

PART 1

TASK 1

1) Start Hadoop Distributed Filesystem (HDFS)

start-dfs.sh

2) Create the folder “files” in HDFS

hadoop fs -mkdir ~/files

3) Upload the files in the ~/files folder in the HDFS.

**hadoop fs -put departmentsR.csv movies.csv movie_genres.csv ratings.csv employeesR.csv
~/files**

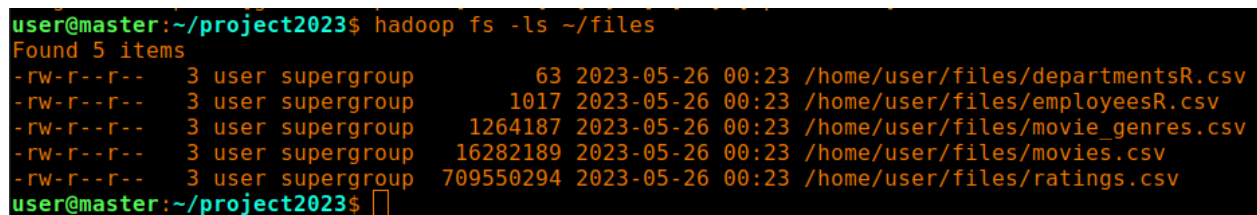
4) Show that the files are uploaded

hadoop fs -ls ~/files

Output:

Found 5 items

```
-rw-r--r--  3 user supergroup      63 2023-05-26 00:23 /home/user/files/departmentsR.csv
-rw-r--r--  3 user supergroup    1017 2023-05-26 00:23 /home/user/files/employeesR.csv
-rw-r--r--  3 user supergroup  1264187 2023-05-26 00:23 /home/user/files/movie_genres.csv
-rw-r--r--  3 user supergroup  16282189 2023-05-26 00:23 /home/user/files/movies.csv
-rw-r--r--  3 user supergroup  709550294 2023-05-26 00:23 /home/user/files/ratings.csv
```



```
user@master:~/project2023$ hadoop fs -ls ~/files
Found 5 items
-rw-r--r--  3 user supergroup      63 2023-05-26 00:23 /home/user/files/departmentsR.csv
-rw-r--r--  3 user supergroup    1017 2023-05-26 00:23 /home/user/files/employeesR.csv
-rw-r--r--  3 user supergroup  1264187 2023-05-26 00:23 /home/user/files/movie_genres.csv
-rw-r--r--  3 user supergroup  16282189 2023-05-26 00:23 /home/user/files/movies.csv
-rw-r--r--  3 user supergroup  709550294 2023-05-26 00:23 /home/user/files/ratings.csv
user@master:~/project2023$
```

5) Convert csv to parquet files and upload them to HDFS :

I created the simple script **csvToParquet.py** to turn the csv files to parquet files. It is included in the src directory.

Then we can display the files using **hadoop fs -ls** :

```

user@master:~/project2023$ hadoop fs -ls /home/user/files
Found 5 items
-rw-r--r-- 3 user supergroup      63 2023-05-26 00:23 /home/user/files/departmentsR.csv
-rw-r--r-- 3 user supergroup    1017 2023-05-26 00:23 /home/user/files/employeesR.csv
-rw-r--r-- 3 user supergroup  1264187 2023-05-26 00:23 /home/user/files/movie_genres.csv
-rw-r--r-- 3 user supergroup  16282189 2023-05-26 00:23 /home/user/files/movies.csv
-rw-r--r-- 3 user supergroup  709550294 2023-05-26 00:23 /home/user/files/ratings.csv
user@master:~/project2023$ nano csvToParquet.py
user@master:~/project2023$ spark-submit csvToParquet.py
23/05/26 18:47:19 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
CREATING PARQUET FILES FROM .CSV FILES.WAIT PLEASE.
user@master:~/project2023$
user@master:~/project2023$ hadoop fs -ls /home/user/files
Found 10 items
drwxr-xr-x - user supergroup      0 2023-05-26 18:48 /home/user/files/departments.parquet
-rw-r--r-- 3 user supergroup      63 2023-05-26 00:23 /home/user/files/departmentsR.csv
drwxr-xr-x - user supergroup      0 2023-05-26 18:48 /home/user/files/employees.parquet
-rw-r--r-- 3 user supergroup    1017 2023-05-26 00:23 /home/user/files/employeesR.csv
drwxr-xr-x - user supergroup      0 2023-05-26 18:48 /home/user/files/genres.parquet
-rw-r--r-- 3 user supergroup  1264187 2023-05-26 00:23 /home/user/files/movie_genres.csv
-rw-r--r-- 3 user supergroup  16282189 2023-05-26 00:23 /home/user/files/movies.csv
drwxr-xr-x - user supergroup      0 2023-05-26 18:47 /home/user/files/movies.parquet
-rw-r--r-- 3 user supergroup  709550294 2023-05-26 00:23 /home/user/files/ratings.csv
drwxr-xr-x - user supergroup      0 2023-05-26 18:48 /home/user/files/ratings.parquet
user@master:~/project2023$

```

After running the script we see that we have both the **.csv** files and the **.parquet** files in our distributed filesystem. In order to achieve the above results we used the script : **csvToParquet.py**

TASK 2

The scripts for each query are included in the **src** directory.

TASK 3

The scripts for each query are included in the **src** directory.

NOTES on Task 2 , 3 :

To run the queries we first initialize spark:

1)start-all.sh

Then run each query using :

2) spark-submit query.py

After every query, the resulting csv file is written into the hadoop distributed filesystem .So in order to retrieve the result from the distributed filesystem into our own local filesystem we need to use the command :

hadoop fs -get home/user/files/file_name.csv ~/project2023/

to download the file “file_name.csv” into the local ‘project2023’ folder.

