National Technical University of Athens
School of Electrical and Computer Engineering
Inter-Faculty Postgraduate Studies Programme
Data Science and Machine Learning

# Applications of Apache Spark in Databases

An assignment written in partial fulfillment of the requirements for the completion of the Big Data core course of the Data Science & Machine Learning post-graduate studies programme.

*Instructors*
Dr. I. Konstantinou
N. Provatas

[redacted]

July 23, 2022

# Table of Contents

## 1 Introduction

The recent inflation of data volumes utilized in industry and research poses tremendous computational challenges, since data sizes have outpaced the capabilities of single machines and therefore new systems and architectures are required to scale out computations to multiple nodes. The purpose of the present assignment is an educational-scale introduction to Apache Spark [1]: perhaps the most efficient (to date) open-source unified analytics engine for large-scale data processing. To this end, a dataset containing data scraped from Spotify is utilized, in order to perform a series of analytics tasks. These tasks are carried out in the context of the MapReduce [2] framework implemented on Spark's RDD [3] objects, as well as in the context of SQL queries performed using Spark's DataFrame API [4]. In what follows, we present a comparative analysis with respect to the queries' execution times following each approach. In addition, as far as the SQL queries are concerned, we execute them by parsing both `.csv`, as well as `.parquet` format files, in order to investigate the potential speedup that the latter induce in the total execution time. Finally, we dedicate a section to discuss the results of an experiment designed to highlight the impact of Spark's DataFrame join optimizer on query execution times.

## 2 Parquet File Format

The first step of the experimental procedure consists of setting up the virtual machines provided by Okeanos (GRNET's cloud service) and installing Spark. Afterwards, we create two directories inside HDFS: a directory named `files`, where we move the four `.csv` files that correspond to the given dataset and a directory named `outputs`, where the results of our experiments will be printed. Before proceeding with the development of the code for the required queries, we load the `.csv` files and transform them into `.parquet` format files, so that the SQL Queries (see Section 3) can be performed in both file formats.

Parquet is a data file format, provided by Apache, which loads a tabular object into a columnar form in a way that ensures the optimization of I/O tasks and the minimization of memory usage. In addition, `.parquet` files are self-describing, in the sense that they encapsulate additional information about the tabular object they represent, such as statistical properties or details about its schema. The code shown in Snippet 1 is an example of one "`.csv` into `.parquet` loading-transformation process", while the full code is given alongside this report in the `conversion.py` file.

```python
# Load the .csv as a DataFrame
artists = spark.read.option("header","false").option("inferSchema","true").\
      csv(path+"files/artists.csv")

# Rename its columns
artists = artists.withColumnRenamed("_c0","ART_ID").\
          withColumnRenamed("_c1","ART_NAME")

# Store as Parquet file
artists.write.parquet(path+"files/artists.parquet")
```

Python Code Snippet 1: Example of .parquet file transformation

We need to make two remarks as far as this process is concerned. First, note that the option `inferSchema` is invoked during the loading of the `.csv` file, in order to ensure that the resulting DataFrame's column types are well defined (otherwise all data are parsed as strings). However, this is not required when saving the file in `.parquet` form, due to the aforementioned characteristic of self-describability. The second remark is that throughout our analysis we use renamed columns for each feature, so that the corresponding code is more readable. The new column names for each `.csv` file can be seen in Appendix C.

## 3 Comparative Analysis & Execution Times

With all the `.csv` and `.parquet` files inside the `files` directory, we proceed with the development of the code for the data analytics queries. In what follows, each Query is discussed separately in its own sub-section. However, one common element (which will not be repeated henceforth) is that the map transformation

```
map(lambda x: list(csv.reader([x], delimiter=',', quotechar=""))[0])
```

is used during the parsing of each `.csv` file to be used with the RDD API. This is done to avoid reading the comma as a delimiter in song names. Note that the code provided in the following snippets corresponds only to the map-reduce type of queries, as directed by the assignment's instructions. The full code for each type of query is given alongside this report in the `rdd_api.py`, `sql_csv.py` and `sql_parquet.py` files.

### 3.1 Query No. 1

The code developed for the first query using the RDD API is shown in Snippet 2. The `rdd` object corresponds to the parsed `.csv` file, while the dummy `x` corresponds to an array, the element indices of which are the indices of the parsed `.csv` file's columns. For example, `x[1]` corresponds to the **`SONG_NAME`** column, using the names presented in Appendix C. First, a filter is performed in order to keep the relevant entries with respect to the given song and chart, and then the map-reduce procedure simply sums the number of streams.

