National Technical University of Athens

Interdisciplinary Master's Programme in
Data Science and Machine Learning

BIG DATA MANAGEMENT SYSTEMS
SEMESTER ASSIGNMENT

Apache Spark in Databases

[redacted]

July 24, 2022

Contents

Short description

In this semester assignment we use the Apache Spark engine for the computation of analytical queries in large scale datasets. The goal of the assignment is to explore the high-level APIs provided by Apache Spark for the manipulation of large scale datasets. We make use of the Hadoop Distributed File System [2] (HDFS) in order to store our data. HDFS employs a NameNode and DataNode architecture to implement a distributed file system that provides high-performance access to data across highly scalable Hadoop clusters. In our case, the cluster consists of two virtual machines provided by GRNET's cloud service. Both machines share the same specifications; 2 CPU's and 4GB RAM each. One of the two machines acts as the DataNode and NameNode at the same time while the second one is the second DataNode of the cluster. The dataset that we use to perform the queries is a dataset consisting of song charts obtained from spotify[1]. The *tar.gz* file contains 4 *csv* files: *artists.csv, chart_artist_mappping.csv, charts.csv, regions.csv*. The main purpose of this assignment is to use two different APIs of Apache Spark; the RDD API and the Dataframe API / Spark SQL and compare the performance of these two APIs. In addition to the *csv* we use the *parquet* file format designed for efficient data storage and retrieval. We compare the three different approaches with respect to the time duration of executing six queries independently and provide explanations for the results.

1 & 2 Creating and storing files in HDFS

After the installation of the cluster the next step is to create a new directory named *files* in the HDFS to store all of our data. We store the four *csv* files in the HDFS files directory. Moreover, using the *csv* files we create the *parquet* files. Apache Parquet is an open source, column-oriented data file format designed for efficient data storage and retrival. It provides efficient data compression and encoding schemes with enhanced performance to handle complex data in bulk. Moreover, the required space to load a *parquet* file format in RAM is less than the *csv* as well as the corresponding required space in hard drive, and for that reason, it optimizes the I/O tasks by reducing the execution time. Another advantage of the *parquet* files over the *csv* files is that the former ones keep additional information about the data. For example, for a block of integer values, *parquet* contains the information about the max/min values of the block which in turn optimizes the filtered retrieval of the data.

[1]The dataset can be downloaded from the following link (be careful, downloading on click).

3 RDD API and Spark SQL queries

We now perform a set of 6 queries in three different ways resulting in a total of 18 queries. The queries implemented are the ones detailed in the following table.
The first implementation uses *csv* files as input files and the RDD API to query them and derive the desired results. Then, Spark SQL is also used to query the input files in both *csv* and *parquet* format and derive the same results.

Query number	Implementation	Input file format
Q1	RDD API	csv
Total number of streams for the "Shape of You" song	Spark SQL	csv
according to the top200 charts.	Spark SQL	parquet
Q2	RDD API	csv
Song with the longest mean remaining time	Spark SQL	csv
on position #1 for each chart.	Spark SQL	parquet
Q3	RDD API	csv
Mean daily number of streams of song at #1	Spark SQL	csv
of top200 charts for each month of each year.	Spark SQL	parquet
Q4	RDD API	csv
Song(s) in viral50 charts with maximum days	Spark SQL	csv
remaining in the charts for each country.	Spark SQL	parquet
Q5	RDD API	csv
Artists with maximum mean number of streams	Spark SQL	csv
in top200 charts for each year.	Spark SQL	parquet
Q6	RDD API	csv
Artist(s) with maximum number of consecutive days in #1	Spark SQL	csv
for any of their songs in Greek charts for each year.	Spark SQL	parquet

4 Queries execution and results

In this section, we execute all 18 scripts that correspond to the 6 previously described queries and record the results and execution times for each implementation. The query results are saved in *csv* format in the hdfs. The execution time is measured at the time of query evaluation in order to be as objective as possible, allowing comparison between implementations.
In the following subsections, we first define the query, then record the results generated and finally present the execution times for the three implementations of each query in a bar plot.

4.1 Query 1

Description: Query 1

Total number of streams for the "Shape of You" song according to the top200 charts.

Total streams
2324245979

Below we see the execution times for the three implementations.