Since by default the partitions are created in a directory with a particular name, when we use the get method we get the whole directory. We can then cd into the directory and find our **part-*.csv** files which are the different file partitions that were stored in the HDFS .

Coalesce() :

(1)When we use “.coalesce(1)” after each query we force the system to output the result into a single partition thereby merging all the resulting partitions into one. Thus when we inspect the directory with name **file_name.csv** we can see only one file which is the final result , and can easily be displayed with **cat file_name.csv**

(2) If on the other hand the `coalesce(1)` is not used we get different csv files (partitions) and can display each one with `cat` but in order to get the total result we should merge all the different partitions.

I use the (1) alternative for the **RDD** queries but only to create the final (single) output and not for the execution timing since `coalesce()` means shuffling files and transporting them to other partitions and has an impact on time. **I only time the `collect()` action after every query on the default partitions that were produced.**

Regarding comments and each query explanation, queries on csv (**I name them “dataframe” queries**) and rdd queries are filled with comments to highlight particular important points in each solution but **parquet queries do not have a lot of comments since parquet queries are almost identical to the csv queries except the fact that when reading the parquet files we do not need to define the schema again, since it is precomputed (from the conversion we did at the start).** For both parquet and dataframe(csv) queries I use the same **SparkSQL API**.

Regarding the output of each query :

- For the RDD queries : I use the function **`final_rdd.coalesce(1).saveAsTextFile("/home/user/files/Q1_alt_rdd_results.csv")`** to write only the output tuples into the output file.
- For the Csv/Parquet queries I use : **`spark-submit Q3_dataframe.py | grep -v Time > Q3_alt_csv_results.csv`** to isolate the output results and redirect them to a output file without including the execution time .I also use this in Rdd – Query 3. I implemented this query in 2 ways : 1) Sorting the final revenues of the Animation movies and taking the top 2) Reducing the revenues keeping each time the max revenue until we get only the max revenue animation movie. The results included are based on (2) but I also include (1) solution for completeness.
- For the timing i just run the queries commenting out the lines that produce or redirect to output and I print only the Execution Time by using `grep Time` now (instead of inverting) : **`spark-submit Q3_dataframe.py | grep Time`**
- Regarding **Query 2** using RDD I solved it in 2 ways : 1) Using simple reduce side join which is pretty expensive as shown in the results . 2) Using broadcast join , broadcasting the movie to all the mappers and doing the join there. For the total plot I use (1) but I also present a comparison between the 2 and include both of them in the src directory.

TASK 4

I present here a screenshot of the execution times of the queries .I will also provide a csv file with the output and the plot(jpg) file . It was generated out of the cluster after the screenshots were taken and the execution times were measured. It is also possible to create a dataframe and populate it in each script with its execution time , but this requires downloading pip and pandas on the cluster and to be honest I did not want to have the possibility of any pip package error / inconsistencies with python inside the vm , so I opted for the other approach , though not out of laziness.

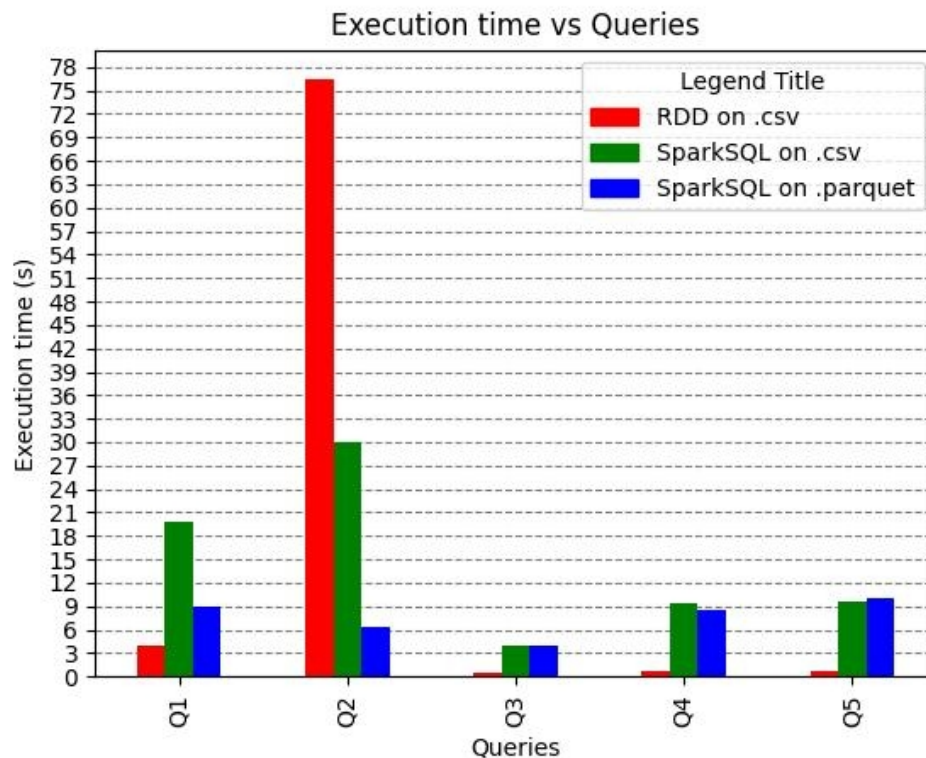
Here is the screenshot of the execution times for all combinations of queries and rdd/csv/parquet .The order of each query is

- 1)Rdd (Example: Q1_rdd.py)
- 2)Csv (Example: Q1_dataframe.py)

