**Owen Galvin**

**HU Extension Assignment 06 E63 Big Data Analytics**

Issued on: March 05, 2016 Due by 11:30PM EST, March 11, 2016

Please, describe every step of your work and present all intermediate and final results in a Word document. Please, copy past text version of all essential command and snippets of results into the Word document. We cannot retype text that is in JPG images. Please, always submit a copy of original, working scripts and/or class files you used as separate files. Sometimes we need to run your code and retyping is too costly. Please, submit to the class drop box. For issues and comments visit the class Discussion Board. You can solve these problems using any language of your choice.

**Problem 1.** Go to an online newspaper and select paragraphs from two articles in a similar field, about politics, art, movies, or any other topic of your choice. Let those paragraphs be moderately small, a few lines and around 100 words. Save those paragraphs as .txt files and then import them into two Spark RDD objects, paragraphA and paragraphB. Use Spark transformation functions to transform those initial RDD-s into RDD-s that contain only words. List for us the first 10 words in each RDD. Subsequently create RDD-s that contain only unique words in each of paragraphs. Then create an RDD that contains only words that are present in paragraphA but not in paragraphB. Finally create an RDD that contains only the words common to two paragraphs.

**Solution:**

**I took text from two online stories that reviewed the new movie Zootopia, and created new text files on the quickstart VM, named as below:**

1. **L06\_paraA.txt**
   1. [**www.nytimes.com/2016/03/04/movies/zootopia-review.html**](http://www.nytimes.com/2016/03/04/movies/zootopia-review.html)
2. **L06\_paraB.txt**
   1. [**http://www.hollywoodreporter.com/review/zootopia-film-review-863843**](http://www.hollywoodreporter.com/review/zootopia-film-review-863843)

**Before actually creating those two files I made a directory structure that looks like below, with the .py files involved in each of the problems residing in the root of Lec06. I’ve provided cmd text to create the directories at the end but I actually created the folders using the File Browser because it was easier. Either way the .txt files wind up in a lec06/data directory and below shows the overall tree structure.**

|  |
| --- |
|  |

**Reviewing the python code (p01.py) for the solution, begin by noting that the template code was taken from my python solutions to Lecture 4. At the beginning we have some items related to setup. Import re for later regex involved in cleaning words, os and shutil for dealing with items on the file level, followed of course by importing the modules necessary for using Spark in python. The paragraph files are quite small, no reason to work with HDFS, so I pull in the two paragraph files from my local /data directory. Note that at the same folder level there will be an /out directory holding the output file I later create using Python commands. Create variables related to all those folder paths. Next is a re\_clean variable that holds the regex expression that will be used to remove extraneous characters (basically anything not a letter or digit). It is compiled to save time during loops (Assignment 4), though here the files are so small it wouldn’t make any difference. Then I create the /out directory if it doesn’t already exist.**

|  |
| --- |
| import re, os, shutil  from pyspark import SparkConf, SparkContext  spark\_root\_dir = *'file:///home/cloudera/proj/lec06'*  file\_root\_dir = *'/home/cloudera/proj/lec06'*  output\_dir = file\_root\_dir + *'/out'*  re\_clean = re.compile(*r'[^A-Za-z0-9 ]'*)  if not os.path.exists(output\_dir):  os.mkdir(output\_dir) |

**Next is a very small function that takes in each word and removes non-word characters using the compiled regex expression. Then it converts the word to all lower-case characters, which I think makes sense in this scenario.**

|  |
| --- |
| def **clean**(word):  return re.sub(re\_clean, *''*, word).lower() |

**The final setup-type code, create the Spark configuration and context variables, followed by using the textFile method on spark context to load contents of each text files as a series of lines (though everything is all in one paragraph in each text file, i.e. only a single line)**

|  |
| --- |
| conf = SparkConf().setMaster(*'local'*).setAppName(*'py01'*)  sc = SparkContext(conf = conf)  linesA = sc.textFile(spark\_root\_dir + *'/data/L06\_paraA.txt'*)  linesB = sc.textFile(spark\_root\_dir + *'/data/L06\_paraB.txt'*) |

**The first part of task 2 is below, which involves splitting the line in text file using example syntax first found in Lecture 4, calling flatMap to return a single list of words. Then run an additional map on those in order to run my clean() function on each to clarify the text to words-only and lower-case the characters. I could have put the splitting & cleaning into a single function but this works fine as-is.**

|  |
| --- |
| #2) Use Spark transformation functions to transform those initial RDD-s into RDD-s that contain only words.  paragraphA = linesA.flatMap(lambda line: line.split(*' '*)).map(lambda word: clean(word))  paragraphB = linesB.flatMap(lambda line: line.split(*' '*)).map(lambda word: clean(word)) |

**To ‘uniquify’ the contents of each RDD produced above, simply use distinct() function on each, which effectively results in an RDD ‘set’ object even though this is not an actual type of RDD.**

|  |
| --- |
| #3) Subsequently create RDD-s that contain only unique words in each of paragraphs.  uniqueA = paragraphA.distinct()  uniqueB = paragraphB.distinct() |

