Università degli studi di Verona
Dipartimento di Informatica

Final project for the "Big Data" course:

Querying the MovieLens Dataset using PySpark

A.A 2020/2021

Studenti:

Luca Marzari VR457336
Deborah Pintani VR464800

Professore:

Damiano Carra

Contents

Chapter 1

Introduction to Spark

1.1 What is Spark? (briefly)

Apache Spark is a unified analytics engine for big data processing. It provides high-level APIs in Java, Scala, Python and R. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing.

A Spark application consists of a driver program that runs the user's main function and executes various parallel operations on a cluster. The main abstraction Spark provides is a *resilient distributed dataset* (`RDD`), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. It is possible to persist an `RDD` in memory using the `persist` method, in which case Spark will keep the elements on the cluster for faster access the next time. Another advantage of `RDDs` is that they automatically recover from node failures.

1.2 Operation in a `RDD`

It is possible to perform two types of operations with `RDDs`: transformations, which create a new dataset from an existing one; and actions, which return a value to the driver program after running a computation on the dataset.

For example, map is a transformation that passes each dataset element through a function and returns a new `RDD` representing the results. On the other hand, reduce is an action that groups all the elements of the `RDD` using a provided function and returns the final result to the main program.

All transformations in Spark are lazy, as they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset. The transformations are only computed when an action requires a result to be returned to the main program (e.g. print the results). This allows Spark to be more efficient.

For example, we can realize that a dataset created through map will be used in a reduce and return only the result of the reduce to the driver, rather than the larger mapped dataset. Each transformed `RDD` has to be re-executed each time an action runs on it.

1.3 DataFrames and why we use them

A `DataFrame` is a distributed collection of data organized into named columns. It is conceptually equal to a table in a relational database. On the other hand, `RDDs` does not infer the schema of the data and requires the user to specify it.

Since our dataset is made up of structured tables, we decided to use `DataFrames`. Moreover, a lot of queries required to join the tables to produce a result, and `DataFrames` provide simple methods to perform these operations. `DataFrames` also support SQL operations, like `select`, `groupBy` and so on. Since the queries were made to be solved using PostgreSQL, we decided to use SQL functions when they were necessary.

Chapter 2

Implementation and results

2.1 Data preparation

The dataset provided by the Movielens website [1] under the section "recommended for education and development" was not suitable to execute the queries contained in the given PDF file [2]. They asked about actors, which are not present in the original dataset.

In order to align with the PDF's specifics, we decided to use the same dataset as the one used in the given file. The issue of this choice was that the given data was in `.dat` format. `.csv` format is more suitable than `.dat` format because **pySpark** provides a list of methods, such as `map`, which allows to convert `.csv` files to `DataFrame`.

To convert the original five `.dat` files (`movies.dat`, `actors.dat`, `genres.dat`, `tags.dat`, `tag_names.dat`) to `.csv` files, we written a small snippet of code, as shown in Listing 2.1.

```
1 import csv
2
3 with open('movies.dat') as dat_file, open('movies.csv', 'w')
    as csv_file:
4     csv_writer = csv.writer(csv_file)
5
6     for line in dat_file:
7         row = [field.strip() for field in line.split('\t')]
8         csv_writer.writerow(row)
```

Listing 2.1: `.csv` converter

The Listing 2.1 opens a `.dat` file (in the example, `movies.dat`), and an

empty `.csv` file, with the same filename. Then, it opens a `.csv` writer using the Python `csv` library. For each line in the `.dat` file, it removes unwanted spaces and then splits the line when a tabulation is found. Each split string becomes an element of a list of strings, that is passed to the csv `writeRow` function, which converts the list into a valid `.csv` line (each element is separated with a comma).

2.1.1 `DataFrame` creation

After the conversion, we got five `.csv` files: `movies.csv`, `actors.csv`, `genres.csv`, `tags.csv`, `tag_names.csv`. We used these files, first to create `RDD` collections, and then to generate `DataFrames` based on the `RDDs`, as shown in Listing 2.2.

We again used the `csv` library provided by Python to correctly read and split lines, and feed them to the `map` function.

```
# we separate the fields for each table in csv format
data_movies = movies_file.map(lambda row: next(csv.reader(row
  .splitlines(), skipinitialspace=True)))
data_genres = genres_file.map(lambda row: next(csv.reader(row
  .splitlines(), skipinitialspace=True)))
data_actors = actors_file.map(lambda row: next(csv.reader(row
  .splitlines(), skipinitialspace=True)))
data_tagNames = tagNames_file.map(lambda row: next(csv.reader
  (row.splitlines(), skipinitialspace=True)))
data_tags = tags_file.map(lambda row: next(csv.reader(row.
  splitlines(), skipinitialspace=True)))

# we create the DataFrame for each data generated
table_movies = spark.createDataFrame(data_movies, ['mid', '
  title','year','rating','num_ratings'])
table_genres = spark.createDataFrame(data_genres, ['mid', '
  genre'])
table_actors = spark.createDataFrame(data_actors, ['mid', '
  name', 'cast_position'])
table_tagNames = spark.createDataFrame(data_tagNames, ['tid',
   'tag'])
table_tags = spark.createDataFrame(data_tags, ['mid', 'tid'])
```

Listing 2.2: `RDDs` and `DataFrames` creation

The result after converting the `RDDs` into `DataFrames` is depicted in the image Figure 2.1.

mid	title	year	rating	num_ratings
1	Toy story	1995	3.7	102338
2	Jumanji	1995	3.2	44587
3	Grumpy Old Men	1993	3.2	10489
4	Waiting to Exhale	1995	3.3	5666
5	Father of the Bri...	1995	3	13761
6	Heat	1995	3.9	42785
7	Sabrina	1954	3.8	12812
8	Tom and Huck	1995	2.7	2649
9	Sudden Death	1995	2.6	3626
10	GoldenEye	1995	3.4	28260
11	The American Pres...	1995	3.2	8320
12	Dracula: Dead and...	1995	2.8	10078
13	Balto	1995	3.2	9195
14	Nixon	1995	3.5	3256
15	Cutthroat Island	1995	2.6	3350
16	Casino	1995	3.9	66463
17	Sense and Sensibi...	1995	3.8	32782
18	Four Rooms	1995	3.5	14266
19	Ace Ventura: When...	1995	3.2	87306
20	Money Train	1995	2.7	5263

only showing top 20 rows

mid	name	cast_position
1	Annie Potts	10
1	Bill Farmer	20
1	Don Rickles	3
1	Erik von Detten	13
1	Greg Berg	17
1	Jack Angel	6
1	Jan Rabson	19
1	Jim Varney	4
1	Joan Cusack	24
1	Joe Ranft	16
1	John Morris	23
1	John Ratzenberger	12
1	Kendall Cunningham	21
1	Laurie Metcalf	8
1	Patrick Pinney	9
1	Penn Jillette	15
1	Philip Proctor	11
1	R. Lee Ermey	14
1	Sarah Freeman	22
1	Scott McAfee	18

only showing top 20 rows

Figure 2.1: On the left: movies `DataFrame`. On the right: actors `DataFrame`

2.2 Time comparison between `RDDs` and `DataFrame` methods

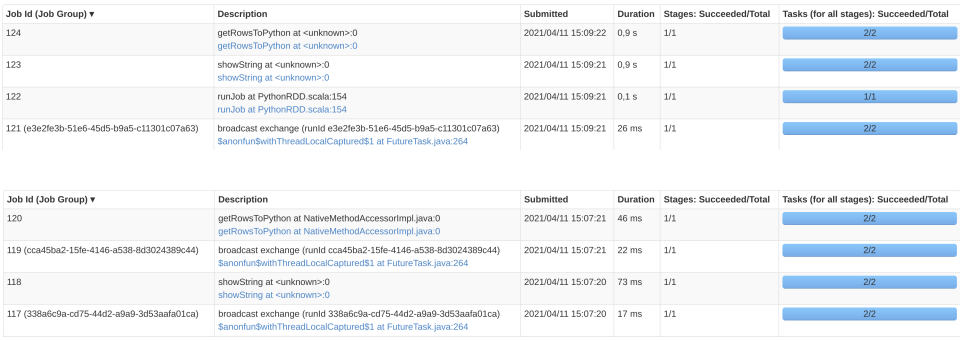
We noticed that in many queries we could not use the `map` and `reduce` functions as the queries were not structured to be performed using those particular functions, although we tried to implement them using as many `maps` and `reduces` as possible.

As stated in Section 1.3, a conversion to `DataFrames` is necessary to perform join operations, and the final result is as well always converted into a `DataFrame`.

However, converting a `DataFrame` to a `RDDs` to perform a `map` operation and then returning again to a `DataFrame` is computationally expensive. As proof of this inefficiency, we measured the execution time needed to perform the first query using two different ways.