3)parquet (Example: Q1_parquet.py)

```
user@master:~/project2023/project_v2$ spark-submit Q1_rdd.py | grep Time
23/07/14 02:28:16 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 3.94 s
user@master:~/project2023/project_v2$ spark-submit Q1_dataframe.py | grep Time
23/07/14 02:28:46 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 19.80 s
user@master:~/project2023/project_v2$ spark-submit Q1_parquet.py | grep Time
23/07/14 02:29:35 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 8.87 s
user@master:~/project2023/project_v2$ spark-submit Q2_rdd.py | grep Time
23/07/14 02:30:49 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 76.40 s
user@master:~/project2023/project_v2$ spark-submit Q2_dataframe.py | grep Time
23/07/14 02:32:28 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 29.97 s
user@master:~/project2023/project_v2$ spark-submit Q2_parquet.py | grep Time
23/07/14 02:33:44 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 6.36 s
user@master:~/project2023/project_v2$ spark-submit Q3_rdd.py | grep Time
23/07/14 02:34:37 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 0.40 s
user@master:~/project2023/project_v2$ spark-submit Q3_dataframe.py | grep Time
23/07/14 02:35:04 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 3.94 s
user@master:~/project2023/project_v2$ spark-submit Q3_parquet.py | grep Time
23/07/14 02:35:44 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 3.90 s
user@master:~/project2023/project_v2$ spark-submit Q4_rdd.py | grep Time
23/07/14 02:36:28 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 0.62 s
user@master:~/project2023/project_v2$ spark-submit Q4_dataframe.py | grep Time
23/07/14 02:36:59 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 9.38 s
user@master:~/project2023/project_v2$ spark-submit Q4_parquet.py | grep Time
23/07/14 02:37:54 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 8.51 s
user@master:~/project2023/project_v2$ spark-submit Q5_rdd.py | grep Time
23/07/14 02:38:49 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 0.63 s
user@master:~/project2023/project_v2$ spark-submit Q5_dataframe.py | grep Time
23/07/14 02:39:15 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 9.56 s
user@master:~/project2023/project_v2$ spark-submit Q5_parquet.py | grep Time
23/07/14 02:40:10 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 9.95 s
```

I present now the bar plot of the execution times for each query :



Q2

For Q2 **with the Rdd** api I also opted to implement it additionally using broadcast join instead of the default join algorithm. The script

Q2_broadcast_rdd.py that implements Q2 using this idea is also included in the src directory. I present below the execution time with this join in comparison to the original.

```
user@master:~/project2023$ spark-submit Q2_broadcast_rdd.py
23/07/12 23:58:42 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 46.30 s
Presenting 1 sample : [(96821, (5096, 3.8701923076923075))]
```

```
user@master:~/project2023/project_v2$ spark-submit Q2_rdd.py | grep Time
23/07/14 02:30:49 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 76.40 s
```

We can see that we have saved a lot of time but it still takes long to execute and from my understanding we still have a lot of shuffling after the join. I first broadcast the particular movie to all the mappers that have the ratings partitions, emitting the key-value pair (0, particular_movie_id) dictionary and then in each ratings partition I filter and keep only the movies that have movie_id = particular_movie_id which can be done in $O(N)$ for N movies in a particular partition. But to calculate the average and the count of ratings for this particular movie we still need to reduceBy the particular_movie_id and thus we still shuffle and move data into the same reducer from different mappers. So we still have slow execution time but definitely improved in comparison to the original version.

BAR PLOT ANALYSIS

Generally by observing the bar plot we can see that in RDD API we have more control over how the query will be implemented since we can change the order of operations, define our own groupBy/partitioning functions and even repartition data. That is why we can get faster implementations by utilizing the RDD API. Only in Query 2 we observe slow execution since I use a simple reduce-side join and therefore it is unoptimized and a lot of shuffling is required. By implementing repartition join we could get better execution times as we got with broadcast join. Comparing now the csv and the parquet queries we generally get faster retrieval with parquet files since parquet is a columnar format designed to for efficient storage and retrieval. With parquet we skip the scheme building phase because the scheme is already present and also the retrieval is faster. The result is that in most cases we get better execution times with parquet and for smaller datasets we get similar performances. Csv in comparison is row based and slower as we see in most cases. If we need to read whole rows from relations csv will be preferred while for more restrictive (requiring part of row) finetuned queries parquet will be better.

