National Technical University of Athens

School of Electrical
and Computer Engineering

Προχωρημένα Θέματα Βάσεων Δεδομένων

9ο Εξάμηνο - Ροή Λ

Ναυσικά Αμπατζή 03117198

Καλλιόπη Ελευθερία Γιαννακοπούλου 03117111

Περιεχόμενα

1 Μέρος 1

1.1 Ζητούμενο 1

Αριχκά φορτώσαμε στο Hadoop τα 3 CSV αρχεία που μας δόθηκαν σε ένα φάκελο files.

```
user@master:~$ hadoop fs -ls hdfs://master:9000/files/.
Found 6 items
-rw-r--r--   2 user supergroup    1264187 2022-01-12 20:28 hdfs://master:9000/files/movie_genres.csv
drwxr-xr-x   - user supergroup          0 2022-01-12 20:51 hdfs://master:9000/files/movie_genres.parquet
-rw-r--r--   2 user supergroup   17466695 2022-01-12 20:29 hdfs://master:9000/files/movies.csv
drwxr-xr-x   - user supergroup          0 2022-01-12 20:51 hdfs://master:9000/files/movies.parquet
-rw-r--r--   2 user supergroup  709550294 2022-01-12 20:30 hdfs://master:9000/files/ratings.csv
drwxr-xr-x   - user supergroup          0 2022-01-12 20:52 hdfs://master:9000/files/ratings.parquet
```

Figure 1: Folder Files

1.2 Ζητούμενο 2

Με τη χρήση του αρχείου csvtoparquet.py μετατρέψαμε τα .CSV αρχεία σε Parquet. Έτσι στο hdfs έχουμε πλέον 3 Parquet και 3 CSV αρχεία.

1.3 Ζητούμενο 3

Για κάθε ένα από τα πέντε δοσμένα queries υλποιήσαμε μία λύση με το RDD API και μία με Spark SQL, η οποία θα μπορεί να διαβάζει είτε αρχεία CSV χρησιμοποιώντας το option inferSchema είτε αρχεία Parquet. Ακολουθεί ο ψευδοκώδικας σε Map Reduce που περιγράφει τις υλοποιήσεις μας για το κάθε ερώτημα με το RDD API.

```
# Query 1 Map Reduce Pseudocode

\begin{minted}[
frame=lines,
framesep=2mm,
baselinestretch=1.2,
fontsize=\footnotesize,
linenos
]{python}

map(key, value) :-
        title = value.split(",")[1]
        date = value.split(",")[3]
        cost = value.split(",")[6]
        income = value.split(",")[7]
        if (date != ""):
                if (int(date.split("-")) >= 2000 and cost != 0 and income != 0):
                        profit = (income - cost) * 100 / cost
                        emit(year, (name, profit))

reduce(year, list_movies) :-
        max = list_movies[0][1]
        max_title = list_movies[0][0]
        for movie in list_movies:
                if movie[1] > max:
                        max = movie[1]
                        max_title = movie[0]
```

```
        emit(year, (max_title, max)
```

```
# Query 2 Map Reduce Pseudocode
# map user ids to ratings
map1(ratings,value):
        for line in ratings:
                rate_record = line.split(",")
                user_id = rate_record[0]
                rating = rate_record[2]
                emit(user_id,rating)

reduce1(user_id,list(rating)):
        counter = 0
        rating_sum = 0
        for rate in list(rating):
                rating_sum = rating_sum + rate
                counter +=1
        avg = rating_sum/counter
        emit(user_id,avg)

map2(user,avg)):
        emit(user,avg))

reduce2(_,list_of_avg):
        total_users = len(list_of_avg)
        users_3 = 0
        for rating in list_of_avg:
                if rating >=3:
                        users_3 +=1
        final_percentage = 100*users_3/total_users
```

```
# Query 3 Map Reduce Pseudocode
# average movie rate

map(ratings,value):
        for line in ratings:
                rate_record = line.split(",")
                movie_id = rate_record[1]
                rating = rate_record[2]
                emit(movie_id,rating)

reduce(movie_id,list(rating)):
        counter = 0
        rating_sum = 0
        for rate in list(rating):
                rating_sum = rating_sum + rate
                counter +=1
        avg = rating_sum/counter
        emit(movie_id,avg)

# το αποτέλεσμα του emit είναι μία λίστα(movie_ratings) με (movie_id,average rating)

all = Join(genres,movie_ratings)

map(all,values):
        for line in all:
                genre_rating = line.split(",")
```

```
                genre = genre_rating[1]
                rating = genre_rating[2]
                emit(genre,rating)

reduce(genre,list(rating)):
        sum = 0
        rating_sum = 0
        counter = 0
        sum = len(list(rating))
        for x in list(rating)):
                rating_sum = rating_sum + x
                counter +=1
        avg = rating_sum/counter
        emit(genre,(avg,sum))
```

```
# Query 4 Map Reduce Pseudocode


# genres file
map(genres,value):
        for line in genres:
                genre_record = line.split(",")
                genre = genre_record[0]
                movieid = genre_record[1]
                if(genre == 'Drama'):
                        emit(movie_id,genre)
# movies file
map(movies,value):
        for line in movies:
                movies_record = line.split(",")
                movieid = movies_record[0]
                description = movies_record[2]
                if(description != ""):
                        date = movies_record[3]
                        year = date[3].spli("-")
                        descr_length = len(description)
                        if(year >= 2000 and year <= 2019):
                                emit((movieid,year),1)

reduce((movieid,year)listof(1)):
        cnt = 0
        for v in listof(1):
                cnt += v
        emit(movieid,(year,cnt))

joined = Join(movies,genres) // joined = (movieid,((year,cnt), genre))

map(joined):
        if year >= 2000 and x <= 2004:
                emit('2000-2004',(cnt,1) )
        else if x >= 2005 and x <= 2009:
                emit('2005-2009',(cnt,1) )
        else if x >= 2010 and x <= 2014:
                emit('2010-2014',(cnt,1) )
        else if x >= 2015 and x <= 2019:
                emit('2015-2019',cnt )
reduce(date_period,listof(cnt)):
        total = 0
        for v in values:
                total += cnt
```

```
        emit(date_period,cnt/len(listof(cnt))