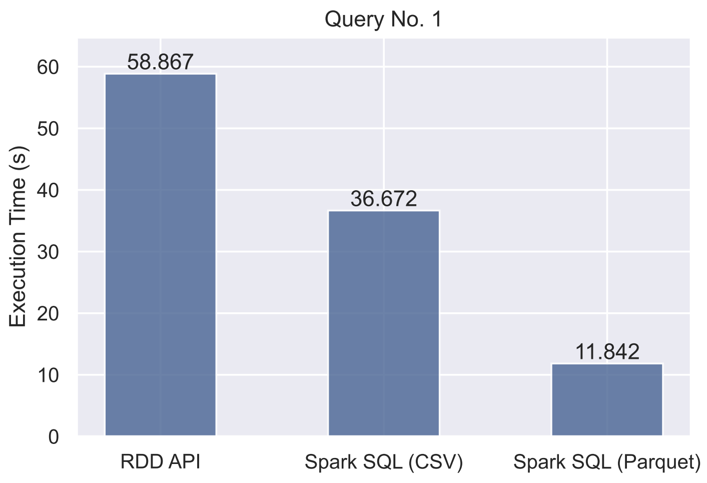

Figure 3.1: The execution times for Query No. 1.

```
rdd.filter(lambda x : (x[1]=="Shape of You") and (x[5]=="top200")).\
    map(lambda x: int(x[7])).reduce(lambda a,b: a+b)
```

Python Code Snippet 2: Map-Reduce code developed for the first query

The execution time for this query is 58.867 seconds and the result is the integer 2324245979, corresponding to the total number of streams for the song “Shape of You”, according to the top200 charts. Obviously, the SQL queries performed using the DataFrame API yield the same result, however the execution times differ: the query on the `.csv` file requires 36.672 seconds and the query for the `.parquet` file requires 11.842 seconds. These results are summarized in the barplot shown in Fig. 3.1.

## 3.2 Query No. 2

The main map-reduce code developed for the second query using the RDD API is shown in Snippet 3. The 1st-place entries of the top200 chart are filtered and only the required columns are kept. Then, the number of each song’s appearance is counted using the `countByKey` function. The results are subsequently sorted and divided by 69, in order to produce the song with the longest average time at 1st position for the top200 chart. A completely analogous procedure is followed for the viral50 chart.

```python
rdd.filter(lambda x: (x[2]=='1') and (x[5]=="top200")).\
    map(lambda x: [x[1], x[5]]).countByKey().items()
```

Python Code Snippet 3: Map-Reduce code developed for the second query

The execution time for this query is 112.784 seconds and the results are shown in Table 3.1.

| **Chart** | **Song** | **Avg. Days No. 1** |
|---|---|---|
| viral50 | Calma - Remix | 24.985507 |
| top200 | Shape of You | 54.246377 |

Table 3.1: Results of Query No. 2.

The SQL queries performed using the DataFrame API yield the same result, with the query on the `.csv` file requiring 54.869 seconds and the query for the `.parquet` file requiring 22.890 seconds to run. These results are summarized in the barplot shown in Fig. 3.2.

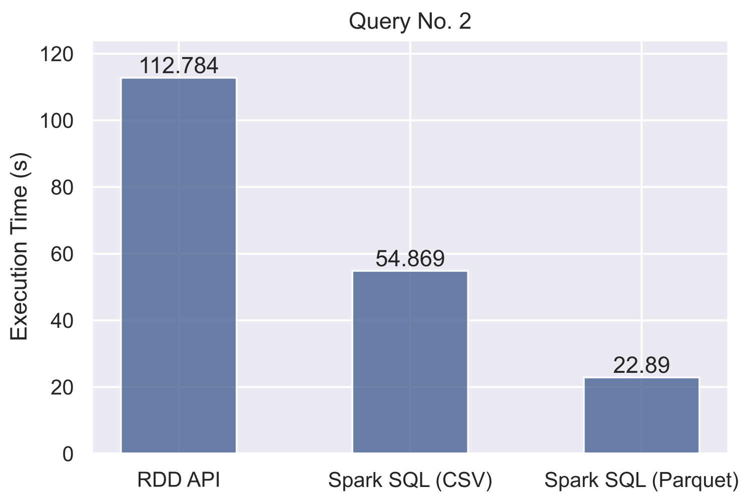

Figure 3.2: The execution times for Query No. 2.

## 3.3 Query No. 3

The code developed for the third query using the RDD API is shown in Snippet 4. First, a filtering keeps only the entries that have appeared at the top of the required chart and then keys are created based on year, month and day. Afterwards, an aggregation is performed on the number of streams for each year-month pair. A workaround to keep track of the number of each month's days is their substitution with the number 1 and a subsequent summing thereof. The first `reduceByKey` function's output is divided by the second's, in order to get a per-month average and finally the results are sorted by year and month, respectively.

```python
rdd.filter(lambda x: (x[5]=="top200") and (x[2]=='1')).\
    map(lambda x: ((int((x[3].split('T')[0][:4])),\
        int(x[3].split('T')[0][5:7]),\
        int(x[3].split('T')[0][8:])),\
        int(x[7]))).\
    reduceByKey(lambda x,y: x+y).\
    map(lambda x: ((x[0][0], x[0][1]), (1, x[1]))).\
    reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).\
    map(lambda x: (x[0][0], x[0][1], x[1][1]/x[1][0])).\
    sortBy(lambda x : (x[0], x[1]))
```

Python Code Snippet 4: Map-Reduce code developed for the third query

The execution time for this query is 197.265 seconds. Due to their length, the results are shown in Table A.1 of Appendix A and they correspond to each month's song with the highest number of average daily streams on the top200 charts. As for the corresponding SQL queries, the one performed on the `.csv` file is executed successfully after 64.454 seconds, while the one performed on the `.parquet` file requires 6.348 seconds. These results are summarized in the barplot shown in Fig. 3.3.