Part 2

Task 1

I present here the screenshot of running broadcast_join and repartition_join. The scripts are included in the src directory , and have comments inside explaining the process.

```
user@master:~/project2023$ spark-submit broadcast_join.py
23/07/13 01:01:22 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 0.46 s
Presenting 1 sample : [(('Dep A', (6, 'Timothy Minert')), ('Dep A', (22, 'Katherine Roundtree')), ('Dep A', (28, 'Douglas Frye')), ('Dep A', (34, 'Bobby Benz')), ('Dep A', (38, 'Michael Lovecchio')), ('Dep A', (41, 'Jamie Jennings')), ('Dep A', (49, 'Joseph Faust')), ('Dep B', (2, 'Nancy Blanchard')), ('Dep B', (11, 'Dona Kneip')), ('Dep B', (12, 'Kelly Brummett')), ('Dep B', (14, 'Thomas Jensen')), ('Dep B', (15, 'Kathleen Flannery')), ('Dep B', (17, 'Robert Hawkin')), ('Dep B', (18, 'Leone Regan')), ('Dep B', (19, 'Cindy Alexander')), ('Dep B', (20, 'Joanna Robles')), ('Dep B', (24, 'Arthur Sera')), ('Dep B', (26, 'Mario Cabral')), ('Dep B', (27, 'Ebony Viner')), ('Dep B', (30, 'Hattie Dorko')), ('Dep B', (31, 'Shannon Wood')), ('Dep B', (33, 'Matthew Capple')), ('Dep B', (36, 'Magdalena Kelly')), ('Dep B', (42, 'Martha Cooper')), ('Dep C', (7, 'Suzanne Adams')), ('Dep C', (8, 'Anthony Lovell')), ('Dep C', (13, 'James Brunson')), ('Dep D', (3, 'Dustin Tureson')), ('Dep D', (9, 'James Wickstrom')), ('Dep D', (37, 'Benito Cunningham')), ('Dep D', (39, 'Rosa Alexander')), ('Dep E', (21, 'Edward Hodges')), ('Dep E', (32, 'George Paul')), ('Dep E', (35, 'Ted Conant')), ('Dep F', (16, 'Sara Wier')), ('Dep F', (29, 'Alicia Nolen')), ('Dep F', (43, 'Nellie Sawchuk')), ('Dep F', (45, 'Blanche Strand')), ('Dep F', (46, 'Angela Hatfield')), ('Dep F', (47, 'Barbara Gill')), ('Dep G', (1, 'Elizabeth Jordan')), ('Dep G', (4, 'Melissa Mcglone')), ('Dep G', (5, 'William Varela')), ('Dep G', (10, 'Tommy Cannon')), ('Dep G', (23, 'Johannie Vogel')), ('Dep G', (25, 'Vanessa Gazaille')), ('Dep G', (40, 'Ahmed Prickett')), ('Dep G', (44, 'Benjamin Hill')), ('Dep G', (48, 'Crystal Milligan')), ('Dep G', (50, 'Patrick Carruthers'))]
user@master:~/project2023$ spark-submit repartition_join.py
23/07/13 01:01:50 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Time for execution is : 0.76 s
Presenting 1 sample : [(('Dep A', (6, 'Timothy Minert')), ('Dep A', (22, 'Katherine Roundtree')), ('Dep A', (28, 'Douglas Frye')), ('Dep A', (34, 'Bobby Benz')), ('Dep A', (38, 'Michael Lovecchio')), ('Dep A', (41, 'Jamie Jennings')), ('Dep A', (49, 'Joseph Faust')), ('Dep B', (2, 'Nancy Blanchard')), ('Dep B', (11, 'Dona Kneip')), ('Dep B', (12, 'Kelly Brummett')), ('Dep B', (14, 'Thomas Jensen')), ('Dep B', (15, 'Kathleen Flannery')), ('Dep B', (17, 'Robert Hawkin')), ('Dep B', (18, 'Leone Regan')), ('Dep B', (19, 'Cindy Alexander')), ('Dep B', (20, 'Joanna Robles')), ('Dep B', (24, 'Arthur Sera')), ('Dep B', (26, 'Mario Cabral')), ('Dep B', (27, 'Ebony Viner')), ('Dep B', (30, 'Hattie Dorko')), ('Dep B', (31, 'Shannon Wood')), ('Dep B', (33, 'Matthew Capple')), ('Dep B', (36, 'Magdalena Kelly')), ('Dep B', (42, 'Martha Cooper')), ('Dep C', (7, 'Suzanne Adams')), ('Dep C', (8, 'Anthony Lovell')), ('Dep C', (13, 'James Brunson')), ('Dep D', (3, 'Dustin Tureson')), ('Dep D', (9, 'James Wickstrom')), ('Dep D', (37, 'Benito Cunningham')), ('Dep D', (39, 'Rosa Alexander')), ('Dep E', (21, 'Edward Hodges')), ('Dep E', (32, 'George Paul')), ('Dep E', (35, 'Ted Conant')), ('Dep F', (16, 'Sara Wier')), ('Dep F', (29, 'Alicia Nolen')), ('Dep F', (43, 'Nellie Sawchuk')), ('Dep F', (45, 'Blanche Strand')), ('Dep F', (46, 'Angela Hatfield')), ('Dep F', (47, 'Barbara Gill')), ('Dep G', (1, 'Elizabeth Jordan')), ('Dep G', (4, 'Melissa Mcglone')), ('Dep G', (5, 'William Varela')), ('Dep G', (10, 'Tommy Cannon')), ('Dep G', (23, 'Johannie Vogel')), ('Dep G', (25, 'Vanessa Gazaille')), ('Dep G', (40, 'Ahmed Prickett')), ('Dep G', (44, 'Benjamin Hill')), ('Dep G', (48, 'Crystal Milligan')), ('Dep G', (50, 'Patrick Carruthers'))]
```

Analysis on the join algorithms

Broadcast Join

Regarding hash join the algorithm works in the following way. We first read the smaller relation departments and build a dictionary where for each department id we assign the corresponding department name (example : 0 → department A). Department id is used as the key. Then we broadcast this dictionary (index) into all the mappers. We then read the employees relation which is bigger. We again use as key the department id and emit (employee_id, employee_name). To find the department_name of each employee we can easily now probe the dictionary that is present in all the mappers and ask for the name corresponding to this employee's department_id. So if the employee in the current mapper has dep_id = 0 we probe broadcast_value[0] and get the corresponding department name. This method is simple and also very fast since for each mapper we just probe a dictionary in O(1) or O(2) steps N times if we have N employees in this partition. There is no wide dependency involved as well as no sorting and no shuffling.

Repartition Join

Regarding repartition join we read the departments relation and use as a key the department_id while we also emit as a value : (1, department_name) where 1 stands for dataset 1. We do the same for the employees dataset emitting as a value : (2, (employee_id, employee_name)) where 2 stands for dataset 2. The key is again the department_id. Then we sort both relations on the key (taken from the suggested paper) in order to create the partitions. To merge the 2 datasets we use the union operation. Then we use groupBy operation to create an iterable for each key (department_id). After that we parse each iterable and call the function we created to distinguish the datasets in order to have a consistent order in the results : First the department_name, then employee_id, employee_name. Without tagging the dataset and using this function we could possibly get results with mixed order, with some results having department_name first and others having the (employee_id, employee_name) first. Testing this implementation with and without the sorting of each relation I observed that without sorting the 2 relations before the union, the query execution time is a lot slower. Also sorting the relation produced from the union before using groupBy accelerates the execution which probably means that the partitions with the same key are a lot faster to retrieve and send to the reducer this way than searching to find all the data within a partition. Simple repartition join still requires reducers to complete and has wide dependencies so it is slower than the broadcast join but faster than attempting to join the partitions from a random unordered dataset.

