



HELLENIC REPUBLIC
National and Kapodistrian
University of Athens

PROJECT

NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

FACULTY OF INFORMATICS AND TELECOMMUNICATIONS

M111: Big Data Management

Author:

Michael Darmanis* (SID: 7115152200004)

Date: July 20, 2023

*mdarm@di.uoa.gr

Part 1 Introduction

Task 1.1

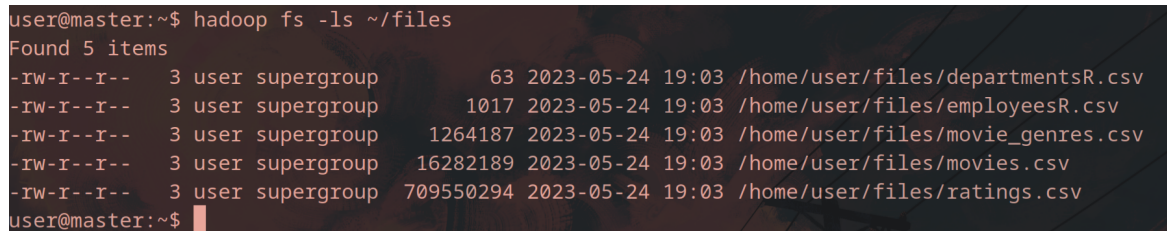
For creating a files directory in the HDFS, the following command was used

```
hadoop fs -mkdir -p ~/files
```

and to populate it with the project's .csv files

```
hadoop fs -put *.csv ~/files .
```

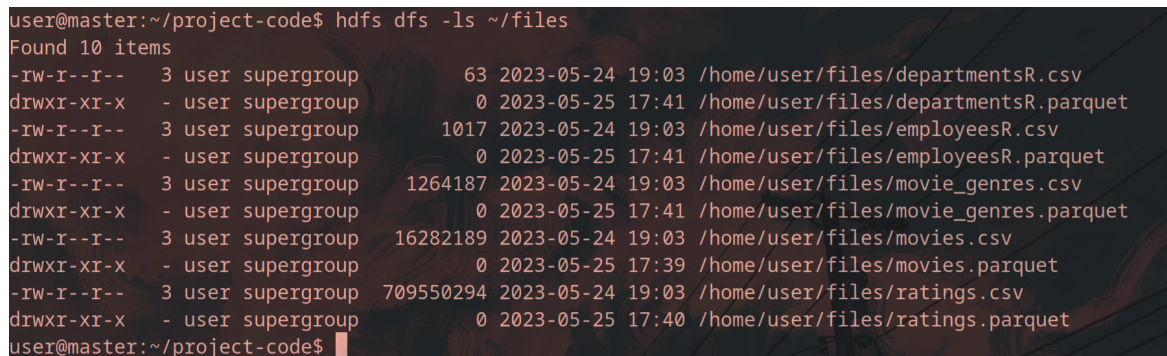
Since the only .csv files populating the working directory were the project's ones, it was fairly easy to fetch all of them using *.csv. Fig. 1 shows a print-screen of the .csv files lying within the created files directory.



```
user@master:~$ hadoop fs -ls ~/files
Found 5 items
-rw-r--r--  3 user supergroup          63 2023-05-24 19:03 /home/user/files/departmentsR.csv
-rw-r--r--  3 user supergroup       1017 2023-05-24 19:03 /home/user/files/employeesR.csv
-rw-r--r--  3 user supergroup    1264187 2023-05-24 19:03 /home/user/files/movie_genres.csv
-rw-r--r--  3 user supergroup    16282189 2023-05-24 19:03 /home/user/files/movies.csv
-rw-r--r--  3 user supergroup   709550294 2023-05-24 19:03 /home/user/files/ratings.csv
user@master:~$
```

Figure 1: HDFS directory containing the project's datasets in .csv format.

The same files were then saved in .parquet format, using Snip. 1, as can be seen in the print-screen of Fig 2.



```
user@master:~/project-code$ hdfs dfs -ls ~/files
Found 10 items
-rw-r--r--  3 user supergroup          63 2023-05-24 19:03 /home/user/files/departmentsR.csv
drwxr-xr-x  - user supergroup          0 2023-05-25 17:41 /home/user/files/departmentsR.parquet
-rw-r--r--  3 user supergroup       1017 2023-05-24 19:03 /home/user/files/employeesR.csv
drwxr-xr-x  - user supergroup          0 2023-05-25 17:41 /home/user/files/employeesR.parquet
-rw-r--r--  3 user supergroup    1264187 2023-05-24 19:03 /home/user/files/movie_genres.csv
drwxr-xr-x  - user supergroup          0 2023-05-25 17:41 /home/user/files/movie_genres.parquet
-rw-r--r--  3 user supergroup    16282189 2023-05-24 19:03 /home/user/files/movies.csv
drwxr-xr-x  - user supergroup          0 2023-05-25 17:39 /home/user/files/movies.parquet
-rw-r--r--  3 user supergroup   709550294 2023-05-24 19:03 /home/user/files/ratings.csv
drwxr-xr-x  - user supergroup          0 2023-05-25 17:40 /home/user/files/ratings.parquet
user@master:~/project-code$
```

Figure 2: HDFS directory containing the project's datasets in both .csv and .parquet formats.

