ΣΑΡΑΝΤΙΝΟΠΟΥΛΟΣ ΕΥΣΤΑΘΙΟΣ
031 16801

Μερος 1
Ζητούμενο 1

Δημιουργώτον φάκελο files χρησιμοποιώντας την ετνολή:

```
hadoop fs -mkdir hdfs://master:9000/files
```

κατόπιν, φορτώνω τα αρχεία στο hdfs με την εντολή

```
hadoop fs -put ./file hdfs://master:9000/files
```

Ζητούμενο 2:
Για την μετατροπή csv σε parquet χρησιμοποιω το script csv2pq.py που βρίσκεται στον φάκελο code του παραδοτέου. Το αρχείο είναι στον φάκελο code του VM και δημιουργεί τα αρχεία κατευθείαν στον φάκελο files του hdfs.

Ζητούμενο 3:
Υλοποιήθηκαν απο 2 αρχεία για κάθε ερώτημα΄ ενα που κάνει χρήση του RDD API και ενα με SPARK SQL. Για τα SQL συγκεκριμένα, ο χρήστης επιλέγει αν θα γίνει χρήση των csv η parquet αρχείων στα arguments της εκτέλεσης. Ακολουθεί λογική και ψευδοκώδικας για κάθε RDD query.

Query 1:

Το συγκεκριμένο ερώτημα είναι αρκετά απλό και λύνεται με 1MR. Κόβουμε τις ταινίες που βγήκαν πριν το 2000, κάνουμε map ώστε να έχουμε κλειδί την χρονολογία values ταινία και κέρδος και με ενα reduce κρατάμε για κάθε χρονιά την ταινία με max κέρδος
Ακολουθεί ο ψευδοκώδικας:

```
movies.csv (id, title, resume, date(), length, cost, revenue, score)
0 1 2 3 4 5 6 7


map(key, value):
# null, line
#split on commas .split(",")
movie = value[1]
date = value[3][0:4]
cost = value[5]
rev = value[6]
cash = rev-cost/cost * 100 #formula to calculate profits
filter (date > 2000, cost != 0, rev !=0)#satisfy conditions
emit ( date, (movie,profit) )



reduce(date, values)
# ^ , (movie,profits)
```

```
# 0, 1(0, 1)
for value in values
if value[1] > max
max = value[1]
movie = value[0]
emit (date,movie,max)
```

Query 2:

Στο συγκεκριμένο ερώτημα κάνουμε map τους χρήστες με κλειδί το user id και και values κριτικές και εναν counter. Κάνουμε reduce ωστε να βρουμε για κάθε χρήστη τον αριθμό τον κριτικών και το άθροισμα των κριτικών ωστ ενα τα διαιρέσουμε για να υπολογίσουμε την average κριτική του. Με χρήση count βρισκουμε τον συνολικό αριθμό χρηστών → n_users
Αφού βρούμε το avg κρατάμε αυτούς που έχουν avg > 3 με μια filter και μετράμε τον αριθμό τους πάλι με χρήση count. → total_users
Η απάντηση είναι το ποιλίκο total_users/n_users*100.
Ακολουθεί ο ψευδοκώδικας:

```
ratings.csv(uid, mid, rating, date)
0 1 2 3

map(key, value)
# null, ratings line
x.split #split on commas
user = [0]
rating = [1]
emit( user, (rating,1) )
#1 to count them

reduce (user, ratings)
# ^ ,(ratings, 1)
# 0 , 1(0, 1)
for rating in ratings
total_ratings += ratings
n_ratings += 1
emit (user, (total_ratings, n_ratings) )

map(user, values)
# ^ , total_ratings, n_ratings
user_avg = total_ratings/n_ratings
emit ( user, user_avg)
# -> users

n_users = users.count()
total_users = users.filter(user_avg > 3)
# x[1]
percentage = total_users/n_users * 100
```

Query 3:

Στο συγκεκριμένο ερώτημα, αρχικά κάνουμε 1MR για να βρούμε την average βαθμολογία κάθε ταινίας, με αντίστοιχο τρόπο με το Q2. Στην συνέχεια κάνουμε το αποτέλεσμα join με τα movie_genres (κοινό κλειδί movie_id) ώστε να βρούμε και την average βαθμολογία κάθε είδους με αντίστοιχο τρόπο με πριν. Μετα το join κάνουμε map για να βάλουμε κλειδί το genre, και values μια τούπλα με την average βαθμολογία της ταινίας και εναν counter που θα μας χρειαστέι για να βρούμε τον συνολικό αριθμό ταινιών κάθε είδους. Βρίσκουμε τον συνολο των βαθμολογιών του είδους καθώς και τον αριθμό ταινιών του και με το ποιλίκο τους έχουμε την average βαθμολογία του είδους.
Ακολουθεί ο ψευδοκώδικας.

```
movies.csv (id, title, resume, date(), length, cost, revenue, score)
0 1 2 3 4 5 6 7
ratings.csv(uid, mid, rating, date)
0 1 2 3
move_genres.csv(mid, genre)
0 1


#parse ratings
#get movie-rating(+counter) pairs from ratings
#map to calculate average rating for each movie
map(key, value)
x.split
mid = [1]
rating = [2]
emit(mid, (rating,1)) #1 for counter

reduce(mid, ratings)
for rating in ratings
total_ratings += ratings
n_ratings += 1
emit (mid, (total_ratings, n_ratings))
map(mid, values)
# ^ , total_ratings, n_ratings
movie_avg = total_ratings/n_ratings
emit ( mid, movie_avg)
#-> this is the movie_avg variable in RDD
#parse movie_genres
map(key, value)
x.split
```

```
mid = [0]
genre = [1]
emit (mid, genre)

#join with movie genres based on m_id
join[ (mid, movie_avg) - (mid, genre) ]on mid
(mid, (movie_avg, genre) )

#like before, map to calculate avg rating for each genre
map (key, values)
#mid, (avg,genre)
emit(genre, (movie_avg, 1)# 1 for counter

reduce(genre, avgs)
for movie in genres
genre_total += move_avg
n_movies += 1
emit (genre, (genre_total, n_movies))

#calculate avg rating for genre
map(genre, values)
# ^ , genre_total, n_movies
genre_avg = genre_total/n_movies
emit ( genre, (genre_avg, n_movies) )



#map with key
```

Query 4

Για το συγκεκριμένο ερώτημα, διαβάζουμε το movie_genres, και κρατάμε μόνο τις drama ταινίες. Επίσης απο το movies, κρατάμε τις ταινίες με έτος κυκλοφορίας >= 2000 και με μια συνάρτηση τις χωρίζουμε στις 5 περιοόδους κυκλοφορίας (μεταβλητή period) . Κάνουμε join τα 2 σύνολα με κοινό κλειδί το movie_id. Το αποτέλεσμα έχει κλειδί movie_id και values την περίοδο και το μήκος της περίληψης για την κάθε ταινία. Κάνουμε ενα map για να πάρουμε την περίοδο ως κλειδί και να προσθέσουμε ενα counter στα values και με ένα reduce έχουμε το average μήκος περίληψης για κάθε περίοδο.
Ακολουθεί ο ψευδοκώδικας.

```
movies.csv (id, title, resume, date(), length, cost, revenue, score)
0 1 2 3 4 5 6 7
move_genres.csv(mid, genre)
0 1

#parse movie_genres, keep only dramas
map(key, value)
# null, line
x.split # split on commas
```

```
mid = [0]
genre = [1]
filter (genre == drama)
emit (mid, genre)


#parse movies, keep year >= 2000, split in 5y periods, calculate resume length
map (key, value)
# null,line
x.split
mid = [0]
resume = [2]
res_len = len(resume.split()) #get word count, split on spaces
year = [3][0:4]
filter (year > 2000, resume != 0)
if year 2000-2004 -> period = 2000-2004 # same for the rest of the perids
emit(mid, period, res_len)#

#join with movie genres based on m_id
join [ (mid, genre) - (mid, period, res_len) ] ON mid
(mid, (genre, period, res_len) )

#restructure in preparation of reduce
#we map it so we get to reduce by period
map (key, values)
# mid, (genre, period, res_len)
emit( period , (res_len, 1) ) #1 for counter to calculate avg res_len

reduce(period, resumes)
for resume in resumes
len_total += res_len
n_resumes += 1
avg_res_len = len_total / n_resumes
emit (period , res_len)
#average resume length for each period.
```

Query 5.