Task 2

We run the query with the unoptimized version using sort merge join and with the optimized version using broadcast hash join. We expect the broadcast hash join to be a lot faster since it involves no shuffle + sort whereas the sort merge join involves shuffling-repartitioning and sorting. I present the query script given. It was modified in a minor way so the optimization on and off could reflect the parameters Y and N, and in the if statements I added the proper spark configurations.

As we can see from the script below I deactivate the Broadcast Join with **spark.conf.set("spark.sql.autoBroadcastJoinThreshold",-1)** in the N part. We force spark to dismiss the broadcast join independently of the broadcast relation size. Spark uses the sort merge join

as default join algorithm (fallback from broadcast join) if broadcast is not available or if the relation size is larger than the threshold. So if we run this command spark will use Sort merge join since it prioritizes the deactivation of broadcast join and uses Sort merge join after that. In the Y part of the if statement, Spark uses the Broadcast Hash Join as the default optimal join algorithm.

```

"""
Created on Tue Jul  4 19:47:02 2023

@author: st_ko
"""
from pyspark.sql import SparkSession
import sys, time

disabled = sys.argv[1]
spark = SparkSession.builder.appName('query1-sql').getOrCreate()

if disabled == "N":
    spark.conf.set("spark.sql.autoBroadcastJoinThreshold",-1)
    #spark.conf.set("spark.sql.join.preferSortMergeJoin",False)
    #spark.conf.set("spark.sql.crossJoin.enabled",True)
    #spark.conf.set("spark.sql.adaptive.skewJoin.enabled",False)
    #spark.conf.set("spark.sql.adaptive.enabled",True)
elif disabled == 'Y':
    pass
else:
    raise Exception ("This setting is not available.")

df = spark.read.format("parquet")
df1 = df.load("hdfs://master:9000/home/user/files/ratings.parquet")
df2 = df.load("hdfs://master:9000/home/user/files/genres.parquet")
df1.registerTempTable("ratings")
df2.registerTempTable("movie_genres")

sqlString = \
"SELECT * " + \
"FROM " + \
" (SELECT * FROM movie_genres LIMIT 100) as g, " + \
" ratings as r " + \
"WHERE " + \
" r.movie_id = g.movie_id"

t1 = time.time()
spark.sql(sqlString).show()
t2 = time.time()

# explain the query
spark.sql(sqlString).explain()
print("Time with choosing join type  %s is %.4f sec."%( "enabled - Using sortmerge" if
disabled == 'N' else "disabled - Using Hash Join", t2-t1))

```

Regarding the output of the 2 join algorithms I first create a temporary output file for each join (temp1.txt ,temp2.txt) where I redirect the output of each join algorithm (Y and N).Then I use grep again with the command : **grep '|'+---** to get only the sql (row) results of the query and redirect them to an output file for each join algorithm. These 2 files are called **query_optimization_on.txt (hash join)** and **query_optimization_off.txt (sort merge)** and are the sql query results and included in the output directory. To isolate the Time we can use **grep Time** with the temporary files. All these processes are

shown below .

Running with or without optimization and Redirection of outputs

```
user@master:~/project2023/optimization_scripts$ spark-submit query optimization.py N > temp1.txt
23/07/18 01:03:28 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
user@master:~/project2023/optimization_scripts$ spark-submit query optimization.py Y > temp2.txt
23/07/18 01:04:14 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
user@master:~/project2023/optimization_scripts$ cat temp1.txt | grep '\|+---' > query_optimization_off.txt
user@master:~/project2023/optimization_scripts$ cat temp2.txt | grep '\|+---' > query_optimization_on.txt
```

Execution times .First is **Sort merge join** then **Hash join**.

```
user@master:~/project2023/optimization_scripts$ cat temp1.txt | grep Time
Time with choosing join type enabled is 12.9137 sec.
user@master:~/project2023/optimization_scripts$ cat temp2.txt | grep Time
Time with choosing join type disabled is 7.3400 sec.
```

