National Technical University of Athens

School of Electrical & Computer Engineering

Big Data Management

Professors:

I. Konstantinou, N. Kozyris

Pyspark and MapReduce frameworks

Pantos Athanasios
Prospective MSc student
A.M. ΕΔΕΜΜ: 03400026
e-mail: pantos.thn@gmail.com

July 25, 2021

Contents

List of Figures

List of Tables

1 Data Extraction - SQL

1.1 Data extraction with different ways

1.1.1 MapReduce implementation

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster. A MapReduce program is composed of a map procedure, which performs filtering and sorting, and a reduce method, which performs a summary operation.The "MapReduce System" (also called "infrastructure" or "framework") orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

The pyspark enviroment extends those capabilities. The first implementation is using the Spark RDD (Resilient Distributed Dataset) API. On top of the RDD lazy transformations are implemented on persisten data.

In the next subsections you can find the results of the implementaion of the two questions of the first part.

The Dataset

This dataset includes trip records from all trips completed in yellow taxis from in NYC from January to June in 2015. Records include fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances. The data used in the attached datasets were collected and provided to the NYC Taxi and Limousine Commission (TLC) by technology providers authorized under the Taxicab Passenger Enhancement Program (TPEP).

What is the average value of longitude as well as latitude of boarding for each hour? Sort the results based on descending order and ignore any problematic data points (such as lat / long with zero values)

In the table 1 below you can find the results for the mapreduce implementation of question 1.

For each vendor, find the maximum Haversine distance of a trip and the duration.

In the table 2 below the maximum distance is depicted for each vendor as well as the duration.

Table 1: Results for the first part of the question - MapReduce implementation

Hours Of Day	**Latitude**	**Longtitude**
00	40.74352391432558	-73.97542119644494
01	40.74144873105477	-73.98015550551409
02	40.74096648053428	-73.98106246096083
03	40.74157920178637	-73.981988010782717
04	40.744919995790845	-73.97779849446763
05	40.74785017767986	-73.97273846432589
06	40.75116285983248	-73.96963386968942
07	40.75413501865128	-73.97124155969401
08	40.754149591596814	-73.9734171341711
09	40.75406090873702	-73.97479352068423
10	40.75453335756495	-73.97366927085308
11	40.75432492996547	-73.97430727880621
12	40.754283962218224	-73.9746343408384
13	40.75370263890706	-73.97432216233956
14	40.753469749883244	-73.97310395038653
15	40.753364065771095	-73.97143887498935
16	40.752952755391675	-73.96993496643499
17	40.75330164005164	-73.97154585281723
18	40.752760254901176	-73.97387697738458
19	40.75127933823094	-73.97518884723644
20	40.749653423341215	-73.9760667030382
21	40.748726699352076	-73.97491147877938
22	40.74776911907728	-73.97503320838544
23	40.74569797772432	-73.97406554954225

Table 2: Results for the second part of the question - MapReduce implementation

Vendor	**Max Distance**	**Duration**
1	7093.165996962807	688.0
2	8170.756255885991	661.0

1.1.2 SparkSQL implementation

Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed. Internally, Spark SQL uses this extra information to perform extra optimizations. There are several ways to interact with Spark SQL including SQL and the Dataset API. When computing a result, the same execution engine is used, independent of which API/language you are using to express the computation. This unification means that developers can easily switch back and forth between different APIs based on which provides the most natural way to express a given transformation.

The main advantage of using SparkSQL is that it allows us to get the same result without actually to write each transformation on the RDDs directly.In addition it also a lot faster as we will see below.

1.1.3 Parquet implementation

Parquet is an open source file format available to any project in the Hadoop ecosystem. Apache Parquet is designed for efficient as well as performance flat columnar storage format of data compared to row based files like CSV or TSV files.