**Now I can use the set-like RDD functions to answer final tasks. First call subtract() to remove from uniqueA RDD any words that are also present in uniqueB. I didn’t think it really made sense to use the original ‘paragraph’ RDDs since those contained intra-RDD duplicate values.**

|  |
| --- |
| #4) Then create an RDD that contains only words that are present in paragraphA but not in paragraphB.  wordsInANotB = uniqueA.subtract(uniqueB) |

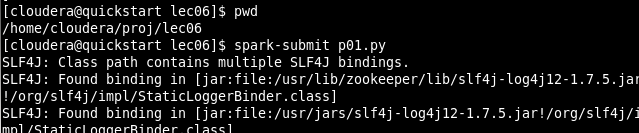
**Final task uses the intersection() function to find words that are common to both ‘unique’ RDDs.**

|  |
| --- |
| #5) Finally create an RDD that contains only the words common to two paragraphs.  wordsInAAndAlsoInB = uniqueA.intersection(uniqueB) |

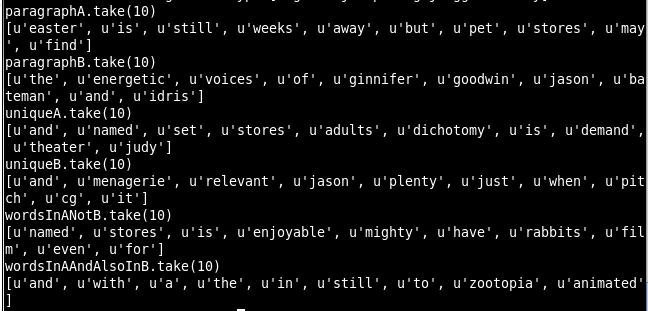
**Now that the actual RDD work is basically done, want to write everything to file, which I’ll include in the upload since even with all data it is a small file. In each case we are dealing with an RDD and so need to take an Action of some kind to create writable output. For the first task take() works since that is what the actual task requires, passing in 10 for number of words to pull out. Then collect() is used on the remaining RDDs since we want to pull everything out and we don’t need to worry about memory issues with such small sample text. Finally, for screen output I just use take(10) to print out 10 elements of each RDD.**

|  |
| --- |
| with open(output\_dir + *'/Problem01\_output.txt'*, *'w'*) as f:  #a. List for us the first 10 words in each RDD.  f.write(*'#ten from paragraphA\n'*)  f.writelines([str(word) + *'\n'* for word in paragraphA.take(10)])  f.write(*'\n#ten from paragraphB\n'*)  f.writelines([str(word) + *'\n'* for word in paragraphB.take(10)])  f.write(*'\n#unique in paragraphA\n'*)  f.writelines([str(word) + *'\n'* for word in uniqueA.collect()])  f.write(*'\n#unique in paragraphB\n'*)  f.writelines([str(word) + *'\n'* for word in uniqueB.collect()])  f.write(*'\n#words in paragraphB but not paragraphA\n'*)  f.writelines([str(word) + *'\n'* for word in wordsInANotB.collect()])  f.write(*'\n#words in paragraphA and also in paragraphB\n'*)  f.writelines([str(word) + *'\n'* for word in wordsInAAndAlsoInB.collect()])  print *'paragraphA.take(10)'*  print paragraphA.take(10)  print *'paragraphB.take(10)'*  print paragraphB.take(10)  print *'uniqueA.take(10)'*  print uniqueA.take(10)  print *'uniqueB.take(10)'*  print uniqueB.take(10)  print *'wordsInANotB.take(10)'*  print wordsInANotB.take(10)  print *'wordsInAAndAlsoInB.take(10)'*  print wordsInAAndAlsoInB.take(10) |

**Now I simply submit my python p01.py file to spark via spark-submit.**



**…**



**Terminal commands.**

|  |
| --- |
| *#PROBLEM 01*  *#starting from cloudera homedir, below to make directories I use in examples*  *mkdir proj*  *cd proj*  *mkdir lec06*  *#this is where the p01.py lives, in lec06*  *cd lec06*  *#below is the directory containing 1) L06\_paraA.txt & 1) L06\_paraB.txt*  *mkdir data*  *pwd*  *spark-submit p01.py* |

**Problem 2**. Consider attached file emps.txt. It contains: name, age and salary of three employees. Create RDD emps by importing that file into Spark. Next create a new RDD emps\_fields by transforming the content of every line in RDD emps into a tuple with three individual elements by splitting the lines on commas. Now comes something new. Spark has a class Row and you need to import it in your script or program. Row comes from the same package as class SQLContext. Row class creates rows with named and typed fields. You need to apply “constructor” Row to every tuple in RDD emps\_fields, like:

employees = emps\_fields.map(lambda e: Row(name = e[0], age = int(e[1]), salary = float(e[2])))

e[0], e[1] and e[2] are the first, second and third elements of the tuple e representing a row (line) in RDD emps\_fields. Note that int and float are types of fields in new rows. Newly create RDD employees is now made of Row elements and is ready to be transformed into a DataFrame. You generate a DataFrame by passing an RDD of Row elements to the method createDataFrame() of class SQLContext. Do it. Show the content of new DataFrame. Transform this DataFrame into a Temporary Table and select names of all employees who have a salary greater than 3500.