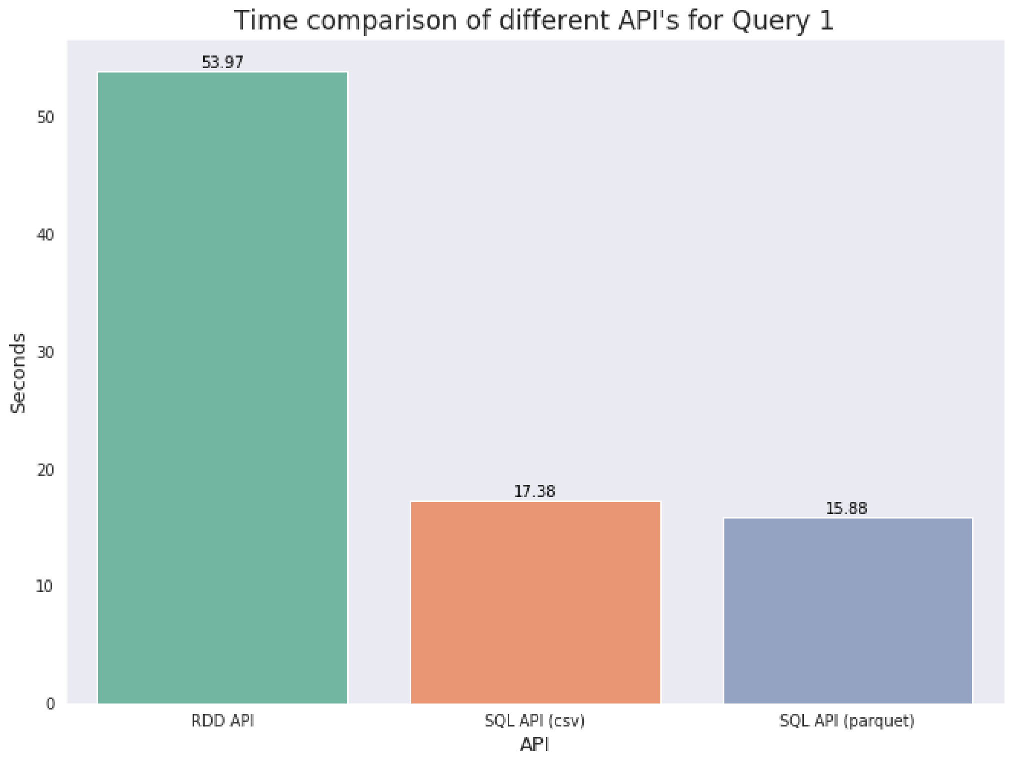

Figure 1: The time comparison for the three different API's for the first query. The RDD API executed the query in 53.97 seconds, the SQL API with the csv file format in 17.38 seconds and the SQL API with the parquet file format in 15.88 seconds.

As we can see from the above table the SQL API of Apache spark with the parquet file format achieves the fastest execution of the query. This is of course expected, since as we have already mentioned the parquet file format keeps additional information for the data and hence requires less time to perform filters on data. The reason why the RDD API requires more time to execute the query with respect to other to approaches is due to the custom implementation of the Map-Reduce. The other two methods optimize the Map-Reduce jobs by minimizing the operations needed to execute the query resulting in lower execution times. In the following pseudocode we describe the custom implementation of the Map-Reduce utilized in the RDD API.

Algorithm 1 Query 1 - Map-Reduce implementation in RDD

```
Require: rdd ← rdd(charts.csv)
1: rdd.filter(song_name =="Shape of You and chart = "top200")
2: rdd.map$(\mathbf{x} \mapsto \mathbf{x}' : (\text{"result"}, \text{number\_of\_streams}))$.ReduceByKey$(x, y : x + y)$
3: return rdd.
```

Query 2

Description: Query 2

Song with the longest mean remaining time on position #1 for each chart (viral50, top200).

Chart	Song	Mean remaining time #1
viral50	Calma - Remix	24.985507246376812
top200	Shape of You	54.2463768115942

In Figure 7 we see the execution times for the three different API's.

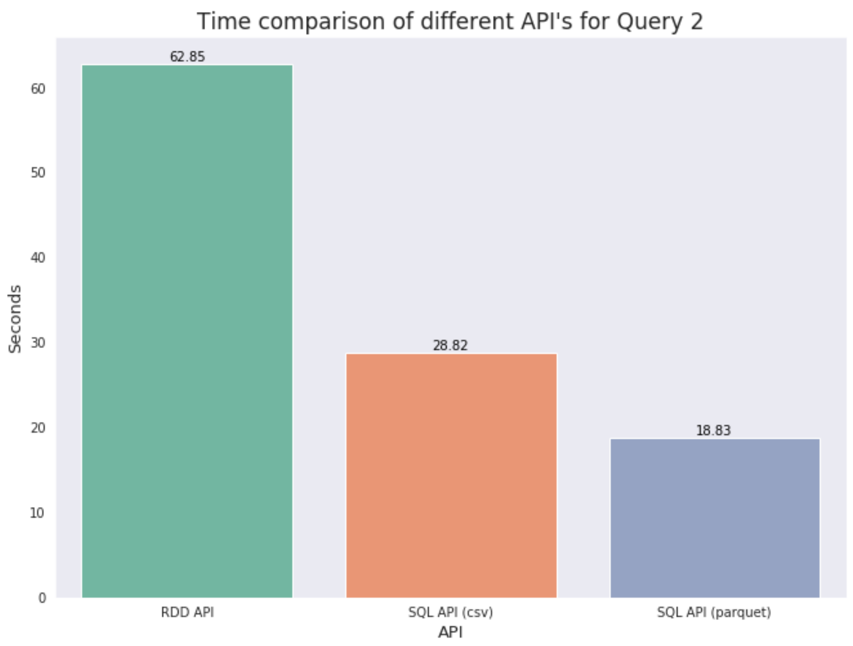

Figure 2: The time comparison for the three different API's for the second query. The RDD API executed the query in 62.85, the SQL API (csv) in 28.82 and the SQL API (parquet) in 18.83 seconds.

Below we see the pseudocode for the Map-Reduce developed for the second query in the RDD API.

