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 21, 2023

*mdarm@di.uoa.gr

Introduction

The following representation provides a structured overview of the project's directory hierarchy, and can serve as a quick reference to the deliverables and components. This layout details the various output files, source code modules and the report itself.

```
$PROJECT_ROOT
¦   # Project's outputs
+-- output
¦   ¦   # Execution times with and without the optimiser
¦   +-- catalyst_times.txt
¦   ¦   # Query outputs using the Dataframe API
¦   +-- df_results
¦   ¦   ¦
¦   ¦   +-- Q1DF.txt
¦   ¦   +-- Q2DF.txt
¦   ¦   +-- Q3DF.txt
¦   ¦   +-- Q4DF.txt
¦   ¦   +-- Q5DF.txt
¦   ¦
¦   +-- join_outputs.txt        # outputs when using broadcast and
↪   repartition joins
¦   +-- join_type_times.txt     # execution times when using broadcast
↪   and repartition joins
¦   +-- non_optimised_plan.txt  # quey plan without using Catalyst
¦   +-- optimised_plan.txt      # query plan with Catalyst
¦   +-- part_1_times.txt        # execution times, for every query,
↪   for all data types
¦   +-- report.pdf              # project's report
¦   ¦   # Query outputs using the RDD API
¦   +-- rdd_results
¦       ¦
¦       +-- Q1RDD.txt
¦       +-- Q2RDD.txt
¦       +-- Q3RDD.txt
¦       +-- Q4RDD.txt
¦       +-- Q5RDD.txt
¦   # Modules for performing all tasks
+-- src
    ¦
    +-- csv_to_parquet.py       # converting the csv files to parquet
    +-- joins.py                # performing broadcast and repartition
     ↪  joins
    +-- main.py                 # running script; executes required
     ↪  tasks
```

```
    +-- optimiser.py            # enabling and disabling Catalyst
    +-- rdd.py                  # rdd queries
    +-- sql_csv.py              # csv queris (SQL-like)
    +-- sql_parquet.py          # parquet queries (SQL-like)
    +-- utils.py                # helper functions
```

All of the code (modules in the `src` files) is also included in Part 3, at the end of this document. Note that because of Java heap errors, the following modification was made in the `spark-defaults.conf` file:

```
spark.driver.memory 1024m
```

Part 1 File Management in HDFS

Task 1.1 Save files in HDFS

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`.
Figure 1 shows a print-screen of the `.csv` files lying within the created `files` directory.

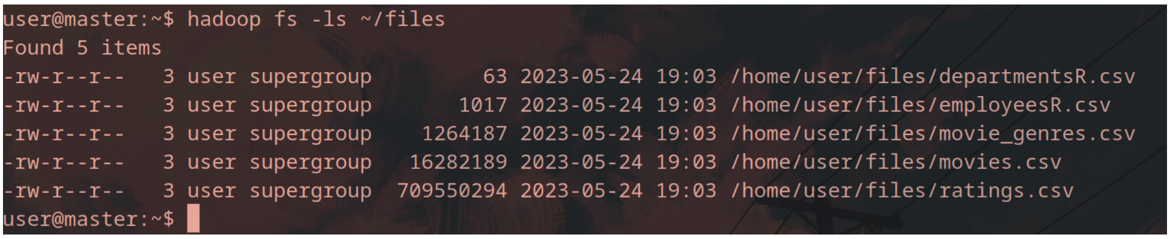

Figure 1: HDFS directory containing the project's datasets in `.csv` format.

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

Task 1.2 RDD queries

All of the RDD queries can be found in the `rdd.py` module (see Snippet 2). All performed joins, when required, were repartition joins. Below is the execution of query 1: "Get the difference betwen revenue and production cost (i.e. profits) of every movie after 1995."

```
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.

```python
    movies = movies_rdd.map(lambda line: line.split(',')) \
                        .filter(lambda field: len(field) > 7 and field[3].isdigit() and field[6].isdigit() and field[5].isdigit()) \
                        .map(lambda field: (int(field[3]), (int(field[6]), int(field[5])))) \
                        .filter(lambda field: field[0] > 1995 and field[1][0] > 0 and field[1][1] > 0) \
                        .map(lambda field: (field[0], str(field[1][0] - field[1][1]))) \
                        .reduceByKey(lambda v1, v2: v1 + ", " + v2) \
                        .sortBy(lambda pair: pair[0])
```

Task 1.3 Dataframe queries

All of the Dataframe queries can be found in the `sql_csv.py` and `sql_parquet.py` modules (see Snippets 3 & 4). Below is the execution of query 1: "Get the difference betwen revenue and production cost (i.e. profits) of every movie after 1995."