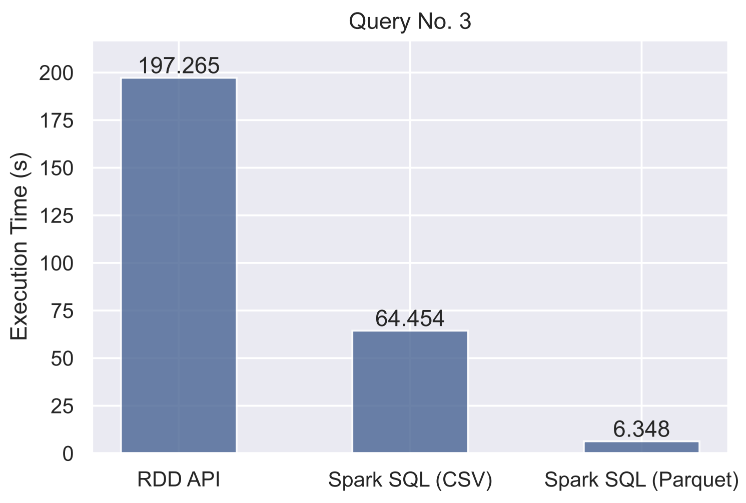

Figure 3.3: The execution times for Query No. 3.

## 3.4 Query No. 4

The code developed for the fourth query using the RDD API is shown in Snippet 5. After filtering on the required chart, we create keys based on region, song ID and song name and perform the aforementioned workaround where we use the number 1 as a placeholder value in order to sum over it. Afterwards, a grouping-by country is performed, followed by a reduction on the country ID key, in order to find the required maximum values with regards to the songs' frequencies. We utilize the flatMapValues function in order to split the entries with the same frequency and perform a join procedure in order to be able to print each country's name based on its ID. Note that the countries object that appears in Snippet 5 is simply the name of the RDD containing the parsed information from the regions.csv file. Finally, the results are formatted according to the format given by the assignment's instructions.

```
rdd.filter(lambda x: x[5] == "viral50").\
    map(lambda x: ((x[4], x[0], x[1]), 1)).reduceByKey(add).\
    map(lambda x: ((x[0][0],x[1]),[x[0][1], x[0][2]])).\
    groupByKey().mapValues(list).\
    map(lambda x: (x[0][0], (x[0][1], x[1]))).\
    reduceByKey(max).map(lambda x: ((x[0], x[1][0]), x[1][1])).\
    flatMapValues(lambda x: x).map(lambda x: (x[0][0], (x[1], x[0][1]))).\
    join(countries).\
    map(lambda x : (x[1][1], int(x[1][0][0][0]), x[1][0][0][1], x[1][0][1])).\
    sortBy(lambda x: x)
```

Python Code Snippet 5: Map-Reduce code developed for the fourth query

This query runs in 127.154 seconds and its results are shown in Table B of Appendix B due to their length. The output corresponds to the song with the longest presence at the viral50 charts per country. The Spark SQL query on the .csv file requires 109.438 seconds to run and the query on the .parquet file requires 34.214 seconds. The results are summarized in the barplot shown in Fig. 3.4.

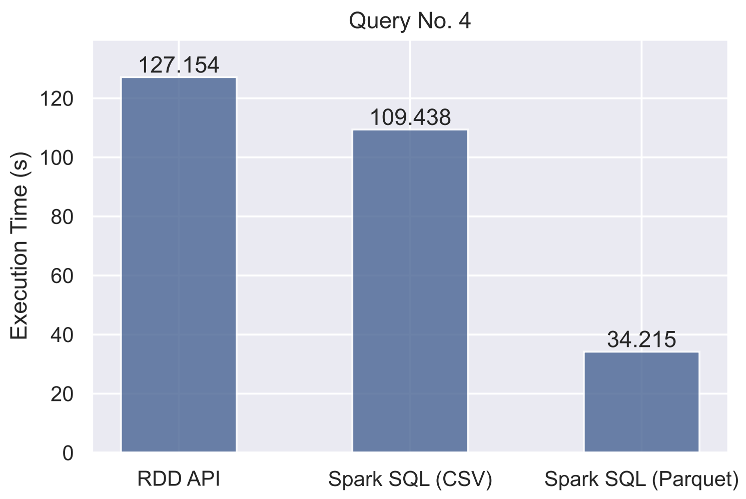

Figure 3.4: The execution times for Query No. 4.

## 3.5 Query No. 5

The code developed for the fifth query using the RDD API is shown in Snippet 6. This query concerns the artist with the highest number of average streams per year in the top200 charts and all functions applied herein have already been discussed in previous queries (the methods used in this query are a combination of the ones used in queries No. 3 and 4). The only remark worth making is that the mapping and artists objects that appear in Snippet 6 are simply the names of the RDDs containing the parsed information from the chart_artist_mapping.csv and artists.csv files, respectively.

```python
rdd.filter(lambda x: (x[5] == "top200") and (x[7]!="")).\
    map(lambda x: ((x[0], x[3][:4]), int(x[7]))).\
    reduceByKey(add).map(lambda x: (x[0][0], (x[1], x[0][1]))).\
    join(mapping).map(lambda x: ((x[1][1], x[1][0][1]), x[1][0][0])).\
    reduceByKey(add).map(lambda x: (x[0], x[1]/69)).\
    map(lambda x: (x[0][1], (x[1], x[0][0]))).\
    reduceByKey(max).map(lambda x: (x[1][1], (x[0], x[1][0]))).\
    join(artists).map(lambda x: (x[1][0][0], (x[1][1], x[1][0][1]))).\
    sortByKey().map(lambda x: (x[0], x[1][0], x[1][1]))
```

Python Code Snippet 6: Map-Reduce code developed for the fifth query

The present query's execution time is equal to 207.844 seconds and its results are shown in Table 3.2.

| Year | Artist | Avg. Streams |
|---|---|---|
| 2017 | Ed Sheeran | 62263262.666667 |
| 2018 | Post Malone | 68126958.681159 |
| 2019 | Post Malone | 66283253.927536 |
| 2020 | Bad Bunny | 77943634.884058 |
| 2021 | Olivia Rodrigo | 64463071.115942 |

Table 3.2: Results of Query No. 5.