I present here the results of running with join optimization off – Using sort merge join.

```
user@master:~/project2023/optimization_scripts$ cat temp1.txt
-----+-----+-----+-----+-----+
movie_id|movie_genre|user_id|movie_id|rating|timestamp|
-----+-----+-----+-----+-----+
451|Drama|390|451|2.0|978580515|
451|Drama|533|451|2.0|974647675|
451|Drama|882|451|3.0|957630720|
451|Drama|2383|451|1.0|956417083|
451|Drama|2975|451|2.0|1188657823|
451|Drama|2981|451|2.0|902036771|
451|Drama|3019|451|3.0|847875644|
451|Drama|3251|451|3.0|849630863|
451|Drama|3792|451|4.0|945277919|
451|Drama|3950|451|2.0|949415361|
451|Drama|4020|451|3.0|927569171|
451|Drama|4066|451|3.0|859107793|
451|Drama|4323|451|3.5|1881878684|
451|Drama|5077|451|3.0|1012862719|
451|Drama|5153|451|3.0|975402624|
451|Drama|5519|451|3.0|981340584|
451|Drama|6201|451|3.0|941209715|
451|Drama|6294|451|3.0|947261311|
451|Drama|6513|451|2.5|1146070190|
451|Drama|6714|451|3.0|1099084203|
-----+-----+-----+-----+
only showing top 20 rows

== Physical Plan ==
*(6) SortMergeJoin [movie_id#8], [movie_id#1], Inner
  -- *(3) Sort [movie_id#8 ASC NULLS FIRST], false, 0
    +- Exchange hashpartitioning(movie_id#8, 200)
      +- *(2) Filter isNotNull(movie_id#8)
        +- *(2) GlobalLimit 100
          +- Exchange SinglePartition
            +- *(1) LocalLimit 100
              +- *(1) FileScan parquet [movie_id#8,movie_genre#9] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://master:9000/home/user/files/genres.parquet], Partition
Filters: [], PushedFilters: [], ReadSchema: struct<movie_id:int,movie_genre:string>
-- *(5) Sort [movie_id#1 ASC NULLS FIRST], false, 0
  +- Exchange hashpartitioning(movie_id#1, 200)
    +- *(4) Project [user_id#0, movie_id#1, rating#2, timestamp#3]
      +- *(4) Filter isNotNull(movie_id#1)
        +- *(4) FileScan parquet [user_id#0,movie_id#1,rating#2,timestamp#3] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://master:9000/home/user/files/ratings.parquet]
Filters: [], PushedFilters: [IsNotNull(movie_id)], ReadSchema: struct<user_id:int,movie_id:int,rating:float,timestamp:int>
Time with choosing join type enabled is 12.9137 sec.
```

And here the results of running with join optimization on – Using broadcast join

```

user@master:~/project2023/optimization_scripts$ cat temp2.txt
-----+-----+-----+-----+-----+
movie_id| movie_genre|user_id|movie_id|rating| timestamp|
-----+-----+-----+-----+-----+
5| Comedy| 2| 5| 3.0| 867039249|
5| Crime| 2| 5| 3.0| 867039249|
1408| Adventure| 15| 1408| 5.0|1346008714|
1408| Action| 15| 1408| 5.0|1346008714|
524| Crime| 24| 524| 2.0| 979870484|
524| Drama| 24| 524| 2.0| 979870484|
1408| Adventure| 24| 1408| 3.0| 979870731|
1408| Action| 24| 1408| 3.0| 979870731|
902| Adventure| 36| 902| 2.0| 965349039|
902| Science Fiction| 36| 902| 2.0| 965349039|
902| Fantasy| 36| 902| 2.0| 965349039|
5| Comedy| 40| 5| 4.0| 862515493|
5| Crime| 40| 5| 4.0| 862515493|
902| Adventure| 41| 902| 4.0|1445255341|
902| Science Fiction| 41| 902| 4.0|1445255341|
902| Fantasy| 41| 902| 4.0|1445255341|
524| Crime| 43| 524| 2.5|1179200291|
524| Drama| 43| 524| 2.5|1179200291|
902| Adventure| 43| 902| 4.5|1179197071|
902| Science Fiction| 43| 902| 4.5|1179197071|
-----+-----+-----+-----+-----+
only showing top 20 rows

== Physical Plan ==
*(3) BroadcastHashJoin [movie_id#8], [movie_id#1], Inner, BuildLeft
-- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as bigint)))
+- *(2) Filter isnotnull(movie_id#8)
   +- *(2) GlobalLimit 100
      +- Exchange SinglePartition
         +- *(1) LocalLimit 100
            +- *(1) FileScan parquet [movie_id#8,movie_genre#9] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://master:9000/home/user/files/genres.parquet], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<movie_id:int,movie_genre:string>
            +- *(3) Project [user_id#0, movie_id#1, rating#2, timestamp#3]
               +- *(3) Filter isnotnull(movie_id#1)
                  +- *(3) FileScan parquet [user_id#0,movie_id#1,rating#2,timestamp#3] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://master:9000/home/user/files/ratings.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(movie_id)], ReadSchema: struct<user_id:int,movie_id:int,rating:float,timestamp:int>
Time with choosing join type disabled is 7.3400 sec.

```

Then let's compare the 2 physical plans only :

Plan with Sort Merge Join – Optimization Off

```

== Physical Plan ==
*(6) SortMergeJoin [movie_id#8], [movie_id#1], Inner
+- *(3) Sort [movie_id#8 ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(movie_id#8, 200)
      +- *(2) Filter isnotnull(movie_id#8)
         +- *(2) GlobalLimit 100
            +- Exchange SinglePartition
               +- *(1) LocalLimit 100
                  +- *(1) FileScan parquet [movie_id#8,movie_genre#9] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://master:9000/home/user/files/genres.parquet], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<movie_id:int,movie_genre:string>
            +- *(5) Sort [movie_id#1 ASC NULLS FIRST], false, 0
               +- Exchange hashpartitioning(movie_id#1, 200)
                  +- *(4) Project [user_id#0, movie_id#1, rating#2, timestamp#3]
                     +- *(4) Filter isnotnull(movie_id#1)
                        +- *(4) FileScan parquet [user_id#0,movie_id#1,rating#2,timestamp#3] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://master:9000/home/user/files/ratings.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(movie_id)], ReadSchema: struct<user_id:int,movie_id:int,rating:float,timestamp:int>
Time with choosing join type enabled is 12.0137 sec.