Part 2 Basics

Task 2.1 Repartition and Broadcast Joins

Task 2.2 Tweaking the Catalyst Optimiser

Without Optimization (Sort Merge Join):

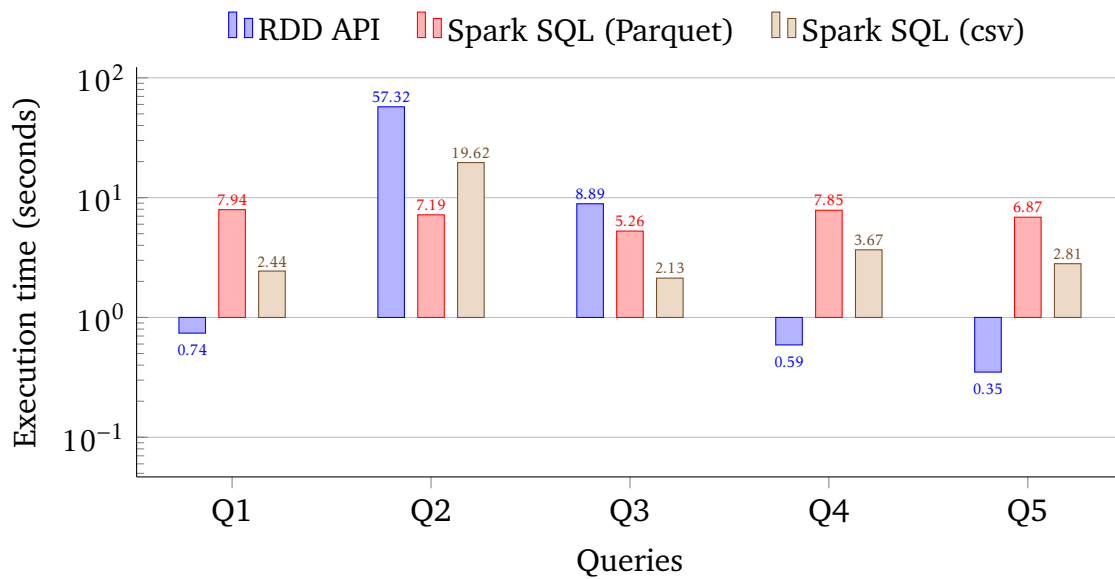


Figure 3: Caption

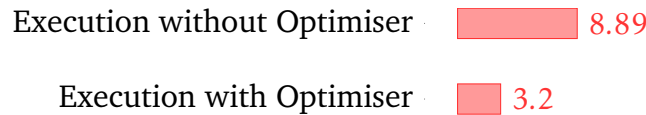


Figure 4: Execution time, for a particular query, with and without Spark's SQL Optimiser.

== Physical Plan ==

```
* (6) SortMergeJoin [mv_id#8], [mv_id#1], Inner
```

This is a Sort Merge Join, which is an operation where two dataframes are joined by sorting the data and then merging the sorted data. This type of join is used when the data is too large to fit into memory for a Broadcast Join.

The stages of this join operation involve filtering data where `mv_id` is not null, limiting the result set to 100, then sorting the data and performing the join operation. The `SortMergeJoin` operation requires that both sides of the join have been partitioned and sorted by the join key (`mv_id` in this case). Hence, there are additional Sort and Exchange operations before the join.

```
+-- * (2) GlobalLimit 100
+-- Exchange SinglePartition
+-- * (1) LocalLimit 100
```

These operations represent the limit that is applied to the "movie_genres" table (the query limits the result set to 100). The `GlobalLimit` means that the limit is applied across the entire dataset, not just per partition.

The time taken for the entire operation is 13.2132 seconds.

With Optimization (Broadcast Hash Join):

```
== Physical Plan ==
```

```
*(3) BroadcastHashJoin [mv_id#8], [mv_id#1], Inner, BuildLeft
```

This is a Broadcast Hash Join, which is a type of join operation where the smaller DataFrame is broadcast to all the nodes containing partitions of the larger DataFrame for comparison and join operations. This type of join can be significantly faster than a sort-merge join because it doesn't require the data to be sorted first, and it can be done entirely in memory if the smaller DataFrame is small enough.

```
:- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0,
  ↳ int, false] as bigint))
```

The BroadcastExchange operation represents broadcasting the smaller DataFrame to the worker nodes. The broadcasting operation will transform the DataFrame into a more efficient data structure (a hash table) to speed up the subsequent join operation.

The time taken for the entire operation is 3.7529 seconds, which is significantly faster than the time taken when the optimization is not enabled. The optimizer has chosen a more efficient join strategy (broadcast hash join instead of sort merge join) based on the size of the data and the nature of the operation, which leads to a faster execution time.

Overall, the difference between these two execution plans demonstrates the power and importance of the Spark Catalyst optimizer in efficiently executing Spark jobs. It can make smart decisions, such as using a BroadcastHashJoin instead of a Sort-MergeJoin, which significantly improves performance.

