



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    # query 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 queries (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

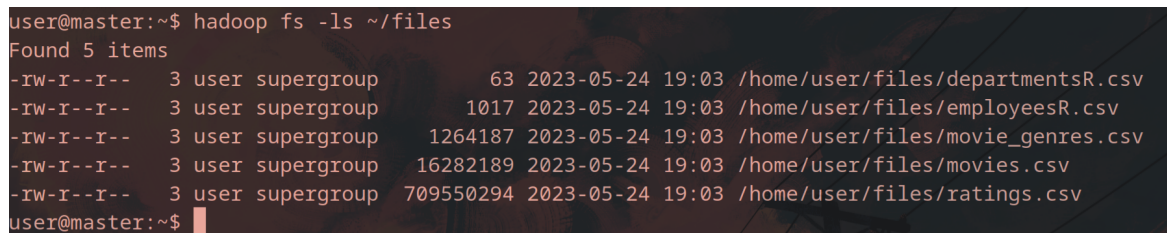
```
hadoop fs -mkdir -p ~/files
```

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.



```
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 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 between 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.

```

18 movies = movies_rdd.map(lambda line: line.split(',')) \
19                      .filter(lambda field: len(field) > 7 and
20                               ↪ field[3].isdigit() and field[6].isdigit()
21                               ↪ and field[5].isdigit()) \
22                      .map(lambda field: (int(field[3]),
23                                          ↪ (int(field[6]), int(field[5])))) \
24                      .filter(lambda field: field[0] > 1995 and
25                               ↪ field[1][0] > 0 and field[1][1] > 0) \
26                      .map(lambda field: (field[0], str(field[1][0] -
27                                          ↪ field[1][1]))) \
28                      .reduceByKey(lambda v1, v2: v1 + ", " + v2) \
29                      .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 between revenue and production cost (i.e. profits) of every movie after 1995.”

```

59 query = """
60     SELECT year, concat_ws(',', collect_list(cast((revenue -
61     ↪ prod_cost) AS string))) AS profit
62     FROM movies
63     WHERE prod_cost > 0 AND revenue > 0 AND year > 1995
64     GROUP BY year
65     ORDER BY year
66 """

```

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.

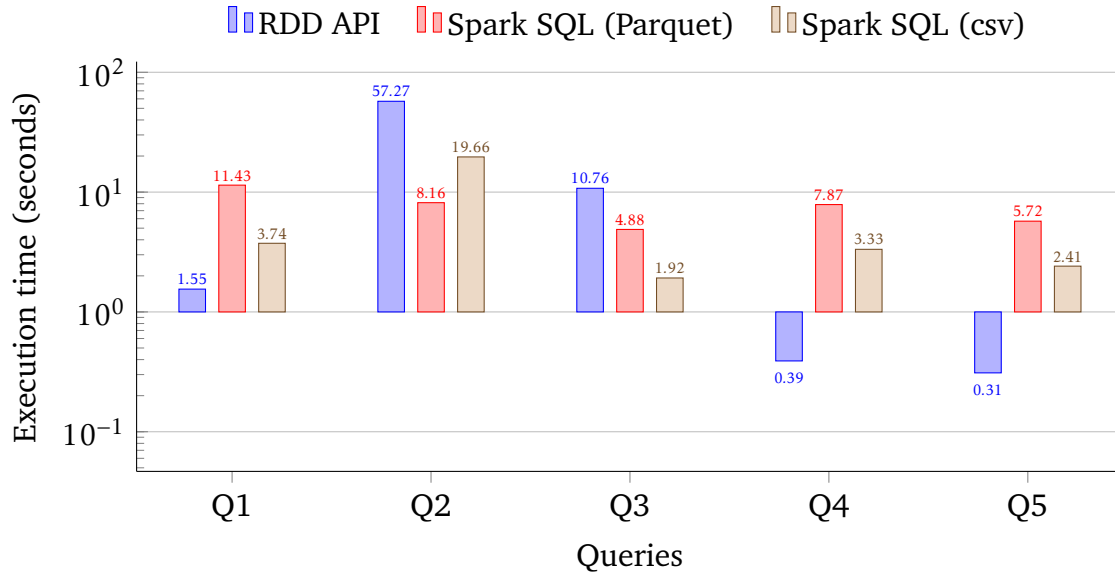


Figure 3: Query execution-times, for all queries, using RDDs, CSVs and Parquet.