```

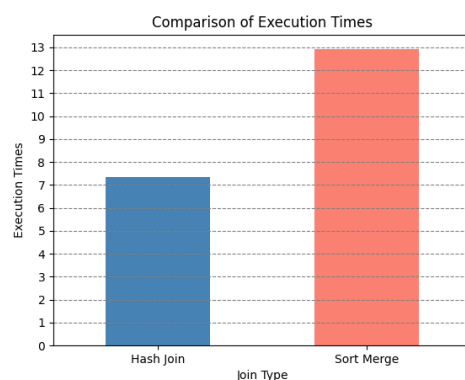
Plan with Broadcast Hash Join – Optimization On

```

== Physical Plan ==
*(3) BroadcastHashJoin [movie_id#8], [movie_id#1], Inner, BuildLeft
-- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as bigint)))
+- *(2) Filter isnotnull(movie_id#8)
   +- *(2) GlobalLimit 100
      +- Exchange SinglePartition
         +- *(1) LocalLimit 100
            +- *(1) FileScan parquet [movie_id#8,movie_genre#9] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://master:9000/home/user/files/genres.parquet], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<movie_id:int,movie_genre:string>
            +- *(3) Project [user_id#0, movie_id#1, rating#2, timestamp#3]
               +- *(3) Filter isnotnull(movie_id#1)
                  +- *(3) FileScan parquet [user_id#0,movie_id#1,rating#2,timestamp#3] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://master:9000/home/user/files/ratings.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(movie_id)], ReadSchema: struct<user_id:int,movie_id:int,rating:float,timestamp:int>
Time with choosing join type disabled is 7.3400 sec.

```

Also here I present the plot of the execution times comparison of the 2 join types :

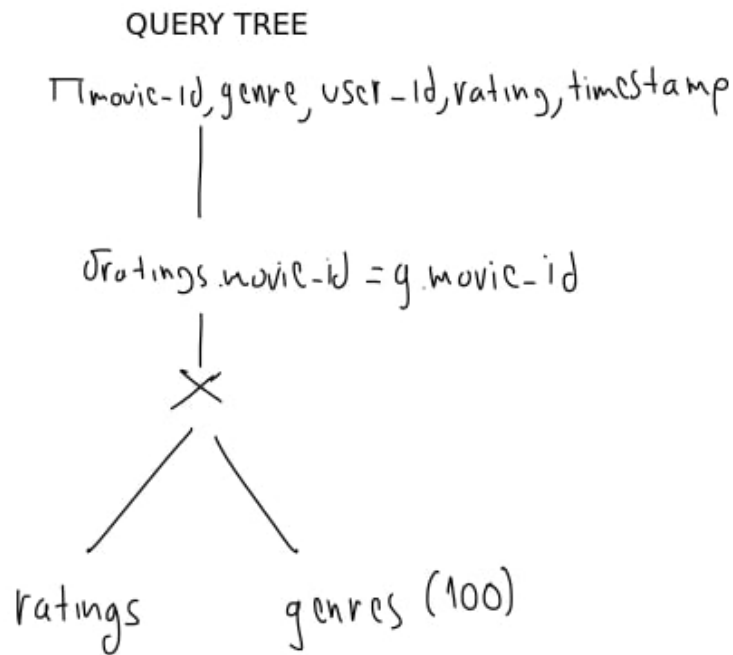


As we can see Broadcast Hash join is indeed faster than sort merge Join

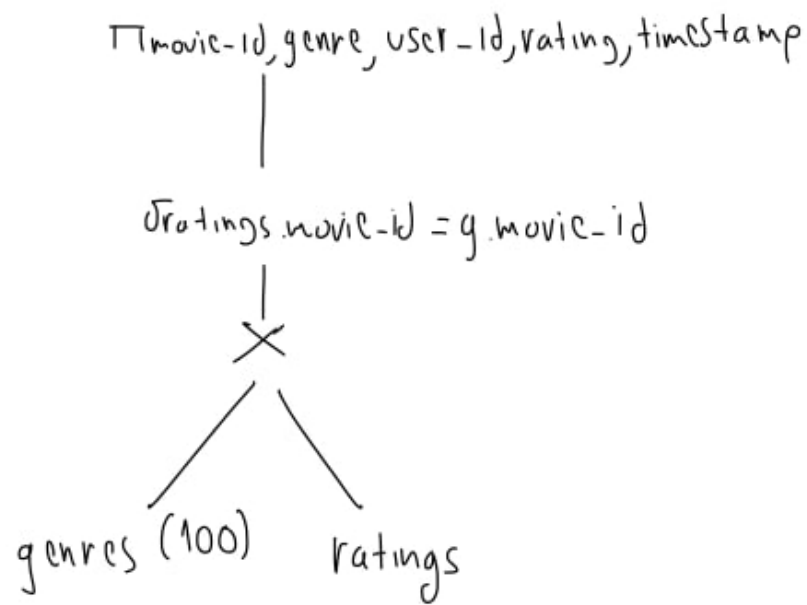
General Logical Optimization

First I present an overview of the sql query we run and how logical optimization takes place for this query and next I present and analyze both join algorithms.

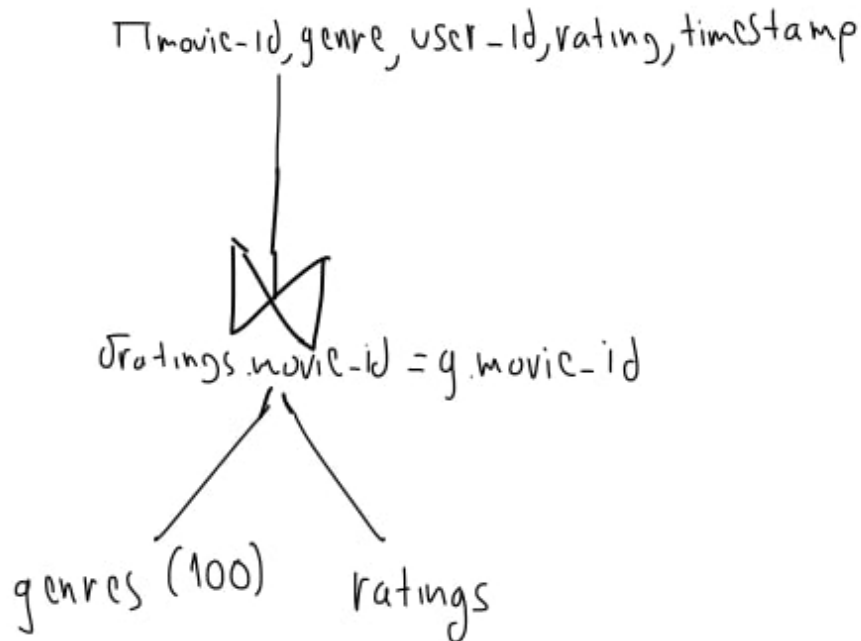
```
SELECT *  
FROM (SELECT * FROM movies_genres LIMIT 100) as g,ratings r  
WHERE r.movie_id = g.movie_id
```