Part 3 Code Snippets

Snippet 1: csv_to_parquet.py

```
1 from pyspark.sql import SparkSession
2 from pyspark.sql.types import StructField, StructType, IntegerType,
  ↳ FloatType, StringType
3
4
5 def convert_csv_to_parquet():
6     # Create spark instance
7     spark = SparkSession \
8         .builder \
9         .appName("Add schemas to CSVs and make Parquet files") \
10        .getOrCreate()
11
12
13     # Set schemas of csv files needed in Part 1 of the project
14     movies_schema = StructType([
15         StructField("mv_id", IntegerType()),
```

```
16     StructField("name", StringType()),
17     StructField("description", StringType()),
18     StructField("year", IntegerType()),
19     StructField("duration", IntegerType()),
20     StructField("prod_cost", IntegerType()),
21     StructField("revenue", IntegerType()),
22     StructField("popularity", FloatType())
23   ])
24
25   ratings_schema = StructType([
26     StructField("usr_id", IntegerType()),
27     StructField("mv_id", IntegerType()),
28     StructField("rating", FloatType()),
29     StructField("time_stamp", IntegerType())
30   ])
31
32   movie_genres_schema = StructType([
33     StructField("mv_id", IntegerType()),
34     StructField("genre", StringType())
35   ])
36
37   # Set schemas of csv files need in Part 2 of the project
38   employeesR_schema = StructType([
39     StructField("id", IntegerType()),
40     StructField("name", StringType()),
41     StructField("dep_id", IntegerType())
42   ])
43
44   departmentsR_schema = StructType([
45     StructField("dep_id", IntegerType()),
46     StructField("dep_name", StringType())
47   ])
48
49
50   # Load the aforementioned csv files into dataframes
51   movies_df = spark.read.format('csv') \
52     .options(header='false') \
53     .schema(movies_schema) \
54     .load("hdfs://master:9000/home/user/files/movies.csv")
55
56   ratings_df = spark.read.format('csv') \
57     .options(header='false') \
58     .schema(ratings_schema) \
59     .load("hdfs://master:9000/home/user/files/ratings.csv")
60
```

```

61 movie_genres_df = spark.read.format('csv') \
62     .options(header='false') \
63     .schema(movie_genres_schema) \
64     .load("hd_
        ↳ fs://master:9000/home/user/files/movie_genres.csv")
65
66 employeesR_df = spark.read.format('csv') \
67     .options(header='false') \
68     .schema(employeesR_schema) \
69     .load("hd_
        ↳ fs://master:9000/home/user/files/employeesR.csv")
70
71 departmentsR_df = spark.read.format('csv') \
72     .options(header='false') \
73     .schema(departmentsR_schema) \
74     .load("hdfs://master:9000/home/user/files/d_
        ↳ epartmentsR.csv")
75
76
77 # Save the dataframes as Parquet
78 movies_df.write.parquet("hd_
        ↳ fs://master:9000/home/user/files/movies.parquet")
79 ratings_df.write.parquet("hd_
        ↳ fs://master:9000/home/user/files/ratings.parquet")
80 movie_genres_df.write.parquet("hd_
        ↳ fs://master:9000/home/user/files/movie_genres.parquet")
81 employeesR_df.write.parquet("hd_
        ↳ fs://master:9000/home/user/files/employeesR.parquet")
82 departmentsR_df.write.parquet("hd_
        ↳ fs://master:9000/home/user/files/departmentsR.parquet")

```

Snippet 2: rdd.py

```

1 from utils import timeit
2
3
4 def create_rdd(spark):
5     movies_rdd = spark.textFile("hd_
        ↳ fs://master:9000/home/user/files/movies.csv")
6     ratings_rdd = spark.textFile("hd_
        ↳ fs://master:9000/home/user/files/ratings.csv")
7     genres_rdd = spark.textFile("hd_
        ↳ fs://master:9000/home/user/files/movie_genres.csv")
8
9     return movies_rdd, ratings_rdd, genres_rdd

```

```
10
11
12 def query1(spark):
13     # Fetch initial RDD from a csv
14     movies_rdd, _, _ = create_rdd(spark)
15
16     # Get the difference between revenue and production cost (i.e.
17     ↪ profits) of every
18     # movie after 1995
19     movies = movies_rdd.map(lambda line: line.split(',')) \
20         .filter(lambda field: len(field) > 7 and
21             ↪ field[3].isdigit() and field[6].isdigit()
22             ↪ and field[5].isdigit()) \
23         .map(lambda field: (int(field[3]),
24             ↪ (int(field[6]), int(field[5])))) \
25         .filter(lambda field: field[0] > 1995 and
26             ↪ field[1][0] > 0 and field[1][1] > 0) \
27         .map(lambda field: (field[0], str(field[1][0]
28             ↪ - field[1][1]))) \
29         .reduceByKey(lambda v1, v2: v1 + ", " + v2) \
30         .sortBy(lambda pair: pair[0])
31
32     execution_time, result = timeit(movies.collect)
33
34     with open('../output/rdd_results/Q1RDD.txt', 'w') as f:
35         for year, movie in result:
36             f.write("year %s, profits [%s]\n" % (year, movie))
37
38     return execution_time
39
40
41 def query2(spark):
42     # Fetch initial RDDs from the csv files
43     movies_rdd, ratings_rdd, _ = create_rdd(spark)
44
45     # Get the movie id, the average rating and the total number of
46     ↪ ratings for the
47     # movie \Cesare deve morire"
48     mapped_movies = movies_rdd.map(lambda line: line.split(',')) \
49         .filter(lambda fields: len(fields) == 8
50             ↪ and fields[3].isdigit() and
51             ↪ fields[6].isdigit()) \
52         .filter(lambda fields: fields[1] ==
53             ↪ "Cesare deve morire") \
54         .map(lambda fields: (int(fields[0]),
55             ↪ ('movies', fields[1])))
```

```

45 mapped_ratings = ratings_rdd.map(lambda line: line.split(',')) \
46     .filter(lambda fields: len(fields) ==
47         ↪ 4) \
48     .map(lambda fields: (int(fields[1]),
49         ↪ (float(fields[2]), 1))) \
50     .reduceByKey(lambda x, y: (x[0] +
51         ↪ y[0], x[1] + y[1])) \
52     .map(lambda pair: (pair[0],
53         ↪ ('ratings', pair[1][0] /
54         ↪ pair[1][1], pair[1][1])))
55
56 union = mapped_movies.union(mapped_ratings)
57
58 movie_stats = union.groupByKey() \
59     .flatMap(lambda kv: [(kv[0], m[1], m[2]) for m
60         ↪ in kv[1] if m[0] == 'ratings' for g in
61         ↪ kv[1] if g[0] == 'movies'])
62
63 execution_time, stats = timeit(movie_stats.collect)
64
65 with open('../output/rdd_results/Q2RDD.txt', 'w') as f:
66     f.write("Movie ID: %i, Number of ratings: %i, Average rating:
67         ↪ %.2f" % (stats[0][0], stats[0][2], stats[0][1]))
68
69 return execution_time
70
71 def query3(spark):
72     # Fetch initial RDDs from the csv files
73     movies_rdd, _, genres_rdd = create_rdd(spark)
74
75     # Get the best Animation movie in terms of revenue for 1995
76     mapped_movies = movies_rdd.map(lambda line: line.split(',')) \
77         .filter(lambda fields: len(fields) == 8
78             ↪ and fields[3].isdigit() and
79             ↪ fields[6].isdigit()) \
80         .filter(lambda fields: int(fields[3])
81             ↪ == 1995 and int(fields[5]) > 0 and
82             ↪ int(fields[6]) > 0) \
83         .map(lambda fields: (int(fields[0]),
84             ↪ ('movies', fields[1],
85             ↪ int(fields[6])))