Algorithm 2 Query 2 - Map-Reduce implementation in RDD

```
Require: rdd(charts.csv)
1: rdd.filter(song_position == "1")
2: rdd.map ($\mathbf{x} \mapsto \mathbf{x}'$ : {(song_name, chart_type), 1}) .reduceByKey(add)
3: rdd.map ($\mathbf{x} \mapsto \mathbf{x}'$ : {(chart_type, total_sum_of_appearances), song_name})
4: rdd.sortByKey(ascending = False)
5: rdd.map($\mathbf{x} \mapsto \mathbf{x}'$ : {(chart_type), (total_sum_of_apperances,song_name)}) .reduceByKey(max)
6: rdd.map ($\mathbf{x} \mapsto \mathbf{x}'$ : {(chart_type, song_name, total_sum_of_appearances/69)})
7: return rdd.
```

Query 3

Description: Query 3

Mean daily number of streams of song at #1 of top200 charts for each month of each year.

Year	Month	Mean daily streams of #1 song, top200 charts
2017	1	7618611.064516129
2017	2	8876450.785714285
2017	3	8955476.41935484
2017	4	8178985.833333333
2017	5	8939831.827586208
2017	6	7790440.0
2017	7	6757058.387096774
2017	8	6599688.064516129
2017	9	7246840.5
2017	10	8138961.129032258
2017	11	7578529.066666666
2017	12	7539380.677419355
2018	1	8135145.774193549
2018	2	9710137.92857143
2018	3	8713800.0

2018	4	9020981.7
2018	5	8654503.193548387
2018	6	8953159.9
2018	7	1.0391241935483871E7
2018	8	8394892.903225806
2018	9	8358084.0
2018	10	8482952.064516129
2018	11	1.01695437E7
2018	12	9371460.225806452
2019	1	1.018902458064516E7
2019	2	1.0817727392857144E7
2019	3	9913579.935483871
2019	4	1.0819890666666666E7
2019	5	1.0465042451612903E7
2019	6	1.0857143233333332E7
2019	7	1.1214812709677419E7
2019	8	1.0708007258064516E7
2019	9	1.04632976E7
2019	10	1.1164322903225806E7
2019	11	1.0746356433333334E7
2019	12	1.1001986903225806E7
2020	1	1.3611543E7
2020	2	1.2275200068965517E7
2020	3	1.1213416838709677E7
2020	4	9582862.366666667
2020	5	9113325.838709677
2020	6	9474377.933333334
2020	7	1.0972215129032258E7
2020	8	1.2249567838709677E7
2020	9	1.2756135466666667E7
2020	10	1.1171908677419355E7
2020	11	1.28059231E7
2020	12	1.204448235483871E7
2021	1	1.2761012483870968E7

2021	2	1.0721677142857144E7
2021	3	1.1661322709677419E7
2021	4	1.2896025933333334E7
2021	5	1.5425066774193548E7
2021	6	1.60089523E7
2021	7	1.4274800677419355E7
2021	8	1.4186574E7
2021	9	1.3402214233333332E7
2021	10	1.2857921677419355E7
2021	11	1.2087256933333334E7
2021	12	3323677.7741935486

In Figure 3 we see the corresponding execution time of the three different approaches for the third query.

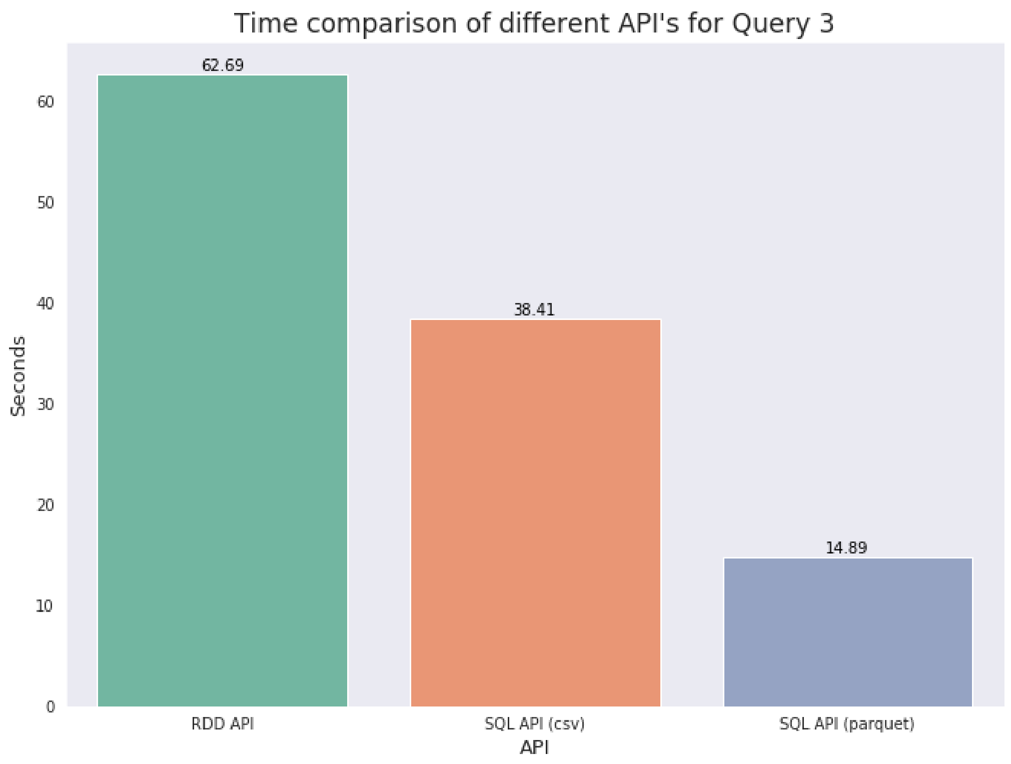

Figure 3: The time comparison for the three different API's for the third query. The RDD API executed the query in 62.69, the SQL API (csv) in 38.41 and the SQL API (parquet) in 14.89 seconds.