1) Rearrange binary operators. Move more restrictive relations into left subtree.



2) Convert cartesian product with selection afterwards into join.



We could also move the projections downwards but all the columns are required in the final projection so I did not implement this step.

***** Below I use a little freely the constants in the complexity referring to tuples while in fact the complexity refers to pages.**

SORT MERGE JOIN

Using sort merge join for both Relations we have to materialize both of them. Running **wc -l ratings.csv** we see that ratings has 26024289 tuples and therefore it is a very large Relation. Running **du -h ratings.csv** we also get that the file is 677Mb.

If we make the assumption that we use a general algorithm for sorting Complexity $O(K \log K)$ then we need for sorting ratings.csv : $O(N \log N)$ and $O(M \log M)$ for sorting genres.csv .

For merge cost we have $O(M + N)$ since genres.csv is the primary relation here and ratings.csv is the foreign relation. Here we have a primary- foreign key (movie_id) relation between genres and ratings since for each movie_id in genres, which defines a partition, we can have many tuples in ratings with the same movie_id, i.e. in the same partition. The 1st relation genres determines how many times we read each corresponding partition in the 2nd relation, here 1 time due to the primary-foreign key relation; this is why the complexity results in $O(M + N)$.

Sort + merge could also be done with less I/O by combining sort merging with join merging if we have appropriate page buffers. We first generate sorted runs of size B for ratings and genres. We allocate one buffer for each run of ratings and one for each run of genres. We then merge the runs of ratings, merge the runs of genres and merge the ratings and genres streams as they are generated. The join condition (`movie_id`) is applied as we merge the 2 relations and discard tuples in the Cartesian product that do not satisfy the join condition. So we can read and write ratings and genres in one pass $O(2N + 2M)$ and read and join them in another pass $O(N+M)$. In total we have $O(3(N+M))$ with the above technique. (Ramakrishnan, R. and Gehrke, J. (1998) *Database Management Systems*, McGraw-Hill, Inc., New York. . p 294-295 An important refinement)

Query Tree Explanation

By observing the query tree we can see the following steps:

- 1a) File scan genres
- 1b) Keep only 100 tuples
- 2) Filter : Check if `movie_id` != Null .Mandatory check if it is key.
- 3a) Then the hash partitioner is called to partition ta genres based on `movie_id` as key and transfer them to the reducers.(shuffle)
- 3b) Then the genres relation is sorted in the reducers according to the `movie_id`.
- 4a) File scan ratings
- 4b) Filter : Check if `movie_id` != Null .Mandatory check if it is key.
- 4c) Projection of fields (`user_id`, `movie_id`, `rating`, `timestamp`) is done. According to query optimization theory projections are moved as low as possible in the query trees.
- 4d) Then the hash partitioner is called to partition the ratings based on the `movie_id` as key (shuffle). The ratings with a specific `movie_id`, belong to the same partition and are transferred to the reducers where the genres with the corresponding `movie_id` are located, so that they are copartitioned to make the join.
- 5) Then the genres relation is sorted in the reducers according to the `movie_id`.
- 6) Finally the join of ratings and genres is done in the reducers.

Because the relations are initially in different partitions and unsorted, this join requires shuffling and sorting and this is costly especially for the larger ratings relation. Using sort merge join here we have map + reduce phase and the join is done on the reducers hence we have reduce side join and wide dependencies.

BROADCAST HASH JOIN

The workers who have been assigned the partitions of ratings Relation, probe for each tuple in their partition in the hashtable of genres and receive the corresponding tuple (if any). If we consider $O(1)$ or $O(2)$ for the probing phase in the hashtable, then the only cost is essentially that of the linear scan of each tuple of ratings in each mapper. We note that neither sort nor shuffle occurs here so the process is very cheap. The broadcasting of genres is also cheap since the genres relation has only 100 tuples and the construction of the hashtable/dictionary is also cheap in memory resources due to the small relation. We could consider that the cost at each mapper for the join is $O(M)$ where M is the number of tuples it has. Using broadcast hash join the join is done on the mappers and we have a map side join which has narrow dependencies. Hence we observe that this join is much cheaper and faster than the above sort merge join.

Query Tree Explanation

Similarly for the broadcast hash join from the query tree we observe the following steps :

- 1a) File scan genres
- 1b) keep only 100 tuples.

2)Filter : Check if movie_id != Null .Mandatory check if it is key.

3a)Broadcast genres relation into all mappers.

3b)File scan ratings

3c)Filter : Check if movie_id != Null

3d)Project the fields (user_id, movie_id, rating, timestamp).According to the theory of query optimization the projections are moved as low as possible in the query trees.

3e)Broadcast Hash join of the relations genres and ratings with respect to the movie_id key. The join is fast since all mappers that have some partitions of ratings have the whole relation genres therefore for each tuple of ratings in their partition they can retrieve the corresponding tuple of genres by lookup in the hashtable/dictionary of genres.A tuple of genres corresponds to more tuples of ratings.