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.

```
11 def repartition_join(spark):
12     # Fetch RDDs from the csv files
13     employees, departments = create_rdd(spark)
14
15     # Tag RDDs
16     employees_tagged = employees \
17         .map(lambda line: line.split(',')) \
18         .map(lambda field: (int(field[0]), field[1],
19             ↪ 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']
```

```

32     departments_records = [field[1] for field in records if
    ↪ field[0] == 'departments']
33     return [(e[1], e[0], d[1]) for e in employees_records for d in
    ↪ departments_records]
34
35     # Extract union result
36     joined_rdd = union.groupByKey() \
37         .flatMap(lambda pair: join_records(pair[1]))
38
39     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.

```

42 def broadcast_join(spark):
43     # Fetch RDDs from the csv files
44     employees, departments = create_rdd(spark)
45
46     employees_rdd = employees \
47         .map(lambda line: line.split(',')) \
48         .map(lambda field: (int(field[0]), field[1],
    ↪ int(field[2])))
49
50     departments_rdd = departments \
51         .map(lambda line: line.split(',')) \
52         .map(lambda field: (int(field[0]), field[1]))
53
54     # Create a dictionary for department data
55     dep_dict = {field[0]: field[1] for field in
    ↪ 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
    ↪ name using
62         # the broadcast variable
63         dep_id = field[2]

```

```

64         dep_name = dep_broadcast.value.get(dep_id)
65
66         # Joined data
67         return (field[1], field[0], dep_name)
68
69     joined_rdd = employees_rdd.map(lambda field: map_func(field))
70
71     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
    """

```

Execution without Optimiser  9.73

Execution with Optimiser  5.86

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

```
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 def query2(spark):
41     # Fetch initial RDDs from the csv files
42     movies_rdd, ratings_rdd, _ = create_rdd(spark)
43
44     # Get the movie id, the average rating and the total number of
45     ↪ ratings for the
46     # movie \Cesare deve morire"
47     mapped_movies = movies_rdd.map(lambda line: line.split(',')) \
48         .filter(lambda fields: len(fields) == 8
49             ↪ and fields[3].isdigit() and
50             ↪ fields[6].isdigit()) \
51         .filter(lambda fields: fields[1] ==
52             ↪ "Cesare deve morire") \
53         .map(lambda fields: (int(fields[0]),
54             ↪ ('movies', fields[1])))
55
56     mapped_ratings = ratings_rdd.map(lambda line: line.split(',')) \
57         .filter(lambda fields: len(fields) ==
58             ↪ 4) \
59         .map(lambda fields: (int(fields[1]),
60             ↪ (float(fields[2]), 1))) \

```

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

```

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

```

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

```

142     execution_time, results = timeit(mapped_movies.collect)
143
144     with open('../output/rdd_results/Q5RDD.txt', 'w') as f:
145         for result in results:
146             f.write("Year %i had an average movie revenue of %.2f\n"
147                     ↪      % (result[0], result[1]))
147
148     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,
   ↪      FloatType, StringType
3  from utils import timeit, save_dataframe_output
4
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)
70     query_output = save_dataframe_output(spark.sql(query))
71
72     return execution_time, query_output
73
74 def query2(spark):
75     # Fetch relations
76     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
112
113 def query4(spark):
114     # Fetch relations
115     create_temp_tables(spark)
116
117     # Get the most popular Comedy movie for each year after 1995
118     query = """
119         WITH ranked_movies AS (

```