```



```
76 mapped_genres = genres_rdd.map(lambda line: line.split(',')) \
77     .filter(lambda fields: len(fields) == 2
78         ↪ and fields[0].isdigit() and
79         ↪ fields[1] == 'Animation') \
80     .map(lambda fields: (int(fields[0]),
81         ↪ ('genres', fields[1])))
82
83 # Union of the two RDDs
84 union = mapped_movies.union(mapped_genres)
85
86 # Group by key and transform the result
87 joined = union.groupByKey() \
88     .flatMap(lambda kv: [(m[1], m[2]) for m in kv[1] if
89         ↪ m[0] == 'movies' for g in kv[1] if g[0] ==
90         ↪ 'genres'])
91
92 # Action takes place through the joined(), so the timeit()
93 ↪ function is placed accordingly
94 execution_time, best_animation_movie = timeit(joined.reduce,
95     ↪ lambda movie, next_movie: movie if movie[1] > next_movie[1]
96     ↪ else next_movie)
97
98 with open('../output/rdd_results/Q3RDD.txt', 'w') as f:
99     f.write("Best Animation Movie of 1995: {}, Revenue:
100     ↪ {}".format(best_animation_movie[0],
101     ↪ best_animation_movie[1]))
102
103 return execution_time
104
105 def query4(spark):
106     # Fetch initial RDDs from the csv files
107     movies_rdd, _, genres_rdd = create_rdd(spark)
108
109     # Get the most popular Comedy movie for each year after 1995
110     mapped_movies = movies_rdd.map(lambda line: line.split(',')) \
111         .filter(lambda field: len(field) == 8
112             ↪ and field[3].isdigit() and
113             ↪ field[6].isdigit()) \
114         .filter(lambda field: int(field[3]) >
115             ↪ 1995 and float(field[7]) > 0) \
116         .map(lambda field: (int(field[0]),
117             ↪ ('movies', int(field[3]), field[1],
118             ↪ float(field[7]))))
```

```

106 mapped_genres = genres_rdd.map(lambda line: line.split(',')) \
107     .filter(lambda field: len(field) == 2
108         ↪ and field[0].isdigit()) \
109     .filter(lambda field: field[1] ==
110         ↪ 'Comedy') \
111     .map(lambda field: (int(field[0]),
112         ↪ ('genres', field[1])))
113
114 # Make union of movies and genres
115 union = mapped_movies.union(mapped_genres)
116
117 # Extract the best comedy per year
118 best_comedy = union.groupByKey() \
119     .flatMap(lambda kv: [(m[1], (m[2], m[3])) for
120         ↪ m in kv[1] if m[0] == 'movies' for g in
121         ↪ kv[1] if g[0] == 'genres']) \
122     .reduceByKey(lambda x, y: x if x[1] > y[1]
123         ↪ else y) \
124     .sortBy(lambda pair: pair[0])
125
126 execution_time, best_comedy = timeit(best_comedy.collect)
127
128 with open('../output/rdd_results/Q4RDD.txt', 'w') as f:
129     for movie in best_comedy:
130         f.write("The most popular Comedy of %i was %s, with a
131             ↪ popularity score of %.2f\n" % (movie[0], movie[1][0],
132             ↪ movie[1][1]))
133
134 return execution_time
135
136 def query5(spark):
137     # Fetch initial RDD from a csv
138     movies_rdd, _, _ = create_rdd(spark)
139
140     # Get the average revenue for each year
141     mapped_movies = movies_rdd.map(lambda line: line.split(',')) \
142         .filter(lambda fields: len(fields) == 8
143             ↪ and fields[3].isdigit() and
144             ↪ fields[6].isdigit()) \
145         .filter(lambda fields: int(fields[3]) >
146             ↪ 0 and int(fields[6]) > 0) \
147         .map(lambda fields: (int(fields[3]),
148             ↪ (int(fields[6]), 1))) \