Το query 5 είναι αρκετά περίπλοκο και εξηγείται πιο αναλυτικά στον ψευδοκώδικα αλλά συνοπτικά μπορεί να χωριστεί σε τρια στάδια. Στο πρώτο κάνουμε join τα genres με ta ratings με σκοπό να βρούμε το αριθμό των βαθμολογιών ανα είδος ανα χρήστη. Για να το πετύχουμε, κάνουμε map το rdd με κλειδί την τούπλα (genre, user_uid) και value την ταινια.

```
movies.csv (id, title, resume, date(), length, cost, revenue, score)
```

```
0 1 2 3 4 5 6 7
ratings.csv(uid, mid, rating, date)
0 1 2 3
move_genres.csv(mid, genre)
0 1



##################################################################
'''
PART 0_
PARSES
'''

#parse genres
map(key, value)
#null, lines from genres
x.split #commas..
mid = [0]
genre = [1]
emit(mid, genre)
#-> genres

#parse movies to get score
map(key,value)
#null, lines from movies
x.split
mid = [0]
title = [1]
score = [7]
emit(mid, (title,score) )
#-> scores

#parse ratings
#get movie-rating counter for MR
map(key, value)
#null, lines from ratings
x.split
uid = [0]
mid = [1]
rating = [2]
n_ratings = 1
emit(mid, (uid, rating, n_ratings))
#key mid in order to join with genres
#-> ratings

#create ratings2
#used later to join with user-genre on uid
map(key, value)
#mid, (uid, rating, n_ratings)
```