The first one used the `map` function (converting from `DataFrame` to `RDDs` and

vice versa, Listing 2.3), while the second one only used `DataFrame` methods, such as the `select` function (Listing 2.4).



Job Id (Job Group) ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
124	getRowsToPython at <unknown>:0 getRowsToPython at <unknown>:0	2021/04/11 15:09:22	0,9 s	1/1	2/2
123	showString at <unknown>:0 showString at <unknown>:0	2021/04/11 15:09:21	0,9 s	1/1	2/2
122	runJob at PythonRDD.scala:154 runJob at PythonRDD.scala:154	2021/04/11 15:09:21	0,1 s	1/1	1/1
121 (e3e2fe3b-51e6-45d5-b9a5-c11301c07a63)	broadcast exchange (runId e3e2fe3b-51e6-45d5-b9a5-c11301c07a63) $anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	2021/04/11 15:09:21	26 ms	1/1	2/2

Job Id (Job Group) ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
120	getRowsToPython at NativeMethodAccessorImpl.java:0 getRowsToPython at NativeMethodAccessorImpl.java:0	2021/04/11 15:07:21	46 ms	1/1	2/2
119 (cca45ba2-15fe-4146-a538-8d3024389c44)	broadcast exchange (runId cca45ba2-15fe-4146-a538-8d3024389c44) $anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	2021/04/11 15:07:21	22 ms	1/1	2/2
118	showString at <unknown>:0 showString at <unknown>:0	2021/04/11 15:07:20	73 ms	1/1	2/2
117 (338a6c9a-cd75-44d2-a9a9-3d53aafa01ca)	broadcast exchange (runId 338a6c9a-cd75-44d2-a9a9-3d53aafa01ca) $anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	2021/04/11 15:07:20	17 ms	1/1	2/2

Figure 2.2: Time comparison for query 1: the fist image depicts the time execution for the map function, the second one for `DataFrame` methods

As we can see from Figure 2.2, solving query 1 with map function requires 1,926 seconds while using `DataFrame` methods requires 0,158 seconds.

2.3 Querying the MovieLens Dataset

For each of the following questions, we wrote and executed a pySpark query in order to achieve the required task. The source code of this project is available at `https://github.com/LM095/BigData-project/blob/main/spark-3.0.2-bin-hadoop2.7/BigData-Project.ipynb`.

2.3.1 Query 1

Print all movie titles starring 'Daniel Craig', sorted in an ascending alphabetical order.

```
1 movies_with_actors = movies.join(actors, movies.mid == actors
    .mid)
2 reduced_movies = movies_with_actors.rdd.map(lambda x: (x.name
    ,x.title)).filter(lambda x: (x[0] == 'Daniel Craig'))
3 movies_with_Craig = reduced_movies.toDF().withColumnRenamed('
    _1','name').withColumnRenamed('_2','title').select('title')
4 movies_with_Craig_sorted =movies_with_Craig.sort(
    movies_with_Craig.title.asc())
```

Listing 2.3: Query 1 using `map` function

It is possible to solve this query in two ways: one using `map` functions as shown in Listing 2.3, and another one just using `DataFrame` methods, which are very similar to SQL instructions, as shown in Listing 2.4.

The following explanation regards the query in Listing 2.3. In the first line, we used a `join` function since we needed the movies starred by 'Daniel Craig', so we had to merge *movies* and *actors* tables. Then, we used `map` and `filter` functions to select only the movies where Daniel Craig starred, using the actor's name as the key and the movie's name as the value. In line 3 we converted the `RDD` in a `DataFrame`, and finally we sort it using the `sort` function. We noticed that using `takeOrdered` on the `RDD` and then converting it in a `DataFrame` was not correct, as the final `DataFrame` did not maintained the correct sorted order.

```
1  movies_with_actors = movies.join(actors, movies.mid == actors
     .mid)
2  movies_title_actors = movies_with_actors.select('title','name
     ')
3  movies_with_Craig = movies_title_actors.filter(
     movies_title_actors.name == 'Daniel Craig').sort(
     movies_title_actors.title.asc()).select('title')
```

Listing 2.4: Query 1 using `DataFrame` functions

As for the query shown in Listing 2.4, in the first line we wrote the same `join` as in the previous query. After this operation, we select only the information useful for the query such as the title of the movies and the actors' names. Then in line 3, we use a `filter` operation to select only the movies starred by 'Daniel Craig' from the new table obtained from the join.

The result of query 1 is shown in Figure 2.3:

title
A Kid in King Arthur's Court
Archangel
Casino Royale
Casino Royale
Elizabeth
Enduring Love
Infamous
Lara Croft: Tomb Raider
Layer Cake
Munich
Quantum of Solace
Renaissance
Road to Perdition
Sorstalanság
Sylvia
The Golden Compass
The Invasion
The Jacket
The Mother
The Mother
The Power of One

Figure 2.3: Result of query 1

We reported a comparison between the time execution of the query 1 using map function and using `DataFrame` methods in Figure 2.2

2.3.2 Query 2

Print names of the cast of the movie 'The Dark Knight' in an ascending alphabetical order.

```
1 movies_with_actors = movies.join(actors, movies.mid == actors
    .mid)
2 reduced_movies = movies_with_actors.rdd.map(lambda x: (x.name
    ,x.title)).filter(lambda x: (x[1] == 'The Dark Knight'))
3 cast_TheDarkKnight = reduced_movies.toDF().withColumnRenamed(
    '_1','name').withColumnRenamed('_2','title').select('name')
4 cast_TheDarkKnight_sorted = cast_TheDarkKnight.sort(
    cast_TheDarkKnight.name.asc())
```

Listing 2.5: Query 2 using `map` function

As before, it is possible to solve the query in two ways: one using a `map` function as shown in Listing 2.5, and another one just using `DataFrame` methods, which are very similar to SQL instructions, as shown in Listing 2.10.

```
movies_with_actors = movies.join(actors, movies.mid == actors
  .mid)
movies_title_actors = movies_with_actors.select('title','name
  ')
cast_TheDarkKnight = movies_title_actors.filter(
  movies_title_actors.title == 'The Dark Knight').sort(
  movies_title_actors.name.asc())
only_cast = cast_TheDarkKnight.select('name')
```

Listing 2.6: Query 2 using `DataFrame` functions

The explanation is pretty straight-forward: we join *actors* and *movies*, then we filter by title and display the actors' names of the movie "The Dark Knight", sorting them.

The result of query 2 is shown in Figure 2.4:

	name
0	Aaron Eckhart
1	Adam Kalesperis
2	Aidan Feore
3	Andrew Bicknell
4	Andy Luther
...	...
86	Walter Lewis
87	Will Zahrn
88	William Armstrong
89	William Fichtner
90	Winston Ellis

91 rows × 1 columns

Figure 2.4: Result of query 2

2.3.3 Query 3

Print the distinct genres in the dataset and their corresponding number of movies N where N is greater than 1000, sorted in the ascending order of N.

```
reduced_genres = genres.rdd.map(lambda x: (x.genre,1)).
  reduceByKey(lambda a,b: a+b).filter(lambda x: (int(x[1]) >
  1000))
reduced_genresdf = reduced_genres.toDF(['genre', 'countMovies
  ']).sort('countMovies')
```

Listing 2.7: Query 3 using `MapReduce` functions

By construction, the genre table is made up of a composite key: *movie id* (*mid*) and *genre*. The query was therefore solved using the `map` function to group all the movies with their genres, then with the `reduceByKey` function we counted the number of movies for each genre and with the `filter` function we only kept those with a value greater than 1000 as required.

The result of query 3 is shown in Figure 2.5:

genre	countMovies
Adventure	1003
Crime	1086
Action	1445
Romance	1644
Thriller	1664
Comedy	3566
Drama	5076

Figure 2.5: Result of query 3

2.3.4 Query 4

For each year, print the movie title, year, and rating, sorted in the ascending order of year and the descending order of movie rating

```
movies_sorted= movies.sort(movies.year.asc(), movies.rating.
  desc())
movies_sortedFiltered = movies_sorted.select('title', 'year',
   'rating')
```

Listing 2.8: Query 4 using `DataFrame` functions

For this query we directly used the methods made available by the `DataFrame` since we only had to print information relating to the movies table. Doing a conversion to `RDD` (with additional computational cost) to use methods like `map` would not make sense here.

The result of query 4 is shown in Figure 2.6:

	title	year	rating
0	The Great Train Robbery	1903	0
1	The Birth of a Nation	1915	3.3
2	Intolerance: Love's Struggle Throughout the Ages	1916	3.8
3	The Immigrant	1917	0
4	Otets Sergiy	1917	0
...	...	...	...
10192	The Last House on the Left	2009	2.7
10193	Miss March	2009	2.7
10194	Megamind	2010	0
10195	The Lady from Shanghai	2010	0
10196	Red Sonja	2011	0

10197 rows × 3 columns

Das Cabinet des D...	1920	4.1
Way Down East	1920	0
The Saphead	1920	0
Der Golem, wie er...	1920	0
The Goat	1921	0
Orphans of the Storm	1921	0
Dr. Mabuse, der S...	1922	4.1
Dr. Mabuse, der S...	1922	4.1
Nosferatu, eine S...	1922	3.9
Häxan	1922	3.8
Nanook of the North	1922	3.7

Figure 2.6: Result of query 4

Figure 2.6 on the right shows the specific result for the year 1920 and 1921, where it can actually be seen that the result has been sorted by ascending order of year and descending order of rating.

