Κοντογιάννης Αλέξανδρος 03115048
Σταματελόπουλος Νικόλαος 03116138

Προχωρημένα Θέματα Βάσεων Δεδομένων
Εξαμηνιαία Εργασία (Apache Spark)
(IP: 83.212.78.31)

Εισαγωγή:

Σκοπός της εργασίας είναι η εισαγωγή και εξοικείωση στην επεξεργασία μεγάλου όγκου δεδομένων (Big Data). Για το σκοπό αυτό μας διατίθενται τρία αρχεία με ένα σύνολο δεδομένων, με πληροφορίες κριτικές και είδη ταινιών στο οποίο καλούμαστε με τη βοήθεια του Apache Spark να σχεδιάσουμε και να υλοποιήσουμε ορισμένες αναζητήσεις (queries) σε αυτά τόσο με χρήση του RDD όσο και του Dataframe API που το εργαλείο μας παρέχει.

Μέρος 1ο

Ζητούμενο 1

Αρχικά καλούμαστε να φορτώσουμε τα αρχεία με τα δεδομένα μας (movies.csv, ratings.csv, movie_genres.csv) στο hdfs, όπου και τοποθετήθηκαν εντός φακέλου με όνομα data.

```
//Download dataset
wget http://83.212.74.119/movie_data.tar.gz

//extract files
tar -xvf movie_data.tar.gz

//make a folder to put the dataset
hadoop fs -mkdir hdfs://master:9000/data

//put data in hdfs
hadoop fs -put movies.csv hdfs://master:9000/data/.
hadoop fs -put movie_genres.csv hdfs://master:9000/data/.
hadoop fs -put ratings.csv hdfs://master:9000/data/.

//check if data is now there
hadoop fs -ls hdfs://master:9000/data/
```

Ζητούμενο 2

Η μετατροπή των αρχείων από .csv σε .parquet έγινε με εκτέλεση στο apache spark του csv2parquet.py που διαβάζει ένα csv μετατρέπει κάθε γραμμή σε dataframe έχοντας κάνει infer το schema των δεδομένων και τελικά την γράφη σε καινούργιο αρχείο ίδιου ονόματος αλλά κατάληξης .parquet όπως φαίνεται και στο script. Το εκτελούμε λοιπόν και για τα τρία αρχεία και καταλήγουμε με 6 αρχεία στο hdfs, 3 csv και 3 parquet. Η εκτέλεση μπορεί να γίνει με τον ακόλουθο τρόπο:

spark-submit csv2parquet.py <csv_filename>
(π.χ. spark-submit csv2parquet.py movies.csv)

Ζητούμενο 3

Για καθένα από τα 5 queries της εκφώνησης δημιουργήθηκαν δύο λύσεις, μία με χρήση RDD και μία με το Dataframe API του Apache Spark τα οποία βρίσκονται εντός του φακέλου code/partA που περιλαμβάνεται στα παραδοτέα. Ακολουθεί και ο ψευδοκώδικας που περιγράφει την υλοποιήση τους σε μορφή MapReduce.

Query 1

```
MAP(key, line (from movies.csv)):
        tokens = line.split(",")
        if tokens.income != 0 && tokens.cost != 0 && tokens.timestamp != "" &&
tokens.timestamp.year >= 2000:
                profit = (tokens.income - tokens.cost) * 100 / tokens.cost
                emit(tokens.timestamp.year, (tokens.title, profit))

REDUCE(key, values):
        year = key
        best = values[0].title
        maxProfit = values[0].profit
        for v in values:
                if v.profit > maxProfit:
                        best = v.title
                        maxProfit = v.profit
        output(year, best, maxProfit)
```

Query 2

```
// (mean rating for every user)
MAP1(key, line (from ratings.csv)):
        tokens = line.split(",")
        emit(tokens.user, tokens.rating)

REDUCE1(key, values): // (user, list(ratings))
        sum = 0
        count = 0
        for r in values:
                count++
                sum += r
        meanRating = sum / count
        emit(1, meanRating) // all emited with same key

// (find how many mean ratings are above 3)
MAP2(key, value):
        emit(key, value)

REDUCE2(key, values):
        sum = 0
        count = 0
        for r in values:
                count++
                if r > 3:
                        sum++
        result = 100 * sum / count
        output(result)
```