```python
    query = """
        SELECT year, concat_ws(',', collect_list(cast((revenue - prod_cost) AS string))) AS profit
        FROM movies
        WHERE prod_cost > 0 AND revenue > 0 AND year > 1995
        GROUP BY year
        ORDER BY year
    """
```

Task 1.4 Query execution-times

To ensure the highest precision in measurement-performance, a Spark benchmarking approach was used. For each individual case, a distinct Spark session was instantiated, and after the run, was terminated. This strategy was imperative to prevent data or intermediate computations from being cached, thereby eliminating any potential biases. This guarantees that the analysis is grounded in somewhat representative query execution-times. The results can be seen in Figure 3.

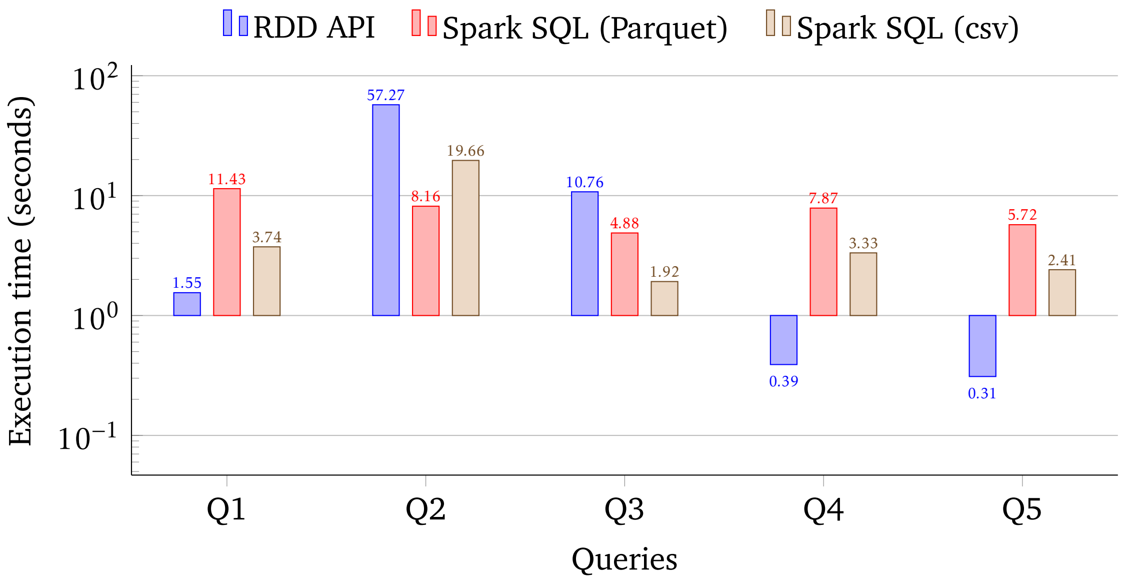

Figure 3: Query execution-times, for all queries, using RDDs, CSVs and Parquet.

A first look at the results reveals contrasting execution times with regards to the RDDs. *Query 1* was completed in a mere 1.55 seconds, while *queries 4 and 5* in under 1 second. On the other hand, *query 2* lasted 57.27 seconds. This variance can largely be traced back to the nature of the operations and transformations inherent to each query. *Query 2* was largely defined by multiple transformations (like 'map', 'filter', 'union', 'flatMap' etc.) and complex computations (like 'reduceByKey' and 'groupByKey') involving shuffling, which is time-intensive.

Furthermore, CSV files must either infer the schema or have it explicitly provided. This schema management can introduce overhead. This is exacerbated for large data sets or complex schema structures. To illustrate: *query 2* took 19.66 seconds, significantly faster than its Parquet counterpart. This behaviour is expected given the inherent row-based orientation of CSVs, which is sub-optimal for Spark's preferred columnar computations. However, in *queries 3, 4 & 5*, all operations on CSV files outperformed their Parquet counterparts.

Even though Parquet often delivers commendable performance, it's crucial to note that, under situations with limited columnar data operations, or when full-file scans are inevitable, Parquet's performance can fall behind that of CSVs due to the overhead from decompressing and decoding columnar data. A very likely scenario for *queries 3, 4 & 5*.

Part 2 Optimised Join Strategies

Task 2.1 Repartition and Broadcast Joins

This part focused on joining two relations, 'departments' and 'employees', on the 'department id' attribute. While there are multiple strategies to do this, two distinct approaches were implemented: the Repartition Join and the Broadcast Join. These operations not only illustrate the diversity in handling data merges but also emphasise the significance of choosing an optimised strategy based on data size and structure. The execution of these joins can be found within the `joins.py` module (see Snippet 5).

2.1.1 Repartition join

This method represents a co-grouping-based join operation. The records from each RDD are tagged with their respective source labels ('employees' or 'departments'). By unionising these tagged RDDs, and grouping by their keys, a co-group operation is achieved. After this, the join operation is performed by matching keys, yielding combined records from both employees and departments.

```python
def repartition_join(spark):
    # Fetch RDDs from the csv files
    employees, departments = create_rdd(spark)

    # Tag RDDs
    employees_tagged = employees \
                        .map(lambda line: line.split(',')) \
                        .map(lambda field: (int(field[0]), field[1], int(field[2]))) \
                        .map(lambda field: (field[2], ('employees', field)))

    departments_tagged = departments \
                        .map(lambda line: line.split(',')) \
                        .map(lambda field: (int(field[0]), field[1])) \
                        .map(lambda field: (field[0], ('departments', field)))

    # Concatenate all RDDS
    union = employees_tagged.union(departments_tagged)

    # Perform the join for each key
    def join_records(records):
        employees_records    = [field[1] for field in records if field[0] == 'employees']
```

```python
        departments_records = [field[1] for field in records if field[0] == 'departments']
        return [(e[1], e[0], d[1]) for e in employees_records for d in departments_records]

    # Extract union result
    joined_rdd = union.groupByKey() \
                    .flatMap(lambda pair: join_records(pair[1]))

    return timeit(joined_rdd.collect)
```

2.1.2 Broadcast join

In this approach, the join operation leverages Spark's broadcasting capability. A dictionary (or lookup map) is created for the 'departments' RDD. This dictionary gets broadcast across the cluster, enabling each node to access it locally. The join is then executed by mapping over the 'employees' RDD and fetching the corresponding department name from the broadcasted dictionary using the department ID. This technique benefits from reduced data shuffling, particularly when one of the datasets (like 'departments') is considerably smaller than the other.