2.3.5 Query 5

Critiques say that some words used in tags to convey emotions are very recurrent. To convey positive and negative emotions, the words 'good' and 'bad', respectively, are used predominantly in tags. Print all movie titles whose audience opinion is split (i.e., has at least one audience who expresses positive emotion and at least one who expresses negative emotion).

```
1 movies_with_tags = movies.join(tags, ['mid']).join(tagNames,
    ['tid'])
2 movies_with_tags_onlyGood = movies_with_tags.filter(
    movies_with_tags.tag.contains('good')) #249
3 movies_with_tags_onlyBad = movies_with_tags.filter(
    movies_with_tags.tag.contains('bad')) #93
4 movies_with_tags_onlyGoodBad = (movies_with_tags_onlyGood.
    select('mid','title')).intersect(movies_with_tags_onlyBad.
    select('mid','title')) #14
```

```
5 result = movies_with_tags_onlyGoodBad.select('title')
```

Listing 2.9: Query 5 using `DataFrame` functions

Since a `join` was required for this query, we thought it would be better to use the methods provided by the `DataFrames` directly. In particular, we used the following strategy: first we find all the movies that have tags **containing** the word 'good'. We then do the same for the movies containing the word 'bad'. At this point we select the movies with conflicting opinions by making an intersection of the two sets: in other words we only keep the movies that had both positive and negative tags.

We originally made a mistake of selecting movies with exactly 'good' or 'bad' tags words. In doing so, the result of the intersection was an empty set. We solved it by using the `contains` method for `DataFrames` and the `in` method inside the `filter` function for `RDDs`.

The result of query 5 is shown in Figure 2.7:

	title
0	Twilight
1	The Forgotten
2	Starship Troopers
3	Bridget Jones's Diary
4	Howard the Duck
5	Hercules in New York
6	C.H.U.D.
7	The Wicker Man
8	Ocean's Eleven
9	Return of the Killer Tomatoes!
10	From Dusk Till Dawn
11	Kakushi-toride no san-akunin
12	Xanadu
13	Lawrence of Arabia

Figure 2.7: Result of query 5

The same query could be solved using the same strategy explained above but with the `map` and `intersection` methods provided for `RDDs` as in the Listing .2.10.

```
1 movies_with_tags = movies.join(tags, ['mid']).join(tagNames,
    ['tid'])
```

```
2  movies_with_tags_onlyGood = movies_with_tags.rdd.map(lambda x
     : (x.mid,x.tag)).filter(lambda x: ('good' in x[1])).map(
     lambda x: (x[0])) #249
3  movies_with_tags_onlyBad = movies_with_tags.rdd.map(lambda x:
      (x.mid,x.tag)).filter(lambda x: ('bad' in x[1])).map(
     lambda x: (x[0])) #93
4  movies_with_tags_onlyGoodBad = movies_with_tags_onlyGood.
     intersection(movies_with_tags_onlyBad) #14
```

Listing 2.10: Query 5 using `DataFrame` functions

The disadvantage of this last approach is that we start from a `DataFrame` and we have to transform it into an `RDD` which, as mentioned in section 2.2, is computationally onerous.

2.3.6 Query 6

One would expect that the movie with the highest number of user ratings is either the highest rated movie or perhaps the lowest rated movie. Let's find out if this is the case here:

6.1

Print all information (mid, title, year, num ratings, rating) for the movie(s) with the highest number of ratings

```
1 num_ratings = movies.filter(movies.num_ratings != '\\N').
    withColumn('num_ratings', col('num_ratings').cast('int'))
2 max_num_rating = num_ratings.select(max('num_ratings'))
3 movies_with_max_num_ratings = movies.filter(movies.num_ratings
     == max_num_rating.collect()[0][0])
```

Listing 2.11: Query 6.1 using `DataFrame` functions

The result of query 6.1 is shown in Figure 2.8:

As mentioned earlier, the query requested data from a single table, therefore it was not necessary to transform the `DataFrame` into a `RDD`. In line 1 of the Listing 2.11 we simply remove the lines that contain *num_rating* with the $'\backslash\backslash N'$ character and then cast the *num_rating* column to `integer`. We then find the maximum *num_rating* using the `max` method and use this result to find all the movies that had a *num_rating* equal to the maximum found.

	mid	title	year	rating	num_ratings
0	4201	Pirates of the Caribbean: At World's End	2007	3.8	1768593
1	53125	Pirates of the Caribbean: At World's End	2007	3.8	1768593

Figure 2.8: Result of query 6.1

6.2

Print all information (mid, title, year, num ratings, rating) for the movie(s) with the highest rating (include tuples that tie), sorted by the ascending order of movie id.

```
1 num_ratings = movies.filter(movies.rating != '\\N').withColumn
    ('rating', col('rating').cast('double'))
2 max_rating = num_ratings.select(max('rating'))
3 movies_with_max_ratings = movies.filter(movies.rating ==
    max_rating.collect()[0][0])
```

Listing 2.12: Query 6.2 using `DataFrame` functions

For this query it was not necessary to transform the `DataFrame` into a `RDD`. In line 1 of the Listing 2.11 we simply remove the lines that contains *rating* with the $'\backslash\backslash N'$ character and then cast the *rating* column to `double`. We then find the maximum *rating* using the `max` method and used this result to find all the movies that had a *rating* equal to the maximum found.
The result of query 6.2 is shown in Figure 2.9:

mid	title	year	rating	num_ratings
4311	1732 Høtten	1998	5	5

Figure 2.9: Result of query 6.2

6.3

Is (Are) the movie(s) with the most number of user ratings among these highest rated movies? Print the output of the query that will check our conjecture (i.e., your query will print the movie(s) that has (have) the highest number of ratings as well as the highest rating).

```
result = movies_with_max_num_ratings.intersect(
    movies_with_max_ratings)
```

Listing 2.13: Query 6.3 using `DataFrame` functions

To answer query 6.3 it was enough to make an intersection between the result of *movies_with_max_num_rating* and result of *movies_with_max_ratings*, because if a movie belongs to both results then it would have confirmed the conjecture. However, as we can see from the result below, the set is empty.

The result of query 6.3 is an empty set as shown in Figure 2.10:

mid	title	year	rating	num_ratings

Figure 2.10: result query 6.3

6.4

Print all information (mid, title, year, num ratings, rating) for the movie(s) with the lowest rating (include tuples that tie), sorted by the ascending order of movie id.

```
num_ratings = movies.filter(movies.rating != '\\N').withColumn
    ('rating', col('rating').cast('double')).withColumn('
    num_ratings', col('num_ratings').cast('int'))
new_num_ratings = num_ratings.filter(num_ratings.num_ratings >
     0)
min_rating = new_num_ratings.select(min('rating'))
movies_with_min_ratings = movies.filter(movies.rating ==
    min_rating.collect()[0][0])
```

Listing 2.14: Query 6.4 using `DataFrame` functions

The solution for this query is very similar to that one for the question 6.2, with the difference that the minimum rating was required here. Furthermore, to have a consistent result we preferred to filter the *num_rating* table by removing all the movies that have not received any rating. The result of query 6.4 is shown in Figure 2.11:

6.5

Is (Are) the movie(s) with the most number of user ratings among these lowest rated movies? Print the output of the query that will

	mid	title	year	rating	num_ratings
0	4230	Too Much Sleep	1997	1.5	3

Figure 2.11: result query 6.4

check our conjecture (i.e., your query will print the movie(s) that has (have) the highest number of ratings as well as the lowest rating).

```
result = movies_with_max_num_ratings.intersect(
  movies_with_min_ratings)
```

Listing 2.15: Query 6.5 using `DataFrame` functions

Once again to test the conjecture in the case of the lower rating it was enough to make the intersection of the result of the previous query with the table *movies_with_max_num_ratings*. The result of query 6.5 is an empty set as shown in Figure 2.12:

mid	title	year	rating	num_ratings

Figure 2.12: result query 6.5

6.6

In conclusion, is our hypothesis or conjecture true for the MovieLens database?

No, the results of query 6.3 and 6.5 show that both the intersection of *max_num_ratings* and *max_rating/min_rating* are empty.