Query 3

```
// (find mean rating for every movie)
MAP1(key, line (ratings.csv)):
        tokens = line.split(",")
        emit(tokens.movie, tokens.rating)

REDUCE1(key, values):
        sum = 0
        count = 0
        for r in values:
                count++
                sum += r
        emit(key, sum/count) // emits (movieId, meanRating)

// (joining new emitted pairs with records from movie_genres.csv file)
// (joining from the start would be unnecessary and inefficient)
MAP2(key, value (emitted pair or line from movie_genres.csv)):
        if value is line from movie_genres.csv:
                tokens = value.split(",")
                emit(tokens.movieId, (genre, "G"))
        else:
                emit(key, (value, "R")) // (movieId, (value, tag))

REDUCE2(key, values):
        for i in values: // find the meanRating in the list (only only exists of course)
                if i.tag = "R":
                        for j in values:
                                if j.tag = "G":
                                        // emit genre/meanRating for every genre that the movie
belongs to
                                        emit(j.genre, i.meanRatings)

// find mean rating and count of movies for every genre
MAP3(key , value):
        emit(key, value)

REDUCE3(key, values):
        sum = 0
        count = 0
        for r in values:
                count++
                sum += r
        result = sum / count
        output(key, result, count) // output pairs of (Genre, MeanRating, MovieCount) for every
genre
```

Query 4

```
// (join movies.csv and movie_genres.csv)
MAP1(key, line (from movies.csv or movie_genres.csv)):
        tokens = line.split(",")
        if line from movies.csv:
                id = tokens.movieId
                plot = tokens.plot
                length = length(plot.split(" ")) // count words in plot
                if tokens.timestamp.year.isBetween(2000, 2004):
                        emit(id, (1, lenght, "M")) // (movieID, (5_year_interval, tag))
                elif tokens.timestamp.year.isBetween(2005, 2009):
                        emit(id, (2, lenght, "M"))
                elif tokens.timestamp.year.isBetween(2010, 2014):
                        emit(id, (3, lenght, "M"))
                elif tokens.timestamp.year.isBetween(2015, 2019):
                        emit(id, (4, lenght, "M"))
        else:
                id = tokens.movieId
                genre = tokens.genre
                if genre = "Drama":
                        emit(id, (genre, "G")) // (movieID, ("Drama", tag))

REDUCE1(key, values):
        for i in values:
                if i.tag == "G": // only emit if it is a drama
                        for j in values:
                                if j.tag == "M":
                                        // emit plot length (with key the 5_year_interval)
                                        emit(j.5_year_interval, j.length)

// (find mean plot_length for every 5_year_interval)
MAP2(key, value):
        emit(key, value)

REDUCE2(key, values):
        sum = 0
        count = 0
        for v in values:
                sum += v
                count++
        result = sum / count
        output(key, result)
```