```
118         SELECT m.year, m.name, m.popularity,
119                ROW_NUMBER() OVER(PARTITION BY m.year ORDER BY
↪ m.popularity DESC) AS rank
120         FROM movies AS m
121         JOIN genres AS mg ON m.mv_id = mg.mv_id
122         WHERE mg.genre = 'Comedy' AND m.year > 1995 AND
↪ m.popularity > 0 AND m.revenue > 0
123     )
124     SELECT year, name, popularity
125     FROM ranked_movies
126     WHERE rank = 1
127     ORDER BY year
128     """
129
130     execution_time, _ = timeit(spark.sql(query).show)
131     query_output = save_dataframe_output(spark.sql(query))
132
133     return execution_time, query_output
134
135
136 def query5(spark):
137     # Fetch relations
138     create_temp_tables(spark)
139
140     # Get the average revenue for each year
141     query = """
142         SELECT year, AVG(revenue) AS avg_revenue
143         FROM movies
144         WHERE year > 0 AND revenue > 0
145         GROUP BY year
146         ORDER BY year DESC
147     """
148
149     execution_time, _ = timeit(spark.sql(query).show)
150     query_output = save_dataframe_output(spark.sql(query))
151
152     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_
      ↳ fs://master:9000/home/user/files/movies.parquet")
8      ratings_df = spark.read.parquet("hd_
      ↳ fs://master:9000/home/user/files/ratings.parquet")
9      genres_df = spark.read.parquet("hd_
      ↳ fs://master:9000/home/user/files/movie_genres.parquet")
10
11     # Create temporary relations
12     movies_df.createOrReplaceTempView("movies")
13     ratings_df.createOrReplaceTempView("ratings")
14     genres_df.createOrReplaceTempView("genres")
15
16
17     def query1(spark):
18         # Fetch relations
19         create_temp_tables(spark)
20
21         # Get the difference between revenue and production cost (i.e.
22         ↳ profits) of every movie after 1995
23         query = """
24             SELECT year, concat_ws(',', collect_list(cast((revenue -
25             ↳ prod_cost) AS string))) AS profit
26             FROM movies
27             WHERE prod_cost > 0 AND revenue > 0 AND year > 1995
28             GROUP BY year
29             ORDER BY year
30         """
31
32         return timeit(spark.sql(query).show)
33
34     def query2(spark):
35         # Fetch relations
36         create_temp_tables(spark)
37
38         # Get the movie id, the average rating and the total number of
39         ↳ ratings for the movie \Cesare deve morire"
40         query = """
41             SELECT m.mv_id, COUNT(r.usr_id) AS user_count, AVG(r.rating)
42             ↳ AS average_rating
43             FROM movies AS m
44             JOIN ratings AS r ON m.mv_id = r.mv_id
45             WHERE m.name = 'Cesare deve morire'
46             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
59         ↪ > 0
60         ORDER BY m.revenue DESC
61         LIMIT 1
62     """
63
64     return timeit(spark.sql(query).show)
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
75             ↪ m.popularity DESC) AS rank
76             FROM movies AS m
77             JOIN genres AS mg ON m.mv_id = mg.mv_id
78             WHERE mg.genre = 'Comedy' AND m.year > 1995 AND
79             ↪ m.popularity > 0 AND m.revenue > 0
80             )
81         SELECT year, name, popularity
82         FROM ranked_movies
83         WHERE rank = 1
84         ORDER BY year
85     """
86
87     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]))) \
21         .map(lambda field: (field[2], ('employees',
22         ↪ field)))
23
24     departments_tagged = departments \
25         .map(lambda line: line.split(',')) \

```

```
23         .map(lambda field: (int(field[0]),
24                               ↪ field[1])) \
25         .map(lambda field: (field[0],
26                               ↪ ('departments', field)))
27
28     # Concatenate all RDDs
29     union = employees_tagged.union(departments_tagged)
30
31     # Perform the join for each key
32     def join_records(records):
33         employees_records = [field[1] for field in records if
34                               ↪ field[0] == 'employees']
35         departments_records = [field[1] for field in records if
36                                ↪ field[0] == 'departments']
37         return [(e[1], e[0], d[1]) for e in employees_records for d
38                 ↪ in departments_records]
39
40     # Extract union result
41     joined_rdd = union.groupByKey() \
42                     .flatMap(lambda pair: join_records(pair[1]))
43
44     return timeit(joined_rdd.collect)
45
46 def broadcast_join(spark):
47     # Fetch RDDs from the csv files
48     employees, departments = create_rdd(spark)
49
50     employees_rdd = employees \
51                     .map(lambda line: line.split(',')) \
52                     .map(lambda field: (int(field[0]), field[1],
53                                         ↪ int(field[2])))
54
55     departments_rdd = departments \
56                        .map(lambda line: line.split(',')) \
57                        .map(lambda field: (int(field[0]), field[1]))
58
59     # Create a dictionary for department data
60     dep_dict = {field[0]: field[1] for field in
61                 ↪ departments_rdd.collect()}
62
63     # Broadcast the department data
64     dep_broadcast = spark.broadcast(dep_dict)
65
66     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_
10         ↪ fs://master:9000/home/user/files/ratings.parquet")
11     genres_dataframe = dataframe.load("hd_
12         ↪ fs://master:9000/home/user/files/movie_genres.parquet")
13
14     ratings_dataframe.registerTempTable("ratings")
15     genres_dataframe.registerTempTable("genres")
16
17 def use_optimiser(spark, disabled = "N"):
18
19     # Fetch relations
20     create_temp_tables(spark)
21
22     if disabled == "Y":
23         spark.conf.set("spark.sql.cbo.enable", False)
24         spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)
25     elif disabled == "N":
26         pass
27     else:
28         raise Exception ("This setting is not available.")

```