```

```
# Query 5 Map Reduce Pseudocode
# Λόγω της μεγάλης έκτασης του query 5 το παρουσιάζουμε λίγο πιο αφαιρετικά σε μερικά σημείο δείχνοντας μόνο τις τελικές ενέργειες
# ratings file - (movieid,userid), name = ratings
map(ratings,value):
    for line in ratings:
        record = line.split(",")
        userid = record[0]
        movieid = record[1]
        emit(movieid,userid)

# genres file (movieid,genre), name = genres
map(genres,value):
    for line in genres:
        record = line.split(",")
        movieid = record[0]
        genre = record[1]
        emit(movieid,genre)

# movies file - (movieid,(namemovie,popularity)), name = popularity
map(movies,value):
    for line in movies:
        movie = line.split(",")
        movieid = movie[0]
        namemovie = movie[3]
        popoularity = movie[7]
        emit(movieid,(namemovie,popularity))

# name = movies2
map(ratings,value):
    for line in ratings:
        record = line.split(",")
        userid = record[0]
        movieid = record[1]
        rating = record[2]
        emit(movieid,(userid,rating))

join1 = movies2.join(popularity).

# name = join1_map
map(join1,value):
    emit(movieid,(userid,title,rating,popularity))

# join2 = (movieid,(userid,title,rating,popularity),genre)
join2 = join1_map.join(genres)

# we consider that the values are splited and just show the final emil
map(join2, value):
    map((genre,userid), (title,rating,popularity))

# user's favourite movie
reduce(map_join2, listof(ratings):
    emit(map_join2, max(listof(rating)))

# user's worst movie
reduce(map_join2, listof(ratings):
    emit(map_join2, min(listof(rating)))