2.3.7 Query 7

Print the movie title, year, and rating of the lowest and highest movies for each year in 2005 – 2011, inclusive, in the ascending order of year. In case of a tie, print the records in the ascending order of title.

```
movies_2005_2011 = movies.filter((movies.year >= 2005) & (
  movies.year <= 2011))
```

```
movies_list = []

for year in range(2005,2012):
    ratings = movies_2005_2011.filter((movies_2005_2011.year
  == year) & (movies_2005_2011.num_ratings != '\\N') & (
  movies_2005_2011.num_ratings > 0)).withColumn('rating', col
  ('rating').cast('double'))

    min_rating = ratings.select(min('rating'))
    max_rating = ratings.select(max('rating'))

    result_min = ratings.filter(ratings.rating == min_rating.
  collect()[0][0]).sort(ratings.title)
    result_max = ratings.filter(ratings.rating == max_rating.
  collect()[0][0]).sort(ratings.title)

    final_result = result_min.union(result_max).select('title
  ','year', 'rating')
        movies_list.extend(final_result.collect())

rdd = sc.parallelize(movies_list)
result = rdd.toDF()
result
```

Listing 2.16: Query 7 using `DataFrame` functions

The query is shown in Listing 2.16. We begin by filtering the *movies* table from year 2005 to 2011, to reduce the table size. Then, to avoid making a manual query for each year, we use a *for* loop to increment a year variable used in the filter inside the loop.

Since we need to store multiple rows for each loop, we had to create an empty list. At first, we decided to declare an empty `DataFrame` instead of an empty list, in which we would have added rows at each loop using a *union* function. Unfortunately this solution did not worked, as `DataFrames` are not created to do this kind of work, and doing it would have been a stretch. In particular, trying to add a row in an empty `DataFrame` resulted in an error. Using a list to add rows and convert it in a `DataFrame` at the end is a better solution.

For each year, we selected the rows where the year is the one considered in the *for* loop, excluding not valid ratings, such as NaN values and movies with zero reviews. Then it was necessary to cast the *rating* column to `double`, as it is used to determine the minimum and maximum value. We store the maximum and minimum value in two variables. They are used to query the `ratings` table, to get the movie details. In this way, movies with the same

maximum rating are selected and then sorted by title.

We merge the minimum and maximum movies result and then add them in the result list using the `extend` function. Note that using `append` instead of `extend` is not correct, as `append` only joins two lists producing a list of lists, while `extend` joins the element of two lists together. When we tried to use `append`, the conversion into an `RDD` using `parallelize` threw an error.

At the end, outside of the loop, we convert the list into a `RDD` and then in a `DataFrame`. The result is depicted in Figure 2.13.

	title	year	rating
0	Alone in the Dark	2005	2.1
1	Alone in the Dark	2005	2.1
2	Alone in the Dark	2005	2.1
3	Son of the Mask	2005	2.1
4	No Direction Home: Bob Dylan	2005	4.3
5	Basic Instinct 2	2006	2.5
6	Basic Instinct 2	2006	2.5
7	Bug	2006	2.5
8	Bug	2006	2.5
9	Doogal	2006	2.5
10	Das Leben der Anderen	2006	4.4
11	D-War	2007	2.3
12	Byôsoku 5 senchimêtoru	2007	4.3
13	No End in Sight	2007	4.3
14	Pete Seeger: The Power of Song	2007	4.3
15	War Dance	2007	4.3
16	Asylum	2008	2.2
17	Asylum	2008	2.2
18	The Dark Knight	2008	4.5
19	Miss March	2009	2.7
20	The Last House on the Left	2009	2.7
21	Star Trek	2009	4.1

Figure 2.13: Result of query 7

As shown in Figure 2.13, the result seems incomplete, since there are no 2010 and 2011 movies. That is because in the *movies* table there are no valid 2010 and 2011 movies, either because they have no rating or because the assigned rating is NaN.

2.3.8 Query 8

Let us find out who are the 'no flop' actors. A 'no flop' actor can be defined as one who has played only in movies which have a

rating greater than or equal to 4. We split this problem into the following steps

8.1

Create a view called high ratings which contains the distinct names of all actors who have played in movies with a rating greater than or equal to 4. Similarly, create a view called low ratings which contains the distinct names of all actors who have played in movies with a rating less than 4. Print the number of rows in each view.

```
movies_with_actors = movies.join(actors,['mid']).withColumn('
    rating', col('rating').cast('double'))
high_ratings = movies_with_actors.filter(movies_with_actors.
    rating >= 4.0).select('name').distinct()
low_ratings = movies_with_actors.filter(movies_with_actors.
    rating < 4.0).select('name').distinct()
high_ratings.count() #13710
low_ratings.count() #87032
```

Listing 2.17: Query 8.1 using `DataFrame` functions

To solve this query, we first join the *movies* and *actors* tables using the `join` method provided by the `DataFrame` methods. In this first operation we also use the `withColumn` method to rename the tables and then cast the *rating* column to `double`. At this point we use the `filter` method to select only actors who participate in a movie with a rating above and below 4 respectively.
Finally we print the required totals with the `count` method.

8.2

Use the above views to print the number of 'no flop' actors in the dataset

```
result = high_ratings.subtract(low_ratings)
result.count() #7015
```

Listing 2.18: Query 8.2 using `DataFrame` functions

From the definition, an actor "no flop" can be defined as one who has only acted in films that have a rating greater than or equal to 4. So we simply take the result of the `DataFrame` *high_ratings* and we subtract all the actors who also appeared in the result of the `DataFrame` *low_ratings*.

8.3

For each 'no flop' actor, print the name of the actor and the number of movies N that he/she played in, sorted in descending order of N. Finally, print the top10 only.

```
noFlopactors_with_mid = actors.join(result,['name'])
reduced_actors = noFlopactors_with_mid.rdd.map(lambda x: (x.
    name,1)).reduceByKey(lambda a,b: a+b)
reduced_actors_df = reduced_actors.toDF(['name', 'countMovies'
    ])
final_result = reduced_actors_df.withColumn('countMovies', col
    ('countMovies').cast('int')).sort(reduced_actors_df.
    countMovies.desc())
```

Listing 2.19: Query 8.3 using `DataFrame` functions

For the last part of query 8, we retrieve the *mid* of each movie of each actor in the *no_flop* `DataFrame`, then we prefer to transform the `DataFrame` into a `RDD` to use the `map` and `reduceByKey` functions. Finally we transfor the result into a `DataFrame` and use the `sort` methods to print the required result.

The result of query 8.3 is an empty set as shown in Figure 2.14:

name	countMovies
Nikolai Grinko	8
John Cazale	7
Paul Frankeur	7
Tsutomu Yamazaki	6
Kuniko Miyake	6
Gunnel Lindblom	6
Allan Garcia	6
Anatoli Solonitsin	5
Timothy T. Mitchum	5
Megan Gallagher	5

Figure 2.14: Result of query 8.3

2.3.9 Query 9

Let us find out who is the actor with the highest 'longevity.' Print the name of the actor/actress who has been playing in movies for the longest period of time (i.e., the time interval between their first movie and their last movie is the greatest).

```
1  movies_with_actors = movies.join(actors,['mid']).withColumn('
     year', col('year').cast('int'))
2  partial_result = movies_with_actors.select('name','year').
     distinct()
3
4  result_max = partial_result.groupBy('name').max('year').
     withColumnRenamed('max(year)', 'recentYear')
5  result_min = partial_result.groupBy('name').min('year').
     withColumnRenamed('min(year)', 'firstYear')
6
7  result_join = result_min.join(result_max,['name'])
8
9  result = result_join.select('name',result_join.recentYear -
     result_join.firstYear).withColumnRenamed('(recentYear -
     firstYear)', 'difference')
10
11 max_difference = result.select(max('difference'))
12
13 final_result = result.filter(result.difference ==
     max_difference.collect()[0][0])
14 final_result
```

Listing 2.20: Query 9 using `DataFrame` functions

The query is shown in Listing 2.20. We begin by joining *movies* with *actors* and casting the *year* column to `integer` as we have to use it to find the maximum and minimum. We then select only the name and the year of the actors, performing a `distinct` since removing attributes generates many duplicates.

We perform a `groupBy` to group the actors by name and for each of them we find the maximum and minimum year. Then we join the results. At this point, the `DataFrame` *result_join* contains for each actor both the minimum and maximum year, separated in two columns.

Using a `select` function we then subtract the maximum year with the minimum one, producing for each actor their activity period. Finally, we find the maximum period and obtain the actor(s)' name(s). In this way, actors with the same maximum activity period are selected. The result of the query is

depicted in Figure 2.15.

name	difference
Morgan Jones	102

Figure 2.15: Result of query 9

As shown in Figure 2.15, "Morgan Jones" is the actor with the highest longevity, 102 years. This result is a bit odd, as an actor with a 102 years career can't exist. The problem here is the *actor* table doesn't contain a unique identification to distinguish between two actors. As a consequence, it is impossible to tell if two actors with the same name are the same person or two different ones.