Below we see the pseudocode for the Map-Reduce developed for the third query in the RDD API.

Algorithm 3 Query 3 - Map-Reduce implementation in RDD

```
Require: rdd ← rdd(charts.csv)
1: rdd.filter(chart_type == "top200" and song_position == "1"
2: rdd.map$(\mathbf{x} \mapsto \mathbf{x}' : \{(\text{year, month,day}), (\text{number\_of\_streams})\})$
3: rdd.reduceByKey(add).map$(\mathbf{x} \mapsto \mathbf{x}' : \{(\text{year,month}), (1, \text{number\_of\_streams})\})$
4: rdd.reduceByKey$(x, y : (x[0]+y[0], x[1]+y[1]))$.sortByKey()
5: rdd.map$(\mathbf{x} \mapsto \mathbf{x}' : \{(\text{year,month, total\_number\_of\_streams/num\_month\_days})\})$
6: return rdd.
```

Query 4

Description: Query 4

Song(s) in viral50 charts with maximum days remaining in the charts for each country.

Country	Song id	Song name	Days in viral50 chart
Andorra	55526	Friday (feat. Mufasa;Hypeman) - Dopamine Re-Edit	251
Argentina	35851	Dance Monkey	253
Australia	35851	Dance Monkey	217
Austria	131808	Roses - Imanbek Remix	233
Belgium	131808	Roses - Imanbek Remix	207
Bolivia	92280	Lost on You	239
Brazil	35851	Dance Monkey	252
Bulgaria	12491	Arcade	231
Canada	131808	Roses - Imanbek Remix	287
Chile	67656	Hookah;Sheridan s	256
Colombia	35851	Dance Monkey	253
Costa Rica	157341	Toast	251
Czech Republic	141605	Slunko	229
Denmark	131808	Roses - Imanbek Remix	198
Dominican Republic	42550	Dream Girl - Remix	223
Ecuador	35851	Dance Monkey	248

Egypt	143230	Someone You Loved	282
El Salvador	35851	Dance Monkey	217
Estonia	70018	I Got Love	338
Finland	120735	Penelope (feat. Clever)	214
France	26355	Calma - Remix	189
Germany	131808	Roses - Imanbek Remix	225
Greece	36928	De Me Theloun	211
Greece	143230	Someone You Loved	211
Guatemala	35851	Dance Monkey	242
Honduras	108941	Ni Gucci Ni Prada	225
Hong Kong	191782	夜にける	343
Hungary	131808	Roses - Imanbek Remix	265
Iceland	64545	Heat Waves	226
India	178233	ily (i love you baby) (feat. Emilee)	154
Indonesia	70406	I Love You but I'm Letting Go	319
Ireland	50796	Fairytale of New York (feat. Kirsty MacColl)	231
Israel	35851	Dance Monkey	223
Italy	85993	La musica non c'‘Í	222
Japan	191782	夜にける	374
Latvia	131808	Roses - Imanbek Remix	226
Lithuania	164633	Vasar‘Ě galvoj minoras	284
Luxembourg	131808	Roses - Imanbek Remix	202
Malaysia	143230	Someone You Loved	268
Mexico	92280	Lost on You	371
Morocco	178496	love nwantiti (feat. ElGrande Toto) - North African Remix	243
Netherlands	131808	Roses - Imanbek Remix	196
New Zealand	131808	Roses - Imanbek Remix	196
Nicaragua	148644	Sweet Night	393
Norway	50796	Fairytale of New York (feat. Kirsty MacColl)	214
Panama	148644	Sweet Night	595
Paraguay	35851	Dance Monkey	228
Peru	35851	Dance Monkey	232

Philippines	134814	Sana	361
Poland	73326	Impreza	168
Portugal	126488	Quando a vontade bater (Participa‘ğ‘čo especial de PK Delas)	298
Romania	131808	Roses - Imanbek Remix	196
Russia	13154	Astronaut In The Ocean	136
Saudi Arabia	148644	Sweet Night	319
Singapore	191782	夜にける	259
Slovakia	131808	Roses - Imanbek Remix	252
South Africa	175518	You’re the One	256
South Korea	113570	OHAYO MY NIGHT	148
Spain	35851	Dance Monkey	215
Sweden	77278	Jerusalema (feat. Nomcebo Zikode)	276
Switzerland	131808	Roses - Imanbek Remix	211
Taiwan	1231	想你想你想你(想你片尾曲)	245
Thailand	177538	comethru	302
Turkey	137312	Seni Dert Etmeler	346
Ukraine	179389	toxin	159
United Arab Emirates	143230	Someone You Loved	269
United Kingdom	50796	Fairytale of New York (feat. Kirsty MacColl)	250
United States	13154	Astronaut In The Ocean	204
Uruguay	35851	Dance Monkey	240
Vietnam	88616	Let Me Down Slowly	290
Vietnam	177538	comethru	290

In Figure 4, we present the execution times of the three different implementations for the fourth query.