Parquet uses the record shredding and assembly algorithm which is superior to simple flattening of nested name spaces. Parquet is optimized to work with complex data in bulk and features different ways for efficient data compression and encoding types. This approach is best especially for those queries that need to read certain columns from a large table. Parquet can only read the needed columns therefore greatly minimizing the IO.

1.1.4 Execution time plots

We see in the figures below that the implementation with dataframes and sparkSQL improves run time allot when compared to the RDD API.The parquet transformation tends to decrease the run time even more . In addition, we also notice that the more complex the applications (Q2) the greater the execution times. This alone states the importance to use such methods in order to minimize the time needed by as much as possible.

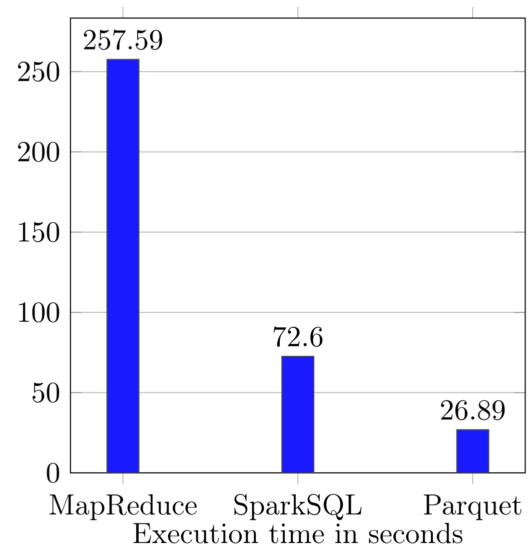

Figure 1: Question 1 Execution Time

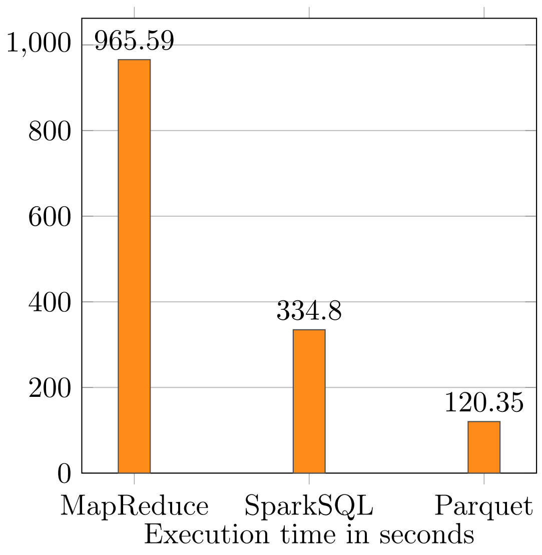

Figure 2: Question 2 Execution Time

1.2 Optimizer test for merging

It is well known that besides the data and the settings that are entered by the user before a particular join, SparkSQL uses an optimizer.A query optimizer is a critical database management system component that analyzes Structured Query Language (SQL) queries and determines efficient execution mechanisms.It also generates one or more query plans for each query, each of which may be a mechanism used to run a query.

Our task is to join the two parquet files that we have and to check which implementation was actually chosen by the system.We then have to change the default setting , run again the procedure and see which optimizer was chosen.