```

```
# combine = ((genre,userid), (title,fav,popularity), (title,morst,popularity))
users_fav_worst = fav.join(worst)

# users_fav_worst_map
map(users_fav_worst,value):
    emit((genre,userid),(titlefav,maxrate,titleworst,minrate))

half = genres.join(ratings)
# half = (movieid, (genre,userid))
map(half,value):
    emit((genre,userid),1)
reduce(res_of_half,value):
    emit((genre,userid), sumof("1"))
map(res_of_reduce,value):
    emit(genre,(userid,sumof("1")))

# keep the user with the most ratings for each genre (name = half):
reduce(res_of_map,value):
    ((genre,user_max_ratings,maxof_numer_of_ratings))

res1 = users_fav_worst_map.join(half)

map(res1,value):
    emit(genre,userid,total reviews, favourite movie, rate, worst movie, rate)
```

1.4 Ζητούμενο 4

Ακολουθούν τα αποτελέσματά από την εκτέλεση των ερωτημάτων του Ζητούμενου 3 και ένα ραβδόγραμμα με όλους του χρόνους εκτέλεσης.

- Query 1

Year	Title	Profit
2000	Billy Elliot	2100.0
2001	千と千尋の神隠し	1732.8339666666666
2002	My Big Fat Greek ...	7274.88088
2003	Tarnation	532933.9449541285
2004	Super Size Me	43861.65846153846
2005	웰컴 투 동막골	4.197475625E8
2006	Facing the Giants	10078.331
2007	Paranormal Activity	1288938.6666666667
2008	Fireproof	6591.2634
2009	The Collector	3252.9411764705883
2010	Catfish	10053.143333333333
2011	From Prada to Nada	2688072.0430107526
2012	Aqui Entre Nos	2.755583E8
2013	Nurse 3-D	9.99999E7
2014	The Quiet Ones	8817.4335
2015	대호	221568.98
2016	Split	2976.911088888889
2017	A Ghost Story	15484.255

Figure 2: Query 1

- Query 2

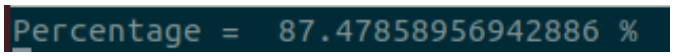
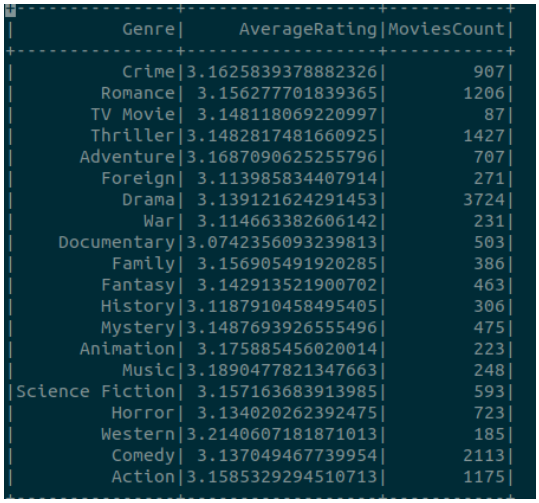


Figure 3: Query 2

- Query 3



Genre	AverageRating	MoviesCount
Crime	3.1625839378882326	907
Romance	3.156277701839365	1206
TV Movie	3.148118069220997	87
Thriller	3.1482817481660925	1427
Adventure	3.1687090625255796	707
Foreign	3.113985834407914	271
Drama	3.139121624291453	3724
War	3.114663382606142	231
Documentary	3.0742356093239813	503
Family	3.156905491920285	386
Fantasy	3.142913521900702	463
History	3.1187910458495405	306
Mystery	3.1487693926555496	475
Animation	3.175885456020014	223
Music	3.1890477821347663	248
Science Fiction	3.157163683913985	593
Horror	3.134020262392475	723
Western	3.2140607181871013	185
Comedy	3.137049467739954	2113
Action	3.1585329294510713	1175

