ΠΡΟΧΩΡΗΜΕΝΑ ΘΕΜΑΤΑ ΒΑΣΕΩΝ ΔΕΔΟΜΕΝΩΝ
**ΕΞΑΜΗΝΙΑΙΑ ΕΡΓΑΣΙΑ**
ΡΟΗ Λ – ΕΞΑΜΗΝΟ 9ο
ΑΚ. ΕΤΟΣ 2021 – 2022

ΣΕΡΓΙΟΣ ΜΠΑΤΣΑΣ
ΕΥΑΓΓΕΛΟΣ ΒΑΓΙΑΝΟΣ

## ΜΕΡΟΣ 1ο:

### Ζητούμενο 1:

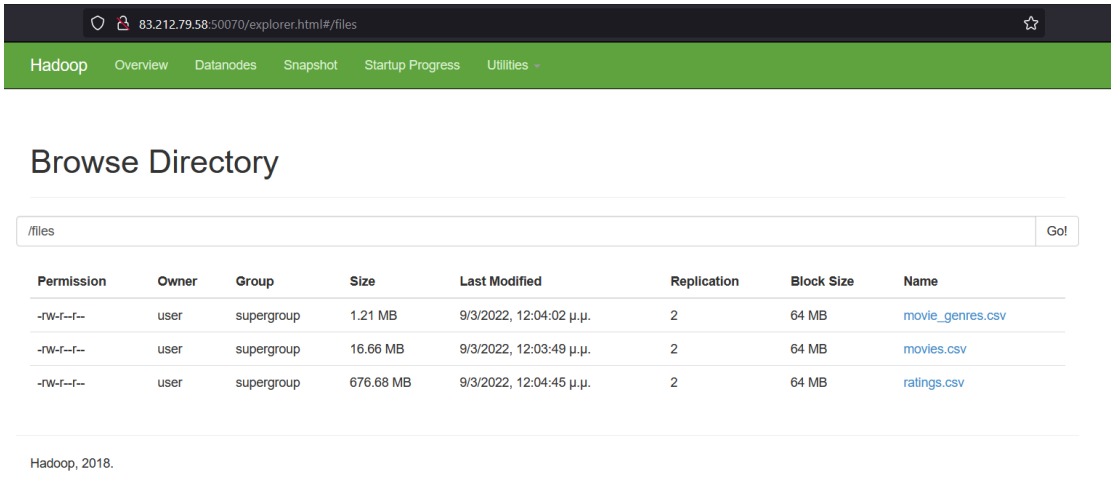
Με την εντολή hadoop fs -mkdir hdfs://master:9000/files φτιάξαμε έναν φάκελο στο hdfs με το όνομα files και με τις εντολές:

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

φορτώσαμε τα ζητούμενα αρχεία csv στον φάκελο files.

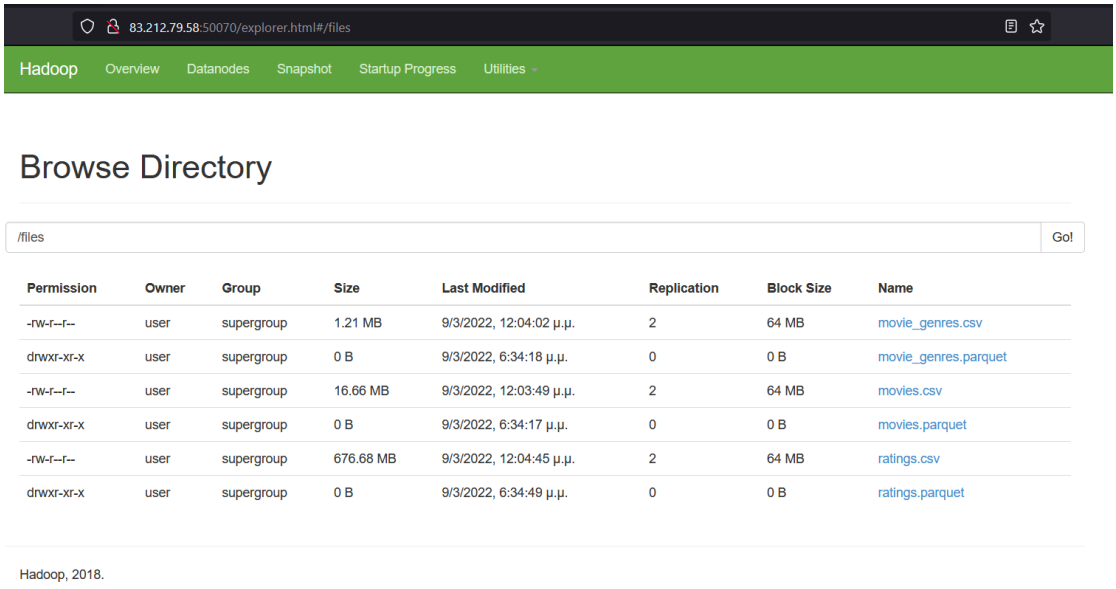
Με χρήση της εντολής hadoop fs -ls hdfs://master:9000/files έχουμε το παρακάτω αποτέλεσμα στο terminal:

```
user@master:~$ hadoop fs -ls hdfs://master:9000/files
Found 3 items
-rw-r--r--   2 user supergroup    1264187 2022-03-09 12:04 hdfs://master:9000/files/movie_genres.csv
-rw-r--r--   2 user supergroup   17466695 2022-03-09 12:03 hdfs://master:9000/files/movies.csv
-rw-r--r--   2 user supergroup  709550294 2022-03-09 12:04 hdfs://master:9000/files/ratings.csv
user@master:~$
```