**Solution:**

**Again, it makes sense to start off with a code review of my p02.py, after noting that the new file is also in /lec06 directory and the downloaded emps.txt file is in /lec06/data folder. The top section of p02.py is very similar to P01 solution, so I’ll just highlight the changes, which include importing SQLContext & Row so that I can access those objects. The other new item is the creation of the SQLContext object, taking the existing SparkContext in its constructor.**

|  |
| --- |
| import os  from pyspark import SparkConf, SparkContext, SQLContext, Row  spark\_root\_dir = *'file:///home/cloudera/proj/lec06'*  file\_root\_dir = *'/home/cloudera/proj/lec06'*  output\_dir = file\_root\_dir + *'/out'*  if not os.path.exists(output\_dir):  os.mkdir(output\_dir)  conf = SparkConf().setMaster(*'local'*).setAppName(*'py01'*)  sc = SparkContext(conf = conf)  sqc = SQLContext(sc) |

**Now the emps RDD is created simply by loading the emps.txt file. Next I use map() on the emps RDD to split into a list of strings representing each value of data for a row. To 100% match the assignment requirements map() is used again to turn the per-row list of strings into a tuple, though everything works as-is without that re-assignment from type = list; perhaps created data types are different in Scala/Java.**

|  |
| --- |
| #1)Create RDD emps by importing that file into Spark.  emps = sc.textFile(spark\_root\_dir + *'/data/emps.txt'*)  #2)Next create a new RDD emps\_fields by transforming the content of every line in RDD emps into  # a tuple with three individual elements by splitting the lines on commas.  emps\_fields = emps.map(lambda line: line.split(*','*)).map(lambda row: tuple(row))  #emps\_fields = emps.map(lambda line: line.split(',')) #works fine |

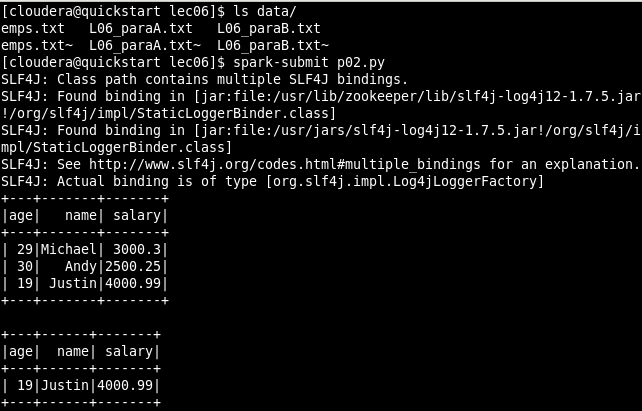
**The next line, re creating an RDD of Row objects, is taken directly from the lecture pdf. Following that create the df DataFrame object by passing that RDD into createDataFrame(), a method on the sqc SQLContext object. Then show() is called on the DataFrame object in order to display its contents in the Terminal window.**

|  |
| --- |
| #3)You need to apply “constructor” Row to every tuple in RDD emps\_fields  employees = emps\_fields.map(lambda e: Row(name = e[0], age = int(e[1]), salary = float(e[2])))  #You generate a DataFrame by passing an RDD of Row elements to the method createDataFrame() of class SQLContext.  df = sqc.createDataFrame(employees)  #Show the content of new DataFrame.  df.show() |

**The final lines involve create a string with the name of the temp table we want to create, to prevent a simple typo later on (as of course I did initially). Call .registerTempTable() on the df DataFrame object to create the temp table by that name = ‘tmp\_employees’. The last line uses the sql method on sqc object to pass in a simple SELECT statement that returns only rows with salary value > 3500, chained to the show() method in order to output to terminal window. Output includes, but isn’t restricted to, employee name.**

|  |
| --- |
| tmp\_table\_name = *'tmp\_employees'*  #Transform this DataFrame into a Temporary Table  df.registerTempTable(tmp\_table\_name)  #select names of all employees who have a salary greater than 3500  sqc.sql(*'SELECT \* FROM {0} WHERE salary > 3500'*.format(tmp\_table\_name)).show() |

**Moving over Terminal window, confirm contents of my /data directory, which includes emp.txt + P01 files + some temp files. Now run my P02.py by submitting to pyspark. Entire output isn’t too lengthy, so below shows everything, from running job to display the two objects, the entire DataFrame followed by the contents of the temp table that match the ‘salary > 3500’ filter.**



**Terminal commands.**

|  |
| --- |
| *#PROBLEM 02*  *ls /data*  *spark-submit p02.py* |

**Problem 3**. Attached file ebay.csv contains information of eBay’s auction history. The Excel file has 9 columns and they represent:

The eBay online auction dataset has the following data fields:

* auctionid - unique identifier of an auction
* bid - the proxy bid placed by a bidder
* bidtime - the time (in days) that the bid was placed, from the start of the auction
* bidder - eBay username of the bidder
* bidderrate - eBay feedback rating of the bidder
* openbid - the opening bid set by the seller
* price - the closing price that the item sold for (equivalent to the second highest bid + an increment)
* item – name of the item being sold
* daystolive – length of the auction.

Using Spark DataFrames you will explore the data with following 4 questions:

1. How many auctions were held?
2. How many bids were made per item?
3. What's the minimum, maximum, and average bid (price) per item?
4. What's the minimum, maximum, and average number of bids per item?
5. Show the bids with price > 100

Import data into an RDD object. Transform that RDD into an RDD of Row-s by assign schema (column names and types). Transform that new RDD into a DataFrame. Call that DataFrame Auction. Show (print) the schema of the DataFrame. Make above queries using DatFrame API. You recall how we applied methods: select(), groupBy(), count() and others to the DataFrame in class. Use those methods.