This is what happened here: in this case "Morgan Jones" seems to be an aggregate of at least two different actors, which existence spans from 1879 to nowadays (Morgan Jones, 1879-1951 and Morgan Jones alive nowadays). Morgan Jones born 1879 starred in the first movie contained in the dataset sorted by year, "The Great Train Robbery" in 1903, while Morgan Jones alive nowadays played in his last movie in 2005, "Breakfast on Pluto". Subtracting these two years results in 102 years ($2005 - 1903 = 102$).

2.3.10 Query 10

Let us find the close friends of Annette Nicole. Print the names of all actors who have starred in (at least) all movies in which Annette Nicole has starred in. Note that it is OK if these actors have starred in more movies than Annette Nicole has played in.

To solve query 10 it was easier to divide it into multiple sub-queries (as also suggested in the directions). In particular, first we find all the films shot by 'Annette Nicole' and then we do a `natural join` on the *mid* key and a `select` to find all the names of the actors who starred with 'Annette Nicole' (we called this result *co-actors*). Then we perform a `cross join` between the result table just obtained (*co-actors*) and the movies shot by 'Annette' to create all the possible couples (*actor*, *movie*), also generating non-existent couples.

To find the pairs that do not actually exist we perform a subtraction between this table just obtained from the cross join and the *actors* table. In doing so, we obtain all the pairs (actor, movie) that do not exist. Finally, to find

what is required by the query, it is enough to subtract the non-existent pairs just found from the result "co-actors". An example scheme of the solution proposed in relational algebra is shown in Figure 2.17.

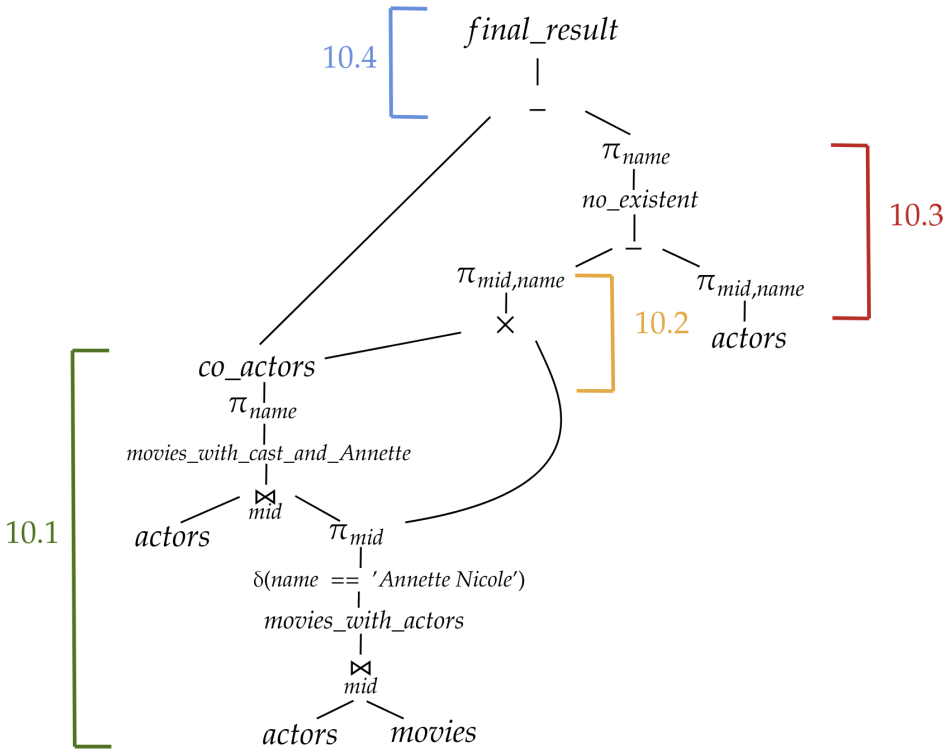


Figure 2.16: Result of query 10

10.1

First, create a view called co_actors, which returns the distinct names of actors who played in at least one movie with Annette Nicole. Print the number of rows in this view.

```
1 movies_with_actors = movies.join(actors,['mid'])
2 movies_with_Annette = movies_with_actors.filter(
    movies_with_actors.name == 'Annette Nicole')
3 movies_with_Annette_mid = movies_with_Annette.select('mid')
4 movies_with_cast_and_Annette = actors.join(
    movies_with_Annette_mid,['mid'])
5 co_actors = movies_with_cast_and_Annette.select('name').
    distinct()
6 co_actors.count() #179
```

Listing 2.21: Query 10.1 using `DataFrame` functions

10.2

Second, create a view called all_combinations which returns all possible combinations of co_actors and the movie ids in which Annette Nicole played. Print the number of rows in this view. Note how that this view contains fake (co_actor, mid) combinations!

```
partial_all_combinations = movies_with_Annette.
    withColumnRenamed('name', 'nameAnnette').crossJoin(
    co_actors)
all_combinations = partial_all_combinations.select('mid','name
    ')
all_combinations.count() #537
```

Listing 2.22: Query 10.2 using `DataFrame` functions

10.3

Third, create a view called non_existent from the view all_combinations by removing all legitimate (co_actor,mid) pairs (i.e., pairs that exist in the actors table). Print the number of rows in this view

```
actors_real = actors.select('mid', 'name')
non_existent = all_combinations.subtract(actors_real)
non_existent.count() #239
```

Listing 2.23: Query 10.3 using `DataFrame` functions

10.4

Finally, from the view co_actors, eliminate the distinct actors that appear in the view non_extistent. Print the names of all co_actors except Annette Nicole.

```
final = co_actors.subtract(non_existent.select('name'))
final.filter(final.name != 'Annette Nicole')
```

Listing 2.24: Query 10.4 using `DataFrame` functions

The result of query 10 is shown in Figure 2.17:

2.3.11 Query 11

Let us find out who is the most social actor. A social actor is the one with the highest number of distinct co-actors. We will break this into two sub-tasks:

name
Kristen Connolly
Christian Perry

Figure 2.17: Result query 10

11.1

For the actor Tom Cruise, print his name and the number of distinct co-actors

```
1 movies_with_Tom = actors.filter(actors.name == 'Tom Cruise').
    select('mid','name')
2 movies_with_coActors = movies_with_Tom.join(actors.
    withColumnRenamed('name', 'nameCoActors'),['mid'])
3 final_coActors_Tom = movies_with_coActors.filter(
    movies_with_coActors.nameCoActors != 'Tom Cruise').select('
    name', 'nameCoActors').distinct().groupBy('name').count().
    withColumnRenamed('count', 'countCoActors')
```

Listing 2.25: Query 11.1 using `DataFrame` functions

The query is shown in Listing 2.25. In case of only one person, is it possible to filter by the actor's name. Then, we join this table with the actors' name, using the movie id *mid* as the key. It produces a table with Tom Cruise and the corresponding co-actors that starred in a film with him.

We exclude the rows where Tom Cruise is the co-actor, then we select the distinct pairs `(Tom Cruise - co-actor name)`. Finally we count the rows by grouping them by Tom Cruise's name. In this way, we deleted the duplicates (meaning, we excluded people that worked with Tom Cruise several times in different movies) and we didn't count Tom Cruise himself in the final result. The result is depicted in Figure 2.18.

name	countCoActors
Tom Cruise	1238

Figure 2.18: Result query 11.1

11.2

For each actor, compute the number of distinct co-actors. For the highest such number, print the name of the actor and the number

of distinct co-actors. In case of a tie, print the records sorted in alphabetical order by name.

```
maxCoActors = 0
actors_list = actors.select('name').distinct().collect()

for actor in actors_list:
    movies_with_specific_actor = actors.filter(actors.name ==
   actor[0]).select('mid','name')
    movies_with_coActors = movies_with_specific_actor.join(
   actors.withColumnRenamed('name', 'nameCoActors'),['mid'])
    final_coActors_specific_actor = movies_with_coActors.
   filter(movies_with_coActors.nameCoActors != actor[0]).
   select('name', 'nameCoActors').distinct().groupBy('name').
   count().withColumnRenamed('count', 'countCoActors')
    most_social_actor.append((actor[0],
   final_coActors_specific_actor.collect()[0][1]))

rdd = sc.parallelize(most_social_actor)
result = rdd.toDF()
max_coActors = result.select(max('_2'))

final_result = result.filter(result._2 == max_coActors.collect
   ()[0][0])
final_result
```

Listing 2.26: Query 11.2 using `DataFrame` functions and a `for` loop - wrong approach

At first, we thought of implementing the same method seen in Subsubsection 2.3.11, where we solved the query 11.1 using a filter on the actor's name. The initial query, shown in Listing 2.26, is **very** inefficient: it loops using a `for` loop on a list containing the actors' names, and it uses the names as the string comparator in the `filter` function. Due to the fact that a `for` loop is not parallelizable and because the data is large, this was not the solution.