1.2.1 Which join was used by Spark framework?

For the first scenario (default) BroadcastJoin was selected and this was due to one file being considerably smaller than the other one.

```
== Parsed Logical Plan ==
'Join UsingJoin(Inner,Buffer(id))'
:- Relation[id#84,start#85,end#86,lon_start#87,lat_start#88,lon_end#89,\\
lat_end#90,cost#91] parquet
+- GlobalLimit 100
   +- LocalLimit 100
      +- Project [id#80, vendor#81]
         +- SubqueryAlias `tempdf1`
            +- Relation[id#80,vendor#81] parquet

== Analyzed Logical Plan ==
id: string, start: string, end: string, lon_start: string, lat_start: string, \\
lon_end: string, lat_end: string, cost: string, vendor: string
Project [id#84, start#85, end#86, lon_start#87, lat_start#88, lon_end#89, \\
lat_end#90, cost#91, vendor#81]
+- Join Inner, (id#84 = id#80)
   :- Relation[id#84,start#85,end#86,lon_start#87,lat_start#88,lon_end#89,\\
   lat_end#90,cost#91] parquet
   +- GlobalLimit 100
      +- LocalLimit 100
         +- Project [id#80, vendor#81]
            +- SubqueryAlias `tempdf1`
               +- Relation[id#80,vendor#81] parquet

== Optimized Logical Plan ==
Project [id#84, start#85, end#86, lon_start#87, lat_start#88, lon_end#89,\\
lat_end#90, cost#91, vendor#81]
+- Join Inner, (id#84 = id#80)
   :- Filter isnotnull(id#84)
   :  +- Relation[id#84,start#85,end#86,lon_start#87,lat_start#88,\\
   lon_end#89,lat_end#90,cost#91] parquet
   +- Filter isnotnull(id#80)
      +- GlobalLimit 100
         +- LocalLimit 100
            +- Relation[id#80,vendor#81] parquet

== Physical Plan ==
*(3) Project [id#84, start#85, end#86, lon_start#87, lat_start#88,\\
lon_end#89, lat_end#90, cost#91, vendor#81]
+- *(3) 'BroadcastHashJoin' [id#84], [id#80], Inner, BuildRight
   :- *(3) Project [id#84, start#85, end#86, lon_start#87, lat_start#88,\\
   lon_end#89, lat_end#90, cost#91]
   :  +- *(3) Filter isnotnull(id#84)
   :     +- *(3) FileScan parquet [id#84,start#85,end#86,lon_start#87,\\
   lat_start#88,lon_end#89,lat_end#90,cost#91] Batched: true, \\
   Format: Parquet, \\Location: InMemoryFileIndex \\
   [hdfs://master:9000/input/df2_temp.parquet],\\
   PartitionFilters: [], PushedFilters: [IsNotNull(id)], ReadSchema: \\
   lat_start:string,lon_end:string,lat_end...\\
   +- BroadcastExchange HashedRelationBroadcastMode \\
   (List(input[0, string, false]))
      +- *(2) Filter isnotnull(id#80)
         +- *(2) GlobalLimit 100
            +- Exchange SinglePartition
               +- *(1) LocalLimit 100
                  +- *(1) FileScan parquet \\
                  PartitionFilters: [], PushedFilters: [],
                  ReadSchema: struct<id:string,vendor:string>
```

1.2.2 After implementing the appropriate setting which optimizer was chosen?

For this task we manually assign BroadcastJoinThreshold = -1 and with this way we disable broadcast join.We execute the same code and we can detect that now SortMergeJoin which is an implementation of repartition join.