Next transform your Auction DataFrame into a table and make the same 4 inquiries using regular SQL queries.

Persist your DataFrame as a Parquet file and show that you could exit your pyspark shell and come back in it and you will be still able to read the data from that file and create a DataFrame and an SQL like table that you can issue queries agains.

**Solution:**

**Begin with a code review. Except for the highlighted line, top part of the .py file are all stuff that have been in previous solutions. The import that was added isn’t necessary to solve this problem but I do use the functions module to do number formatting later on.**

|  |
| --- |
| from pyspark import SparkConf, SparkContext, SQLContext, Row  from pyspark.sql import functions as F  spark\_root\_dir = *'file:///home/cloudera/proj/lec06'*  conf = SparkConf().setMaster(*'local'*).setAppName(*'py03'*)  sc = SparkContext(conf = conf)  sqc = SQLContext(sc) |

**The first real task is to import the downloaded ebay.csv file, which is located in same data folder used for previous problems, using the textFile command. Then create an ebay\_lines RDD by splitting on the comma delimiter, winding up with a list of split strings for each line of text. To transform that RDD into an RDD of pyspark.sql.Row objects, use map to pass each data item to the Row constructor. Each item is retrieved by its sequential index in the list, and converted to a non-string data type as appropriate. (To properly calculate values involving money, e.g. for the bid column, float is to be avoided in favor of something like a Python Decimal, but I didn’t get around to being able to play with that.) Create the primary auction DataFrame by passing the RDD of rows into createDataFrame function. Then simply call the printSchema() function on that object to display schema in output.**

|  |
| --- |
| #Import data into an RDD object.  ebay\_data = sc.textFile(spark\_root\_dir + *'/data/ebay.csv'*) #\_short.csv')  ebay\_lines = ebay\_data.map(lambda line: line.split(*','*))  #Transform that RDD into an RDD of Row-s by assign schema (column names and types).  ebay\_rows = ebay\_lines.map(lambda e: Row(auctionid = e[0],  bid = float(e[1]),  bidtime = float(e[2]),  bidder = e[3],  bidderrate = int(e[4]),  openbid = float(e[5]),  price = float(e[6]),  item = e[7],  daystolive = int(e[8])))  # Transform that new RDD into a DataFrame. Call that DataFrame Auction.  auction = sqc.createDataFrame(ebay\_rows)  #Show (print) the schema of the DataFrame.  auction.printSchema() |

**Here is where I persist that auction RDD to parquet, in this case to an HDFS directory named auction\_parquet. I included overwrite mode while in testing phase since the script was being called a number of times. (Later on a separate .py file will retrieve the data from the parquet format.)**

|  |
| --- |
| #Persist your DataFrame as a Parquet file  auction.write.parquet(*'auction\_parquet'*, mode=*'overwrite'*) |

**The two primary tasks of issuing queries using DataFrame vs. tabular objects have been broken down into two functions. The first, as\_dataframe() unsurprising uses the DataFrame approach. It receives the auction DataFrame, as a variable named auc.**

**Breaking into sections:**

**#How many auctions were held?**

* **Each auction is assigned an auctionid, so we simply need a count of the distinct values of that column, using both distinct() and count().**

**#How many bids were made per item?**

* **I interpreted this as the number of bids for each item across all of the auctions and as such only need a .count() called on the groupBy of item values. There were only three items and doing so will sum up the number of rows for each item and with each row representing a single bid we get an answer.**

**#What's the minimum, maximum, and average bid (price) per item?**

* **Compared to aggregate calculations involving the number of bids this one seemed rather simple, so I added it in after doing the former. The assumption here is that again we are once looking across all auctions at once. A groupBy is called on item once again, and then the appropariate aggregate function (min/max/avg) is called upon that intermediate grouping to get the final value.**

|  |
| --- |
| def **as\_dataframe**(auc):  print *'------------- AS DATAFRAME ------------------------------------'*    #How many auctions were held?  print *'Number of auctions:'*  print auc.select(*"auctionid"*).distinct().count()    #How many bids were made per item?  print *'Number of bids per item (across all auctions):'*  auc.groupBy(*'item'*).count().show()    #What's the minimum, maximum, and average bid (price) per item?  print *'Minimum price/bid per item:'*  auc.groupBy(*'item'*).min(*'bid'*).show()  print *'Maximum price/bid per item:'*  auc.groupBy(*'item'*).max(*'bid'*).show()  print *'Average price/bid per item:'*  auc.groupBy(*'item'*).avg(*'bid'*).show() |

**Continuation of above, broken up to make review easier.**

**#What's the minimum, maximum, and average number of bids per item?**

* **This question doesn’t make sense unless handled on a per-auction basis since ‘across all auctions’ min/max/avg are the same value for each item. So to handle per-auction create what is apparently a GroupedData object by grouping on both item and auctionid (uncomment aucGroup1.show() to display results in output) so that I have the equivalent of an item/auctionid/NumberOfBidsThisAuction dataframe. Only with that done can the aggregate calculations be performed, at least with my approach. The aucGroup2 grouping is created pretty much to drop the auctionid column, since we want calculations at the item-level only at this point. The next three statements are very similar, calling min/max/avg functions on the result of a groupBy of the item, performing calcs on the count, i.e. number of bids, at this level. Some formatting specifically for the avg calc so that it is rounded down to reasonable value, and using that then allows for an alias() on the results of that function, to further pretty up the results.**