```python
def broadcast_join(spark):
    # Fetch RDDs from the csv files
    employees, departments = create_rdd(spark)

    employees_rdd = employees \
                      .map(lambda line: line.split(',')) \
                      .map(lambda field: (int(field[0]), field[1], int(field[2])))

    departments_rdd = departments \
                        .map(lambda line: line.split(',')) \
                        .map(lambda field: (int(field[0]), field[1]))

    # Create a dictionary for department data
    dep_dict = {field[0]: field[1] for field in departments_rdd.collect()}

    # Broadcast the department data
    dep_broadcast = spark.broadcast(dep_dict)

    def map_func(field):
        # Extract the join key (department ID) and the department name using
        # the broadcast variable
        dep_id = field[2]
```

```
        dep_name = dep_broadcast.value.get(dep_id)

        # Joined data
        return (field[1], field[0], dep_name)

    joined_rdd = employees_rdd.map(lambda field: map_func(field))

    return timeit(joined_rdd.collect)
```

Task 2.2 Tweaking the Catalyst Optimiser

Running a query with, and without, the Catalyst optimiser is performed in the `optimiser.py` module (see Snippet 6). Figure 4 shows the execution times for the following query:

```
query = """
        SELECT *
        FROM (SELECT * FROM genres LIMIT 100) AS g, ratings AS r
        WHERE r.mv_id = g.mv_id
"""
```

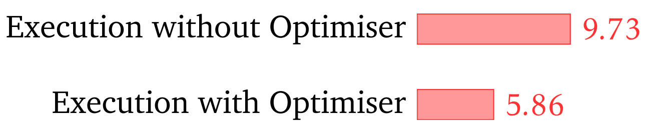

Figure 4: Execution time, for the aforementioned query, with and without Spark's Catalyst Optimiser.

The full query-plans, for using and without using Catalyst, can be found at the `optimised_plan.txt` and `non_optimised_plan.txt` files respectively.

2.2.1 Without Catalyst (Sort Merge Join)

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

When not using Catalyst, a Sort Merge Join is opted. This is a process where two dataframes are merged post-sorting. Such joins are prevalent when data cannot fit into memory for a Broadcast Join (this is a fallacious statement in this case of course). The join requires both data sides to be partitioned and sorted by the join key (here, 'mv_id').

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

The above operations limit the results to 100 for the "movie_genres" relation, applying the limit globally. Execution time was 9.73 seconds.

2.2.2 With Catalyst (Broadcast Hash Join)

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

When utilising Catalyst, a Broadcast Hash Join was used. In this kind of join, the smaller DataFrame is broadcast to the nodes of the larger one. This in-memory operation is quicker than the sort-merge join, especially when the smaller DataFrame is adequately sized.

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

The above indicates broadcasting the smaller DataFrame for optimized joining. Execution time post-optimisation: 5.86 seconds, a significant improvement.
To sum up, the difference in execution-plans underlines Spark Catalyst optimiser's role in enhancing Spark job performance. Through optimal decision-making, such as opting for a BroadcastHashJoin over a SortMergeJoin, due to the relations' difference in size, execution times were significantly reduced.

Part 3 Code Snippets

Snippet 1: `csv_to_parquet.py`

```python
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField, StructType, IntegerType,
↪  FloatType, StringType


def convert_csv_to_parquet():
    # Create spark instance
    spark = SparkSession \
            .builder \
            .appName("Add schemas to CSVs and make Parquet files") \
            .getOrCreate()


    # Set schemas of csv files needed in Part 1 of the project
    movies_schema = StructType([
        StructField("mv_id", IntegerType()),
        StructField("name", StringType()),
        StructField("description", StringType()),
        StructField("year", IntegerType()),
        StructField("duration", IntegerType()),
        StructField("prod_cost", IntegerType()),
        StructField("revenue", IntegerType()),
```

```
        StructField("popularity", FloatType())
        ])

    ratings_schema = StructType([
        StructField("usr_id", IntegerType()),
        StructField("mv_id", IntegerType()),
        StructField("rating", FloatType()),
        StructField("time_stamp", IntegerType())
        ])

    movie_genres_schema = StructType([
        StructField("mv_id", IntegerType()),
        StructField("genre", StringType())
        ])

    # Set schemas of csv files need in Part 2 of the project
    employeesR_schema = StructType([
        StructField("id", IntegerType()),
        StructField("name", StringType()),
        StructField("dep_id", IntegerType())
        ])

    departmentsR_schema = StructType([
        StructField("dep_id", IntegerType()),
        StructField("dep_name", StringType())
        ])