The SQL queries performed using the DataFrame API yield the same result, with the query on the `.csv` file requiring 120.228 seconds and the query for the `.parquet` file requiring 30.423 seconds to run. These results are summarized in the barplot shown in Fig. 3.5.

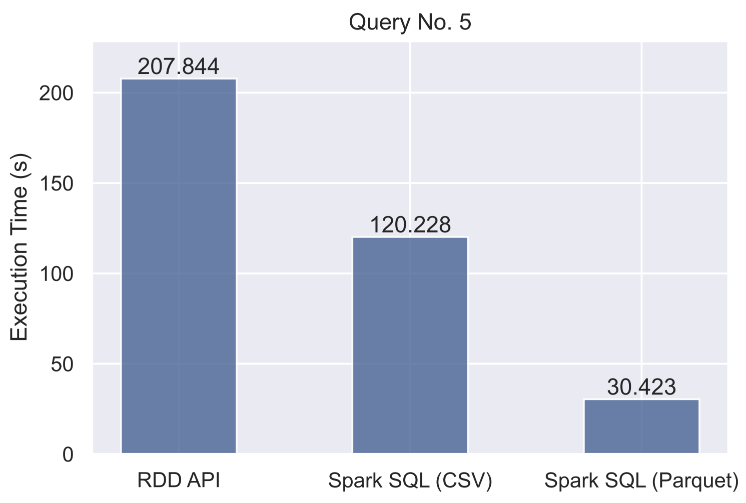

Figure 3.5: The execution times for Query No. 5.

## 3.6 Query No. 6

As far as the sixth query is concerned, we need to make a few remarks before moving on to the map-reduce code developed for its execution. To our mind, the assignment's instructions are ambiguous when it comes to the sixth query, which is why we contacted the instructors via email on July 2, 2022, since no forum was constructed for the assignment (at least to our knowledge). Having received no reply as of July 23, 2022, we found two possible interpretations for the sixth query.

The first among the two possible interpretations is that we are asked to find the maximum number of consecutive days for which an artist's song has remained on the top of a chart during a year. While this interpretation does not explain the "most times" (*περισσότερες φορές*) part of the instructions, its complexity justifies the fact that 20% of the whole assignment's grade is allocated to it: in terms of the SQL implementation, this task requires several filterings and partition windows which are in general considered to be higher-than-introductory level. This version of the query was implemented using the Spark SQL API only and can be found commented out in the `sql_csv.py` and `sql_parquet.py` files. Its results are shown in Table 3.3.

| Chart | Year | Artist | Max Consecutive No. 1 |
|---|---|---|---|
| top200 | 2017 | Ed Sheeran | 108 |
| top200 | 2018 | Drake | 68 |
| top200 | 2019 | iLLEOo | 74 |
| top200 | 2019 | Ypo | 74 |
| top200 | 2019 | Sin Boy | 74 |
| top200 | 2019 | Mad Clip | 74 |
| top200 | 2020 | SNIK | 28 |
| top200 | 2021 | Light | 35 |
| top200 | 2021 | Hawk | 35 |
| viral50 | 2017 | Louis Tomlinson | 13 |
| viral50 | 2018 | Ariana Grande | 23 |
| viral50 | 2019 | Trevor Daniel | 38 |
| viral50 | 2020 | Trevor Daniel | 43 |
| viral50 | 2021 | Masked Wolf | 30 |
| viral50 | 2021 | Shouse | 30 |

Table 3.3: Results of Query No. 6 based on the first interpretation.

It becomes evident that the sample results given in the assignment's instructions are not confirmed. This implies that our first interpretation of the query was incorrect, thus leaving us with the second interpretation: we need to find the number of separate **times** that a song appeared on the top of a chart during a year for consecutive days. For example, following this interpretation, a song that remained on the top for 70 consecutive days and then never returned gets a count of 1, while a song that remained on the top for 4 consecutive days, was replaced by another song and then a few days later returned on the top for 5 more consecutive days gets a count of 2. While this interpretation is closer to the assignment's instructions (and simpler to implement compared to the first interpretation), it also fails to reproduce the sample results.