Figure 4: Query 3

- Query 4

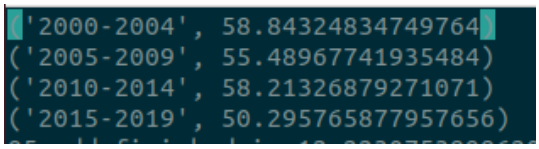


Figure 5: Query 4

- Query 5

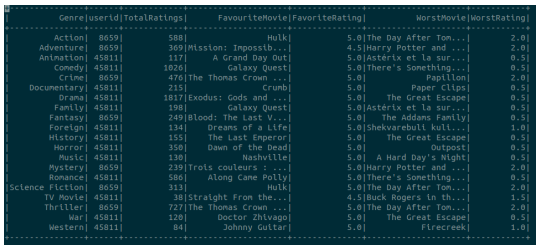


Figure 6: Query 5

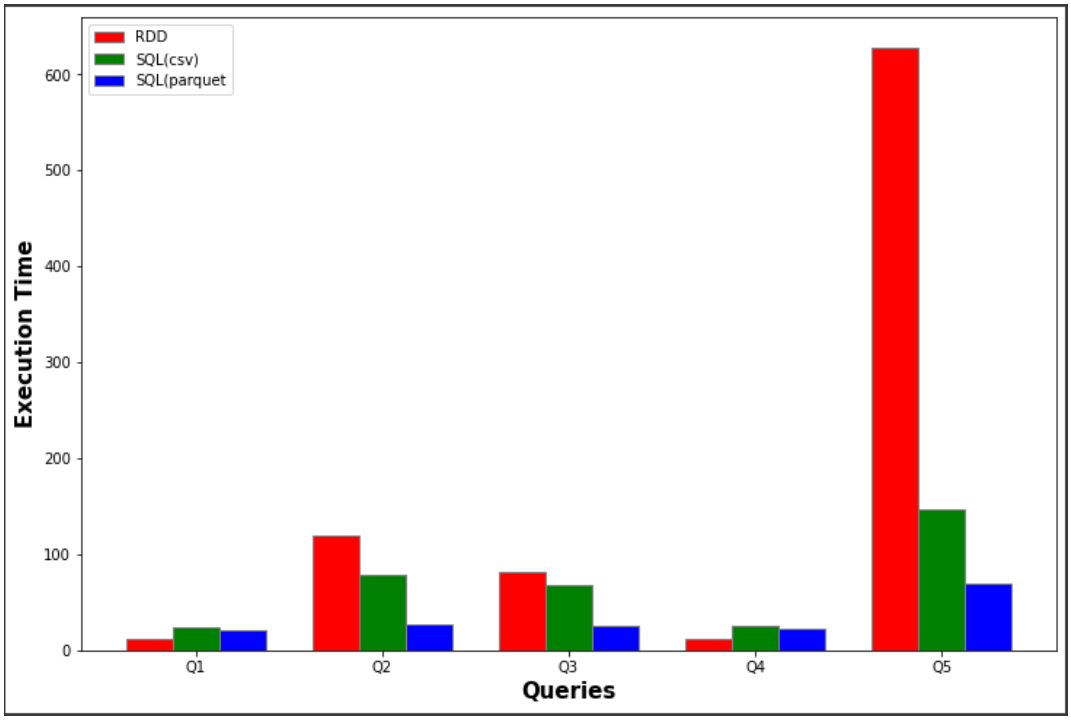


Figure 7: Execution Times

Σχολιασμός

Όπως προκύπτει και από το ραβδόγραμμα με τους χρόνους εκτέλεσης όλων των περιπτώσεων, όσον αφορά την SQL και την RDD υλοποίηση παρατηρούμε ότι στα Q2,Q3 και Q5 η RDD υλοποίηση είναι πιο χρονοβόρα. Ειδικά στο Q5, που ήταν ένα αρκετά περίπλοκο query χρειάστηκαν πάνω από 600 seconds! Επομένως στις περιπτώσεις που χρησιμοποιήσαμε περισσότερα από ένα map reduce η SQL ήταν πιο γρήγορη. Σχετικά με τα Q1 και Q4, όπου η SQL καθυστέρησε θεωρούμε ότι αυτό ίσως οφείλεται και σε δική μας μη βέλτιστη υλοποίηση. Η καλύτερη επίδοση της SQL ήταν αναμενόμενη, καθώς περιέχει έναν βελτιστοποιητή ο οποίος "προσπαθεί" να βελτιώσει την επίδοση των queries. Επιπλέον στην υλοποίηση με RDD, η διαδικασία του grouping απαιτεί τη μεταφορά των δεδομένων πάνω από το δίκτυο, κάτι το οποίο επίσης επιφέρει καθυστερήσεις.
Σχετικά με τη σύγκριση της εκτέλεσης των ερωτημάτων σε sql πάνω σε csv ή parquet αρχεία, όπως περιμέναμε η parquet μορφή έχει πιο σύντομα αποτελέσματα. Τα csv είναι row-orientated, ενώ τα parquet column-orientated όσον αφορά την αποθήκευση. Τα parquet αρχεία χρησιμοποιούν λιγότερο χώρο στο δίσκο σε σχέση με τα csv και επομένως σκανάρονται πιο εύκολα. Από την άλλη τα csv χρησιμοποιούν το InferSchema, κάτι το οποίο καθυστερεί την επεξεργασία των δεδομένων.