    # Load the aforementioned csv files into dataframes
    movies_df = spark.read.format('csv') \
            .options(header='false') \
            .schema(movies_schema) \
            .load("hdfs://master:9000/home/user/files/movies.csv")

    ratings_df = spark.read.format('csv') \
            .options(header='false') \
            .schema(ratings_schema) \
            .load("hdfs://master:9000/home/user/files/ratings.csv")

    movie_genres_df = spark.read.format('csv') \
            .options(header='false') \
            .schema(movie_genres_schema) \
            .load("hdfs://master:9000/home/user/files/movie_genres.csv")

```

```
    employeesR_df = spark.read.format('csv') \
            .options(header='false') \
            .schema(employeesR_schema) \
            .load("hdfs://master:9000/home/user/files/employeesR.csv")

    departmentsR_df = spark.read.format('csv') \
            .options(header='false') \
            .schema(departmentsR_schema) \
            .load("hdfs://master:9000/home/user/files/departmentsR.csv")


    # Save the dataframes as Parquet
    movies_df.write.parquet("hdfs://master:9000/home/user/files/movies.parquet")
    ratings_df.write.parquet("hdfs://master:9000/home/user/files/ratings.parquet")
    movie_genres_df.write.parquet("hdfs://master:9000/home/user/files/movie_genres.parquet")
    employeesR_df.write.parquet("hdfs://master:9000/home/user/files/employeesR.parquet")
    departmentsR_df.write.parquet("hdfs://master:9000/home/user/files/departmentsR.parquet")
```

Snippet 2: `rdd.py`

```
from utils import timeit


def create_rdd(spark):
    movies_rdd    = spark.textFile("hdfs://master:9000/home/user/files/movies.csv")
    ratings_rdd   = spark.textFile("hdfs://master:9000/home/user/files/ratings.csv")
    genres_rdd    = spark.textFile("hdfs://master:9000/home/user/files/movie_genres.csv")

    return movies_rdd, ratings_rdd, genres_rdd


def query1(spark):
    # Fetch initial RDD from a csv
    movies_rdd, _, _ = create_rdd(spark)

```

```python
    # Get the difference betwen revenue and production cost (i.e. profits) of every
    # movie after 1995
    movies = movies_rdd.map(lambda line: line.split(',')) \
                       .filter(lambda field: len(field) > 7 and field[3].isdigit() and field[6].isdigit() and field[5].isdigit()) \
                       .map(lambda field: (int(field[3]), (int(field[6]), int(field[5])))) \
                       .filter(lambda field: field[0] > 1995 and field[1][0] > 0 and field[1][1] > 0) \
                       .map(lambda field: (field[0], str(field[1][0] - field[1][1]))) \
                       .reduceByKey(lambda v1, v2: v1 + ", " + v2) \
                       .sortBy(lambda pair: pair[0])

    execution_time, result = timeit(movies.collect)

    with open('../output/rdd_results/Q1RDD.txt', 'w') as f:
        for year, movie in result:
            f.write("year %s, profits [%s]\n" % (year, movie))

    return execution_time


def query2(spark):
    # Fetch initial RDDs from the csv files
    movies_rdd, ratings_rdd, _ = create_rdd(spark)

    # Get the movie id, the average rating and the total number of ratings for the
    # movie \Cesare deve morire"
    mapped_movies = movies_rdd.map(lambda line: line.split(',')) \
                              .filter(lambda fields: len(fields) == 8 and fields[3].isdigit() and fields[6].isdigit()) \
                              .filter(lambda fields: fields[1] == "Cesare deve morire") \
                              .map(lambda fields: (int(fields[0]), ('movies', fields[1])))

    mapped_ratings = ratings_rdd.map(lambda line: line.split(',')) \
                                .filter(lambda fields: len(fields) == 4) \
                                .map(lambda fields: (int(fields[1]), (float(fields[2]), 1))) \
```

```python
                                        .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1])) \
                                        .map(lambda pair: (pair[0], ('ratings', pair[1][0] / pair[1][1], pair[1][1])))

    union = mapped_movies.union(mapped_ratings)

    movie_stats = union.groupByKey() \
                                .flatMap(lambda kv: [(kv[0], m[1], m[2]) for m in kv[1] if m[0] == 'ratings' for g in kv[1] if g[0] == 'movies'])

    execution_time, stats = timeit(movie_stats.collect)


    with open('../output/rdd_results/Q2RDD.txt', 'w') as f:
        f.write("Movie ID: %i, Number of ratings: %i, Average rating: %.2f" % (stats[0][0], stats[0][2], stats[0][1]))

    return execution_time


def query3(spark):
    # Fetch initial RDDs from the csv files
    movies_rdd, _, genres_rdd = create_rdd(spark)

    # Get the best Animation movie in terms of revenue for 1995
    mapped_movies = movies_rdd.map(lambda line: line.split(',')) \
                                    .filter(lambda fields: len(fields) == 8 and fields[3].isdigit() and fields[6].isdigit()) \
                                    .filter(lambda fields: int(fields[3]) == 1995 and int(fields[5]) > 0 and int(fields[6]) > 0) \
                                    .map(lambda fields: (int(fields[0]), ('movies', fields[1], int(fields[6]))))

    mapped_genres = genres_rdd.map(lambda line: line.split(',')) \
                                    .filter(lambda fields: len(fields) == 2 and fields[0].isdigit() and fields[1] == 'Animation') \
                                    .map(lambda fields: (int(fields[0]), ('genres', fields[1])))
```

