Final Project

Chalkiopoulos Georgios[1] and Konomi Evdhoksia[2]

[1]p3352124 , [2]p3352105

March 27, 2022

Part 1.

Task 1 Create a files directory on HDFS and then upload the CSVs to the files directory: provide the commands needed for the directory creation and the files upload. Provide a print screen that shows the csv files in the directory you created (0.5 points). Convert the files to parquet and upload them on HDFS (0.5 points)

The files were downloaded using the Dropbox link provided in a folder named `project` in the `home` directory:

```
> wget https://www.dropbox.com/s/qidkffat5gukkjb/datasets
.tar.gz?dl=0
> mv datasets.tar.gz?dl=0 datasets.tar.gz
> tar -xzf datasets.tar.gz
```

After unzipping the files we ended up with the `.csv` files.

```
> ls
departmentsR.csv  movie_genres.csv  ratings.csv
employeesR.csv    movies.csv
```

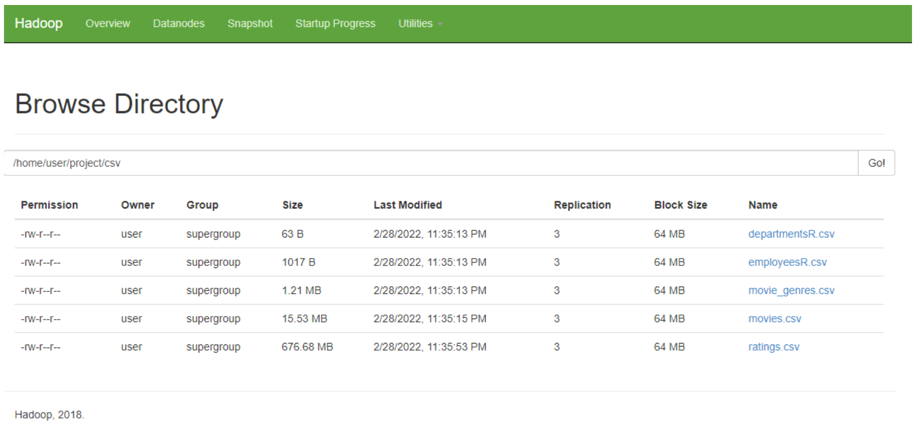
Then we created an HDFS directory to upload our files:

```
> hadoop fs -mkdir -p ~/project/csv
```

Finally we uploaded the files to the created HDFS directory:

```
> hadoop fs -put *.csv ~/project/csv
```

The files were successfully been uploaded to Hadoop as shown in Figure 1:



Hadoop Overview Datanodes Snapshot Startup Progress Utilities

Browse Directory

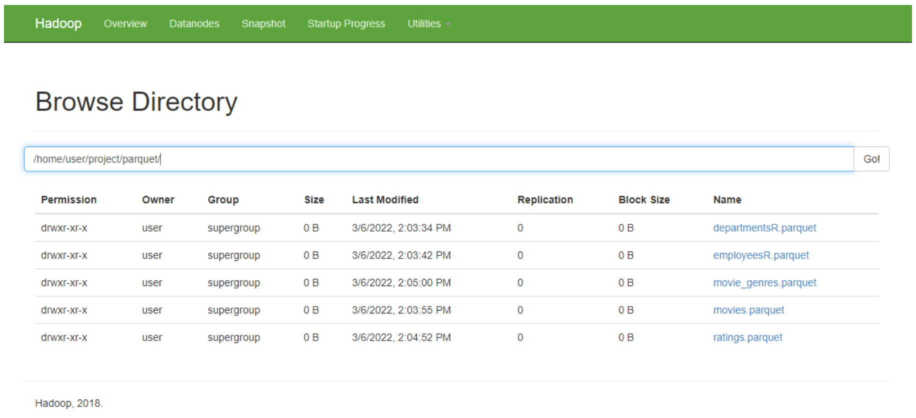
/home/user/project/csv

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	user	supergroup	63 B	2/28/2022, 11:35:13 PM	3	64 MB	departmentsR.csv
-rw-r--r--	user	supergroup	1017 B	2/28/2022, 11:35:13 PM	3	64 MB	employeesR.csv
-rw-r--r--	user	supergroup	1.21 MB	2/28/2022, 11:35:13 PM	3	64 MB	movie_genres.csv
-rw-r--r--	user	supergroup	15.53 MB	2/28/2022, 11:35:15 PM	3	64 MB	movies.csv
-rw-r--r--	user	supergroup	676.68 MB	2/28/2022, 11:35:53 PM	3	64 MB	ratings.csv

Hadoop, 2018.