These observations lead us to a third interpretation of the query's instructions: we need to find the maximum number of distinct pairs of consecutive days (as in a Level-2 Markov Chain) for which a song appears on the top of a chart during a year. For example, the value for a song that remained on the top for 4 consecutive days, was replaced by another song and then a few days later returned on the top for 5 more consecutive days gets a count of $(4 - 1) + (5 - 1) = 7$. To be honest, this interpretation was not obvious to us, which is why we had to "reverse-engineer" the sample results in order to arrive at possible interpretations. While this third interpretation is the easiest to implement compared to the first two, it seems to be the only one that reproduces the sample results. We therefore assume that this is the required version of the query and move on to present the map-reduce code developed for its execution using the RDD API (see Snippet 7).

| **Chart** | **Year** | **Artist** | **Most Times No. 1** |
|---|---|---|---|
| top200 | 2017 | Ed Sheeran | 107 |
| top200 | 2018 | Drake | 67 |
| top200 | 2019 | iLLEOo | 78 |
| top200 | 2019 | Ypo | 78 |
| top200 | 2019 | Sin Boy | 78 |
| top200 | 2019 | Mad Clip | 78 |
| top200 | 2020 | Roddy Ricch | 47 |
| top200 | 2021 | Saske | 79 |
| top200 | 2021 | Rack | 79 |
| viral50 | 2017 | Post Malone | 13 |
| viral50 | 2017 | 21 Savage | 13 |
| viral50 | 2018 | Gigi D'Agostino | 29 |
| viral50 | 2018 | Dynoro | 29 |
| viral50 | 2019 | Trevor Daniel | 37 |
| viral50 | 2020 | CJ | 45 |
| viral50 | 2021 | Masked Wolf | 34 |

Table 3.4: Results of Query No. 6 based on the interpretation that reproduces the sample results.

While the functions utilized for the execution of this query have already been discussed, we provide a basic description of Snippet 7, so that our final interpretation of the query becomes clear. First, the entries that have remained in the first position without moving are filtered (which corresponds to the subtraction by 1 performed in the example of the previous paragraph, since we simply don't account for the `MOVE_UP` action that brings the song at the top). Then, a join operation is performed in order to get results only for Greece. The aforementioned workaround using the digit 1 is employed in order to track the frequency of each song ID in each chart's first position and then two join operations are performed using the remaining two `.csv` files, in order to perform a mapping between song ID and artist name. The results are grouped by chart and year, while the calculated frequencies and the corresponding artists are set as key-value pairs. Finally, a maximum aggregation is performed using the frequency keys, in order to draw the maximum frequency corresponding to each chart and year alongside the artists' names.

```
rdd.filter(lambda x: x[2] =="1" and x[6] == "SAME_POSITION").\
    map(lambda x: (x[4], (x[0], x[3][:4], x[5]))).\
    join(countries.filter(lambda x: x[1] == "Greece")).\
    map(lambda x: (x[1][0],1)).reduceByKey(lambda x,y: x+y).\
    map(lambda x: (x[0][0], (x[0][1], x[0][2], x[1]))).\
    join(mapping).map(lambda x: (x[1][1], x[1][0])).join(artists).\
    map(lambda x: ((x[1][0][1], x[1][0][0], x[1][1]), x[1][0][2])).\
    map(lambda x: ((x[0][0], x[0][1], x[1]), x[0][2])).\
    groupByKey().mapValues(list).\
    map(lambda x: ((x[0][0], x[0][1]), (x[0][2], x[1]))).reduceByKey(max).\
    map(lambda x: ((x[0][0], x[0][1], x[1][0]), x[1][1])).\
    flatMapValues(lambda x: x).\
    map(lambda x: (x[0][0], x[0][1], x[1], x[0][2])).\
    sortBy(lambda x: (x[0], x[1]))
```

Python Code Snippet 7: Map-Reduce code developed for the sixth query

The execution time for the query using the RDD API is equal to 77.441 seconds and the corresponding results are shown in Table 3.4. As for the Spark SQL queries of this interpretation, the query on the `.csv` file is executed after 70.362 seconds, while the query on the `.parquet` file is executed after 15.132 seconds. These results are summarized in the barplot of Fig. 3.6.

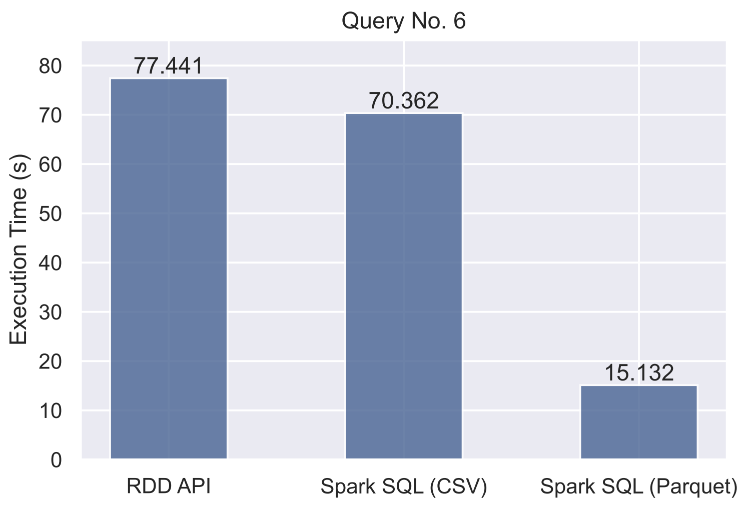

Figure 3.6: The execution times for Query No. 6.