```python

    # Union of the two RDDs
    union = mapped_movies.union(mapped_genres)

    # Group by key and transform the result
    joined = union.groupByKey() \
                  .flatMap(lambda kv: [(m[1], m[2]) for m in kv[1] if m[0] == 'movies' for g in kv[1] if g[0] == 'genres'])

    # Action takes place through the joined(), so the timeit() function is placed accordingly
    execution_time, best_animation_movie = timeit(joined.reduce, lambda movie, next_movie: movie if movie[1] > next_movie[1] else next_movie)

    with open('../output/rdd_results/Q3RDD.txt', 'w') as f:
        f.write("Best Animation Movie of 1995: {}, Revenue: {}".format(best_animation_movie[0], best_animation_movie[1]))

    return execution_time


def query4(spark):
    # Fetch initial RDDs from the csv files
    movies_rdd, _, genres_rdd = create_rdd(spark)

    # Get the most popular Comedy movie for each year after 1995
    mapped_movies = movies_rdd.map(lambda line: line.split(',')) \
                              .filter(lambda field: len(field) == 8 and field[3].isdigit() and field[6].isdigit()) \
                              .filter(lambda field: int(field[3]) > 1995 and float(field[7]) > 0) \
                              .map(lambda field: (int(field[0]), ('movies', int(field[3]), field[1], float(field[7]))))

    mapped_genres = genres_rdd.map(lambda line: line.split(',')) \
                              .filter(lambda field: len(field) == 2 and field[0].isdigit()) \
                              .filter(lambda field: field[1] == 'Comedy') \
                              .map(lambda field: (int(field[0]), ('genres', field[1])))
```

```python

    # Make union of movies and genres
    union = mapped_movies.union(mapped_genres)

    # Extract the best comedy per year
    best_comedy = union.groupByKey() \
                       .flatMap(lambda kv: [(m[1], (m[2], m[3])) for m in kv[1] if m[0] == 'movies' for g in kv[1] if g[0] == 'genres']) \
                       .reduceByKey(lambda x, y: x if x[1] > y[1] else y) \
                       .sortBy(lambda pair: pair[0])

    execution_time, best_comedy = timeit(best_comedy.collect)

    with open('../output/rdd_results/Q4RDD.txt', 'w') as f:
        for movie in best_comedy:
            f.write("The most popular Comedy of %i was %s, with a popularity score of %.2f\n" % (movie[0], movie[1][0], movie[1][1]))

    return execution_time


def query5(spark):
    # Fetch initial RDD from a csv
    movies_rdd, _, _ = create_rdd(spark)

    # Get the average revenue for each year
    mapped_movies = movies_rdd.map(lambda line: line.split(',')) \
                              .filter(lambda fields: len(fields) == 8 and fields[3].isdigit() and fields[6].isdigit()) \
                              .filter(lambda fields: int(fields[3]) > 0 and int(fields[6]) > 0) \
                              .map(lambda fields: (int(fields[3]), (int(fields[6]), 1))) \
                              .reduceByKey(lambda revenue, next_revenue: (revenue[0] + next_revenue[0], revenue[1] + next_revenue[1])) \
                              .map(lambda fields: (fields[0], fields[1][0] / fields[1][1])) \
                              .sortBy(lambda pair: pair[0])

```

```
    execution_time, results = timeit(mapped_movies.collect)

    with open('../output/rdd_results/Q5RDD.txt', 'w') as f:
        for result in results:
            f.write("Year %i had an average movie revenue of %.2f\n" % (result[0], result[1]))

    return execution_time
```

Snippet 3: `sql_csv.py`

```
from pyspark.sql.functions import collect_list
from pyspark.sql.types import StructField, StructType, IntegerType, FloatType, StringType
from utils import timeit, save_dataframe_output


# Set schemas of csv files
movies_schema = StructType([
    StructField("mv_id", IntegerType()),
    StructField("name", StringType()),
    StructField("description", StringType()),
    StructField("year", IntegerType()),
    StructField("duration", IntegerType()),
    StructField("prod_cost", IntegerType()),
    StructField("revenue", IntegerType()),
    StructField("popularity", FloatType())
])

ratings_schema = StructType([
    StructField("usr_id", IntegerType()),
    StructField("mv_id", IntegerType()),
    StructField("rating", FloatType()),
    StructField("time_stamp", IntegerType())
])

movie_genres_schema = StructType([
    StructField("mv_id", IntegerType()),
    StructField("genre", StringType())
])


def create_temp_tables(spark):
    # Load the aforementioned csv files into dataframes
    movies_df = spark.read.format('csv') \
```

```
            .options(header='false') \
            .schema(movies_schema) \
            .load("hdfs://master:9000/home/user/files/movies.csv")

    ratings_df = spark.read.format('csv') \
            .options(header='false') \
            .schema(ratings_schema) \
            .load("hdfs://master:9000/home/user/files/ratings.csv")

    movie_genres_df = spark.read.format('csv') \
            .options(header='false') \
            .schema(movie_genres_schema) \
            .load("hdfs://master:9000/home/user/files/movie_genres.csv")