**Ζητούμενο 2:**

Για την μετατροπή των αρχείων CSV σε Parquet έγινε χρήση του αρχείου csv_to_parquet.py και πλέον έχουμε 6 αρχεία




**Ζητούμενο 3:**

Δομή πινάκων

Movies.csv

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| ID | Title | Summary | Release_Date | Duration | Prod_Cost | Revenue | Popularity |

Movie_genres.csv

| 0 | 1 |
|---|---|
| ID | Genre |

Ratings.csv

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| User X | Movie Y | Rating Z | Date |

Έγιναν 2 υλοποιήσεις σε κάθε ζητούμενο του πίνακα, μια με χρήση του RDD API και μια με την Spark SQL. Στην περίπτωση της δεύτερης, υπάρχει η επιλογή να γίνει εκτέλεση σε αρχεία csv ή σε αρχεία parquet με την αντίστοιχη προσθήκη (csv ή parquet) στο τέλος της εντολής εκτέλεσης.
Παρακάτω έχουμε τον ψευδοκώδικα σε MapReduce για την υλοποίηση με RDD API

Q1:

```
map(movies, value):
    for line in movies:
        data = line.split(',')
        if((data[3].split('-')[0]>=2000)
        and data[5]!=0 and data[6]!=0):
            name = data[1]
            year = data[3].split('-')[0]
            profit = ((data[6]-data[5])/data[5])*100
            emit(year, (name, profit))

reduce(year, list((name, profit)...)):
    maxname = '0'
    maxprofit = 0
    for pair in values:
        if (pair[1] > maximum)
            maxname = pair[0]
            maxprofit = pair[1]
    emit(year, (maxname, maxprofit))
```

Q2:

```
map(ratings, value):
    for line in ratings:
        userID = line.split(',')[0]
        rating = line.split(',')[2]
        emit(userID, (rating, 1))

reduce(userID, list((rating, 1)...)):
    sunolo = count = 0
    for pair in values:
        sunolo += pair[0]
        count += pair[1]
    emit(userID, (sunolo, count))

map(userID, (sunolo, count)):
    if ((sunolo/count)>3):
        emit(1, (1, 1))
    else:
        emit(1, (0, 1))

reduce(key, list((polite, 1)...)):
    sunolo = count = 0
    for pair in values:
        sunolo += pair[0]
        count += pair[1]
    emit(key, (sunolo, count))

#to polite einai 0 i 1 analoga me to rating

map(key, (sunolo, count)):
    emit(1, sunolo*100/count)
```