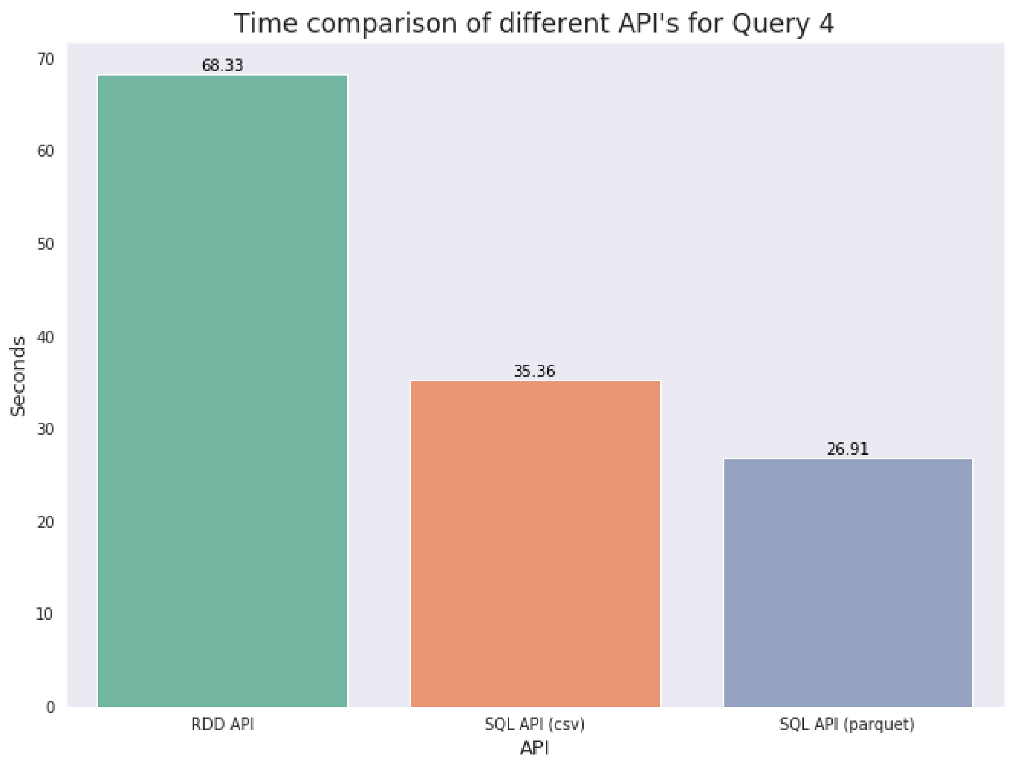

Figure 4: The time comparison for the three different API's for the fourth query. The RDD API executed the query in 68.33, the SQL API (csv) in 35.36 and the SQL API (parquet) in 26.91 seconds.

Below we see the pseudocode for the Map-Reduce developed for the fouth query in the RDD API.

Query 5

Description: Query 5

Artists with maximum mean number of streams in top200 charts for each year.

In Figure 5, we present the execution times of the three different implementations for the fifth query.

Algorithm 4 Query 4 - Map-Reduce implementation in RDD

```
Require: rdd ← rdd(charts.csv), regions ← rdd(regions.csv)
1: rdd.filter(chart_type == "viral50").map ($\mathbf{x} \mapsto \mathbf{x}'$ : {(country_id, song_id, song_name) , 1})
2: rdd.reduceByKey(add).map ($\mathbf{x} \mapsto \mathbf{x}'$ : {(country_id,total_sum), [song_id,song_name]})
3: rdd.groupByKey().mapValues(list)
4: rdd.map ($\mathbf{x} \mapsto \mathbf{x}'$ : {country_id, (total_sum, [song_id,song_name])})
5: rdd.reduceByKey(max).map ($\mathbf{x} \mapsto \mathbf{x}'$ : {(country_id, total_sum), [song_id,song_name]})
6: rdd.flatMapValues($x$ : return($x$))
7: rdd.map. ($\mathbf{x} \mapsto \mathbf{x}'$ : {country_id, ([song_id,song_name] , total_sum)}) .join(regions)
8: rdd.map ($\mathbf{x} \mapsto \mathbf{x}'$ : {country_name, song_id, song_name, total_sum}) .sortBy($\mathbf{x}$ :
   ($x[0], x[1], x[2], x[3]$))
9: return rdd.
```

Year	Artist	Max mean number of streams, top200 chart
2017	Ed Sheeran	6.226326266666664E7
2018	Post Malone	6.812695868115942E7
2019	Post Malone	6.628325392753623 4E7
2020	Bad Bunny	7.794363488405797E7
2021	Olivia Rodrigo	6.446307111594203E7