    # Create temporary tables
    movies_df.createOrReplaceTempView("movies")
    ratings_df.createOrReplaceTempView("ratings")
    movie_genres_df.createOrReplaceTempView("genres")


def query1(spark):
    # Fetch relations
    create_temp_tables(spark)

    # Get the difference betwen revenue and production cost (i.e. profits) of every movie after 1995
    query = """
        SELECT year, concat_ws(',', collect_list(cast((revenue - prod_cost) AS string))) AS profit
        FROM movies
        WHERE prod_cost > 0 AND revenue > 0 AND year > 1995
        GROUP BY year
        ORDER BY year
    """

    execution_time, _ = timeit(spark.sql(query).show)
    query_output = save_dataframe_output(spark.sql(query))

    return execution_time, query_output


def query2(spark):
    # Fetch relations
    create_temp_tables(spark)
```

```python

    # Get the movie id, the average rating and the total number of ratings for the movie \Cesare deve morire"
    query = """
        SELECT m.mv_id, COUNT(r.usr_id) AS user_count, AVG(r.rating) AS average_rating
        FROM movies AS m
        JOIN ratings AS r ON m.mv_id = r.mv_id
        WHERE m.name = 'Cesare deve morire'
        GROUP BY m.mv_id
    """

    execution_time, _ = timeit(spark.sql(query).show)
    query_output = save_dataframe_output(spark.sql(query))

    return execution_time, query_output


def query3(spark):
    # Fetch relations
    create_temp_tables(spark)

    # Get the best Animation movie in terms of revenue for 1995
    query = """
        SELECT m.name AS movie_name, m.revenue AS revenue
        FROM movies AS m
        JOIN genres AS mg ON m.mv_id = mg.mv_id
        WHERE mg.genre = 'Animation' AND m.year = 1995 AND m.revenue > 0
        ORDER BY m.revenue DESC
        LIMIT 1
    """
    execution_time, _ = timeit(spark.sql(query).show)
    query_output = save_dataframe_output(spark.sql(query))

    return execution_time, query_output


def query4(spark):
    # Fetch relations
    create_temp_tables(spark)

    # Get the most popular Comedy movie for each year after 1995
    query = """
        WITH ranked_movies AS (
```

```
            SELECT m.year, m.name, m.popularity,
            ROW_NUMBER() OVER(PARTITION BY m.year ORDER BY m.popularity DESC) AS rank
            FROM movies AS m
            JOIN genres AS mg ON m.mv_id = mg.mv_id
            WHERE mg.genre = 'Comedy' AND m.year > 1995 AND m.popularity > 0 AND m.revenue > 0
            )
        SELECT year, name, popularity
        FROM ranked_movies
        WHERE rank = 1
        ORDER BY year
    """

    execution_time, _ = timeit(spark.sql(query).show)
    query_output = save_dataframe_output(spark.sql(query))

    return execution_time, query_output


def query5(spark):
    # Fetch relations
    create_temp_tables(spark)

    # Get the average revenue for each year
    query = """
        SELECT year, AVG(revenue) AS avg_revenue
        FROM movies
        WHERE year > 0 AND revenue > 0
        GROUP BY year
        ORDER BY year DESC
    """

    execution_time, _ = timeit(spark.sql(query).show)
    query_output = save_dataframe_output(spark.sql(query))

    return execution_time, query_output
```

Snippet 4: sql_parquet.py

```
from pyspark.sql.functions import collect_list
from utils import timeit


def create_temp_tables(spark):
```

```
    # Fetch data
    movies_df = spark.read.parquet("hdfs://master:9000/home/user/files/movies.parquet")
    ratings_df = spark.read.parquet("hdfs://master:9000/home/user/files/ratings.parquet")
    genres_df = spark.read.parquet("hdfs://master:9000/home/user/files/movie_genres.parquet")

    # Create temporary relations
    movies_df.createOrReplaceTempView("movies")
    ratings_df.createOrReplaceTempView("ratings")
    genres_df.createOrReplaceTempView("genres")


def query1(spark):
    # Fetch relations
    create_temp_tables(spark)

    # Get the difference betwen revenue and production cost (i.e. profits) of every movie after 1995
    query = """
        SELECT year, concat_ws(',', collect_list(cast((revenue - prod_cost) AS string))) AS profit
        FROM movies
        WHERE prod_cost > 0 AND revenue > 0 AND year > 1995
        GROUP BY year
        ORDER BY year
   """

    return timeit(spark.sql(query).show)


def query2(spark):
    # Fetch relations
    create_temp_tables(spark)

    # Get the movie id, the average rating and the total number of ratings for the movie \Cesare deve morire"
    query = """
        SELECT m.mv_id, COUNT(r.usr_id) AS user_count, AVG(r.rating) AS average_rating
        FROM movies AS m
        JOIN ratings AS r ON m.mv_id = r.mv_id
        WHERE m.name = 'Cesare deve morire'
        GROUP BY m.mv_id
```

```
    """

    return timeit(spark.sql(query).show)


def query3(spark):
    # Fetch relations
    create_temp_tables(spark)

    # Get the best Animation movie in terms of revenue for 1995
    query = """
        SELECT m.name AS movie_name, m.revenue AS revenue
        FROM movies AS m
        JOIN genres AS mg ON m.mv_id = mg.mv_id
        WHERE mg.genre = 'Animation' AND m.year = 1995 AND m.revenue > 0
        ORDER BY m.revenue DESC
        LIMIT 1
    """

    return timeit(spark.sql(query).show)


def query4(spark):
    # Fetch relations
    create_temp_tables(spark)