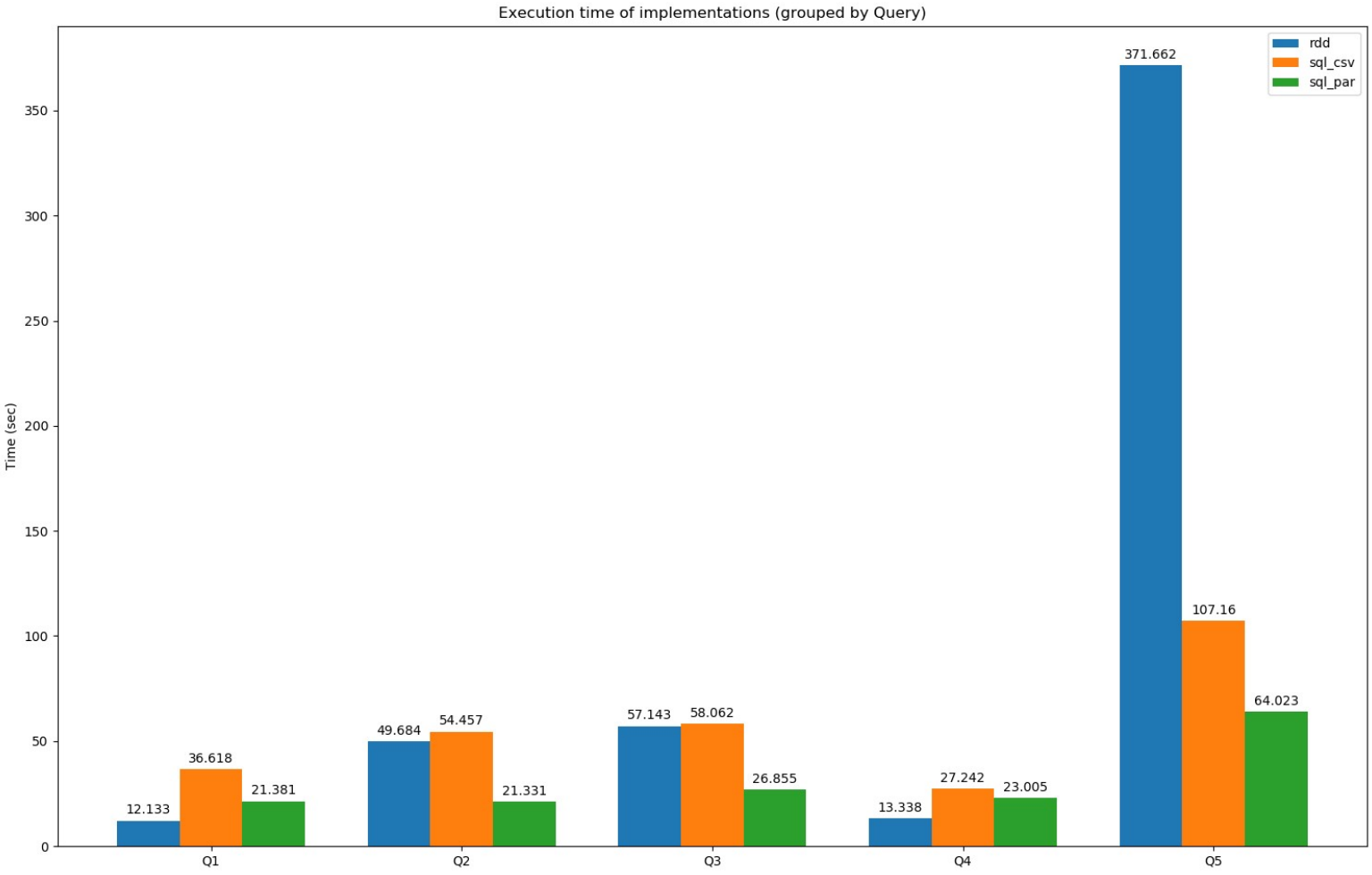
Query 5

```
// join all records on movieID from all 3 files
MAP1(key, line):
        tokens = line.split(",")
        if line from movies.csv:
                emit(tokens.movieId, (tokens.title, tokens.popularity, "M"))
        elif line from ratings.csv:
                emit(tokens.movieId, (tokens.user, tokens.rating, "R"))
        else:
                emit(tokens.movieId, (tokens.genre, "G"))

REDUCE1(key, values): // key is the movieID
        for i in values:
                if i.tag == "G":
                        for j in values:
                                if j.tag == "R":
                                        for k in values:
                                                if k.tag == "M":
                                                        emit(key, (j, k, i)) // emit everything after join

// emit with (Genre, User) as key
MAP2(key, value):
        emit((value.genre, value.user), (value.rating, popularity, value.title))

// for every (Genre, User) combination find count of ratings
// along with best and worst movie for the user on that specific genre only
REDUCE2(key, values):
        count = 0
        best = values[0]
        worst = values[0]
        for v in values:
                count++
                if v > best: //mind the way the values/tuples are constructed
                        best = v
                if v[0] < worst[0] || (v[0] == worst[0] && v[1] > worst[1]):
                        worst = v
        emit(Genre, (user, count, best, worst))

// now found most ratings user for every genre
MAP3(key, value)
        emit(key, value)

REDUCE3(key, values):
        genre = key
        max = values[0].count
        result = values[0]
        for v in values:
                if v.count > max:
                        max = v.count
                        result = v
        output(genre, result.user, max, result.best.title, result.best.rating, result.worst.title,
result.worst.rating)
```

Ζητούμενο 4

Για λόγους μείωσης της έκτασης της αναφορά προτιμήθηκε να αφήσουμε τις εξόδους των εκτελέσεων των queries εντός txt αχρείων εντός του φακέλου outputs ενώ δεν τις παραθέτουμε κι εδώ. Ακολουθούν μόνο οι χρόνοι εκτέλεσης των queries και τα αντίστοιχα διαγράμματα.