Q3:

```
map(ratings, value):
    for line in ratings:
        movieID = line.split(',')[1]
        rating = line.split(',')[2]
        emit(movieID, (rating, 1))

reduce(movieID, list((rating, 1)...)):
    sunolo = count = 0
    for pair in values:
        sunolo += pair[0]
        count += pair[1]
    emit(movieID, (sunolo/count, count))
#opou sunolo/count = AvgMovieRating

map(movie_genres, value):
    for distinct(line) in movie_genres:
        movieID = line.split(',')[0]
        genre = line.split(',')[1]
        emit(movieID, genre)

Join(ratings, movie_genres)

map(movieID, ((AvgMovieRating, count), genre)):
    genre = values[1]
    AvgMovieRating = values[0][0]
    emit(genre, (AvgMovieRating, 1))

reduce(genre, list((AvgMovieRating, 1))):
    for pair in values:
        sunolo += pair[0]
        count += pair[1]
    emit(genre, (sunolo/count, count))
```

Q4:

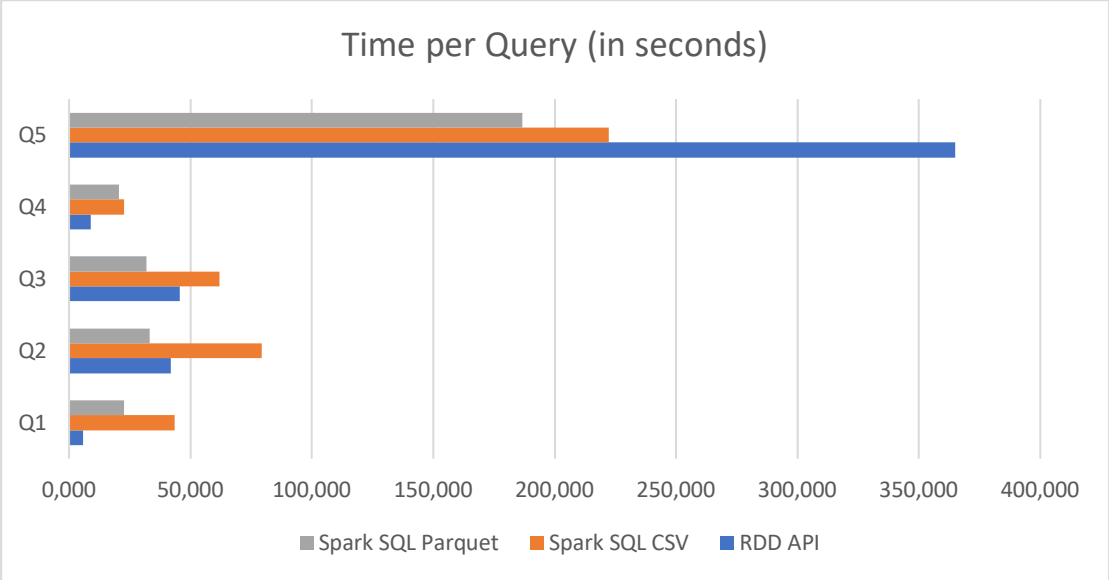
```
map(movies, value):
    for line in movies:
        data = line.split(',')
        if(2000<=(data[3].split('-')[0])<=2019):
            movieID = data[0]
            year = data[3].split('-')[0]
            summary = data[2]
            for word in summary.split(' '):
                emit((movieID, year), 1)

reduce((movieID, year), list(1)):
    for i in values:
        count += i
    emit(movieID, (year, count))
#(movieID, (year, #ofWords))

map(movie_genres, value):
    for line in movie_genres:
        if (line.split(',')[1] == 'Drama'):
            movieID = line.split(',')[0]
            genre = line.split(',')[1]
            emit(movieID, genre)

Join(movies, movie_genres)

map(movieID, ((year, numofWords), genre)):
    if (2000<=values[0][0]<=2004):
        period = "2000-2004"
    elif (2005<=values[0][0]<=2009):
        period = "2005-2009"
    elif (2010<=values[0][0]<=2014):
        period = "2010-2014"
    elif (2015<=values[0][0]<=2019):
        period = "2015-2019"
    numofWords = values[0][1]
    emit(period, (numofWords, 1))

reduce(period, list((numofWords, 1))):
    sunolo = count = 0
    for pair in values:
        sunolo += values[0]
        count += values[1]
    avgRatingLength = sunolo/count
    emit(period, avgRatingLength)
```