We had to change approach and take advantage of what `DataFrame` functions offer.

To understand the query and to structure the solution, we began by using a simple example. The method applies to this specific example, as well as a generalized solution to the problem.

At first, we used `map` and `reduceByKey` functions to group and count the actors that starred in each movie using the `actors` table. Then, we joined the produced table with the `actors` table again to obtain the actors' names. The joined table is shown in Figure 2.19. As depicted, the table contains

for each movie the total actors count and the starred actors.

mid	*name*
1	*A*
1	*B*
1	*C*
2	*C*
2	*D*
2	*A*
.	.
.	.
.	.
.	.

$\Longrightarrow$

mid	*countActors*	*name*
1	3	*A*
1	3	*B*
1	3	*C*
2	20	*C*
2	20	*D*
2	20	*A*

Figure 2.19: First table: starting `actors` table. Second table: joined table between counted actors and actors' names

At this point, the `countActor` column contains the actor themselves and the non-distinct co-actors.

Then, we use `map` and `reduceByKey` functions: in the `map`, the key is the actor's name, the value is the `countActors` -1. It is -1 because we subtract the actor themselves from the total count of the co-actors. Note that after performing the `reduceByKey` function, we have a table containing for each actor the corresponding **non-distinct** co-actors. The produced table called `result_countActors` is shown in Figure 2.20 and it will be used later on. For example, if we take the third row of Figure 2.20, the pair `(C, 21)`, we can see it is the result of $(3-1)+(20-1)=21$.

mid	*countActors*	*name*
1	3	*A*
1	3	*B*
1	3	*C*
2	20	*C*
2	20	*D*
2	20	*A*

$\Longrightarrow$

result_CountActors

name	*countCoActors*
A	21
B	2
C	21
D	19

Figure 2.20: First table: the previous joined table. Second table: table `result_countActors` after the `map` and `reduceByKey` functions

Let us 'pause' the `result_countActors` table and focus on building a second table: the one that will contain the duplicate co-actors for each actor.

First, let us show that the count in the `countCoActors` column is incorrect, by taking the pair `(C, 21)` shown in row 3 in Figure 2.20. The count `21` should be `20`, because the actor C worked with the actor A two times, as shown in Figure 2.19: we have to count the actor A only one time, and not twice.

To build the second table, we join `actors` with `actors` itself, on the `mid` key. The produced table has for each movie all the pairs of starred actors, as shown in Figure 2.21.

actors ⋈ *actors*

mid	*name*	*coActor*
1	*A*	*A*
1	*A*	*B*
1	*A*	*C*
1	*B*	*A*
1	*B*	*B*
1	*B*	*C*
1	*C*	*A*
1	*C*	*B*
1	*C*	*C*
2	*C*	*C*
2	*C*	*D*
2	*C*	*A*
2	*D*	*C*
2	*D*	*D*
2	*D*	*A*
2	*A*	*C*
2	*A*	*D*
2	*A*	*A*

Figure 2.21: Join between `actors` and `actors` on `mid` key

We then group and count the pairs `name - coActors`. We tried to use `map` and `reduceByKey` functions, but they were very slow. Surprisingly, `groupBy` and `count` functions were faster, so we stuck with them, even in the following phases of this query. Note that the same pairs that worked together in more than one movie have count greater than one. The generated table is depicted in Figure 2.22.

Since we don't need pairs where the elements are the same, we get rid of them, as shown in Figure 2.23.

Then, we subtract 1 from all the totals. Let us remind that we are building a table that counts the duplicates co-actors for each actor. We only need to keep the duplicates, and ignore if an actor has worked with another actor

actors ⋈ *actors*

mid	*name*	*coActor*
1	*A*	*A*
1	*A*	*B*
1	*A*	*C*
1	*B*	*A*
1	*B*	*B*
1	*B*	*C*
1	*C*	*A*
1	*C*	*B*
1	*C*	*C*
2	*C*	*C*
2	*C*	*D*
2	*C*	*A*
2	*D*	*C*
2	*D*	*D*
2	*D*	*A*
2	*A*	*C*
2	*A*	*D*
2	*A*	*A*

⟹

groupBy(). *count*()

name	*coActor*	*CountcoActors*
A	*A*	2
A	*B*	1
A	*C*	2
B	*B*	1
B	*A*	1
B	*C*	1
C	*A*	2
C	*B*	1
C	*C*	2
C	*D*	1
D	*D*	1
D	*C*	1
D	*A*	1
A	*D*	1

Figure 2.22: First image: previously joined table. Second image: grouped pairs

filter

name	*coActor*	*CountcoActors*
~~*A*~~	~~*A*~~	~~2~~
A	*B*	1
A	*C*	2
~~*B*~~	~~*B*~~	~~1~~
B	*A*	1
B	*C*	1
C	*A*	2
C	*B*	1
~~*C*~~	~~*C*~~	~~2~~
C	*D*	1
~~*D*~~	~~*D*~~	~~1~~
D	*C*	1
D	*A*	1
A	*D*	1

Figure 2.23: Remove rows where `coActor` is equals to `countCoActors`

just once. We want to remove 1 from the totals because later on we will use this table to subtract the duplicates from the 'paused' `result_countActors` table. A row where the count is 1 only shows that a pair has worked together once, therefore we don't want to subtract a 'legit' pair from the total. If we don't remove the 1 from the current table, every final count of co-actors will be 0. The produced table is shown in Figure 2.24.

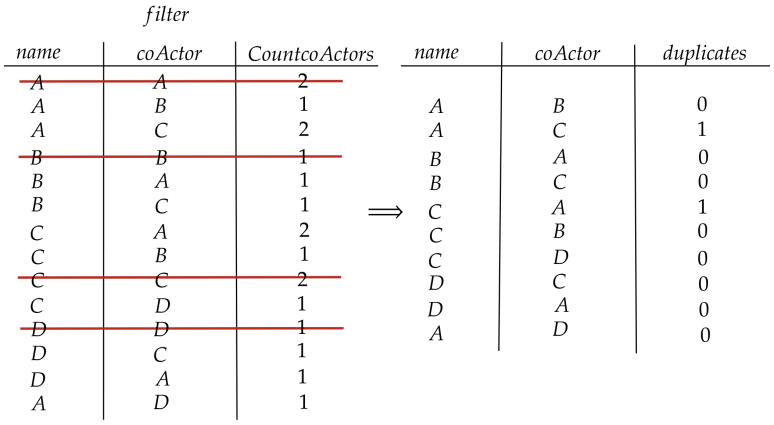


Figure 2.24: First image: the previous table. Second image: the same table but with the decreased count

We group the previous table on the name, and sum the totals based on the actors' names, getting rid of the `coActor` column. We can exclude the `coActor` column because the corresponding pair (eg. `(A, C) and (C, A)`) doesn't need to be counted twice. The result table, called `result_couples`, is shown in Figure 2.25.

result_Couples

name	*sum(duplicates)*
A	1
B	0
C	1
D	0

Figure 2.25: `result_couples` table, grouped by name, where the `duplicate` column is a sum

As we can see, actors A and C worked together twice, therefore the duplicate is counted for both of them. On the other hand, the actor B has never worked more than once with the same person, therefore it has no duplicates.

At this point, we build the necessary tables: `result_couples` and `result_countCoActors`. Now we want to subtract the two totals, to get the real number of co-actor for each actor. To do so, we join the two tables on the actors' name, as shown in Figure 2.26.

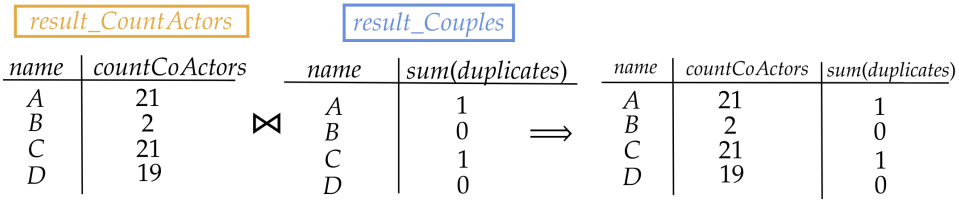


Figure 2.26: Join between `result_couples` and `result_countCoActors` on the `name` column

We now have two columns that need to be subtracted: we use the `select` function to do so on the columns `countCoActors` and `duplicates`. The generated table has for each actor the distinct number of co-actors in the column `countCoActors`. The result is depicted in Figure 2.27.