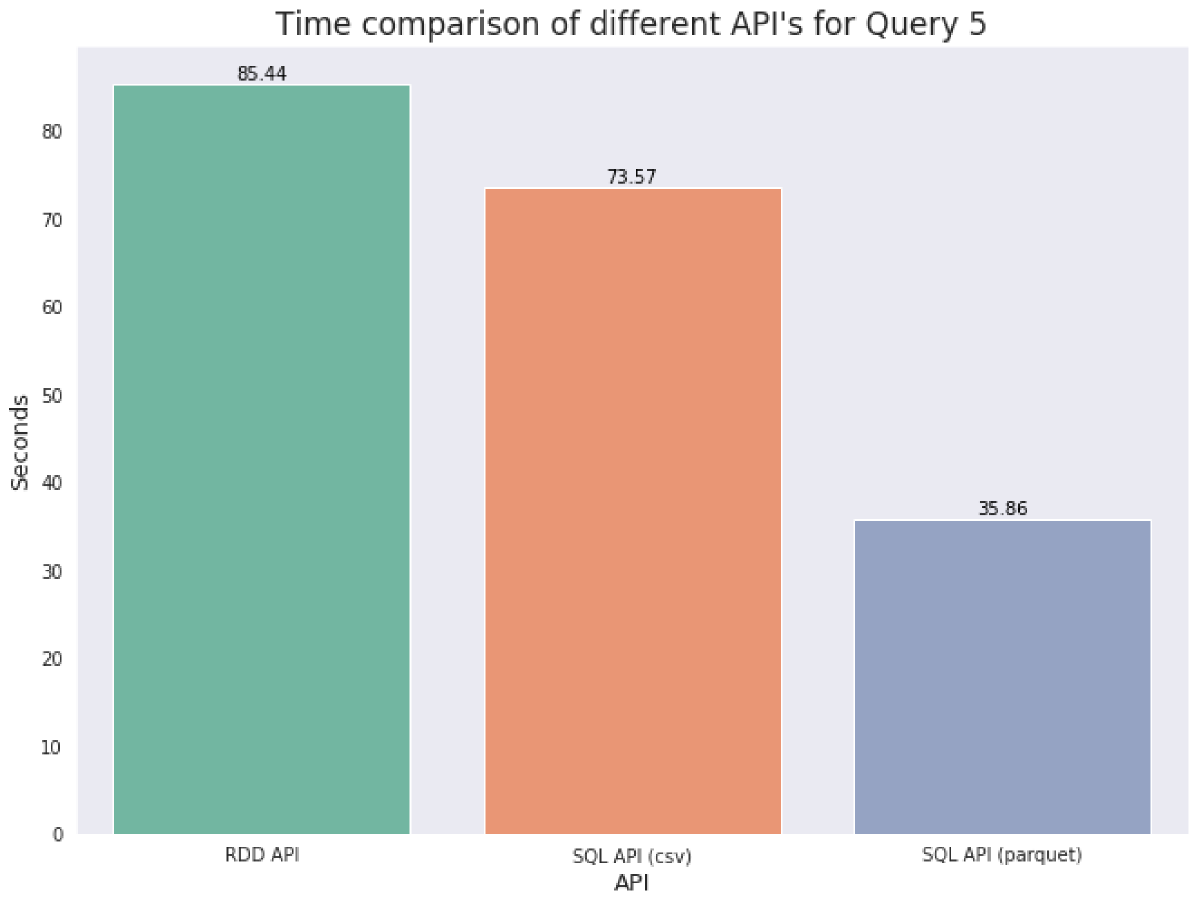

Figure 5: The time comparison for the three different API's for the fifth query. The RDD API executed the query in 85.44, the SQL API (csv) in 73.57 and the SQL API (parquet) in 35.86 seconds.

Below we see the pseudocode for the Map-Reduce developed for the fifth query in the RDD API.

Algorithm 5 Query 5 - Map-Reduce implementation in RDD

```
Require: rdd ← rdd(charts.csv), mapping ← rdd(chart_artist_mapping.csv),
    artists ← rrd(artists.csv)
 1: rdd.filter(chart_type == "top200" and number_of_streams != "")
 2: rdd.map ($\mathbf{x} \mapsto \mathbf{x}'$ : {(song_id,year) , (number_of_streams)})
 3: rdd.reduceByKey(add).map ($\mathbf{x} \mapsto \mathbf{x}'$ : {(song_id), (total_streams_sum,year)})
 4: rdd.join(mapping).rdd.map ($\mathbf{x} \mapsto \mathbf{x}'$ : {(artist_id,year) , (total_streams_sum)})
 5: rdd.reduceByKey(add).map ($\mathbf{x} \mapsto \mathbf{x}'$ : {(artist_id,year), (mean_year_streams_sum)})
 6: rdd.map ($\mathbf{x} \mapsto \mathbf{x}'$ : {(year) , (mean_year_streams_sum, artist_id)}) .reduceByKey(max)
 7: rdd.map ($\mathbf{x} \mapsto \mathbf{x}'$ : {(artist_id), (year, max_mean_year_streams_sum)}) .join(artists)
 8: rdd.map ($\mathbf{x} \mapsto \mathbf{x}'$ : {(year) , (artist_name,max_mean_year_streams_sum)})
 9: rdd.sortByKey().map ($\mathbf{x} \mapsto \mathbf{x}'$ : {(year, artist_name,max_mean_year_streams_sum )})
10: return rdd.
```

Query 6

Description: Query 6

Artist(s) with maximum number of consecutive days in #1 position for any of their songs in Greek charts for each year.