Best case scenario the performance will be equal to that of broadcastjoin if the smaller file is projected onto the bigger one.In practice though, this does not hold true for two reason. First for speed reasons and because the repartition join does not always pick the smaller file.

```
== Parsed Logical Plan ==
'Join UsingJoin(Inner,Buffer(id))'
:- Relation[id#84,start#85,end#86,lon_start#87,lat_start#88,lon_end#89,lat_end#90,\\
cost#91] parquet
+- GlobalLimit 100
   +- LocalLimit 100
      +- Project [id#80, vendor#81]
         +- SubqueryAlias `parquetd1f`
            +- Relation[id#80,vendor#81] parquet

== Analyzed Logical Plan ==
id: string, start: string, end: string, lon_start: string,\\
lat_start: string, lon_end: string, lat_end: string, cost: string, vendor: string
Project [id#84, start#85, end#86, lon_start#87, lat_start#88,\\
lon_end#89, lat_end#90, cost#91, vendor#81]
+- Join Inner, (id#84 = id#80)
   :- Relation[id#84,start#85,end#86,lon_start#87,lat_start#88,\\
   lon_end#89,lat_end#90,cost#91] parquet
   +- GlobalLimit 100
      +- LocalLimit 100
         +- Project [id#80, vendor#81]
```

```
            +- SubqueryAlias `parquetd1f`
               +- Relation[id#80,vendor#81] parquet

== Optimized Logical Plan ==
Project [id#84, start#85, end#86, lon_start#87, lat_start#88, \\
lon_end#89, lat_end#90, cost#91, vendor#81]
+- Join Inner, (id#84 = id#80)
   :- Filter isnotnull(id#84)
   :  +- Relation[id#84,start#85,end#86,lon_start#87,lat_start#88,\\
   }lon_end#89,lat_end#90,cost#91] parquet
   +- Filter isnotnull(id#80)
      +- GlobalLimit 100
         +- LocalLimit 100
            +- Relation[id#80,vendor#81] parquet

== Physical Plan ==
*(6) Project [id#84, start#85, end#86, lon_start#87, lat_start#88,
lon_end#89, lat_end#90, cost#91, vendor#81]
+- *(6) 'SortMergeJoin' [id#84], [id#80], Inner
   :- *(2) Sort [id#84 ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(id#84, 200)
   :     +- *(1) Project [id#84, start#85, end#86, lon_start#87, lat_start#88,
   lon_end#89, lat_end#90, cost#91]
   :        +- *(1) Filter isnotnull(id#84)
   :           +- *(1) FileScan parquet [id#84,start#85,end#86,lon_start#87,\\
   lat_start#88,
   lon_end#89,lat_end#90,cost#91] Batched: true, Format: Parquet, Location:
   PartitionFilters: [], PushedFilters: [IsNotNull(id)], ReadSchema:
   +- *(5) Sort [id#80 ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(id#80, 200)
         +- *(4) Filter isnotnull(id#80)
            +- *(4) GlobalLimit 100
               +- Exchange SinglePartition
                  +- *(3) LocalLimit 100
                     +- *(3) FileScan parquet [id#8
```

2 Machine Learning - Text classifier

In this section we will use the consumer complaint data set, which describes complaints customers in relation to financial products and services. Specifically, we will use a subset of the data which we downloaded and then uploaded to hdfs. As for the file format, it is a comma-delimited csv file where the first field is the date of the complaint, the second field is the category of the product or service and the third is the customer's complaint.

To extract features for the purpose of training machine learning models we use the TF-IDF technique,

$$tfidf(t, d, D) = tf(t, d) * idf(t, D)$$

Where t, d, D the term, the document and the collection of documents based on which the calculations are executed.The terms tf and idf are calculated based on the following formulas:

- $$tf(t,d) = 0.5 + 0.5 * \frac{f_{t,d}}{\max f_{t',d} : t' \in d}$$

- $$idf(t,D) = \log \frac{N}{|d \in D : t \in d|}$$

Below are the calculation steps in order to transform the data to the right format and train the machine learning algorithm.

1. In the first step we downloaded the file via the bash command on our master server at oceanos. We then uploaded the csv file to hdfs.

2. In addition we proceeded with the cleaning of the data.The data that we have are separated in three columns (date, category, comment). We only keep the rows that their date starts with 201 and we discard which ever rows have empty comment. We used the startswith command for the date filtering so we end up with a tuple (id, date, comment).

3. In the third step we are trying to find all the different words that exist in user's comments. In the mapreduce framework we use the re library and the stop-words, unnecessary blanks and numbers are removed.In addition to that we lower all the characters. The stopwords were removed with the help of the NLTK library.