Figure 1: Hadoop `/project/csv` directory including the files.

The next step was to create the parquet files using the `.csv` files. The code used to create the files may be found in the corresponding `csv2parquet.py` file and the HDFS directory containing the files is shown in Figure 2.



Hadoop Overview Datanodes Snapshot Startup Progress Utilities

Browse Directory

/home/user/project/parquet/

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
drwxr-xr-x	user	supergroup	0 B	3/6/2022, 2:03:34 PM	0	0 B	departmentsR.parquet
drwxr-xr-x	user	supergroup	0 B	3/6/2022, 2:03:42 PM	0	0 B	employeesR.parquet
drwxr-xr-x	user	supergroup	0 B	3/6/2022, 2:05:00 PM	0	0 B	movie_genres.parquet
drwxr-xr-x	user	supergroup	0 B	3/6/2022, 2:03:55 PM	0	0 B	movies.parquet
drwxr-xr-x	user	supergroup	0 B	3/6/2022, 2:04:52 PM	0	0 B	ratings.parquet

Hadoop, 2018.

Figure 2: Hadoop `/project/parquet` directory including the files.

Task 2 Using RDDs write code to answer the following queries (Q1-Q5) you can use the csv or parquet files you uploaded on HDFS (2.75 points, 0.55x5)

This part of the project was performed using `.csv` files. The codes, along with the results can be found in the corresponding folder (Part1_Task2). Each `.py` file was executed in the master and the output was saved in a file, locally using the following command (`@` is the Task and is the query):

```
spark-submit @_Q#.py > results_@_Q#.txt
```

Before discussing the specific queries some general information will be given. In order to load the RDDs we created a sparksession, which also includes the sparkContext at the end, as follows:

```
    sc = SparkSession \
        .builder \
        .appName(<name of the app>) \
        .getOrCreate() \
        .sparkContext
```

The text files were read using the `textFile` method. Since the files are `.csv` we used a `.map` method using the default lambda function as follows:

```
table = sc.textFile(<hdfs path>) \
            .map(lambda x: (x.split(",")))
```

A high level overview for each question may be found below:

Q1 For every year after 1995 print the difference between the money spent to create the movie and the revenue of the movie (revenue – production cost)?

For this query we did the first map which filtered out the empty records (in these files the empty records contain a space). Using the output we performed another map in order to keep the movies after 1995 and only positive revenue/production values ($>$ 0). Finally a final map function was done in which we keep the movie id, the year and calculate the difference between the revenue and the cost.

Q2 For the movie "Cesare deve morire" find and print the movies id and then search how many users rated the movie and what the average rating was?

In this case we loaded two files (movies and rating) and only kept the movie 'Cesare deve morire'. As a result we didn't have to apply any sanity checks regrading the revenue/year/movie name. We then filtered the ratings file, keeping all the records for the movie, and used a map method to only keep the ratings. From there the count and mean methods provide the needed results which are also printed.

Q3 What was the best in term of revenue Animation movie of 1995?

Similar to Q1 we excluded empty values from the movies RDD and only kept the Animation genre from the genre RDD. After applying a map, to filter the movies after 1995 with positive revenue we created key, value pairs and joined the two RDDs. Finally we converted the revenue to integer and after sorting (sortbykey) we kept the first record.

Q4 Find and print the most popular Comedy movies for every year after 1995

The process here is similar to the previous step until the point of joining the two RDDs. The difference here is that after joining the two RDDs (using the k, v pairs) we used a groupByKey method and for each year a new list was created (with the movie and rating). Finally we sorted the lists (for every year) using the rating value (float) in descending order. In order to print the results, we sorted the RDD in ascending order.