# 4 Spark's Join Optimizer

This final part of the experimental procedure involves a comparison on the physical execution plan produced by Apache Spark and its corresponding execution time when its join optimizer is enabled, versus when it is not. Spark's join optimizer works as follows: when the framework is instructed to join two tabular objects, the optimizer's default option is to check if one of the two is "small enough", so that it can be broadcast to all worker nodes without exceeding their memory limits. If this is the case, a hash join (also known as broadcast join) is performed, since it is the most optimized type of join procedure in terms of time complexity. This check for a "small enough" tabular object is configured by the `spark.sql.autoBroadcastJoinThreshold` parameter, which sets the maximum size in bytes for the tabular object and has a default value equal to 10 MB.

For the purposes of our experiment, we attempt to perform a join between the `charts.csv` and the `regions.csv` files on the `REG_ID` column. While the former is a large tabular object, the latter's size is smaller than 1 KB, which indicates that a hash join can be performed to minimize the execution time. By running the `join_optim.py` file, which is provided alongside this report, we find that the execution time required for the join with the optimizer enabled is equal to 29.834 seconds. Furthermore, the physical plan constructed is shown below. Note that `BroadcastHashJoin` has been used, as expected.

```
== Physical Plan ==
(2) Project [SONG_ID#0, SONG_NAME#1, POS#2, DATE#3, REG_ID#4, CHART#5,
ACTION#6, STREAMS#7, REG_NAME#17 AS region_name#39]
+-(2) BroadcastHashJoin [REG_ID#4], [REG_ID#16], Inner, BuildRight
:- (2) Project [SONG_ID#0, SONG_NAME#1, POS#2, DATE#3, REG_ID#4, CHART#5,
ACTION#6, STREAMS#7]
:  +-(2) Filter isnotnull(REG_ID#4)
:     +- (2) FileScan parquet [SONG_ID#0,SONG_NAME#1,POS#2,
DATE#3,REG_ID#4,CHART#5,ACTION#6,STREAMS#7] Batched: true, Format: Parquet,
Location: InMemoryFileIndex[hdfs://master:9000/files/charts.parquet],
PartitionFilters: [], PushedFilters: [IsNotNull(REG_ID)],
ReadSchema: struct<SONG_ID:int,SONG_NAME:string,POS:int,DATE:timestamp,
REG_ID:int,CHART:string,ACTION:string,...
+- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0,
int, true] as bigint)))
+-(1) Project [REG_ID#16, REG_NAME#17]
+- (1) Filter isnotnull(REG_ID#16)
+-(1) FileScan parquet [REG_ID#16,REG_NAME#17] Batched: true, Format: Parquet,
Location: InMemoryFileIndex[hdfs://master:9000/files/regions.parquet],
PartitionFilters: [], PushedFilters: [IsNotNull(REG_ID)],
ReadSchema: struct<REG_ID:int,REG_NAME:string>
```

The join optimizer can be effectively disabled simply by setting the aforementioned parameter equal to -1. In this case, the framework's default join operation becomes the sort-merge join, which is computationally more time consuming compared to the hash join, however it is more general purpose since it does not require at least one of the two objects to be small in size. By disabling the join optimizer, the execution time required for the join increases to 64.392 seconds and the corresponding physical plan is shown below. Note that the physical plan includes a `SortMergeJoin`, as expected.

```
== Physical Plan ==
(5) Project [SONG_ID#49, SONG_NAME#50, POS#51, DATE#52, REG_ID#53,
CHART#54, ACTION#55, STREAMS#56, REG_NAME#66 AS region_name#88]
+-(5) SortMergeJoin [REG_ID#53], [REG_ID#65], Inner
:- (2) Sort [REG_ID#53 ASC NULLS FIRST], false, 0
:  +- Exchange hashpartitioning(REG_ID#53, 200)
:     +-(1) Project [SONG_ID#49, SONG_NAME#50, POS#51, DATE#52,
REG_ID#53, CHART#54, ACTION#55, STREAMS#56]
:        +- (1) Filter isnotnull(REG_ID#53)
:           +-(1) FileScan parquet [SONG_ID#49,SONG_NAME#50,POS#51,
DATE#52,REG_ID#53,CHART#54,ACTION#55,STREAMS#56] Batched: true, Format: Parquet,
Location: InMemoryFileIndex[hdfs://master:9000/files/charts.parquet],
PartitionFilters: [], PushedFilters: [IsNotNull(REG_ID)],
ReadSchema: struct<SONG_ID:int,SONG_NAME:string,POS:int,
DATE:timestamp,REG_ID:int,CHART:string,ACTION:string,...
+- (4) Sort [REG_ID#65 ASC NULLS FIRST], false, 0
+- Exchange hashpartitioning(REG_ID#65, 200)
+-(3) Project [REG_ID#65, REG_NAME#66]
+- (3) Filter isnotnull(REG_ID#65)
+-(3) FileScan parquet [REG_ID#65,REG_NAME#66] Batched: true, Format: Parquet,
Location: InMemoryFileIndex[hdfs://master:9000/files/regions.parquet],
PartitionFilters: [], PushedFilters: [IsNotNull(REG_ID)],
ReadSchema: struct<REG_ID:int,REG_NAME:string>
```

Closing this final experiment, we present the two execution times in the barplot of Fig. 4.1 for completeness.