Q5:

```
map(movie_genres, values):
    for distinct(line) in movie_genres:
        movieID = line.split(',')[0]
        genre = line.split(',')[1]
        emit(movieID, genre)

map(movies, values):
    for line in movies:
        movieID = data[0]
        name = data[3].split('-')[0]
        popularity = data[7]
        emit(movieID, (name, popularity))

movie_info = Join(movie_genres, movies)

map(movieID, (genre, (name, popularity))):
    emit(movieID, (genre, name, popularity))

map(ratings, values):
    for line in movie_genres:
        movieID = line.split(',')[1]
        userID = line.split(',')[0]
        rating = line.split(',')[2]
        emit(movieID, (userID, rating))

Join(movie_info, ratings)

map(movieID, ((genre, name, popularity), (userID, rating))):
    MaxName=MinName=name
    MaxRating=MinRating=rating
    MaxPopularity=MinPopularity=popularity
    emit((genre, userID), (1, MaxName, MaxRating, MaxPopularity, MinName, MinRating, MinPopularity))

reduce(key, list(values)):
    count = 0
    MaxRating=MaxPopularity=0
    MinRating=MinPopularity=inf
    for item in values:
        count += item[0]
        if ((item[2]>MaxRating) or ((item[2]==MaxRating) and (item[3]>MaxPopularity))):
            MaxName=item[1]
            MaxRating=item[2]
            MaxPopularity=item[3]
        if ((item[5]<MinRating) or ((item[5]==MinRating) and (item[6]>MinPopularity))):
            MinName=item[4]
            MinRating=item[5]
            MinPopularity=item[6]
    emit((genre, userID), (count, MaxName, MaxRating, MaxPopularity, MinName, MinRating, MinPopularity))

map((genre, userID), (count, MaxName, MaxRating, MaxPopularity, MinName, MinRating, MinPopularity)):
    emit(genre, (userID, count, MaxName, MaxRating, MinName, MinRating))

reduce(key, list(values)):
    MaxCount=0
    for item in values:
        if (item[1]>MaxCount):
            MaxCount = item[1]
            userID = item[0]
            MaxName = item[2]
            MaxRating = item[3]
            MinName = item[4]
            MinRating = item[5]
    emit(genre, (userID, MaxCount, MaxName, MaxRating, MinName, MinRating))
```

**Ζητούμενο 4:**

Έπειτα από την εκτέλεση όλων των υλοποιήσεων του ζητούμενου 3 για κάθε query, συγκεντρώσαμε τους χρόνους εκτέλεσης για όλες τις περιπτώσεις RDD API, Spark SQL με CSV αρχείο και Spark SQL με parquet αρχείο. Με τους χρόνους αυτούς κατασκευάσαμε το παρακάτω διάγραμμα:




Όπως διαπιστώνουμε και από το διάγραμμα, ως προς την SparkSQL για όλα τα queries, η χρήση Parquet αρχείων επιφέρει καλύτερα αποτελέσματα. Αυτό ήταν αναμενόμενο καθώς οι Parquet πίνακες είναι αποθηκευμένοι σε ένα columnar format που βελτιστοποιεί το I/O και τη χρήση μνήμης, μειώνοντας έτσι το χρόνο εκτέλεσης, ενώ ταυτόχρονα διατηρεί παραπάνω πληροφορία για το dataset. Αντίθετα, ο χρόνος εκτέλεσης για τον υπολογισμό ερωτημάτων αναλυτικής επεξεργασίας πάνω σε αρχεία CSV δεν είναι αποδοτικός. Χρησιμοποιώντας το infer schema, το οποίο προσθέτει καθυστέρηση, προσπαθεί να μαντέψει αυτόματα τους τύπους δεδομένων για κάθε πεδίο. Με αυτό τον τρόπο, το API διαβάζει κάποιες ενδεικτικές εγγραφές από το αρχείο ώστε να βγάλει συμπεράσματα για το σχήμα του.
Συγκρίνοντας τις υλοποιήσεις με RDD API και με Spark SQL, συμπεραίνουμε ότι για τα πιο απλά queries που θέλουμε μετασχηματισμούς χαμηλού επιπέδου, βέλτιστα αποτελέσματα δίνει η χρήση RDD API, ενώ για τα πιο περίπλοκα ερωτήματα που χρειαζόμαστε παραπάνω MapReduce διεργασίες, η βέλτιστη επιλογή είναι η χρήση της SQL. Η τελευταία διαθέτει αυτόματο βελτιστοποιητή σε αντίθεση με την λύση RDD, όπου η βελτιστοποίηση αφήνεται στα χέρια του προγραμματιστή.
Γενικά, επιλέγουμε RDD API για καλύτερη λειτουργικότητα χαμηλού επιπέδου και έλεγχο, ενώ Spark SQL όταν θέλουμε υψηλού επιπέδου λειτουργίες, μικρότερη δέσμευση χώρου και καλύτερους χρόνους σε πιο περίπλοκα ερωτήματα.