(Time in sec)	Q1	Q2	Q3	Q4	Q5
rdd	12.133	49.684	57.143	13.338	371.662
sql_csv	36.618	54.457	58.062	27.242	107.160
sql_par	21.381	21.331	26.855	23.005	64.023



Προφανώς τα αποτελέσματα από κάθε query είναι ίδια για όλες τις υλοποιήσεις, ενώ μικρές διαφοροποιήσεις μπορεί να παρατηρηθούν μόνο σε μερικές περιπτώσεις κατά τη στρογγυλοποίηση των αποτελεσμάτων (ή διαφορετική σειρά εκτέλεσης κάποιων πράξεων) που όμως είναι αμελητέες.

Η χρήση του parquet αμέσως μπορούμε να δούμε ότι κάνει την επεξεργασία των δεδομένων πολύ πιο αποδοτική ως προς το χρόνο εκτέλεσης. Αυτό συμβαίνει γιατί μπορεί να διατηρεί κάποια στατιστικά πάνω στο σύνολο των δεδομένων και κάποια queries του Dataframe API μπορούν να επωφεληθούν από αυτά. Επιπλέον δεν είναι απαραίτητη η χρήση του infer schema διότι ο τύπος των δεδομένων έχει ήδη εξαχθεί και διατηρείται εντός του αρχείου η αντίστοιχη πληροφορία (αντίθετα στην περίπτωση του csv είναι απαραίτητη η χρήση του infer schema για την αναγνώριση των datatypes που οδηγεί σε ένα επιπλέον πέρασμα των δεδομένων αποκλειστικά για αυτόν τον λόγο). Έτσι δικαιολογείται γιατί η χρήση των parquet οδηγεί πάντα σε μικρότερο χρόνο εκτέλεσης από το csv.

Απευθυνόμενοι στη συνολική εικόνα των χρόνων εκτέλεσης βλέπουμε ότι για πιο απλά queries η υλοποίηση με χρήση των rdd είναι και η πιο γρήγορη.

Γενικά το RDD API εκτελεί τις εργασίες που εμείς ορίζουμε στον κώδικα, που σημαίνει ότι ο προγραμματιστής είναι υπεύθυνος για την υλοποίηση και τελικά το σχέδιο εκτέλεσης του query.

Αντίθετα το Dataframe API (Spark SQL) έχει τη δυνατότητα αναδιαμόρφωσης και καθορισμού του execution schedule βάση του προβλεπόμενου κόστους που κάθε επεξεργασία θα επιφέρει. Αυτό σε περιπτώσεις πιο σύνθετων query μπορεί να είναι πολύ χρήσιμο. Από την άλλη όμως, το scheduling αυτό γίνεται στην αρχή, μετά την "αίτηση" και πριν εκκινήσει η εκτέλεση του query.

Βάση των παραπάνω μπορούμε να συμπεράνουμε ότι δικαίως στα Q1 και Q4 που η επεξεργασία των δεδομένων ήταν πολύ απλή, η εκτέλεση με χρήση των rdd ήταν και η πιο γρήγορη, ενώ το scheduling του Spark SQL πήρε περισσότερο από την ίδια την εκτέλεση του Query με αποτέλεσμα να αποτελέσει και το overhead στην εκτέλεση του.

Στην περίπτωση των Q2 και Q3 όπου είναι λίγο πιο σύνθετα τα ερωτήματα παρατηρούμε ότι η αποδοτική υλοποίηση με RDD αρχίζει και δυσκολεύει με το χρόνο εκτέλεση τους να πλησιάζει αυτόν του Spark SQL (η χρήση των parquet αρχείων είναι πολύ πιο γρήγορη, ενώ η Spark SQL μπορεί να κάνει αυτόματα χρήση και broadcast join σε περιπτώσεις όπου το ένα σύνολο δεδομένων είναι σημαντικά πολυπληθέστερο του άλλου).

Τέλος, στο Q5 είναι προφανές ότι λόγω της σύνθετης φύσης του το RDD API επιφέρει σημαντικά μεγαλύτερο χρόνο εκτέλεσης από ότι η Spark SQL.

Μέρος 2ο

Ζητούμενα 1 και 2

Υλοποιήθηκαν με χρήση του RDD API τόσο broadcast όσο και repartition join στα αντίστοιχα αρχεία του στον code/partB στα παραδοτέα. Για λόγους πληρότητας παραθέτουμε και ψευδοκώδικα για την περιγραφή τους (κατά την εκτέλεση απαιτείται να περαστούν ως παράμετροι τα ονόματα των δύο αρχείων καθώς και η θέση του στοιχείου/στήλης βάση του οποίου γίνεται το join με αρίθμηση που ξεκινά από το 1).