2 Μέρος 2

2.1 Ζητούμενο 1

Βασιζόμενοι στο paper που μας δόθηκε υλοποιήσαμε το broadcast join ως εξής:

```
import csv
import time

from pyspark.sql import SparkSession
from io import StringIO
import csv
from itertools import product
import time
import sys


begin = time.time()

def data_split(x):
    return list(csv.reader(StringIO(x), delimiter=','))[0]

def broad_join(tup):
    key = tup[0]
    value = tup[1]
    if key in broad_targetR.value:
        return ((key, (genre, value)) for genre in broad_targetR.value[key])
    else:
        return []


spark = SparkSession.builder.appName("broadcast").getOrCreate()
sc = spark.sparkContext

# R_Record and L_Record variables may change

R_Record = "hdfs://master:9000/files/movie_genres_100.csv"
L_Record = "hdfs://master:9000/files/ratings.csv"

keyr = 0
```

```
keyl = 1


targertR = sc.textFile(R_Record). \
    map(lambda line: data_split(line)). \
    map(lambda line: (int(line[0]), line[1])). \
    groupByKey(). \
    mapValues(list).collectAsMap()


broad_targetR = sc.broadcast(targertR)




targetL = sc.textFile(L_Record) \
    .map(lambda line: data_split(line)) \
    .map(lambda line: (int(line[1]), (int(line[0]), float(line[2]), line[3]))).flatMap(broad_join)

for i in targetL.collect():
        print(i)

end = time.time()
print("Broadcast join finished in " +str(end - begin) + " seconds" )
```

2.2 Ζητούμενο 2

Βασιζόμενοι στο paper που μας δόθηκε υλοποιήσαμε το repartition join ως εξής:

```
import csv
import time

spark = SparkSession.builder.appName("repartition").getOrCreate()
sc = spark.sparkContext

begin = time.time()

def repart(map_values):
        targ_R = []
        targ_L = []
        for value in map_values:
                if value[0] == 'R':
                        targ_R.append(value)
                elif value[0] == 'L':
                        targ_L.append(value)
        return [(valueR, valueL) for valueR in targ_R for valueL in targ_L]


def data_split(f):
        return list(csv.reader(StringIO(f), delimiter=','))[0]

# R_Record and L_Record variables may change

R_Record = "hdfs://master:9000/files/moviegenres100.csv"
L_Record = "hdfs://master:9000/files/ratings.csv"

keyr = 0
keyl = 1


```

```
targetR = \
        sc.textFile(R_Record). \
        map(lambda x : (data_split(x))). \
        map(lambda x : (x.pop(keyr), ('R', x)))

targetL = \
        sc.textFile(L_Record). \
        map(lambda x : (data_split(x))). \
        map(lambda x : (x.pop(keyl), ('L', x)))


union_targets = targetR.union(targetL)

res =   union_targets. \
        groupByKey(). \
        flatMapValues(lambda x : repart(x)). \
        map(lambda x : (x[0], x[1][0][1], x[1][1][1]))

for i in res.collect():
        print(i)

end = time.time()
print("Repartition join finished in " +str(end - begin) + " seconds" )
```