Q5 For every year print the average movie revenue

After reading and filtering negative and empty values we created a k, v pair using the year and revenue. Having the year as the key, we created a tuple for the revenue as follows: (revenue, 1). Then we used the reducebykey method, using a custom function which calculated the sum for each element of the tuple (_ + _). That means that for each year we had the sum of the revenue along with the sum of 1s which corresponds to the total count. Finally we used a map to calculate the average (sum/count) and sorted by key in order to print the values.

Task 3 Using DataFrames write code to answer the following queries (Q1-Q5) using the parquet files you created and uploaded to HDFS (2.75 points, 0.55x5)

The use of dataframes is much simpler, compared to RDDs. The main difference here is that we don't need the `.sparkContext` during the `Sparksession` creation.

Q1 For every year after 1995 print the difference between the money spent to create the movie and the revenue of the movie (revenue - production cost)?

After reading the file, we used the `.filter` method to keep years after 1995 and positive cost and revenues. After that the difference is calculated per column and using the `.select` method we assign the data to a new variable which is printed using the `.show` method. We also use the `False` argument in the show `.show` in order to print the entire column content.

Q2 For the movie "Cesare deve morire" find and print the movies id and then search how many users rated the movie and what the average rating was?

In Q2 we load the two parquet files, filter the needed movie ('Cesare deve morire') and join the ratings dataframe on the movie id key. After that the count and average methods are used and printed.

Q3 What was the best in term of revenue Animation movie of 1995?

Similar to Q2 the dataframes are loaded, filtered (for the year, revenue and animation) and then joined on the movie id key. It is easy to sort the results and keep the first value.

Q4 Find and print the most popular Comedy movies for every year after 1995

The procedure is the same as Q3, with a main difference. Once the data is joined, a typical sql syntax using a partition by method is used to find the highest rating for each year.

Q5 For every year print the average movie revenue

Similar to Q1 after loading and filtering the following methods are applied:
`.groupBy("year").avg("revenue").sort("year")`. The code is self explanatory.

Task 4 For every query (Q1-Q5) measure the execution time of each scenario:
1. Map/Reduce - RDD API
2. Spark SQL on csv files (you need to define the schema of each file)
3. Spark SQL on parquet files

Create a bar chart with the execution times grouped by query number and write a paragraph explaining the results (0.5 points).

Using python's `time` library we changed the scripts in order to calculate the time from start to finish. Before running each script the bash scripts `stop-all.sh` and `start-all.sh` were executed in order to restart the workers etc. For each script under the folder we executed the following commands:

```
> stop-all.sh
> start-all.sh
> spark-submit 4_<scenario>_Q\#.py
```

The time was saved in a file names `time.txt` for each type of text. The execution time of each script can be found in Figure:

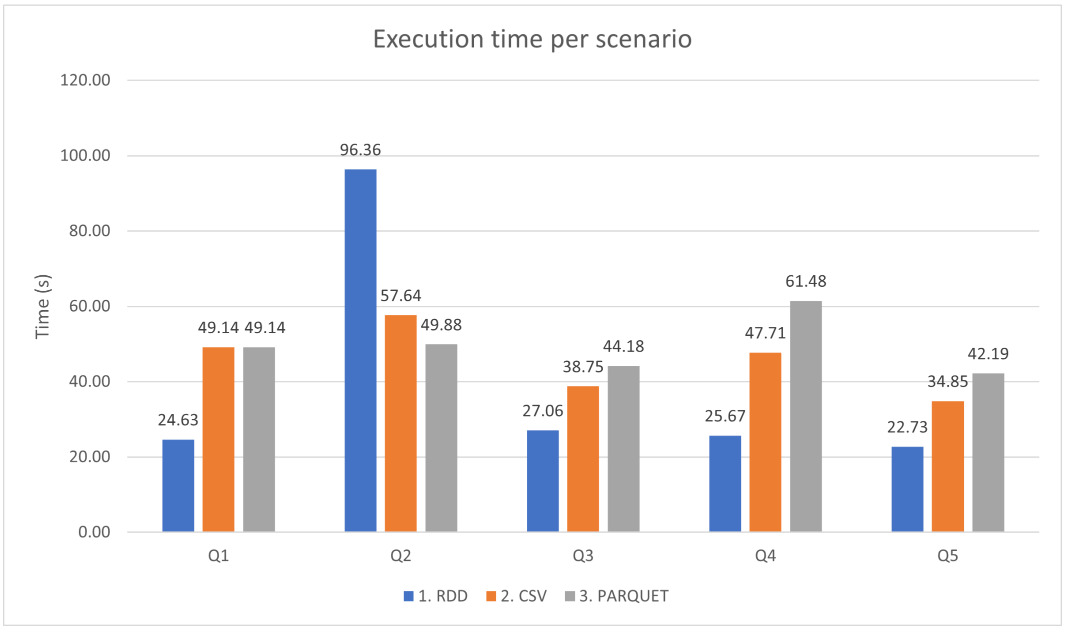


Figure 3: Execution time per scenario using RDDs, csv and Parquet files.

Looking at Figure 3 we can see that for all queries, expect the second, the RDDs have better performance, followed by the CSVs and finally the Parquet files. However, in the second query the performance is the other way around, with parquet files having the fastest time and RDD the slowest.

The first comment to be made is the difference between query 2 and the other queries. In query two we are using the ratings table, which is the only big table (∼600MB). Given that parquet files are column based files, it might make sense that we are getting the best performance in query two since, in fact, we only care about one column of a very big table. Parquet files store the data in columns format and can retrieve the data faster compared to csv, which is row oriented. In the rest of the queries, we are fetching multiple columns and also apply reduce jobs, so it might make sense that the csv, which store data per row, have the better performance. That said, the files we have are, generally speaking, small in size, and the same queries could scale differently in bigger sizes.

Finally, the RDDs have an excellent performance on queries that fetch small(-er) tables. As we know the advantage of RDDs is that they can be stored in the RAM and are distributed in the workers. For small files, this proves to have a big advantage in computational time, with the drawback being that the code preparation is much more complex.

Part 2.

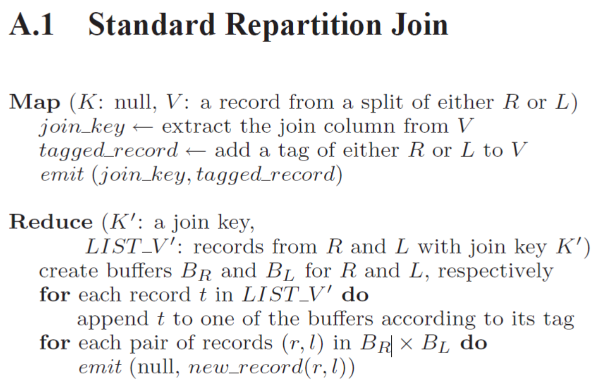
In this part of the final project, you will evaluate different join algorithms. More specifically you will implement the broadcast join algorithm (aka Map side join) and repartition join algorithm (aka Reduce side join). You can read more about the algorithms and their implementations here (for broadcast join Paragraph 3.2 , Pseudocode A.4. and for repartition join Paragraph 3.1, Pseudocode A.1)

Task 1 Write code with Map/Reduce jobs that implements the broadcast join (RDD API) (1 point)

In this part of the project we implemented the broadcast join using the reference paper. The first step is to lead the files, `departmentsR` and `employeesR`. In this case the departmentsR file is the corresponding `R` table and the employeesR file corresponds to the `L` table. To implement the broadcast join, we have to load the R table into memory, and in practice perform a left join with the bigger, L, table. A prerequisite of the broadcast join is to hash the small table into partitions on the join key. Given that the inputs (keys) are co-partitioned this dependency is a narrow one, thus considered a map only job. For this reason we counted the distinct number of keys and we used this number as an argument in the left join. More info on the leftouterjoin may be found in the documentation. Having the small table loaded into the memory, we performed the leftouterjoin which takes care of the hashing needed to implement the broadcast join. The results may be found in the corresponding file.