    # Get the most popular Comedy movie for each year after 1995
    query = """
        WITH ranked_movies AS (
            SELECT m.year, m.name, m.popularity,
            ROW_NUMBER() OVER(PARTITION BY m.year ORDER BY m.popularity DESC) AS rank
            FROM movies AS m
            JOIN genres AS mg ON m.mv_id = mg.mv_id
            WHERE mg.genre = 'Comedy' AND m.year > 1995 AND m.popularity > 0 AND m.revenue > 0
            )
        SELECT year, name, popularity
        FROM ranked_movies
        WHERE rank = 1
        ORDER BY year
    """

    return timeit(spark.sql(query).show)
```

```python


def query5(spark):
    # Fetch relations
    create_temp_tables(spark)

    # Get the average revenue for each year
    query = """
        SELECT year, AVG(revenue) AS avg_revenue
        FROM movies
        WHERE year > 0 AND revenue > 0
        GROUP BY year
        ORDER BY year DESC
    """

    return timeit(spark.sql(query).show)
```

Snippet 5: `joins.py`

```python
from utils import timeit


def create_rdd(spark):
    employees   = spark.textFile("hdfs://master:9000/home/user/files/employeesR.csv")
    departments = spark.textFile("hdfs://master:9000/home/user/files/departmentsR.csv")

    return employees, departments


def repartition_join(spark):
    # Fetch RDDs from the csv files
    employees, departments = create_rdd(spark)

    # Tag RDDs
    employees_tagged = employees \
                        .map(lambda line: line.split(',')) \
                        .map(lambda field: (int(field[0]), field[1], int(field[2]))) \
                        .map(lambda field: (field[2], ('employees', field)))

    departments_tagged = departments \
                          .map(lambda line: line.split(',')) \
```

```python
                                .map(lambda field: (int(field[0]), field[1])) \
                                .map(lambda field: (field[0], ('departments', field)))

    # Concatenate all RDDS
    union = employees_tagged.union(departments_tagged)

    # Perform the join for each key
    def join_records(records):
        employees_records   = [field[1] for field in records if field[0] == 'employees']
        departments_records = [field[1] for field in records if field[0] == 'departments']
        return [(e[1], e[0], d[1]) for e in employees_records for d in departments_records]

    # Extract union result
    joined_rdd = union.groupByKey() \
                      .flatMap(lambda pair: join_records(pair[1]))

    return timeit(joined_rdd.collect)


def broadcast_join(spark):
    # Fetch RDDs from the csv files
    employees, departments = create_rdd(spark)

    employees_rdd = employees \
                        .map(lambda line: line.split(',')) \
                        .map(lambda field: (int(field[0]), field[1], int(field[2])))

    departments_rdd = departments \
                        .map(lambda line: line.split(',')) \
                        .map(lambda field: (int(field[0]), field[1]))

    # Create a dictionary for department data
    dep_dict = {field[0]: field[1] for field in departments_rdd.collect()}

    # Broadcast the department data
    dep_broadcast = spark.broadcast(dep_dict)

    def map_func(field):
```

```
        # Extract the join key (department ID) and the department
        ↪ name using
        # the broadcast variable
        dep_id = field[2]
        dep_name = dep_broadcast.value.get(dep_id)

        # Joined data
        return (field[1], field[0], dep_name)

    joined_rdd = employees_rdd.map(lambda field: map_func(field))

    return timeit(joined_rdd.collect)
```

Snippet 6: `optimiser.py`

```
import io
import contextlib
from utils import timeit


def create_temp_tables(spark):
    dataframe = spark.read.format("parquet")

    ratings_dataframe = dataframe.load("hd
    ↪ fs://master:9000/home/user/files/ratings.parquet")
    genres_dataframe = dataframe.load("hd
    ↪ fs://master:9000/home/user/files/movie_genres.parquet")

    ratings_dataframe.registerTempTable("ratings")
    genres_dataframe.registerTempTable("genres")


def use_optimiser(spark, disabled = "N"):

   # Fetch relations
   create_temp_tables(spark)

   if disabled == "Y":
      spark.conf.set("spark.sql.cbo.enable", False)
      spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)
   elif disabled == "N":
      pass
   else:
      raise Exception ("This setting is not available.")

```

```
    query = """
            SELECT *
            FROM (SELECT * FROM genres LIMIT 100) AS g, ratings AS r
            WHERE r.mv_id = g.mv_id
        """

    stdout = io.StringIO()
    with contextlib.redirect_stdout(stdout):
        spark.sql(query).explain()

    # Get the captured standard output
    query_plan = stdout.getvalue()

    return timeit(spark.sql(query).show), query_plan
```

Snippet 7: `utils.py`

```
import io
import time
import contextlib


def timeit(func, *args, **kwargs):
    """
    Measure execution time of a function for a single run.

    Args:
        func: Function to be executed.
        *args: Variable length argument list for the function.
        **kwargs: Arbitrary keyword arguments for the function.