2.3 Ζητούμενο 3

Αρχικά μέσω του αρχείου movie_genres_100_generate.py που βρίσκεται στον φάκελο code, απομονώσαμε 100 γραμμές του πίνακα movie genres σε ένα άλλο CSV. Εκτελέσαμε τις δύο παραπάνω υλοποιήσεις για την συνένωση των 100 γραμμών με τον πίνακα ratings. Ακολουθεί το ραβδρόγραμμα με τους χρόνους εκτέλεσης:

Από το παραπάνω ραβδόγραμμα είναι εμφανές ότι η εκτέλεση με Repartition Join απαιτεί παραπάνω από τον τριπλάσιο χρόνο σε σχέση με την εκτέλεση με Broadcast Join. Κάτι τέτοιο ήταν φυσικά αναμενόμενο, καθώς στη μία περίπτωση πρόκειται για Map Side και στην άλλη για Reduce Side Join. Το Broadcast join είναι πολύ πιο αποδοτικό σε περιπτώσεις συνένωσης ενός μικρού και ενός πολύ μεγάλου πίνακα. Αυτό συμβαίνει διότι γίνεται broadcast του μικρού πίνακα σε όλο το δίτκυο, με αποτέλεσμα να αποθηκεύται τοπικά σε όλα τα μηχανήματα. Έτσι δεν χρειάζεται να μεταφερθεί ολόκληρος ο μεγάλος πίνακας στο δίκτυο, κάτι το οποίο θα προκαλούσε μεγάλες καθυστερήσεις, όπως συμβαίνει στην περίπτωση του repartition join. Φυσικά τα αποτελέσματα που προκύπτουν και με τους δύο τύπους join είναι τα ίδια.

2.4 Ζητούμενο 4

Στο ζητούμενο αυτό, ασχοληθήκαμε με τον βελτιστοποιητή ερωτημάτων (query optimizer) που μας παρέχεται από το Dataframe API του SparkSQL. Σύμφωνα με την εκφώνηση, μία τέτοια βελτιστοποίηση είναι ότι επιλέγει αυτόματα την υλοποίηση που θα χρησιμοποιήσει για ένα ερώτημα join λαμβάνοντας υπόψη το μέγεθος των δεδομένων και πολλές φορές αλλάζει και την σειρά ορισμένων τελεστών προσπαθώντας να μειώσει τον συνολικό χρόνο εκτέλεσης του ερωτήματος. Αν ο ένας πίνακας είναι αρκετά μικρός (με βάση ένα όριο που ρυθμίζει ο χρήστης) θα χρησιμοποιήσει το broadcast join, αλλιώς θα κάνει ένα repartition join. Για να ενεργοποιήσουμε τη δυνατότητα βελτιστοποίησης θέτουμε τη μεταβλητή spark.sql.autoBroadcastJoinThreshold σε -1 στο script που μας δώθηκε. Εκτελέσαμε το script με τη βελτιστοποίηση ενεργοποιημενη και απενεργοποιημένη και πήραμε τα εξής αποτελέσματα:

. .

.