**#Show the bids with price > 100**

* **The final one is quite simple compared to previous, use filter() function to filter results to only those with bid/price > 100, pass in ‘10’ to show so only 10 get displayed. I had played around with different ways of accessing the dataframe members and I think here is the only time I used attribute/dot notation directly on the auc dataframe.**

|  |
| --- |
| #What's the minimum, maximum, and average number of bids per item?  aucGroup1 = auc.groupBy(*'item'*,*'auctionid'*).count()  #aucGroup1.show() #debug, to show my initial grouping  print *'Minimum number of bids per item:'*  aucGroup2 = aucGroup1.select(aucGroup1[*'item'*],aucGroup1[*'count'*])  (aucGroup2.groupBy(*'item'*).min(*'count'*)  .select(*'item'*, F.format\_number(*'min(count)'*,0)  .alias(*'MinPerBid'*)).show())  print *'Maximum number of bids per item:'*  (aucGroup2.groupBy(*'item'*).max(*'count'*)  .select(*'item'*, F.format\_number(*'max(count)'*,0)  .alias(*'MaxPerBid'*)).show())  print *'Average number of bids per item:'*  (aucGroup2.groupBy(*'item'*).avg(*'count'*)  .select(*'item'*, F.format\_number(*'avg(count)'*,4)  .alias(*'AvgPerBid'*)).show())  print *'Bids with price > 100 (limit = 10)'*  auc.filter(auction.bid > 100).show(10) |

**Now for the table approach, first thing is to take that auc DataFrame that gets passed in and make it a temporary table in SqlContext. Since this is a nested function sqc is still directly available from global environment, use registerDataFrameAsTable on that object to create the queryable temp table.**

**#How many auctions were held?**

* **COUNT(DISTINCT xyz) is standard SQL for doing just what it says, getting number of distinct auctionids, i.e. distinct auctions. I use collect() to get the list of Rows then 0 index to get first (only) Row and again to get first (only) value within. Show() would work fine but this helped understand more of what was going on under the covers.**

**#How many bids were made per item?**

* **As with dataframe, did this ‘across all auctions’. Group on item column and COUNT() based on that to sum up the number of bids.**

**#What's the minimum, maximum, and average bid (price) per item?**

* **With SQL can do this easily in one statement, GROUP BY on item and then call each applicable aggregate function on the results.**

**#What's the minimum, maximum, and average number of bids per item?**

* **This one is also much more straightforward vs. the dataframe solution, at least using my approach. The inner A subquery takes the place of aucGroup1 in as\_dataframe() and from there can simply call the aggregate functions on that inner grouping, using results of COUNT(bid), i.e. the number of bids for each item in a given auction.**

**#Show the bids with price > 100**

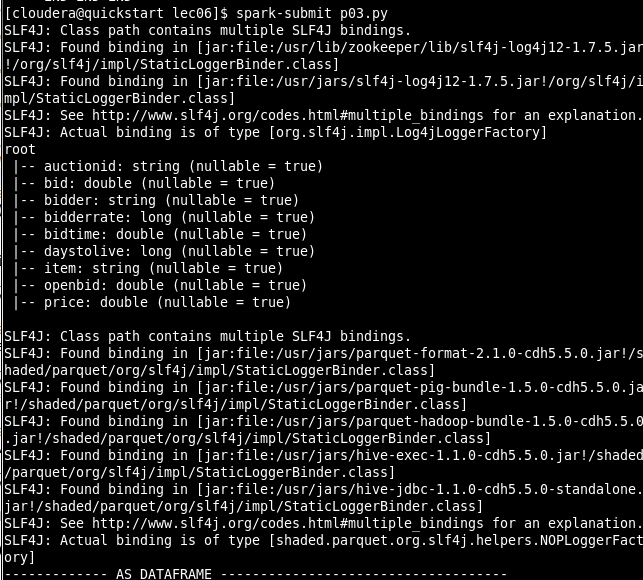
* **Use standard WHERE to filter rows to those with bid/price > 100 and LIMIT 10 statement so that we only see the first 10. Wasn’t asking for the ‘highest’ bids so I didn’t bother with an ORDER BY clause.**