Broadcast Join (Προηγείται Broadcast του μικρού συνόλου στοιχείων και δημιουργία HashMap αυτού ως προς το "κλειδί συνένωσης", έστω HashMap)

```
MAP(key, line (from big table)):
        tokens = line.split(",")
        key = tokens[key_index]
        big_val = tokens
        small_vals = HashMap.lookup(key)
        if small_val is not None:
                for v in small_vals:
                        emit(key, (big_val, v))

REDUCE(key, values):
        for v in values:
                emit(key, v) // RDD can't support multiple emits but for the sake of the algorithm…
```

Τέλος διαγραφεί της broadcasted μεταβλητής από όλολλυς τους workers.

Repartition join

```
MAP(key, line (from tables 1 or 2)):
        tokens = line.split(“,”)
        if line from table 1:
                key = tokens[key_index1]
                tag = 1
        else: //from table 2
                key = tokens[key_index2]
                tag = 2
        emit(key, (tokens, tag))

REDUCE(key, values):
        for v1 in values:
                if v1.tag == 1:
                        for v2 in values:
                                if v2.tag == 2:
                                        emit(key, (v1.value, v2.value))
```

Ζητούμενα 3

// Isolate 100 lines from movie_genres.csv
head -100 movie_genres.csv > movie_genres_sample.csv
// Upload sample to hdfs
hadoop fs -put movie_genres_sample.csv hdfs://master:9000/data/.

Εκτέλεση των broadcast και repartition joins πάνω στα δεδομένα του movie_genres_sample.csv και ratings.csv

spark-submit rdd_broadcast_join.py movie_genres_sample.csv 1 ratings.csv 2
spark-submit rdd_repartition_join.py movie_genres_sample.csv 1 ratings.csv 2

Broadcast Join => 76.306 sec
Repartition Join => 600.374 sec

Όπως είναι προφανές το Broadcast Join εκτελείται πολύ πιο γρήγορα από το repartition join. Αυτό συμβαίνει διότι η μεταφορά της συνένωσης στην πλευρά του map ευνοεί τον παραλληλισμό ενώ επίσης δε χρειάζεται να συσσωρεύεται όλη η πληροφορία για κάθε κλειδί σε έναν κόμβο για να επιτευχθεί η συνένωση (όπως γίνεται στο repartition join όπου έχουμε τη συνένωση στην πλευρά του Reduce). Ειδικά το τελευταίο φάνηκε να δημιουργεί προβλήματα λόγω περιορισμών στη μνήμη με το default configuration οπότε κια χρειάστηκε να αυξήσουμε τη μνήμη που επιτρέπεται να χρησιμοποιούν οι workers. Αυτό όμως δεν αποτελεί ένα γενικό κανόνα καθώς το Broadcast Join απαιτεί την αποθήκευση του ενός εκ των δύο συνόλου δεδομένων, τοπικά σε κάθε worker, δηλαδή υπάρχει η απαίτηση να μπορεί να χωρέσει αυτό στη μνήμη σε κάθε worker ξεχωριστά. Έτσι το broadcast join είναι αρκετά ωφέλιμο/ταχύτερο αλλά μπορεί να χρησιμοποιηθεί μόνο στις περιπτώσεις όπου το ένα σύνολο δεδομένων είναι αρκετά μικρό ώστε να μπορεί να αποθηκευτεί στη RAM.

Ζητούμενο 4

Εκτέλεση του join με ενεργοποιημένο τον optimizer.

user@master:~/project_code/partB$ time spark-submit sql_optimizer.py N

```
== Physical Plan ==
*(3) BroadcastHashJoin [_c0#8], [_c1#1], Inner, BuildLeft
:- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as bigint)))
:  +- *(2) Filter isnotnull(_c0#8)
:     +- *(2) GlobalLimit 100
:        +- Exchange SinglePartition
:           +- *(1) LocalLimit 100
:              +- *(1) FileScan parquet [_c0#8,_c1#9] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://
master:9000/data/movie_genres.parquet], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<_c0:int,_c1:string>
+- *(3) Project [_c0#0, _c1#1, _c2#2, _c3#3]
   +- *(3) Filter isnotnull(_c1#1)
      +- *(3) FileScan parquet [_c0#0,_c1#1,_c2#2,_c3#3] Batched: true, Format: Parquet, Location:
InMemoryFileIndex[hdfs://master:9000/data/ratings.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(_c1)],
ReadSchema: struct<_c0:int,_c1:int,_c2:double,_c3:int>
Time with choosing join type enabled is 11.3605 sec.
```

Εκτέλεση του join με απενεργοποιημένο τον optimizer.