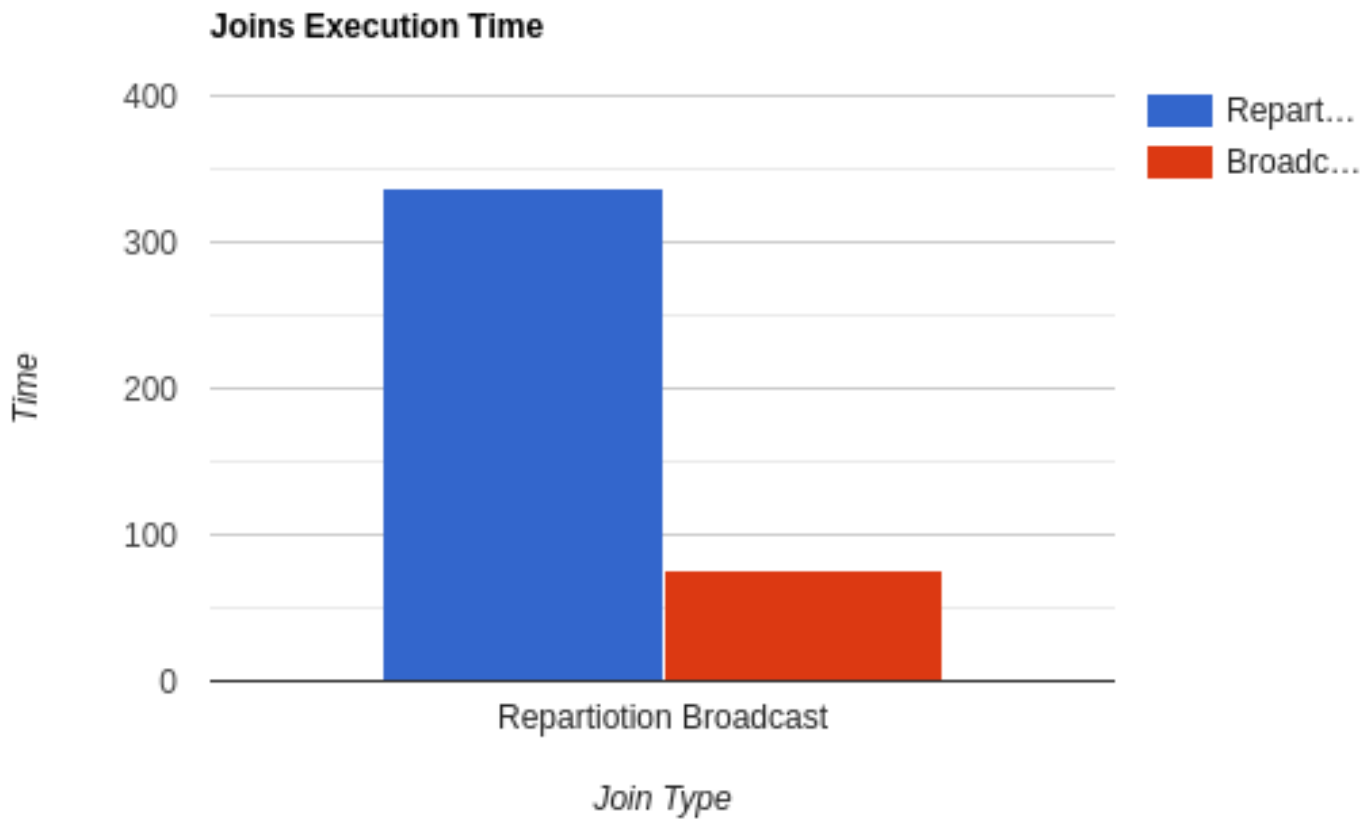


Figure 8: Broadcast - Repartition Joins

.