```
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
24    return execution_time, result
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

```

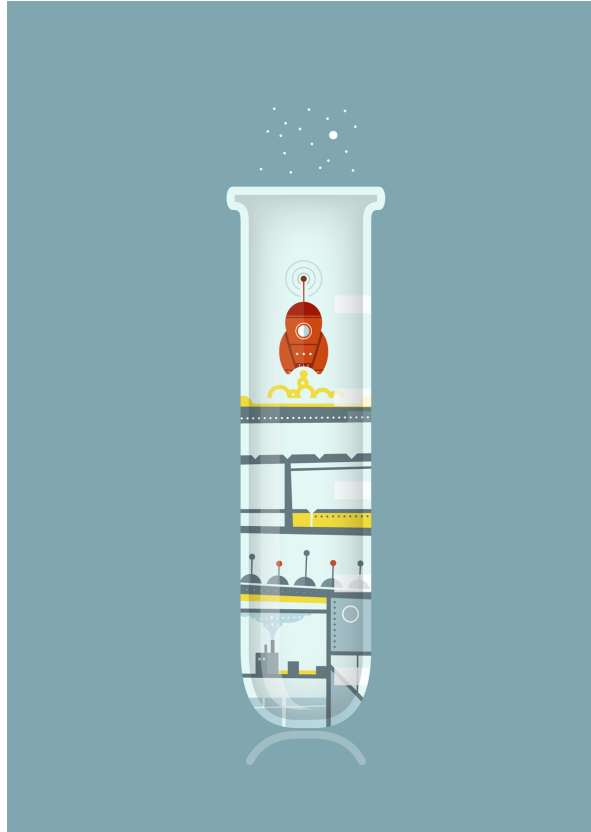
```
20 from sql_parquet import query2 as sql_parquet_query2
21 from sql_parquet import query3 as sql_parquet_query3
22 from sql_parquet import query4 as sql_parquet_query4
23 from sql_parquet import query5 as sql_parquet_query5
24
25 # Import RDD joins
26 from joins import broadcast_join
27 from joins 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_csv = SparkSession \
49             .builder \
50             .appName("All-use session") \
51             .getOrCreate()
52         spark_parquet = spark_csv
53         spark_rdd = spark_csv.sparkContext
54
55         rdd_query_name = 'rdd_query%s' % (i)
56         parquet_query_name = 'sql_parquet_query%s' % (i)
57         csv_query_name = 'sql_csv_query%s' % (i)
58
59         times[rdd_query_name] =
60             ↪ globals()[rdd_query_name](spark_rdd)
61         times[parquet_query_name], _ =
62             ↪ globals()[parquet_query_name](spark_parquet)
63         times[csv_query_name], query_output =
64             ↪ globals()[csv_query_name](spark_csv)
```

```

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

```

```
105
106     ##### Task 2 #####
107     times = {}
108
109     # Two instances are created since Spark tends to keep
110     # metadata from each run in order to optimise reading
111     # and calculating future queries.
112     spark = SparkSession \
113         .builder \
114         .appName('Using Catalyst') \
115         .getOrCreate()
116     sc = spark
117
118     times["Using Catalyst"], with_catalyst =
119         ↪ use_optimiser(spark)
120     times["Without using Catalyst"], without_catalyst =
121         ↪ use_optimiser(sc, disabled="Y")
122
123     spark.stop()
124     sc.stop()
125
126     # Compute execution times and write to a text file
127     with open('../output/catalyst_times.txt', 'w') as f:
128         for query, execution_time in times.items():
129             f.write('%s: %.2f seconds\n' % (query,
130                 ↪ execution_time[0]))
131
132     # Save the optimised query plan to text file
133     with open('../output/optimised_plan.txt', 'w') as f:
134         f.write(with_catalyst)
135
136     # Save the non-optimised query plan to text file
137     with open('../output/non_optimised_plan.txt', 'w') as f:
138         f.write(without_catalyst)
139
140 if __name__ == "__main__":
141     part1()
142     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/>.