**ΜΕΡΟΣ 2ο:**

**Ζητούμενο 1:**

Με βάση το paper που μας δόθηκε στην εκφώνηση, υλοποιήσαμε το broadcast join στο RDD API (Map Reduce) με τον παρακάτω κώδικα:

```
from pyspark.sql import SparkSession
import csv, time
from io import StringIO

start_time = time.time()

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

R_PATH = "hdfs://master:9000/files/movie-genres-100.csv"
L_PATH = "hdfs://master:9000/files/ratings.csv"


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

def combines(key, val):
        combined = []
        if BrMap.value.get(key, "-") == "-":
                return combined
        for r in BrMap.value.get(key):
                combined.append((key, (r, val)))
        return combined

broadcast_data = sc.textFile(R_PATH). \
        map(lambda row: split_complex(row)). \
        map(lambda x : (x[0], (x[1]) ) ). \
        groupByKey(). \
        map(lambda x : (x[0], list(x[1]))). \
        collectAsMap()

BrMap = sc.broadcast(broadcast_data)

result = \
        sc.textFile(L_PATH). \
        map(lambda row: split_complex(row)). \
        map(lambda x: ( x[1], (x[0], x[2], x[3]) )). \
        flatMap(lambda x: combines(x[0],x[1]))

print(result.collect())
print("Total time: {} sec".format(time.time()-start_time))
```