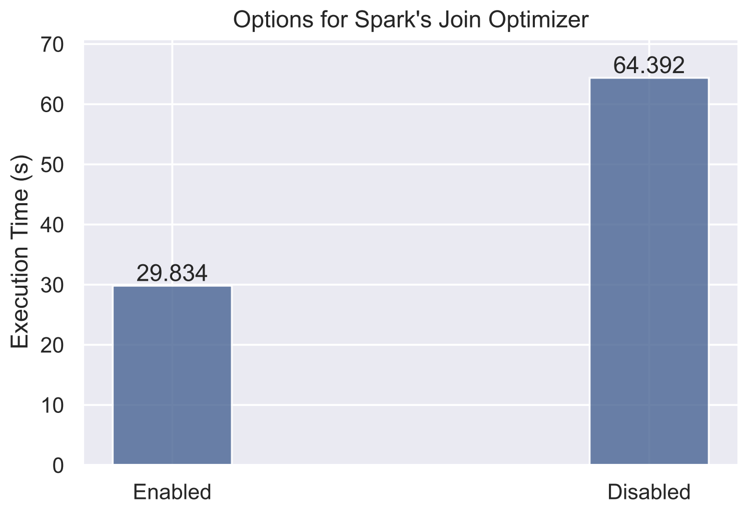

Figure 4.1: Barplot depicting the join's execution time with enabled (left) and disabled (right) optimizer.

## 5 Conclusions

In summation, through this assignment we demonstrated how the framework provided by Apache Spark does not only enable the handling of large datasets in a distributed manner, but also significantly reduces all execution times by performing a series of optimizations. First and foremost, while the MapReduce framework used to dominate the field of data analytics tasks for large volumes, it becomes evident that Spark's DataFrame API is by far superior, as the execution times for all queries using the DataFrame API are considerably smaller compared to the ones where the RDD API was used with map-reduce tasks. The only exception to this rule of thumb is the case of Query No. 6, where the RDD API and the Spark SQL (CSV) execution times are comparable, which can be attributed to a highly optimized map-reduce code on our part (or a not-so-optimized sql code).

One of the main reasons for the superiority of the DataFrame API is the fact that it uses the Catalyst optimizer, which offers rule-based (indicates how to execute the query from a set of defined rules) and cost-based (generates multiple execution plans and compares them to choose the lowest cost one) optimization. To highlight this, we have grouped all the barplots shown in Figs. 3.1-3.6 in a single barplot shown in Fig. 5.1.

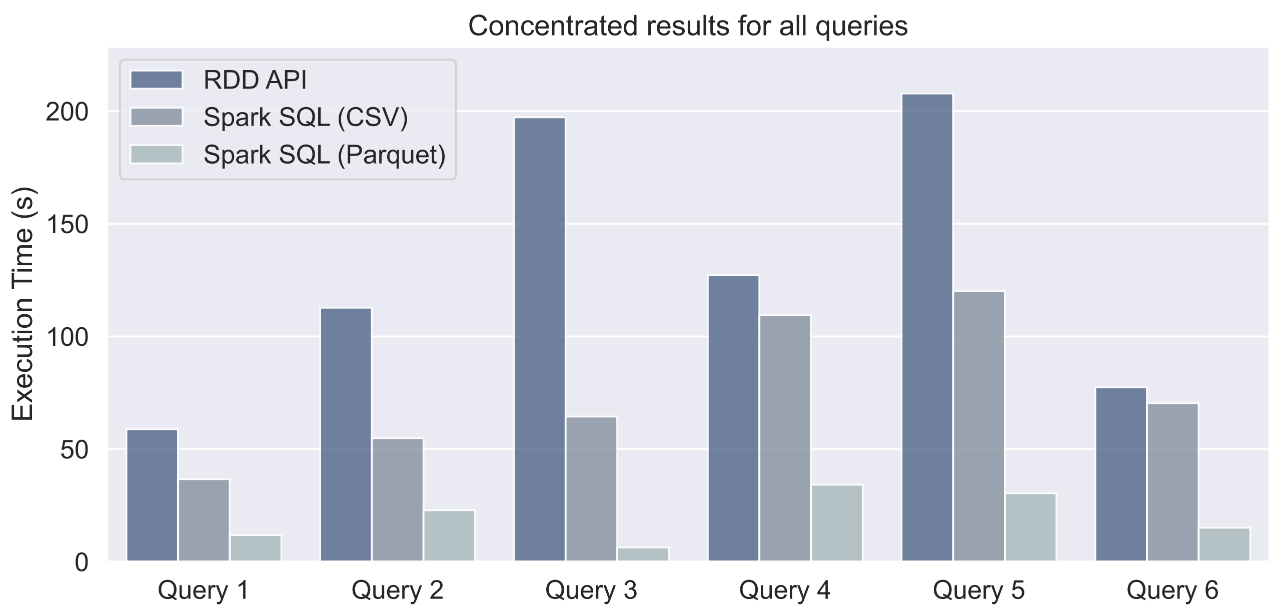

Figure 5.1: A summary for all queries' execution times.

Secondly, the same queries performed with the DataFrame API but on different file formats can also greatly impact the resulting execution time: by using `.parquet` files instead of `.csv` files, we managed to further reduce the total run times, in all cases without exception by a large factor. Last but not least, we showed in practice how various components built in the DataFrame API manage this decrease in total execution time, by enabling and then disabling the join optimizer module and finding an increase of 115.83% in the execution time with the disabled optimizer.