```

```
138         .reduceByKey(lambda revenue,
139                        ↪ next_revenue: (revenue[0] +
140                        ↪ next_revenue[0], revenue[1] +
141                        ↪ next_revenue[1])) \
142     .map(lambda fields: (fields[0],
143                        ↪ fields[1][0] / fields[1][1])) \
144     .sortBy(lambda pair: pair[0])
145
146     execution_time, results = timeit(mapped_movies.collect)
147
148     with open('../output/rdd_results/Q5RDD.txt', 'w') as f:
149         for result in results:
150             f.write("Year %i had an average movie revenue of %.2f\n"
151                    ↪ % (result[0], result[1]))
152
153     return execution_time
```

Snippet 3: sql_csv.py

```
1 from pyspark.sql.functions import collect_list
2 from pyspark.sql.types import StructField, StructType, IntegerType,
3   ↪ FloatType, StringType
4 from utils import timeit, save_dataframe_output
5
6 # Set schemas of csv files
7 movies_schema = StructType([
8     StructField("mv_id", IntegerType()),
9     StructField("name", StringType()),
10    StructField("description", StringType()),
11    StructField("year", IntegerType()),
12    StructField("duration", IntegerType()),
13    StructField("prod_cost", IntegerType()),
14    StructField("revenue", IntegerType()),
15    StructField("popularity", FloatType())
16 ])
17
18 ratings_schema = StructType([
19     StructField("usr_id", IntegerType()),
20     StructField("mv_id", IntegerType()),
21     StructField("rating", FloatType()),
22     StructField("time_stamp", IntegerType())
23 ])
24
25 movie_genres_schema = StructType([
```

```

26     StructField("mv_id", IntegerType()),
27     StructField("genre", StringType())
28 ])
29
30
31 def create_temp_tables(spark):
32     # Load the aforementioned csv files into dataframes
33     movies_df = spark.read.format('csv') \
34         .options(header='false') \
35         .schema(movies_schema) \
36         .load("hdfs://master:9000/home/user/files/movies.csv")
37
38     ratings_df = spark.read.format('csv') \
39         .options(header='false') \
40         .schema(ratings_schema) \
41         .load("hdfs://master:9000/home/user/files/ratings.csv")
42
43     movie_genres_df = spark.read.format('csv') \
44         .options(header='false') \
45         .schema(movie_genres_schema) \
46         .load("hdfs://master:9000/home/user/files/movie_genres.csv")
47
48     # Create temporary tables
49     movies_df.createOrReplaceTempView("movies")
50     ratings_df.createOrReplaceTempView("ratings")
51     movie_genres_df.createOrReplaceTempView("genres")
52
53
54 def query1(spark):
55     # Fetch relations
56     create_temp_tables(spark)
57
58     # Get the difference between revenue and production cost (i.e.
59     ↪ profits) of every movie after 1995
60     query = """
61         SELECT year, concat_ws(',', collect_list(cast((revenue -
62         ↪ prod_cost) AS string))) AS profit
63         FROM movies
64         WHERE prod_cost > 0 AND revenue > 0 AND year > 1995
65         GROUP BY year
66         ORDER BY year
67     """
68
69     execution_time, _ = timeit(spark.sql(query).show)

```

```
68     query_output = save_dataframe_output(spark.sql(query))
69
70     return execution_time, query_output
71
72
73 def query2(spark):
74     # Fetch relations
75     create_temp_tables(spark)
76
77     # Get the movie id, the average rating and the total number of
78     ↪ ratings for the movie \Cesare deve morire"
79     query = """
80         SELECT m.mv_id, COUNT(r.usr_id) AS user_count, AVG(r.rating)
81         ↪ AS average_rating
82         FROM movies AS m
83         JOIN ratings AS r ON m.mv_id = r.mv_id
84         WHERE m.name = 'Cesare deve morire'
85         GROUP BY m.mv_id
86     """
87
88     execution_time, _ = timeit(spark.sql(query).show)
89     query_output = save_dataframe_output(spark.sql(query))
90
91     return execution_time, query_output
92
93 def query3(spark):
94     # Fetch relations
95     create_temp_tables(spark)
96
97     # Get the best Animation movie in terms of revenue for 1995
98     query = """
99         SELECT m.name AS movie_name, m.revenue AS revenue
100        FROM movies AS m
101        JOIN genres AS mg ON m.mv_id = mg.mv_id
102        ↪ WHERE mg.genre = 'Animation' AND m.year = 1995 AND m.revenue
103        > 0
104        ORDER BY m.revenue DESC
105        LIMIT 1
106    """
107
108     execution_time, _ = timeit(spark.sql(query).show)
109     query_output = save_dataframe_output(spark.sql(query))
110
111     return execution_time, query_output
```

```

110
111 def query4(spark):
112     # Fetch relations
113     create_temp_tables(spark)
114
115     # Get the most popular Comedy movie for each year after 1995
116     query = """
117         WITH ranked_movies AS (
118             SELECT m.year, m.name, m.popularity,
119                 ROW_NUMBER() OVER(PARTITION BY m.year ORDER BY
120 ↪ m.popularity DESC) AS rank
121             FROM movies AS m
122             JOIN genres AS mg ON m.mv_id = mg.mv_id
123             WHERE mg.genre = 'Comedy' AND m.year > 1995 AND
124 ↪ m.popularity > 0 AND m.revenue > 0
125         )
126         SELECT year, name, popularity
127         FROM ranked_movies
128         WHERE rank = 1
129         ORDER BY year
130     """
131
132     execution_time, _ = timeit(spark.sql(query).show)
133     query_output = save_dataframe_output(spark.sql(query))
134
135     return execution_time, query_output
136
137 def query5(spark):
138     # Fetch relations
139     create_temp_tables(spark)
140
141     # Get the average revenue for each year
142     query = """
143         SELECT year, AVG(revenue) AS avg_revenue
144         FROM movies
145         WHERE year > 0 AND revenue > 0
146         GROUP BY year
147         ORDER BY year DESC
148     """
149
150     execution_time, _ = timeit(spark.sql(query).show)
151     query_output = save_dataframe_output(spark.sql(query))
152
153     return execution_time, query_output