Task 2 Write code with Map/Reduce jobs that implements the repartition join (RDD API) (1 point)

The repartition algorithm may be found in Figure 4. As seen in the algorithm the



A.1 Standard Repartition Join

```
Map ($K$: null, $V$: a record from a split of either $R$ or $L$)
    $join\_key \leftarrow$ extract the join column from $V$
    $tagged\_record \leftarrow$ add a tag of either $R$ or $L$ to $V$
    $emit$ $(join\_key, tagged\_record)$

Reduce ($K'$: a join key,
        $LIST\_V'$: records from $R$ and $L$ with join key $K'$)
    create buffers $B_R$ and $B_L$ for $R$ and $L$, respectively
    for each record $t$ in $LIST\_V'$ do
        append $t$ to one of the buffers according to its tag
    for each pair of records $(r, l)$ in $B_R \times B_L$ do
        $emit$ (null, $new\_record(r, l)$)
```

Figure 4: Repartition algorithm.

first step is to create the ¡k,v¿ pairs but the extra step of this method is that each value tuple is tagged with either R or L. In our case, for the sake of simplicity, we used the values 1 and 2. We then perform a union which leads to the reduce step. In The reduce part of the repartition join algorithm we groupbykey the data, which creates a unique ¡k,v¿ pair for each key. The values of each pair, contain the data, along with the tags, from the two tables. The next step is to use a function that will arrange the data, bringing the tag containing data from L table (tag 1 in out case) to the left part of the ¡k,v¿ pair. Having that information we can separate the records in order to kave a new ¡k,v¿ pair which will contain the joined data from the two tables, but the ¡k,v¿ pair now corresponds to a distinct record.

Task 3 Spark SQL includes different join algorithms implementations. It also includes a query optimizer named "Catalyst". With the help of catalyst Spark SQL chooses the best join algorithm based on the tables we want to join. We can stop Spark from choosing a join algorithm automatically. You can find more information here.
You can use the script to stop spark from choosing a join algorithm automatically. Execute the query with the query optimizer enabled and with the query optimizer disabled. Create a bar graph with the execution time and explain the results. Take a screenshot of the SQL plan printed by the command spark.sql(query).explain() and give a brief explanation of the SQL plan (1 point).

By modifying the script and assigning the proper parameters, we executed the script 2 times using the following commands:

```
> spark-submit 2_3.py Y > results2_3_Y.txt
```

and:

```
> spark-submit 2_3.py N > results2_3_N.txt
```

The times and results are presented in Figure 5. It is clear that using the query optimizer the results are much better, achieving a time of 12s, while the corresponding time without the query optimizer is 21.8s. Looking at the Physical plan, of both scripts, we see that the main difference is the type of Join used. Without the query optimizer the script uses a SortMergeJoin while a BroadcastHashJoin is used with Catalyst enabled.
Using the Broadcast join, several steps are skipped, during the join procedure. As seen from the explain commands, two projections are used for the two tables, a filtering is applied and, finally, the the tables and joined. On the other hand the SortMergeJoin adds an extra step of sorting both tables and then joining which, obviously add overhead, in terms of time and computational power, to the overall calculation.

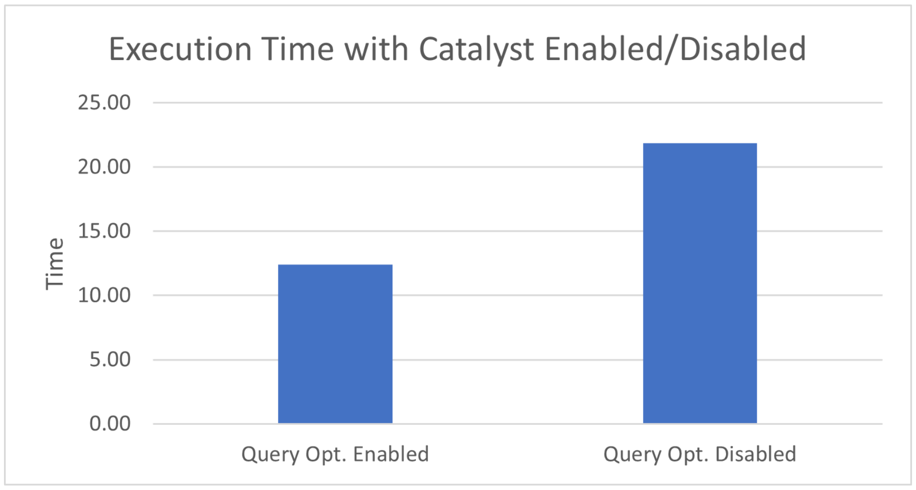


Figure 5: Execution time with Catalysts on-off.