As a final remark, we need to stress the fact that different runs on the Virtual Machine yielded different execution times even when our code was not altered between consecutive executions. This matter was discussed with other teams, who also reported the same issue, which could be attributed to network traffic or factors completely unknown to us. Nonetheless, in order for the above comparisons to be valid, the results shown in this report were acquired from a single run (e.g. we did not execute the SQL (CSV) queries on Monday evening and then the SQL (Parquet) queries on Tuesday noon).

# A Results for Query No. 3

| Year | Month | Avg. Daily Streams | | Year | Month | Avg. Daily Streams |
|---|---|---|---|---|---|---|
| 2017 | 1 | 7618611.064516 | | 2019 | 7 | 11214812.709677 |
| 2017 | 2 | 8876450.785714 | | 2019 | 8 | 10708007.258065 |
| 2017 | 3 | 8955476.419354 | | 2019 | 9 | 10463297.6 |
| 2017 | 4 | 8178985.833333 | | 2019 | 10 | 11164322.903226 |
| 2017 | 5 | 8939831.827586 | | 2019 | 11 | 10746356.433333 |
| 2017 | 6 | 7530758.666667 | | 2019 | 12 | 11001986.903226 |
| 2017 | 7 | 6757058.387097 | | 2020 | 1 | 13611543.0 |
| 2017 | 8 | 6599688.064516 | | 2020 | 2 | 12275200.068966 |
| 2017 | 9 | 7246840.5 | | 2020 | 3 | 11213416.83871 |
| 2017 | 10 | 8138961.129032 | | 2020 | 4 | 9582862.366667 |
| 2017 | 11 | 7578529.066667 | | 2020 | 5 | 9113325.83871 |
| 2017 | 12 | 7539380.677419 | | 2020 | 6 | 9474377.933333 |
| 2018 | 1 | 8135145.774194 | | 2020 | 7 | 10972215.129032 |
| 2018 | 2 | 9710137.928571 | | 2020 | 8 | 12249567.83871 |
| 2018 | 3 | 8713800.0 | | 2020 | 9 | 12756135.466667 |
| 2018 | 4 | 9020981.7 | | 2020 | 10 | 11171908.677419 |
| 2018 | 5 | 8654503.193548 | | 2020 | 11 | 12805923.1 |
| 2018 | 6 | 8953159.9 | | 2020 | 12 | 12044482.354839 |
| 2018 | 7 | 10391241.935484 | | 2021 | 1 | 12761012.483871 |
| 2018 | 8 | 8394892.903226 | | 2021 | 2 | 10721677.142857 |
| 2018 | 9 | 8358084.0 | | 2021 | 3 | 11661322.709677 |
| 2018 | 10 | 8482952.064516 | | 2021 | 4 | 12896025.933333 |
| 2018 | 11 | 10169543.7 | | 2021 | 5 | 15425066.774194 |
| 2018 | 12 | 9371460.225806 | | 2021 | 6 | 16008952.3 |
| 2019 | 1 | 10189024.580645 | | 2021 | 7 | 14274800.677419 |
| 2019 | 2 | 10817727.392857 | | 2021 | 8 | 14186574.0 |
| 2019 | 3 | 9913579.935484 | | 2021 | 9 | 13402214.233333 |
| 2019 | 4 | 10819890.666667 | | 2021 | 10 | 12857921.677419 |
| 2019 | 5 | 10465042.451613 | | 2021 | 11 | 12087256.933333 |
| 2019 | 6 | 10857143.233333 | | 2021 | 12 | 3323677.774194 |

Table A.1: Results of Query No. 3.

## B Results for Query No. 4

| Country | Song ID | Song Name | Days Present |
|---|---|---|---|
| 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 | ⊠⊠⊠⊠⊠ | 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’è | 222 |
| Japan | 191782 | ⊠⊠⊠⊠⊠ | 374 |

| | | | |
|---|---|---|---|
| Latvia | 131808 | Roses - Imanbek Remix | 226 |
| Lithuania | 164633 | Vasarą 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 (Participação 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 | ⌧⌧⌧⌧⌧ | 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 | ”⌧⌧⌧⌧⌧⌧⌧⌧⌧(⌧⌧⌧””⌧⌧⌧””⌧⌧⌧)” | 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 |

Table B.1: Results of Query No. 4.

## C Renamed Columns

We present here the renamed columns for each `.csv` file. For a detailed description of each feature, we refer the reader to the assignment's instructions.

`artists.csv`

```
ART_ID | ART_NAME
```

`charts.csv`

```
SONG_ID | SONG_NAME | POS | DATE | REG_ID | CHART | ACTION | STREAMS
```

`regions.csv`

```
REG_ID | REG_NAME
```

`chart_artist_mapping.csv`

```
SONG_ID | ART_ID
```

## References

[1] B. Chambers and M. Zaharia, *Spark: The Definitive Guide*. O'Reilly Media, Inc, 2018.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Sixth Symposium on Operating System Design and Implementation*, pp. 137–150, 2004. DOI: 10.1145/1327452.1327492.

[3] M. Zaharia, M. Chowdhury, T. Das, *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 15–28, 2012. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf.

[4] M. Armbrust, R. S. Xin, C. Lian, *et al.*, "Spark sql: Relational data processing in spark," *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1383––1394, 2015. DOI: 10.1145/2723372.2742797.