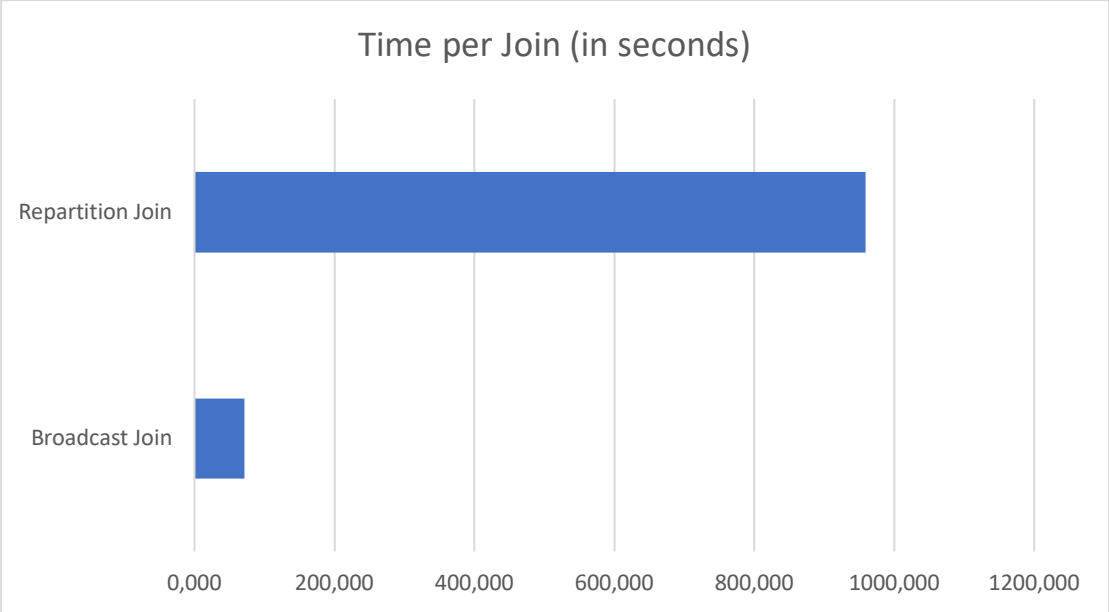
**Ζητούμενο 2:**

Με βάση το paper που μας δόθηκε στην εκφώνηση, υλοποιήσαμε το repartition join στο RDD API (Map Reduce) με τον παρακάτω κώδικα:

```
from pyspark.sql import SparkSession
import csv, time
from io import StringIO

start_time = time.time()

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

R_PATH = "hdfs://master:9000/files/movie-genres-100.csv"
L_PATH = "hdfs://master:9000/files/ratings.csv"

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

def combines(key, list_of_values):
        length = int(len(list_of_values))
        b_r = []
        b_l = []
        for i in range(length):
                elem = list_of_values[i]
                if(elem[0] == 'R'):
                        b_r.append(elem[1:])
                elif(elem[0] == 'L'):
                        b_l.append(elem[1:])
        combined=[]
        for r in b_r:
                for l in b_l:
                        combined.append((key,(r+l)))
        return combined

R_data = \
        sc.textFile(R_PATH). \
        map(lambda row: split_complex(row)). \
        map(lambda x : (x[0],[('R', x[1])] ) )

L_data = \
        sc.textFile(L_PATH). \
        map(lambda row: split_complex(row)). \
        map(lambda x: (x[1],[('L', (x[0], x[2], x[3]) )]))

res = L_data.union(R_data). \
        reduceByKey(lambda x,y: x + y ). \
        flatMap(lambda x: combines(x[0], x[1]))


print(res.collect())
print("Total time: {} sec".format(time.time()-start_time))
```

**Ζητούμενο 3:**

Χρησιμοποιώντας τους κώδικες στα Ζητούμενα 1 και 2, συνενώσαμε τον πίνακα ratings με τον πίνακα movie-genres-100 ο οποίος περιέχει τις πρώτες 100 γραμμές του πίνακα movie genres. Πρώτα κάναμε χρήση του broadcast join και μετά του repartition join. Τα αποτελέσματα στους χρόνους εκτέλεσης φαίνονται στο παρακάτω διάγραμμα.




Όπως βλέπουμε και από το διάγραμμα, υπάρχει πολύ μεγάλη διαφορά στους χρόνους εκτέλεσης. Συγκεκριμένα, το broadcast join είναι σαφώς πιο γρήγορο από το repartition join το οποίο είναι και αναμενόμενο καθώς ο πίνακας movie genres έχει μικρό μέγεθος. Γενικά, το broadcast join θεωρείται το πιο αποδοτικό join όταν έχουμε ένα μεγάλο fact table και ένα σχετικά μικρό dimension table. Το Spark στέλνει ένα αντίγραφο του μικρού πίνακα σε όλους τους executor nodes οπότε δεν χρειάζεται πλέον επικοινωνία όλων με όλους καθώς μπορεί ο καθένας να συνενώσει τα records του μεγάλου dataset με το μικρό broadcasted table.
Το repartition join από την άλλη υλοποιεί μία map και μία reduce διαδικασία. Στη φάση map, κάθε record λαμβάνει μια ετικέτα που δηλώνει από ποιο σύνολο δεδομένων προέρχεται και εξάγεται το (key,value) ζευγάρι. Οι έξοδοι γίνονται partitioned, sorted και merged από το framework. Στη φάση reduce, για κάθε κλειδί, η συνάρτηση χωρίζει και τοποθετεί σε buffers τις εγγραφές εισόδου σε δύο sets ανάλογα με την ετικέτα προέλευσής τους και έπειτα εξάγει κάθε δυνατό συνδυασμό μεταξύ των εγγραφών από τα δύο sets. Με αυτό τον τρόπο, όλες οι εγγραφές για ένα δεδομένο join θα πρέπει να τοποθετηθούν σε buffers το οποίο μπορεί να προκαλέσει πρόβλημα μνήμης.
Στο δικό μας use case έχουμε έναν μικρό πίνακα (movie_genres) και διαθέτουμε μόνο δύο worker nodes, αρα ένα broadcast join σε σύγκριση με ένα repartition είναι προφανώς πιο συμφέρων καθώς δεν θα χρειαστεί να τοποθετήσουμε σε buffers όλες τις εγγραφές του πολύ μεγάλου πίνακα records. Επειδή είναι μικρός ο πίνακας, τοποθετείται στη μνήμη RAM του κάθε worker προκειμένου να αποφύγουμε την επιβάρυνση του δικτύου, ολοκληρώνοντας τη διαδικασία του join από τη φάση map χωρίς τη χρήση reducers.