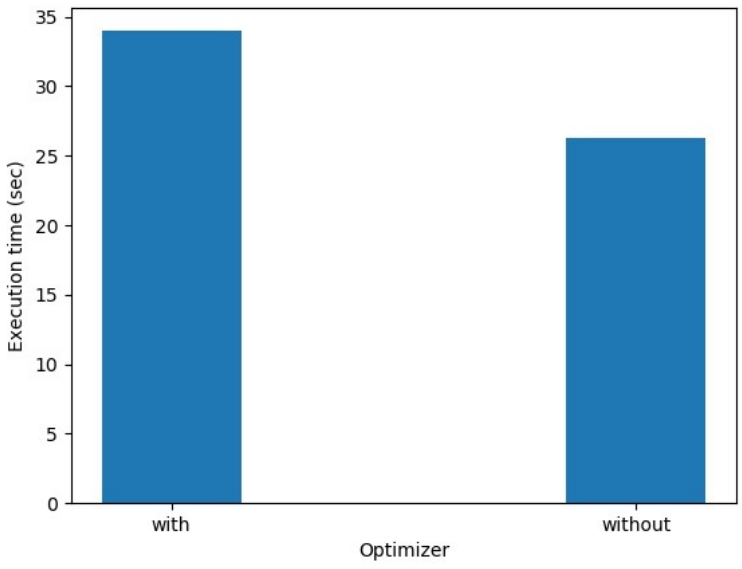
user@master:~/project_code/partB$ time spark-submit sql_optimizer.py Y

```
== Physical Plan ==
*(6) SortMergeJoin [_c0#8], [_c1#1], Inner
:- *(3) Sort [_c0#8 ASC NULLS FIRST], false, 0
:  +- Exchange hashpartitioning(_c0#8, 200)
:     +- *(2) Filter isnotnull(_c0#8)
:        +- *(2) GlobalLimit 100
:           +- Exchange SinglePartition
:              +- *(1) LocalLimit 100
:                 +- *(1) FileScan parquet [_c0#8,_c1#9] Batched: true, Format: Parquet, Location:
InMemoryFileIndex[hdfs://master:9000/data/movie_genres.parquet], PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<_c0:int,_c1:string>
+- *(5) Sort [_c1#1 ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(_c1#1, 200)
      +- *(4) Project [_c0#0, _c1#1, _c2#2, _c3#3]
         +- *(4) Filter isnotnull(_c1#1)
            +- *(4) FileScan parquet [_c0#0,_c1#1,_c2#2,_c3#3] Batched: true, Format: Parquet, Location:
InMemoryFileIndex[hdfs://master:9000/data/ratings.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(_c1)],
ReadSchema: struct<_c0:int,_c1:int,_c2:double,_c3:int>
Time with choosing join type disabled is 13.9659 sec.
```

Total time

with optimizer => 33.977 sec

without optimizer => 26.227 sec



Παρατηρούμε ότι για ενεργοποιημένο τον optimizer επιλέγεται η χρήση Broadcast Join δεδομένου ότι το σύνολο δεδομένων movie_genres είναι μικρότερο από το προεπιλεγμένο threshold. Αντίθετα μετά την απενεργοποίηση του optimizer παρατηρούμε ότι χρησιμοποιείται SortMergeJoin το οποίο γίνεται στην πλευρά του Reduce και δεν έχει κάποια απαίτηση ως προς τη φύση των δεδομένων. Το ενδιαφέρον είναι πως στη δεύτερη περίπτωση ο χρόνος εκτέλεσης είναι μικρότερο από ότι με χρήση του optimizer. Αυτό συμβαίνει διότι η επιλογή της στρατηγικής εκτέλεσης του query απαιτεί ένα χρονικό διάστημα, το οποίο όμως εδώ είναι τελικά συγκρίσιμο/παραπλήσιο με τον ίδιο το χρόνο της εκτέλεσής του, και η ζημία για τον υπολογισμό της στρατηγικής υπερτερεί του οφέλους/επιτάχυνσης που τελικά παρέχει.