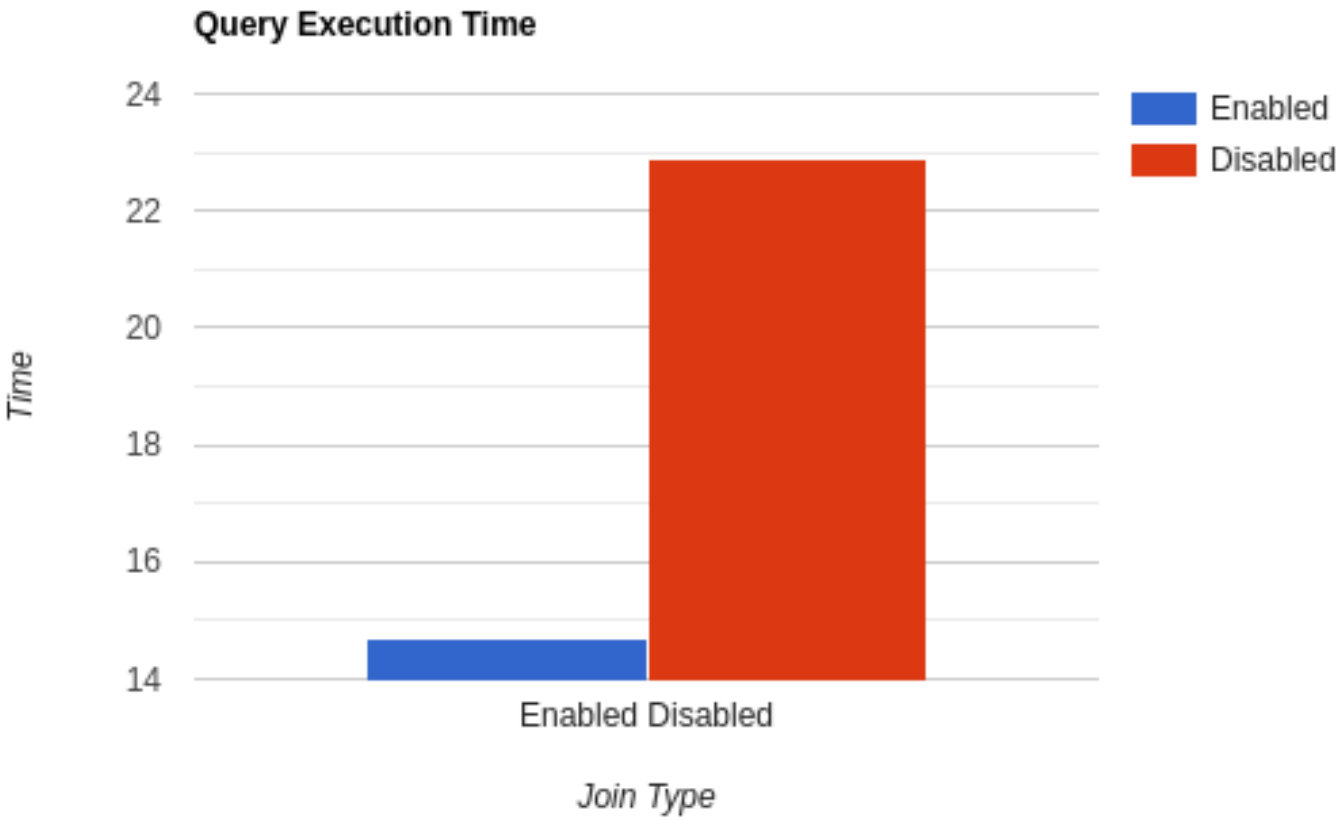


Figure 9: Joins

```
== 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/files/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/files/ratings.parquet], PartitionFilters: [], PushedFilters:
[IsNotNull(_c1)], ReadSchema: struct<_c0:int,_c1:int,_c2:double,_c3:int>
Time with choosing join type enabled` is 14.7016 sec.
```

Figure 10: Joins-N

```
== 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/files/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/files/ratings.parquet], PartitionFilters: [],
PushedFilters: [IsNotNull(_c1)], ReadSchema: struct<_c0:int,_c1:int,_c2:double,_c3:int>
Time with choosing join type disabled is 22.8705 sec.
```

Figure 11: Joins-Y

Παρατηρούμε ότι η εκτέλεση του query χρησιμοποιώντας βελτιστοποιητή είναι πολύ ταχύτερη σε σχέση με τη μη χρήση. Αυτό συμβαίνει διότι ο optimizer "βλέπει" ότι το genres είναι πολύ μικρό

(μόνο 100 γραμμές) κι έτσι χρησιμοποιεί broadcast join. Όπως είδαμε και στο προηγούμενο ερώτημα η συγκεκριμένη αναλογία πίνακα εισόδων με τη χρήση του broadcast join ολοκληρώνεται πολύ γρήγορα. Στην περίπτωση που δεν χρησιμοποιείται ο βελτιστοποιητής, το πρόγραμμα χρησιμοποιεί Sort Merge Join, κάτι το οποίο οδηγεί σε διπλάσιο χρόνο εκτέλεσης.

Σημείωση: Τα αρχεία που μας δώθηκαν περιέχονται στον φάκελο files του hdfs (hdfs://master:9000/files/). Στον φάκελο code υπάρχει η υλοποίηση των ερωτημάτων του 1ου μέρους. Στον φάκελο code υπάρχουν επίσης οι φάκελοι outputs, logs και PART2. Ο φάκελος PART2 περιέχει τις υλοποιήσεις μας για το 2ο μέρος, ο φάκελος outputs περιέχει τα αποτελέσματα από όλα τα queries και ο φάκελος logs περιέχει τα αποτελέσματα απ'όλα τα logs.