**Ζητούμενο 4:**

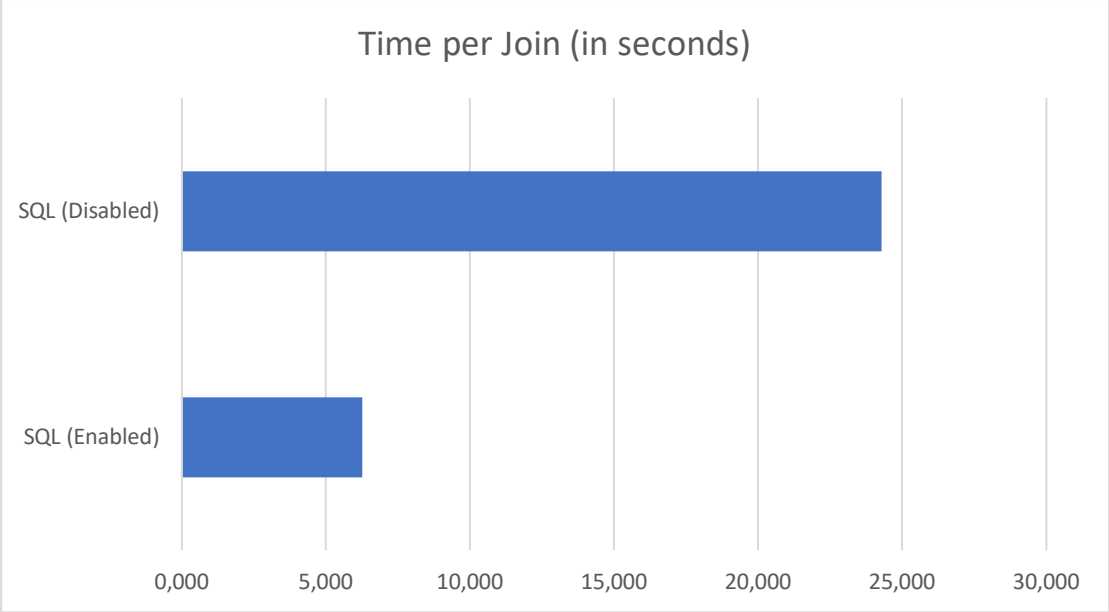
Πλάνο εκτέλεσης χωρίς βελτιστοποιητή:

```
== 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[hd
rtitionFilters: [], PushedFilters: [], ReadSchema: struct<_c0:string,_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: InMemoryFileIn
artitionFilters: [], PushedFilters: [IsNotNull(_c1)], ReadSchema: struct<_c0:string,_c1:string,_c2:string,_c3:string>
Time with choosing join type disabled is 24.2857 sec.
```

Πλάνο εκτέλεσης με ενεργοποιημένο βελτιστοποιητή:

```
== Physical Plan ==
*(3) BroadcastHashJoin [_c0#8], [_c1#1], Inner, BuildLeft
:- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, false]))
:  +- *(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[h
tionFilters: [], PushedFilters: [], ReadSchema: struct<_c0:string,_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: InMemoryFileInde
onFilters: [], PushedFilters: [IsNotNull(_c1)], ReadSchema: struct<_c0:string,_c1:string,_c2:string,_c3:string>
Time with choosing join type enabled` is 6.2620 sec.
```

Όπως βλέπουμε, με την χρήση του βελτιστοποιητή, επιλέγεται το broadcast join αυτόματα ενώ χωρίς τον βελτιστοποιητή, γίνεται ένα Sort Merge Join. Το αποτέλεσμα από τους χρόνους εκτέλεσης φαίνεται από το παρακάτω διάγραμμα:




Όπως είδαμε και στο Ζητούμενο 3, το broadcast join είναι σαφώς καλύτερο για την υλοποίηση του ζητούμενου join (μικρός πίνακας με πολύ μεγαλύτερο).