    Returns:
        tuple: A tuple containing the execution time and the result
    ↪ of the function call.
    """
    start = time.time()
    result = func(*args, **kwargs)
    end = time.time()
    execution_time = end - start

    return execution_time, result


def save_dataframe_output(dataframe):
    """
```

```python
    Function to capture and return the complete output of a PySpark DataFrame.

    Args:
        dataframe (pyspark.sql.DataFrame): The DataFrame whose output is to be captured.

    Returns:
        str: The entire output of the DataFrame as a string.
    """

    # Calculate the number of rows in the DataFrame
    row_count = dataframe.count()

    # Create a StringIO object
    stdout = io.StringIO()

    # Execute show() on DataFrame and capture the output
    with contextlib.redirect_stdout(stdout):
        dataframe.show(n=row_count, truncate=False)

    # Get the captured standard output
    return stdout.getvalue()
```

Snippet 8: `main.py`

```python
# Import SparkSession
from pyspark.sql import SparkSession

# Import RDD queries
from rdd import query1 as rdd_query1
from rdd import query2 as rdd_query2
from rdd import query3 as rdd_query3
from rdd import query4 as rdd_query4
from rdd import query5 as rdd_query5

# Import SQL-on-csv queries
from sql_csv import query1 as sql_csv_query1
from sql_csv import query2 as sql_csv_query2
from sql_csv import query3 as sql_csv_query3
from sql_csv import query4 as sql_csv_query4
from sql_csv import query5 as sql_csv_query5

# Import SQL-on-Parquet queries
from sql_parquet import query1 as sql_parquet_query1
```

```
from sql_parquet import query2 as sql_parquet_query2
from sql_parquet import query3 as sql_parquet_query3
from sql_parquet import query4 as sql_parquet_query4
from sql_parquet import query5 as sql_parquet_query5

# Import RDD joins
from joins import broadcast_join
from joins import repartition_join

# Import Optimiser script
from optimiser import use_optimiser

# Import csv-to-parquet converter
from csv_to_parquet import convert_csv_to_parquet


def part1():
    ###################### Task 1 ######################
    # Convert CSVs to Parquet; run only once. Should you
    # wish to repeat the process, comment it out.
    convert_csv_to_parquet()


    ################## Tasks 2, 3 & 4  ##################
    times = {}

    # Calculate execution times for each query (Tasks 2, 3 & 4)
    for i in range(1, 6):
        spark_csv = SparkSession \
                    .builder \
                    .appName("All-use session") \
                    .getOrCreate()
        spark_parquet = spark_csv
        spark_rdd = spark_csv.sparkContext

        rdd_query_name     = 'rdd_query%s' % (i)
        parquet_query_name = 'sql_parquet_query%s' % (i)
        csv_query_name     = 'sql_csv_query%s' % (i)

        times[rdd_query_name]                 =
        ↪  globals()[rdd_query_name](spark_rdd)
        times[parquet_query_name], _          =
        ↪  globals()[parquet_query_name](spark_parquet)
        times[csv_query_name], query_output =
        ↪  globals()[csv_query_name](spark_csv)
```

```python

        # Save the query-output, in dataframe format, on a text file
        with open('../output/df_results/Q%sDF.txt' % i, 'w') as f:
            f.write(query_output)

        # Consistency in execution times
        spark_csv.stop()
        spark_parquet.stop()
        spark_rdd.stop()
        print(times)

    # Compute execution times and write to a text file
    with open('../output/part_1_times.txt', 'w') as f:
        for query, execution_time in times.items():
            f.write('%s: %.2f seconds\n' % (query, execution_time))


def part2():
    ###################### Task 1 ######################
    times = {}

    spark = SparkSession \
                .builder \
                .appName("All-use session") \
                .getOrCreate() \
                .sparkContext

    times['Broadcast Join'], broadcast_result = broadcast_join(spark)
    times['Repartition Join'], _              = repartition_join(spark)

    # Compute execution times and write to a text file
    with open('../output/join_type_times.txt', 'w') as f:
        for query, execution_time in times.items():
            f.write('%s: %.2f seconds\n' % (query, execution_time))

    # Save the result to text files
    with open('../output/join_outputs.txt', 'w') as f:
        for result in broadcast_result:
            f.write(str(result) + '\n')

    # Consistency in execution times
    spark.stop()

```

```python

    ##################### Task 2 #####################
    times = {}

    # Two instances are created since Spark tends to keep
    # metadata from each run in order to optimise reading
    # and calculating future queries.
    spark = SparkSession \
            .builder \
            .appName('Using Catalyst') \
            .getOrCreate()
    sc = spark

    times["Using Catalyst"], with_catalyst         = use_optimiser(spark)
    times["Without using Catalyst"], without_catalyst = use_optimiser(sc, disabled="Y")

    spark.stop()
    sc.stop()

    # Compute execution times and write to a text file
    with open('../output/catalyst_times.txt', 'w') as f:
        for query, execution_time in times.items():
            f.write('%s: %.2f seconds\n' % (query, execution_time[0]))

    # Save the optimised query plan to text file
    with open('../output/optimised_plan.txt', 'w') as f:
        f.write(with_catalyst)

    # Save the non-optimised query plan to text file
    with open('../output/non_optimised_plan.txt', 'w') as f:
        f.write(without_catalyst)


if __name__ == "__main__":
    part1()
    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/.