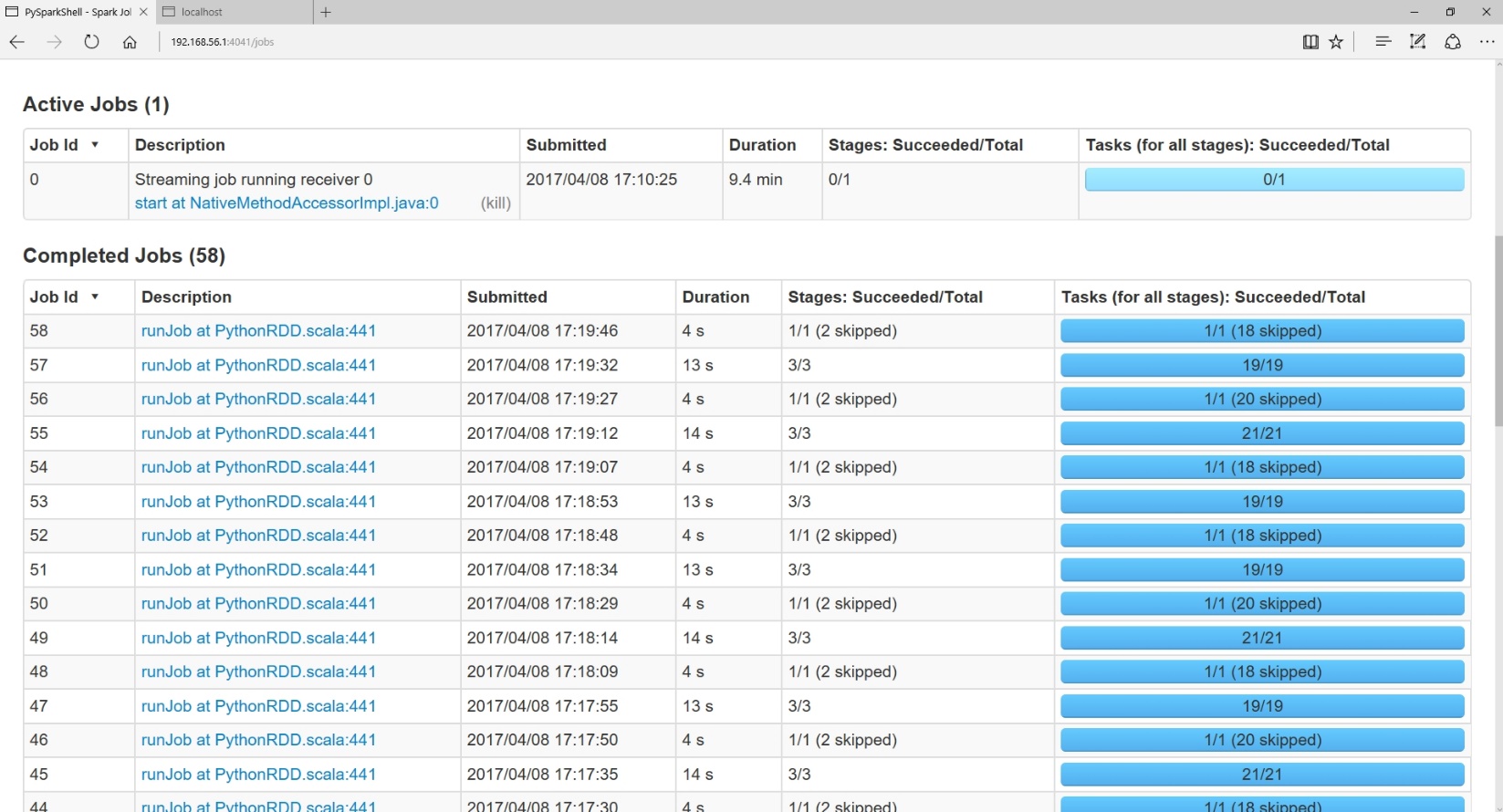
The same update information that I mentioned above for Spark batch processing applys for Spark Streaming as well.

**Optimize Quality**

The same quality optimization information that I mentioned above for Spark batch processing applys for Spark Streaming as well.

**Optimize Performance**

As mentioned above, using reduceByKey instead of groupByKey to reduce the amount of data transferred during shuffling applies with Spark streaming as well. I will try to finish this section better on the next assignment, which is all about Spark Streaming. Below is an image from one of the Streaming jobs I ran.



**Summary**

I want to briefly mention how cool I think Spark is. To solve the same problem, my Hadoop MapReduce code took many more lines of code. Additionally, I really enjoy that I can use Python with Spark. It will take a bit more time and practice working with Spark, but I think this is a technology I would love to work with after graduation this Spring. When I took CS152, I saw the power of functional programming but really did not see how it would be used in the real world. Spark showed me how.