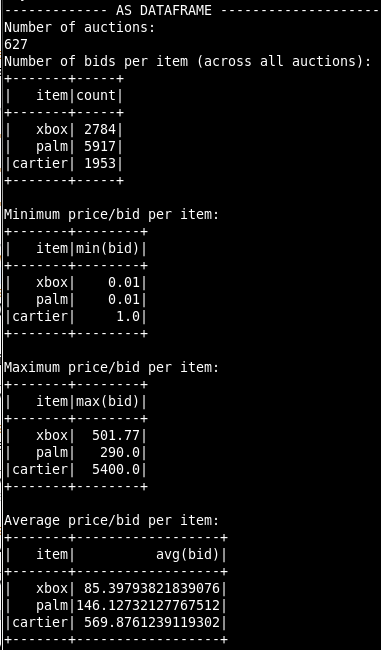
|  |
| --- |
| def **as\_table**(auc):  print *'------------- AS TABLE ------------------------------------'*  sqc.registerDataFrameAsTable(auc, *'auction'*)    #How many auctions were held?  cnt = sqc.sql(*'SELECT COUNT(DISTINCT auctionid) AS auctionCount FROM auction'*).collect()[0][0]  print *'Number of auctions: {0}'*.format(cnt)  #How many bids were made per item?  print *'Number of bids per item (across all auctions):'*  num\_bids\_group = sqc.sql(*'SELECT item, COUNT(bid) AS NumBids FROM auction GROUP BY item'*)  num\_bids\_group.show()  #What's the minimum, maximum, and average bid (price) per item?  print *'Minimum/maximum/average price (bid) per item:'*  sqc.sql(*'''*  *SELECT item, MIN(bid) AS MinBid, MAX(bid) AS MaxBid, AVG(bid) AS AvgBid*  *FROM auction*  *GROUP BY item*  *'''*).show()  #What's the minimum, maximum, and average number of bids per item?  print *'Minimum/maximum/average number of bids per item:'*  sqc.sql(*'''*  *SELECT A.item, MIN(A.NumBids) AS MinNumBids, MAX(NumBids) AS MaxNumBids,*  *AVG(NumBids) AS AvgNumBids*  *FROM*  *(SELECT item, auctionid, COUNT(bid) AS NumBids*  *FROM auction*  *GROUP BY item, auctionid) A*  *GROUP BY A.item*  *'''*).show()  #Show the bids with price > 100  print *'Bids with price > 100'*  sqc.sql(*'SELECT \* FROM auction WHERE bid > 100 LIMIT 10'*).show() |

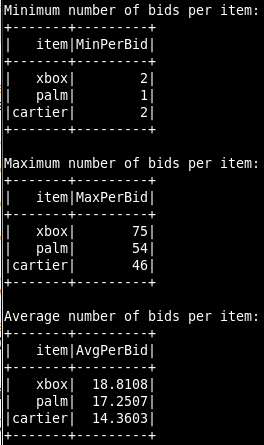
**Calling the defined functions, passing in the auction RDD. Print statements here and in output to visibly separate various actions.**

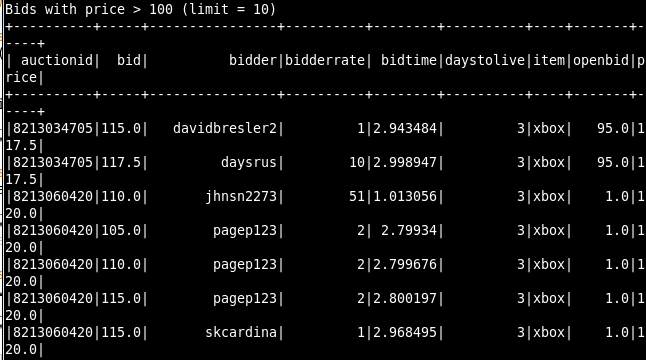
|  |
| --- |
| as\_dataframe(auction)  as\_table(auction)  print *'-----END END END --------------------------------------'* |

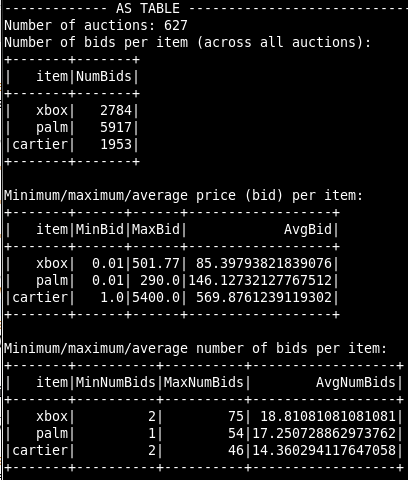
**The final thing to do is submit my p03.py to spark and output results – I don’t see any better way than basically capturing all of the screen output. The ‘------ AS XYZ ------‘ should help delimit DataFrame vs. Temp Table. After Terminal outputs I’ll look at p03a.py, which loads the parquet file in and performs a simple query.**