```

Snippet 4: sql_parquet.py

```
1 from pyspark.sql.functions import collect_list
2 from utils import timeit
3
4
5 def create_temp_tables(spark):
6     # Fetch data
7     movies_df = spark.read.parquet("hd_
8         ↳ fs://master:9000/home/user/files/movies.parquet")
9     ratings_df = spark.read.parquet("hd_
10         ↳ fs://master:9000/home/user/files/ratings.parquet")
11     genres_df = spark.read.parquet("hd_
12         ↳ fs://master:9000/home/user/files/movie_genres.parquet")
13
14     # Create temporary relations
15     movies_df.createOrReplaceTempView("movies")
16     ratings_df.createOrReplaceTempView("ratings")
17     genres_df.createOrReplaceTempView("genres")
18
19 def query1(spark):
20     # Fetch relations
21     create_temp_tables(spark)
22
23     # Get the difference between revenue and production cost (i.e.
24     ↳ profits) of every movie after 1995
25     query = """
26         SELECT year, concat_ws(',', collect_list(cast((revenue -
27         ↳ prod_cost) AS string))) AS profit
28         FROM movies
29         WHERE prod_cost > 0 AND revenue > 0 AND year > 1995
30         GROUP BY year
31         ORDER BY year
32     """
33     return timeit(spark.sql(query).show)
34
35 def query2(spark):
36     # Fetch relations
37     create_temp_tables(spark)
38
39     # Get the movie id, the average rating and the total number of
40     ↳ ratings for the movie \Cesare deve morire"
41     query = """
```

```

39         SELECT m.mv_id, COUNT(r.usr_id) AS user_count, AVG(r.rating)
↪ AS average_rating
40         FROM movies AS m
41         JOIN ratings AS r ON m.mv_id = r.mv_id
42         WHERE m.name = 'Cesare deve morire'
43         GROUP BY m.mv_id
44     """
45
46     return timeit(spark.sql(query).show)
47
48
49 def query3(spark):
50     # Fetch relations
51     create_temp_tables(spark)
52
53     # Get the best Animation movie in terms of revenue for 1995
54     query = """
55         SELECT m.name AS movie_name, m.revenue AS revenue
56         FROM movies AS m
57         JOIN genres AS mg ON m.mv_id = mg.mv_id
58         WHERE mg.genre = 'Animation' AND m.year = 1995 AND m.revenue
↪ > 0
59         ORDER BY m.revenue DESC
60         LIMIT 1
61     """
62
63     return timeit(spark.sql(query).show)
64
65
66 def query4(spark):
67     # Fetch relations
68     create_temp_tables(spark)
69
70     # Get the most popular Comedy movie for each year after 1995
71     query = """
72         WITH ranked_movies AS (
73             SELECT m.year, m.name, m.popularity,
74             ROW_NUMBER() OVER(PARTITION BY m.year ORDER BY
↪ m.popularity DESC) AS rank
75         FROM movies AS m
76         JOIN genres AS mg ON m.mv_id = mg.mv_id
77         WHERE mg.genre = 'Comedy' AND m.year > 1995 AND
↪ m.popularity > 0 AND m.revenue > 0
78         )
79         SELECT year, name, popularity

```



```
80         FROM ranked_movies
81         WHERE rank = 1
82         ORDER BY year
83     """
84
85     return timeit(spark.sql(query).show)
86
87
88 def query5(spark):
89     # Fetch relations
90     create_temp_tables(spark)
91
92     # Get the average revenue for each year
93     query = """
94         SELECT year, AVG(revenue) AS avg_revenue
95         FROM movies
96         WHERE year > 0 AND revenue > 0
97         GROUP BY year
98         ORDER BY year DESC
99     """
100
101     return timeit(spark.sql(query).show)
```

Snippet 5: joins.py

```
1 from utils import timeit
2
3
4 def create_rdd(spark):
5     employees = spark.textFile("hd_
6     ↪ fs://master:9000/home/user/files/employeesR.csv")
7     departments = spark.textFile("hd_
8     ↪ fs://master:9000/home/user/files/departmentsR.csv")
9
10    return employees, departments
11
12 def repartition_join(spark):
13     # Fetch RDDs from the csv files
14     employees, departments = create_rdd(spark)
15
16     # Tag RDDs
17     employees_tagged = employees \
18         .map(lambda line: line.split(',')) \
19         .map(lambda field: (int(field[0]), field[1],
20         ↪ int(field[2]))) \
```

```

19         .map(lambda field: (field[2], ('employees',
20             ↪ field)))
21
22 departments_tagged = departments \
23     .map(lambda line: line.split(',')) \
24     .map(lambda field: (int(field[0]),
25         ↪ field[1])) \
26     .map(lambda field: (field[0],
27         ↪ ('departments', field)))
28
29 # Concatenate all RDDs
30 union = employees_tagged.union(departments_tagged)
31
32 # Perform the join for each key
33 def join_records(records):
34     employees_records = [field[1] for field in records if
35         ↪ field[0] == 'employees']
36     departments_records = [field[1] for field in records if
37         ↪ field[0] == 'departments']
38     return [(e[1], e[0], d[1]) for e in employees_records for d
39         ↪ in departments_records]
40
41 # Extract union result
42 joined_rdd = union.groupByKey() \
43     .flatMap(lambda pair: join_records(pair[1]))
44
45 return timeit(joined_rdd.collect)
46
47 def broadcast_join(spark):
48     # Fetch RDDs from the csv files
49     employees, departments = create_rdd(spark)
50
51     employees_rdd = employees \
52         .map(lambda line: line.split(',')) \
53         .map(lambda field: (int(field[0]), field[1],
54             ↪ int(field[2])))
55
56     departments_rdd = departments \
57         .map(lambda line: line.split(',')) \
58         .map(lambda field: (int(field[0]), field[1]))
59
60     # Create a dictionary for department data
61     dep_dict = {field[0]: field[1] for field in
62         ↪ departments_rdd.collect()}