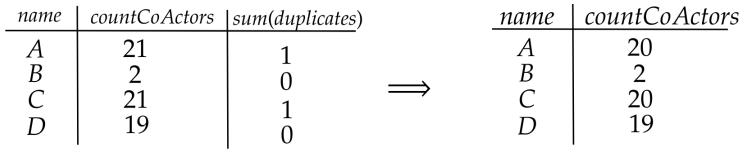


Figure 2.27: First image: the previous table. Second image: the same table but on the `countCoActors` column the result of `countCoActors` − `duplicates`

Finally, we have to find the maximum value in the `countCoActors` column, to determine which actor is the most social. To do so, we use the `max` function to find the highest value, and then obtain the actor(s)' name(s). In this way, actors with the same maximum are selected. The result of the query is depicted in Figure 2.28

name	countCoActors
Samuel L. Jackson	1824

Figure 2.28: Result of query 11.2

The full query is shown in Listing 2.27.

```
1 reduced_actors = actors.rdd.map(lambda x: (x.mid,1)).
    reduceByKey(lambda a,b: a+b)
2 result_countActors = reduced_actors.toDF().withColumnRenamed('
    _1','mid').withColumnRenamed('_2','countActors')
```

```
movies_with_countActors = result_countActors.join(actors,['mid']).select('mid','countActors','name')

movies_with_countActors_reduced = movies_with_countActors.rdd.map(lambda x: (x.name,x.countActors-1)).reduceByKey(lambda a,b: a+b)
result_countActors = movies_with_countActors_reduced.toDF().withColumnRenamed('_1','name').withColumnRenamed('_2','countCoActors')

# search for duplicates, we tried use map and reduceByKey but they were too slow. Surprisingly groupBy and count were faster than map and reduceByKey.
couples_coActors = actors.join(actors.withColumnRenamed('name','coActor'),['mid']).groupBy('name','coActor').count().withColumnRenamed('count', 'countCoActors')
result_partial_couples = couples_coActors.filter(couples_coActors.name != couples_coActors.coActor).select('name', 'coActor',couples_coActors.countCoActors-1).withColumnRenamed('(countCoActors - 1)', 'duplicates').withColumn('duplicates', col('duplicates').cast('int'))
result_couples = result_partial_couples.groupBy('name').sum('duplicates').withColumnRenamed('sum(duplicates)', 'duplicates')

r= result_countActors.join(result_couples,['name'])
final_r = r.select('name', r.countCoActors - r.duplicates).withColumnRenamed('(countCoActors - duplicates)','countCoActors')
max_coActors = final_r.select(max('countCoActors'))
#final_r.filter(final_r.name == 'Tom Cruise') --> 1238 it's correct!
final_result = final_r.filter(final_r.countCoActors == max_coActors.collect()[0][0])
```

Listing 2.27: Query 11.2 using `DataFrame` functions

To prove that the query is correct, we verified that "Tom Cruise"'s total matches with the result produced in query 11.1.

2.3.12 Query 12

Given a user who is known to like the movie "Mr. & Mrs. Smith", write a query that prints the movie title, rating, and similarity percentage (i.e., similarity * 100) for the top 10 movies that are most similar to the "Mr. & Mrs. Smith" movie, ordered by the similarity percentage.

We will now write some queries for a Content-Based Movie Recommendation System such as NetFlix. However, in this project we shall deploy a simple algorithm that may or may not produce optimal recommendations. Content-based recommendations focus on the properties of items, in our case movies. The similarity of two movies is determined by measuring the similarity of their properties. For a movie item, we shall consider the following five properties: actors, tags, genres, year, and rating.
Given two movies X and Y, the similarity of Y to X, sim(X,Y), can be computed as: (fraction of common actors + fraction of common tags + fraction of common genres + age gap + rating gap) /5 where fraction is the number of common elements between X and Y divided by the number of elements of X, age gap is the normalized difference between the production years of X and Y, and rating gap is the normalized difference between the ratings of X and Y. Intuitively, the smaller the gaps are, the better (since movies of the same decade and rating are more likely to be similar). Moreover, note that we divide by five because each property is given an equal weight of 1.

In particular we can define the similarity between two different movies as:

$$\frac{\text{fraction of } \textit{common actors} \ + \ \text{fraction of } \textit{common tags} \\ + \text{fraction of } \textit{common genres} \ + \ \textit{age gap} + \textit{rating gap}}{5}$$

Figure 2.29: Similarity formulation

where:

$$common\_actors = \frac{num\_actors(actors(m_1) \cap actors(m_2))}{num\_actors(m_1)}$$
$$common\_genre = \frac{num\_genre(genre(m_1) \cap genre(m_2))}{num\_genre(m_1)}$$
$$common\_tags = \frac{num\_tags(tags(m_1) \cap tags(m_2))}{num\_tags(m_1)}$$
$$age\_gap = 1 - \frac{|year(m_1) - year(m_2)|}{max(year\_dataset)}$$
$$rating\_gap = 1 - \frac{|rating(m_1) - rating(m_2)|}{max(rating\_dataset)}$$

In our case since we have to find the 10 movies most similar to "Mr. & Mrs Smith", m_1 will always be "Mr. & Mrs Smith" and m_2 will be one movie at a time among all the movies in the dataset (excluding m_1 of course).
To solve the query we first create a support function called *similarity* which receives as parameters two *"mid"* of the movies and returns the percentage of similarity calculated with the formula described above in the Figure 2.30.

```
1  def similarity(mid_movie1, mid_movie2):
2      get_num_actor = len(actors.filter(actors.mid ==
   mid_movie1).select('name').collect())
3      get_num_tags = len(tags.filter(tags.mid == mid_movie1).
   select('tid').distinct().collect())
4      get_num_genres = len(genres.filter(genres.mid ==
   mid_movie1).select('genre').distinct().collect())
5
6      actorsMovie1 = actors.filter(actors.mid == mid_movie1).
   select('name')
7      actorsMovie2 = actors.filter(actors.mid == mid_movie2).
   select('name')
8      intersect_actor =  len(actorsMovie1.intersect(
   actorsMovie2).collect())/get_num_actor
9
10     tagsMovie1 = tags.filter(tags.mid == mid_movie1).select('
   tid')
11     tagsMovie2 = tags.filter(tags.mid == mid_movie2).select('
   tid')
12     intersect_tag = len(tagsMovie1.intersect(tagsMovie2).
   collect())/get_num_tags
13
14     genreMovie1 = genres.filter(genres.mid == mid_movie1).
   select('genre')
```

```
15      genreMovie2 = genres.filter(genres.mid == mid_movie2).
    select('genre')
16      intersect_genre =  len(genreMovie1.intersect(genreMovie2)
    .collect())/get_num_genres
17
18      year_movie1 = int(movies.filter(movies.mid == mid_movie1)
    .select('year').collect()[0][0])
19      year_movie2 = int(movies.filter(movies.mid == mid_movie2)
    .select('year').collect()[0][0])
20      difference = abs(year_movie1 - year_movie2)
21      age_gap =  1 - (difference/maxYear)
22
23
24      rating_movie1 = float(movies.filter(movies.mid ==
    mid_movie1).select('rating').collect()[0][0])
25      rating_movie2 = float(movies.filter(movies.mid ==
    mid_movie2).select('rating').collect()[0][0])
26      difference_gap = abs(rating_movie1 - rating_movie2)
27      rating_gap = 1 - (difference_gap/maxRating)
28      similarity = round((intersect_actor+intersect_tag+
    intersect_genre + age_gap + rating_gap)/5,2)*100
29
30      return similarity
```

Listing 2.28: similarity support function

A trivial solution to this query would therefore be to do a `for` loop on all the *mid* and for each pair (m_1, m_i) calculate the similarity and insert the result in a list. At the end, the list of triples $(m_1, m_i, similarity)$ is sorted by descending order of similarity and the first 10 results are taken. The problem is that this solution is **highly** inefficient, as it requires the use of a `for` loop which, with a large amount of data like the one we have, would lead to very long resolution times as it is a non-parallelizable process.

In fact, we tested this approach on a subset of movies of the dataset, 100 movies to be precise, and the resolution time was about one and a half minute:

```
1  mid_movieMrSmith = get_mid('Mr. & Mrs. Smith')
2  movies_without_MrSmith = movies.filter(movies.mid !=
     mid_movieMrSmith).filter(movies.rating != '\\N').limit(100)
     .select('mid')
3  similarity_for_movies = []
4
5  for mid in movies_without_MrSmith.collect():
6      similarity_for_movies.append([mid_movieMrSmith,mid[0],
     similarity(mid_movieMrSmith,mid[0])])
7
```

```
rdd = sc.parallelize(similarity_for_movies)
result = rdd.toDF().withColumnRenamed('_1','mid_movieMrSmith'
  ).withColumnRenamed('_2','mid').withColumnRenamed('_3','
  similarity')
r = result.join(movies,['mid']).select('title', 'rating','
  similarity')
final_result= r.sort(r.similarity.desc()).limit(10)
final_result
```

Listing 2.29: inefficient solution with for loop

title	rating	similarity
Waiting to Exhale	3.3	61.0
Mighty Aphrodite	3.3	60.0
Gazon maudit	3.4	60.0
The American Pres...	3.2	59.0
Grumpy Old Men	3.2	59.0
Beautiful Girls	3.6	59.0
Bottle Rocket	3.7	59.0
Sabrina	3.8	57.99999999999999
Sense and Sensibi...	3.8	57.99999999999999
Vampire in Brooklyn	2.4	57.99999999999999

Figure 2.30: result inefficient solution

If we think that we have 10.197 movies in our dataset, with the same inefficient approach it would take 2,9 hours to finish.