Chart	**Year**	**Artist**	**Max number of consecutive days in #1 in Greek charts**
top200	2021	Saske	79
top200	2021	Rack	79
top200	2020	Roddy Ricch	47
top200	2019	iLLEOo	78
top200	2019	Ypo	78
top200	2019	Sin Boy	78
top200	2019	Mad Clip	78
top200	2018	Drake	67
top200	2017	Ed Sheeran	107
viral50	2021	Masked Wolf	34
viral50	2020	CJ	45
viral50	2019	Trevor Daniel	37
viral50	2018	Gigi D'Agostino	29
viral50	2018	Dynoro	29
viral50	2017	Post Malone	13
viral50	2017	21 Savage	13

In Figure 6, we present the execution times of the three different implementations for the sixth query.

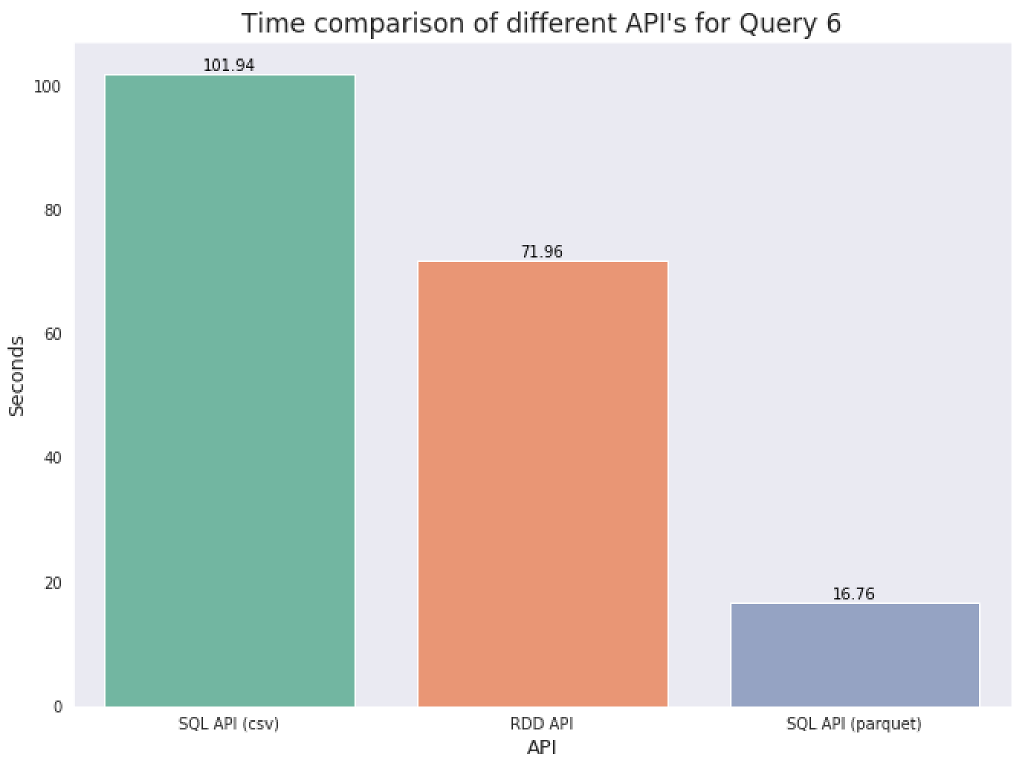

Figure 6: The time comparison for the three different API's for the sixth query. The RDD API executed the query in 101.94, the SQL API (csv) in 71.96 and the SQL API (parquet) in 16.76 seconds.

In the next page we provide the implementation of the sixth query in pseudocode for the Map-Reduce in the RDD API.

Algorithm 6 Query 6 - Map-Reduce implementation in RDD

```
Require: rdd ← rdd(charts.csv), artists ← rdd(artists.csv)
Require: mapping ← rdd(chart_artist_mapping.csv), regions ← rdd(regions.csv)
 1: rdd.filter(song_position == "1" and song_move == "SAME_POSITION")
 2: rdd.map $(\mathbf{x} \mapsto \mathbf{x}' : \{$country_id, (song_id, year, chart_type)$\})$
 3: rdd.join(regions.filter(song_name == "Greece"))
 4: rdd.map $(\mathbf{x} \mapsto \mathbf{x}' : \{$(song_id, year, chart_type), 1$\})$ .reduceByKey$(x, y : x + y)$
 5: rdd.map $(\mathbf{x} \mapsto \mathbf{x}' : \{$song_id, (year, chart_type, total_sum)$\})$
 6: rdd.join(mapping).map$(\mathbf{x} \mapsto \mathbf{x}' : \{$(artist_id, (year, chart_type, total_sum)$\})$.join(artists)
 7: rdd.map $(\mathbf{x} \mapsto \mathbf{x}' : \{$(chart_type,year, artist_name), total_sum$\})$
 8: rdd.map $(\mathbf{x} \mapsto \mathbf{x}' : \{$(chart_type, year, total_sum), artist_name$\})$ .groupByKey()
 9: rdd.mapValues(list).map $(\mathbf{x} \mapsto \mathbf{x}' : \{$(chart_type, year), (total_sum, artist_name)$\})$
10: rdd.reduceByKey(max).map $(\mathbf{x} \mapsto \mathbf{x}' : \{$(chart_type, year, artist_name), total_sum$\})$
11: rdd.flatMapValues$(\mathbf{x} : return(\mathbf{x}))$
12: rdd.map. $(\mathbf{x} \mapsto \mathbf{x}' : \{$chart_type, year, artist_name, total_sum$\})$ .sortBy(x:  (x[0],x[1])
13: return rdd.
```

From the previously presented bar plots, it becomes apparent that using the parquet format for the input files results in a significant reduce in time required for the query execution and thus it is preferred to have the input files in a parquet format instead of a csv.