```

```
56
57     # Broadcast the department data
58     dep_broadcast = spark.broadcast(dep_dict)
59
60     def map_func(field):
61         # Extract the join key (department ID) and the department
62         ↪ name using
63         # the broadcast variable
64         dep_id = field[2]
65         dep_name = dep_broadcast.value.get(dep_id)
66
67         # Joined data
68         return (field[1], field[0], dep_name)
69
70     joined_rdd = employees_rdd.map(lambda field: map_func(field))
71
72     return timeit(joined_rdd.collect)
```

Snippet 6: optimiser.py

```
1  import io
2  import contextlib
3  from utils import timeit
4
5
6  def create_temp_tables(spark):
7      dataframe = spark.read.format("parquet")
8
9      ratings_dataframe = dataframe.load("hd_j
10     ↪ fs://master:9000/home/user/files/ratings.parquet")
11     genres_dataframe = dataframe.load("hd_j
12     ↪ fs://master:9000/home/user/files/movie_genres.parquet")
13
14     ratings_dataframe.registerTempTable("ratings")
15     genres_dataframe.registerTempTable("genres")
16
17
18  def use_optimiser(spark, disabled = "N"):
19
20     # Fetch relations
21     create_temp_tables(spark)
22
23     if disabled == "Y":
24         spark.conf.set("spark.sql.cbo.enable", False)
25         spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)
```

```

24     elif disabled == "N":
25         pass
26     else:
27         raise Exception ("This setting is not available.")
28
29     query = """
30         SELECT *
31         FROM (SELECT * FROM genres LIMIT 100) AS g, ratings AS r
32         WHERE r.mv_id = g.mv_id
33     """
34
35     stdout = io.StringIO()
36     with contextlib.redirect_stdout(stdout):
37         spark.sql(query).explain()
38
39     # Get the captured standard output
40     query_plan = stdout.getvalue()
41
42     return timeit(spark.sql(query).show), query_plan

```

Snippet 7: utils.py

```

1  import io
2  import time
3  import contextlib
4
5
6  def timeit(func, *args, **kwargs):
7      """
8      Measure execution time of a function for a single run.
9
10     Args:
11         func: Function to be executed.
12         *args: Variable length argument list for the function.
13         **kwargs: Arbitrary keyword arguments for the function.
14
15     Returns:
16         tuple: A tuple containing the execution time and the result
17     ↪ of the function call.
18     """
19     start = time.time()
20     result = func(*args, **kwargs)
21     end = time.time()
22     execution_time = end - start

```

```
23     return execution_time, result
24
25
26 def save_dataframe_output(dataframe):
27     """
28     Function to capture and return the complete output of a PySpark
    ↪ DataFrame.
29
30     Args:
31         dataframe (pyspark.sql.DataFrame): The DataFrame whose
    ↪ output is to be captured.
32
33     Returns:
34         str: The entire output of the DataFrame as a string.
35     """
36
37     # Calculate the number of rows in the DataFrame
38     row_count = dataframe.count()
39
40     # Create a StringIO object
41     stdout = io.StringIO()
42
43     # Execute show() on DataFrame and capture the output
44     with contextlib.redirect_stdout(stdout):
45         dataframe.show(n=row_count, truncate=False)
46
47     # Get the captured standard output
48     return stdout.getvalue()
```