A smarter solution then would have been to use a `map` function, where as a key we put a key made up of the two *mid* of the movies and as a value the *similarity* value, calling the function shown in the Listing 2.28.

```
mid_movieMrSmith = get_mid('Mr. & Mrs. Smith')
movies_without_MrSmith = movies.filter(movies.mid !=
  mid_movieMrSmith).filter(movies.rating != '\\N').select('
  mid')
similarity_for_movies = []

```

```
map_film = movies_without_MrSmith.rdd.map(lambda x: ((
  mid_movieMrSmith,x.mid), similarity(mid_movieMrSmith,x.mid)
  ))
```

Listing 2.30: map solution

However, this solution leads to errors caused by the implicit limitations of the PySpark language. In fact, the error encountered is the following: "PicklingError: Could not serialize object" which is basically an error that appears when attempting to broadcast an `RDD` or reference an `RDD` from an action or transformation. `RDD` transformations and actions can only be invoked by the driver, not inside of other transformations.

The problem is that even if we break down the `similarity` function in smaller and simpler parts, we would always need to use the various `DataFrames` (i.e. *actors*, *genre*, *tags*, etc...) to calculate the percentage of similarity. Converting `DataFrames` into `RDDs` also did not solve the issue. We also tried to solve the problem using broadcast functions, but even this path did not lead to any results.

Our final solution:

```
def get_mid(movie):
    return movies.filter(movies.title == movie).select('mid')
  .collect()[0][0]

maxYear = int(movies.select(max(movies.year)).collect()
  [0][0])
maxRating = float(movies.filter(movies.rating != '\\N').
  select(max(movies.rating)).collect()[0][0])
mid_Mr = get_mid('Mr. & Mrs. Smith')
num_actors_Mr = actors.filter(actors.mid == mid_Mr).count()
num_genres_Mr = genres.filter(genres.mid == mid_Mr).count()
num_tags_Mr = tags.filter(tags.mid == mid_Mr).count()
year_Mr = int(movies.filter(movies.mid == mid_Mr).select('
  year').collect()[0][0])
rating_Mr = float(movies.filter(movies.mid == mid_Mr).select(
  'rating').collect()[0][0])

#we create table common_actors
actors_Mr = actors.filter(actors.mid == mid_Mr).select('mid',
  'name').withColumnRenamed('mid','midMr')
# here we don't have 'count = 0' for no relation between
  actors
```

```
16  partial_common_actors = actors.filter(actors.mid != mid_Mr).
      join(actors_Mr,['name']).groupBy('mid').count()
17  # we add 'count = 0' for the remaining movies
18  partial_common_actors_withCount= movies.select('mid').
      withColumnRenamed('mid','midNew')
19  c = partial_common_actors_withCount.join(
      partial_common_actors, partial_common_actors.mid ==
      partial_common_actors_withCount.midNew, how='left').select(
      'midNew','count')
20  common_actors = c.fillna({'count':'0'}).select('midNew', col(
      'count')/num_actors_Mr).withColumnRenamed('midNew','mid').
      withColumnRenamed('(count / 12)','common_actors_value')
21
22  #we create table common_genres
23  genres_Mr = genres.filter(genres.mid == mid_Mr).select('mid',
      'genre').withColumnRenamed('mid','midMr')
24  partial_common_genres = genres.filter(genres.mid != mid_Mr).
      join(genres_Mr,['genre']).groupBy('mid').count()
25  partial_common_genres_withCount= movies.select('mid').
      withColumnRenamed('mid','midNew')
26  g = partial_common_genres_withCount.join(
      partial_common_genres, partial_common_genres.mid ==
      partial_common_genres_withCount.midNew, how='left').select(
      'midNew','count')
27  common_genres = g.fillna({'count':'0'}).select('midNew', col(
      'count')/num_genres_Mr).withColumnRenamed('midNew','mid').
      withColumnRenamed('(count / 2)','common_genres_value')
28
29  #we create table common_tags
30  tags_Mr = tags.filter(tags.mid == mid_Mr).select('mid','tid')
      .withColumnRenamed('mid','midMr')
31  partial_common_tags = tags.filter(tags.mid != mid_Mr).join(
      tags_Mr,['tid']).groupBy('mid').count()
32  partial_common_tags_withCount= movies.select('mid').
      withColumnRenamed('mid','midNew')
33  t = partial_common_tags_withCount.join(partial_common_tags,
      partial_common_tags.mid == partial_common_tags_withCount.
      midNew, how='left').select('midNew','count')
34  common_tags = t.fillna({'count':'0'}).select('midNew', col('
      count')/num_tags_Mr).withColumnRenamed('midNew','mid').
      withColumnRenamed('(count / 3)','common_tags_value')
35
36  #we create table year_gap
37  y = movies.select('mid','year').withColumn('year', col('year'
      ).cast('int')).withColumn('yearMr',lit(year_Mr))
38  year_gap = y.select('mid',1 - ((abs(col('year')-col('yearMr')
      ))/maxYear)).withColumnRenamed('(1 - (abs((year - yearMr))
      / 2011))', 'year_gap')
39
```

```
40 #we create table rating_gap, we change null value to 0.0
41 r = movies.select('mid','rating').withColumn('rating', col('
     rating').cast('double')).fillna({'rating':'0'}).withColumn(
     'ratingMr',lit(rating_Mr))
42 rating_gap = r.select('mid',1 - ((abs(col('rating')-col('
     ratingMr')))/maxRating)).withColumnRenamed('(1 - (abs((
     rating - ratingMr)) / 5.0))', 'rating_gap')
43
44 final_table = common_actors.join(common_genres,['mid']).join(
     common_tags,['mid']).join(year_gap,['mid']).join(rating_gap
     ,['mid'])
45 f = final_table.withColumn('similarity',lit(((final_table.
     common_actors_value+final_table.common_genres_value+
     final_table.common_tags_value+final_table.year_gap+
     final_table.rating_gap)/5)*100))
46
47 partial_result = f.join(movies,['mid']).select('title', '
     rating','similarity').withColumn('similarity', round('
     similarity',2))
48 result = partial_result.sort(partial_result.similarity.desc()
     ).limit(10)
```

Listing 2.31: Solution query 12 with `DataFrames`

We decided to use a different approach creating five different views, one for each single part of the *similarity* formula (Figure 2.30), instead of using the `similarity` function. In particular, first we generate the `common_actors` table, which for every movie contains the number of actors in common with the actors who starred in "Mr. & Mrs Smith". The final number obtained was divided by the total number of actors of "Mr. & Mrs Smith".

In a similar way we produce tables `common_genres` and `common_tags`, as described in the *similarity* function. For the remaining part of the *similarity* formula, we compute two last tables: one called `year_gap` and the other one called `rating_gap`. The first one contains the normalized difference between the production years of "Mr. & Mrs Smith" and all the different movies. The value is between [0,1] where 0 corresponds to a very large gap and 1 is the same year of production. The second one is the same as the first one but it uses the ratings.

Once we compute these support tables we can use them to calculate the final similarity, just by applying the *similarity* formula for each row. To do so, we use the `select` function, multiplying the final result by 100 to get the percentage. Finally, we sort by *similarity* in a descending order and take the first 10 rows.

The produced table contains the top 10 movies similar to "Mr. & Mrs. Smith". The result of query 12 is depicted in Figure 2.31.

	title	rating	similarity
0	Mr. & Mrs. Smith	3.4	80.00
1	Hitch	3.4	66.67
2	Waitress	3.4	66.65
3	Definitely, Maybe	3.5	66.24
4	Bend It Like Beckham	3.2	65.84
5	Walking and Talking	3.6	65.78
6	French Kiss	3.2	65.77
7	The Naked Gun: From the Files of Police Squad!	3.2	65.70
8	The Women	3	65.04
9	L'ultimo bacio	3.8	65.03

Figure 2.31: Result query 12

We notice that the first movie in the list is the homonym of "Mr. & Mrs. Smith", but having a different movie id (probably because it is a different version of the movie). Also, the similarity is not 100% because the intersection between the tags is empty, so the tag similarity value is 0.

Bibliography

[1] MovieLens, "Movielens dataset." [Online]. Available: https://grouplens.org/datasets/movielens/

[2] Q. School of Computer Science Carnegie Mellon University, "Querying the movielens database." [Online]. Available: https://web2.qatar.cmu.edu/~mhhammou/15415-f16/projects/project1/P1_Handout.pdf