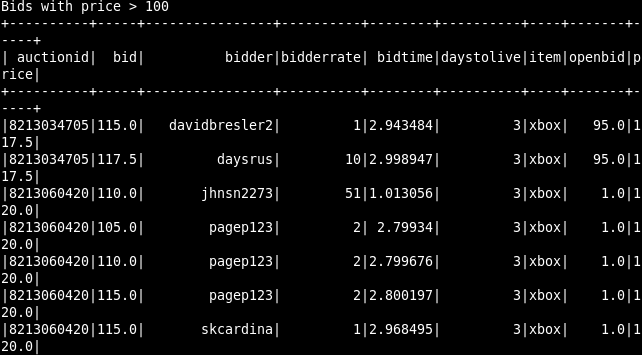








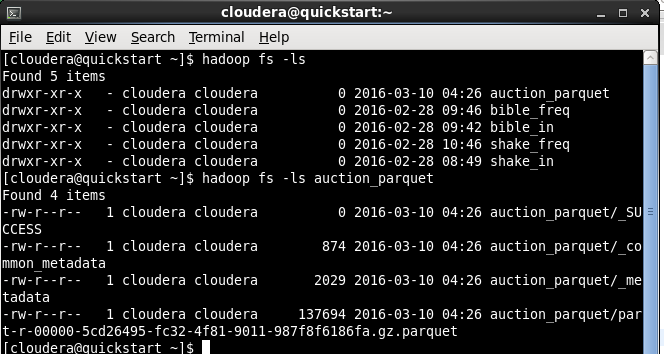




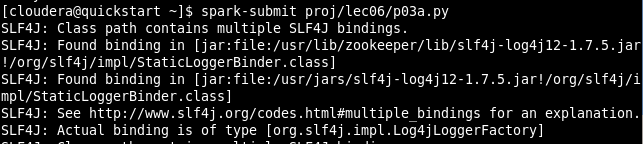
**My p03a.py code for loading in the parquet file is below. Everything has been seen before up until using read.parquet() function via the SQLContext object. The directory is at my HDFS root, so no further pathing needed beyond the directory name itself = “auction\_parquet”. Call .show() to indicate load was successful. Then create a temp table named auction\_p and run a simple query on auction\_p and .show() the results, illustrating that it can indeed be queried**

|  |
| --- |
| from pyspark import SparkConf, SparkContext, SQLContext  conf = SparkConf().setMaster(*'local'*).setAppName(*'py03a'*)  sc = SparkContext(conf = conf)  sqc = SQLContext(sc)  #and show that you could exit your pyspark shell and come back in it  df = sqc.read.parquet(*'auction\_parquet'*)  df.show(5)  sqc.registerDataFrameAsTable(df, *'auction\_p'*)  print *'Bid history for bidder pagep123'*  sqc.sql(*'''*  *SELECT \* FROM auction\_p*  *WHERE bidder = 'pagep123'*  *ORDER BY auctionid, bid*  *'''*).show() |

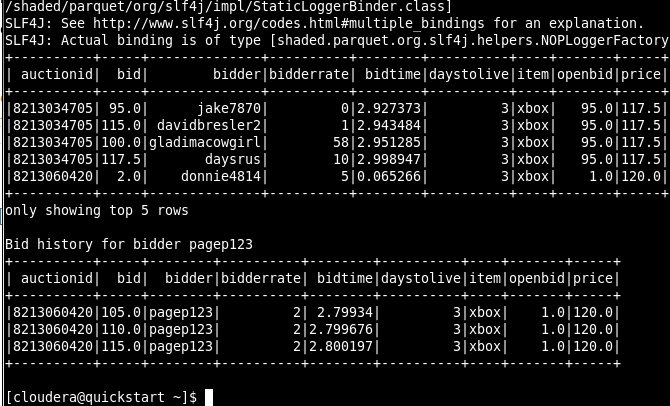
**Here is a brand new Terminal window, with some ls to show the contents of my HDFS, including the auction\_parquet generated by p03.py.**



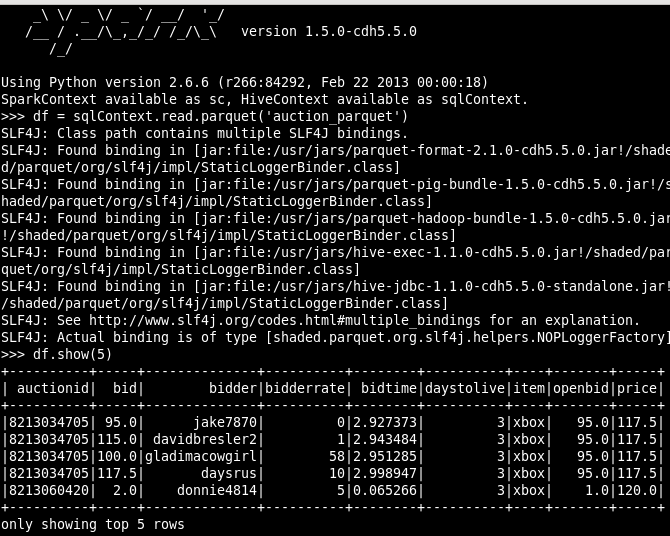
**Submit my p03a.py and output at bottom shows successful running of the embedded queries upon the dataframe as deserialized from the parquet file(s).**



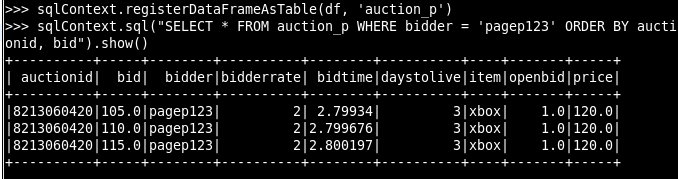
**…**



**Only upon later review did I really notice the reference to ‘pyspark shell ‘ in this last part of the Problem. But running in that environment involved only a few adjustments to the .py code, mostly just a copy & paste. Start pyspark from Terminal, read parquet as in py, only shell comes with sqlContext variable “pre-populated”. Call show() on the df DataFrame.**