Snippet 8: main.py

```
1  # Import SparkSession
2  from pyspark.sql import SparkSession
3
4  # Import RDD queries
5  from rdd import query1 as rdd_query1
6  from rdd import query2 as rdd_query2
7  from rdd import query3 as rdd_query3
8  from rdd import query4 as rdd_query4
9  from rdd import query5 as rdd_query5
10
11 # Import SQL-on-csv queries
12 from sql_csv import query1 as sql_csv_query1
13 from sql_csv import query2 as sql_csv_query2
14 from sql_csv import query3 as sql_csv_query3
```

```

15 from sql_csv import query4 as sql_csv_query4
16 from sql_csv import query5 as sql_csv_query5
17
18 # Import SQL-on-Parquet queries
19 from sql_parquet import query1 as sql_parquet_query1
20 from sql_parquet import query2 as sql_parquet_query2
21 from sql_parquet import query3 as sql_parquet_query3
22 from sql_parquet import query4 as sql_parquet_query4
23 from sql_parquet import query5 as sql_parquet_query5
24
25 # Import RDD joins
26 from joins import broadcast_join
27 from joins import repartition_join
28
29 # Import Optimiser script
30 from optimiser import use_optimiser
31
32 # Import csv-to-parquet converter
33 from csv_to_parquet import convert_csv_to_parquet
34
35
36 def part1():
37     ##### Task 1 #####
38     # Convert CSVs to Parquet; run only once. Should you
39     # wish to repeat the process, comment it out.
40     convert_csv_to_parquet()
41
42
43     ##### Tasks 2, 3 & 4 #####
44     times = {}
45
46     # Calculate execution times for each query (Tasks 2, 3 & 4)
47     for i in range(1, 6):
48         spark = SparkSession \
49             .builder \
50             .appName("All-use session") \
51             .getOrCreate()
52         sc = spark.sparkContext
53
54         rdd_query_name      = 'rdd_query%s' % (i)
55         parquet_query_name  = 'sql_parquet_query%s' % (i)
56         csv_query_name      = 'sql_csv_query%s' % (i)
57
58         times[rdd_query_name] =
            ↪ globals()[rdd_query_name](sc)

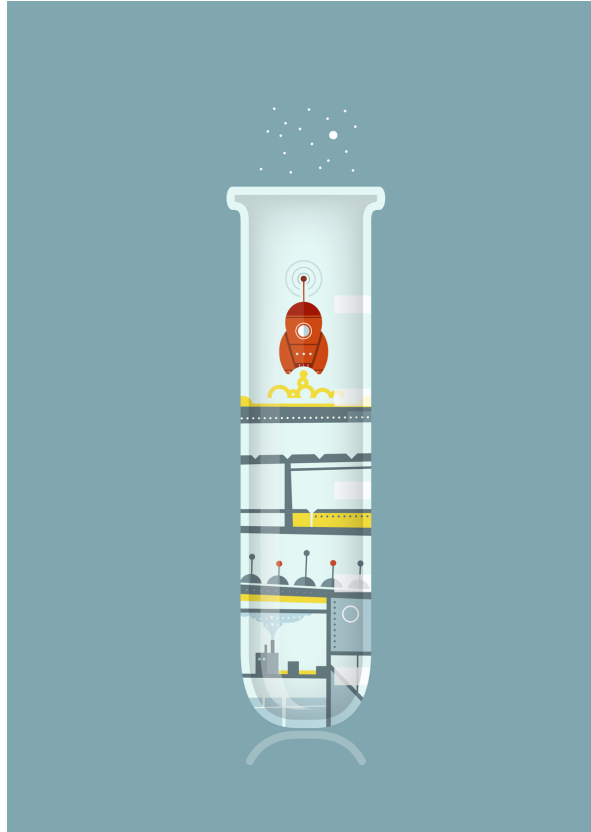
```

```
59     times[parquet_query_name], _ =
60         ↪ globals()[parquet_query_name](spark)
61
62     times[csv_query_name], query_output =
63         ↪ globals()[csv_query_name](spark)
64
65     # Save the query-output, in dataframe format, on a text file
66     ↪
67     with open('../output/df_results/Q%sDF.txt' % i, 'w') as f:
68         f.write(query_output)
69
70     # Consistency in execution times
71     spark.stop()
72     sc.stop()
73     print(times)
74
75     # Compute execution times and write to a text file
76     with open('../output/part_1_times.txt', 'w') as f:
77         for query, execution_time in times.items():
78             f.write('%s: %.2f seconds\n' % (query, execution_time))
79
80
81 def part2():
82     ##### Task 1 #####
83     times = {}
84
85     spark = SparkSession \
86         .builder \
87         .appName("All-use session") \
88         .getOrCreate() \
89         .sparkContext
90
91     times['Broadcast Join'], broadcast_result = broadcast_join(spark)
92     times['Repartition Join'], _ =
93         ↪ repartition_join(spark)
94
95     # Compute execution times and write to a text file
96     with open('../output/join_type_times.txt', 'w') as f:
97         for query, execution_time in times.items():
98             f.write('%s: %.2f seconds\n' % (query, execution_time))
99
100     # Save the result to text files
101     with open('../output/join_outputs.txt', 'w') as f:
102         for result in broadcast_result:
103             f.write(str(result) + '\n')
```

```

100     # Consistency in execution times
101     spark.stop()
102
103
104     ##### Task 2 #####
105     times = {}
106
107     # Two instances are created since Spark tends to keep
108     # metadata from each run in order to optimise reading
109     # and calculating future queries.
110     spark = SparkSession \
111         .builder \
112         .appName('Using Catalyst') \
113         .getOrCreate()
114     sc = spark
115
116     times["Using Catalyst"], with_catalyst =
117         ↪ use_optimiser(spark)
118
119     times["Without using Catalyst"], without_catalyst =
120         ↪ use_optimiser(sc, disabled="Y")
121
122     spark.stop()
123     sc.stop()
124
125     # Compute execution times and write to a text file
126     with open('../output/catalyst_times.txt', 'w') as f:
127         for query, execution_time in times.items():
128             f.write('%s: %.2f seconds\n' % (query,
129                 ↪ execution_time[0]))
130
131     # Save the optimised query plan to text file
132     with open('../output/optimised_plan.txt', 'w') as f:
133         f.write(with_catalyst)
134
135     # Save the non-optimised query plan to text file
136     with open('../output/non_optimised_plan.txt', 'w') as f:
137         f.write(without_catalyst)
138
139 if __name__ == "__main__":
140     part1()
141     part2()

```

THIS COURSEWORK WAS TYPESET using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 12 point kpfonts, a revival of URW Palladio typeface. The above illustration, "Science Experiment 02", was created by Ben Schlitter and released under CC BY-NC-ND 3.0. A template that can be used to format coursework with this look and feel has been released under the permissive CC BY-NC-ND 4.0 license, and can be found online at <https://cs.overleaf.com/latex/>.