4. With the above data we begin to calculate the TFIDF metric. We worked exclusively with RDDs. Initially we find the most common words and keep them as features. Essentially we create a word count lexicon.Then, we broadcast this lexicon for all the workers. With the above data in hand we begin to calculate the tf finding for each comment which words are in the lexicon and returning their position at the lexicon.We discard the sentences that do not have a word in the lexicon and we add an auto incremental index witht he help of zipWithIndex function.After that we calculate the frequency of each word in this sentence with the reduceByKey function and the length of the sentence. For the IDF we need the number of sentences so we use an expensive count(). We then reduceByKey with a word as a key. From the above calculations we end up with an rdd that contains the rows that have each word and how many times the word was shown up in a particular sentence. We then use reduceByKey again to find how many sentences had the certain word.We then divide by the total number of sentences and we the log in order to calculate the IDF component.

   We then join with the key the words and reduceByKey with the key the increasing number of each sentence.With that all the words of each sentence are added to a list along with their corresponding TFIDF score and their index number in the lexicon. We then sort and use the SparceVector to reduce the amount of memory needed.

We end up with the following table just as it was asked in the exercise.

```
('mortgage', SparseVector(128, {0: 0.1014, 21: 0.6369, 76: 0.3016, 97:
0.3245}))
('credit reporting credit repair services or other personal consumer
reports', SparseVector(128, {0: 0.0978, 2: 0.024, 3: 0.0369, 4: 0.0432,
..., 110: 0.1216}))
('debt collection', SparseVector(128, {0: 0.0169, 1: 0.0159, 2: 0.0115,
3: 0.0132,..., 123: 0.0297}))
```

5. The table above is then transformed to a Spark Dataframe so we can use a split and then train a simple perceptron.

6. For the spliting of the data we used a stratified split and we ended up in the following distribution: 75% for the training set and 25% for the test set.For this to take place we find all the different classes-labels and we split accordingly.After the train test split we end up with the following table:

   Table 3: Dataset distribution after split

Set	Number of Rows
Training	355259
Test	118406

   Regarding the number of rows that is assigned to each category we have the following distribution:

7. For the training of simple perceptron we used the two train and test dataframes and the MultilayerPerceptron offered by pyspark.ml . In the begining we trained the model without using the cache mechanism. The reason why cache is usefull is because the data is available throughout the training phase and there is no need for look ups in the storage.When we cache a dataset, all the transformations etc. take place only once in the beggining and the data are used for the next itterations.

Table 4: Number of rows per label

Label	Training Set	Test Set
0	107633	23668
1	80121	24300
2	46039	15340
3	24093	7794
4	23544	6705
5	18965	6262
6	14305	4787
7	14017	4680
8	11178	3679
9	7106	2341
10	6118	2026
11	5958	1968
12	4837	1617
13	1320	416
14	1119	351
15	1111	352
16	217	87
17	10	2

Regarding the times with and without the use of cache we have the following bar plot:

Furthermore, several models have been tested and we present to you our best findings. The accuracy of the model is depicted with and without the use of cache.

Table 5: Accuracy

	With Cache	Without Cache
Accuracy	0.6043726487324423	0.604048422680218

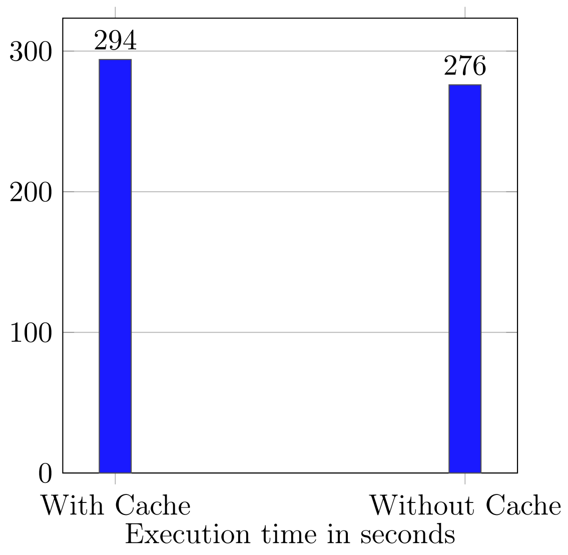

Figure 3: Question 2 Execution Time