Parquet is a file format that loads tabular data in an optimized manner for input and output operations while also minimizing the required memory size of the dataset. Parquet files also include information about the dataset including statistical properties among other. Csv files are loaded with an inferSchema option enabled in order to parse data column types properly and not all as string type. However, parquet files do not this require this since they are self-describable.

5 Query optimizer

In this final section, we experiment with the join optimizer of SparkSQL in order to compare the physical execution plan generated by default and by deactivating it. We also compare the execution time of a script (*join_selection_benchmark.py*) with the optimizer enabled (default) or deactivated.
SparkSQL's join optimizer by default checks the ability to broadcast one of two tables to be joined to all worker nodes so that it can be loaded in their RAM. In case the broadcast can be performed, then a broadcast/hash join is added to the physical plan. This is the fastest type of join, therefore by disabling the join optimizer, a reduce-side join will be performed instead, which is the most general type of join in map-reduce, however, it is also the slowest one.
For demonstration purposes, the *charts.csv* tabular dataset is joined with the *regions.csv* dataset with the optimizer enabled as well as disabled. Spark's *spark.sql.autoBroadcastJoinThreshold* sets a threshold in memory size (bytes) for the smaller dataset with a default value of 10MB. To disable the (broadcast) join optimizer, we set its value to -1 by using
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1).
Executing the script we also observe the two physical execution plans. The one with the optimizer enabled contains a broadcast join, whereas the one with the disabled optimizer contains a sort-merge (or reduce-side) join. Below, we compare the two execution times in a barplot chart.

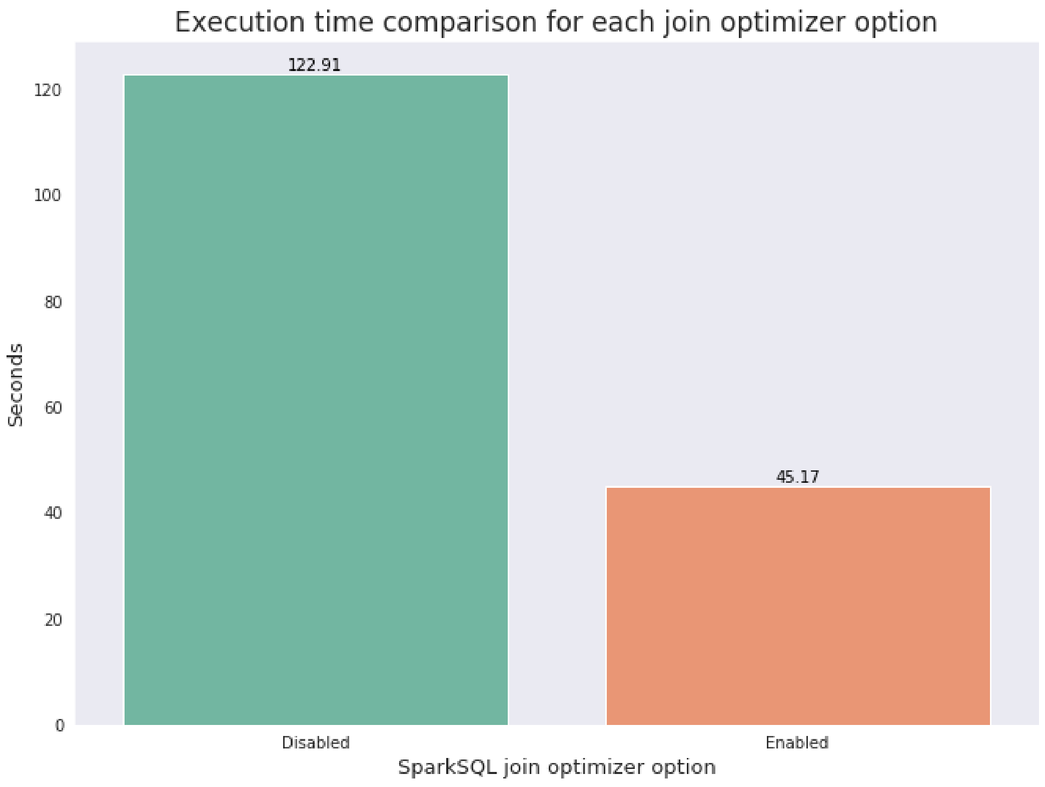

Figure 7: The execution time comparison of *join_selection_benchmark.py* with the SparkSQL join optimizer enabled and disabled.

References

[1] Bill Chambers and Matei Zaharia. *Spark: The definitive guide: Big data processing made simple*. " O'Reilly Media, Inc.", 2018.

[2] Konstantin Shvachko et al. "The hadoop distributed file system". In: *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee. 2010, pp. 1–10.