**And end with copy & paste of .py code, just replacing ‘sqc’ with ‘sqlContext’.**



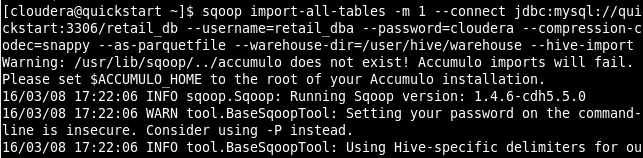
**Terminal commands.**

|  |
| --- |
| *#PROBLEM 03*  *spark-submit p03.py*  *hadoop fs -ls*  *hadoop fs -ls auction\_parquet*  *pyspark*  *>>> df = sqlContext.read.parquet('auction\_parquet')*  *>>> df.show(5)*  *>>> sqlContext.registerDataFrameAsTable(df, 'auction\_p')*  *>>> sqlContext.sql("SELECT \* FROM auction\_p WHERE bidder = 'pagep123' ORDER BY auctionid, bid").show()* |

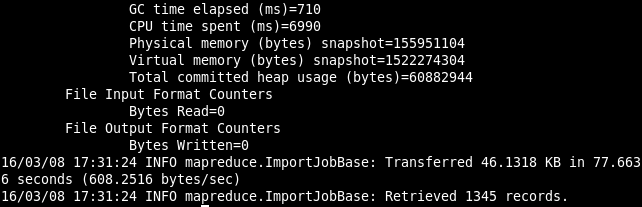
**Problem 4**. Use Sqoop to import all tables in MySQL demo database retail\_db into Hive. Use Spark DataFrame API or Spark Temporary Tables to find orders with the largest number of order items per order.

**Solution:**

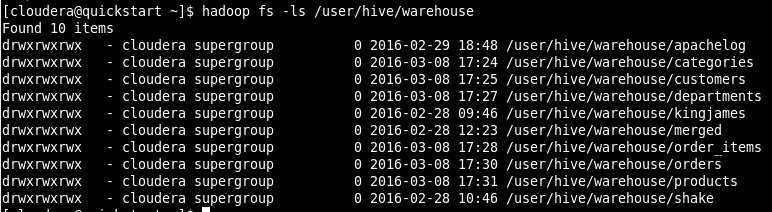
**Importing files into sqoop doesn’t involve much more than copy and paste of the necessary command from lecture pdf. I walked away from my computer but as noted in the pdf it did apparently take a while, a little over 9 minutes going by the Terminal timestamps.**



…



**Check that the data has made its way into HDFS, note categories, customers etc. and similar tables that have date modified around when I imported into sqoop.**



**Perform setup so that Spark will be able to interact with Hive data, first start hiveserver2 in the background.**



**Elevate permissions and copy hive-site.xml into $SPARK\_HOME/conf.**



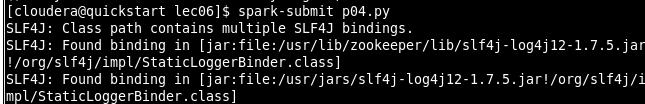
**Now for a review of the code. The setup code is similar to what I’ve had before and the few differences are highlighted – import of HiveContext and then creation of an object of that type, passing SparkContext object into its constructor.**

|  |
| --- |
| from pyspark import SparkConf, SparkContext, SQLContext, HiveContext  conf = SparkConf().setMaster(*'local'*).setAppName(*'py01'*)  sc = SparkContext(conf = conf)  hc = HiveContext(sc) |

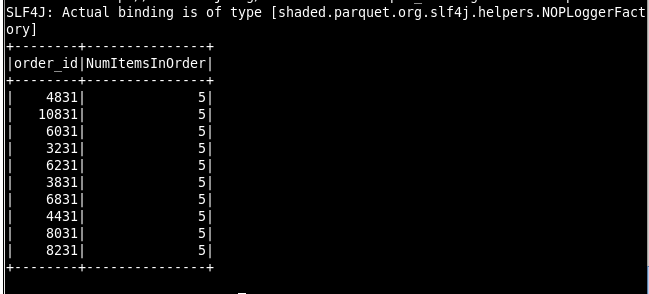
**The new code is quite simple since the retail\_db tables are directly available. To simply answer the assignment in this part may not really need to join to order table but without that you aren’t really returning ‘order’ items, so I went ahead and did so. GROUP BY on the order\_id so that all rows with that same value are grouped together when executing the aggregate COUNT() function. Limited output to the 10 highest, which all have item counts = 5.**

|  |
| --- |
| query = *'''*  *SELECT o.order\_id, COUNT(o.order\_id) as NumItemsInOrder*  *FROM order\_items i*  *INNER JOIN orders o ON o.order\_id = i.order\_item\_order\_id*  *GROUP BY o.order\_id*  *ORDER BY COUNT(o.order\_id) DESC*  *LIMIT 10*  *'''*  dfs = hc.sql(query)  dfs.show() |

**Submit the .py file to spark and observe the output.**



**…**



**Terminal commands.**

|  |
| --- |
| *#PROBLEM 04*  *sqoop import-all-tables -m 1 --connect jdbc:mysql://quickstart:3306/retail\_db --username=retail\_dba --password=cloudera --compression-codec=snappy --as-parquetfile --warehouse-dir=/user/hive/warehouse --hive-import*  *hadoop fs -ls /user/hive/warehouse*  *hiveserver2 &*  *su*  *cp /etc/hive/conf/hive-site.xml /etc/spark/conf/*  *spark-submit p04.py* |