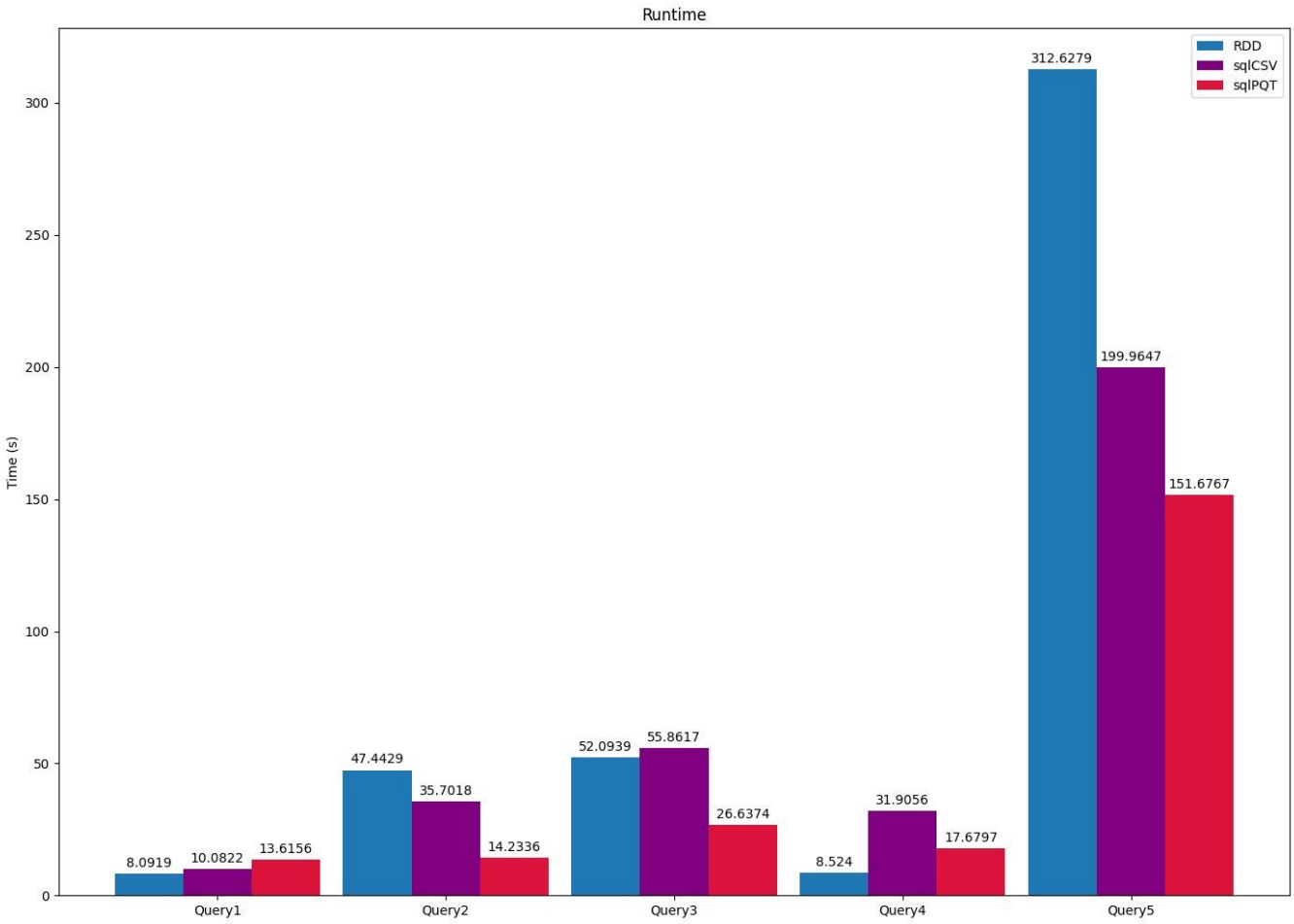
```
emit(uid, (mid, rating))
#-> ratings2

##########################################################
'''
PART A_
join ratings with genres to get number of ratings PER genre PER user
'''

#join ratings with genres on mid

join[ (mid, genre) U (mid, (uid, rating, n_ratings)) ] on mid
-> ( mid, {(genre), (uid, rating, n_ratings)} )

#now that they are joined, we map with genre-uid key
#in order to easier calculate top_users per genre

map(key, value)
#mid, (genre, uid, rating, n_ratings)
genre-user = (genre, uid) #
movie-ratings = (mid, rating, n_ratings)
emit (genre-user , movie-ratings)

#we now reduce to count
#the number of ratings per genre per user

reduce(genre-user, movie-ratings)
for rating in genre-user
n_ratings += n_ratings
emit (genre-user , n_ratings)

#example: (drama-15, 167)
#For genre 'Drama', user '15', has rated '167' movies

#remap to better parse by genre
map(genre-user, n_ratings)
genre = [0][0]
user = [0][1]
n_ratings = [1]
emit(genre, (user, n_ratings))

#reduce to keep only the users with the highest ratings
reduce(genre, (user, n_ratings))
top_user, top_n_ratings = 0
for user in genre
if n_ratings > top_n_ratings
top_user = user
emit(genre, (top_user, top_n_ratings))

#we now have top user per genre with his number of ratings
```

```
# remap to have uid as key to prepare for next join
map(key, value)
#genre, (top user, top ratings)
emit(top_user, (genre, top_n_ratings))



##########################################################################
#############

'''
PART B)
calculate movie POPULARITY PER USER PER GENRE
we have to rejoin with ratings to get mid from our uid
then with genres to filter again
and finally with movies to get scores for genre

ratings2(mid, (uid, rating) )
genres(mid, genre)
movieScores(mid, (title, score) )
'''

#86
#start by joining genreUSer with ratings2

join ratingsGenreUser with ratings2
join[ (top_user'uid', (genre, top_n_ratings) ) U (uid, (mid, rating)) ] on mid
-> ( uid, {(genre,top_n_ratings), (mid, rating)} )

#map result with mid as key to join with genres
map(key, value) #
emit( mid, (genre, top_ratings, uid, rating) )

#join with genres to have filter material
join ratingsGenreUser+ with genres
join [ (mid, (top_genre, uid, top_ratings, rating)) U (mid, genre) ]
-> (mid, {(top_genre, uid, top_ratings, rating), genre)} )

#filter to clear, genre = top_genre
filter (top_genre == genre)
# [1][0][0] == [1][1]

join ratings++ with scores
join [ (mid, {(top_genre, uid, top_ratings, rating), (genre)}) U (mid, {(title, score)}) ]
-> (mid, [{(top_genre, uid, top_ratings, rating), (genre)}, {(title, score)}] )
# 0 1000 1001 1002 1003 1010 1100 1101
#all data collected
#remap with top_genre, top_user, top_ratings as key
#so that we calculate best and worst score movies
#PER TOP_USER per genre
```

```
map(key, value)
#join result
top_genre = [1][0][0][0]
top_user = [1][0][0][1]
top_rating = [1][0][0][2]
genre-user-rating = ( top_genre, top_user, top_rating )
mid = [0]
title = [1][1][0][0]
rating = [1][0][0][3]
score = [1][1][0][1]
emit(genre-user-rating, (rest))
```

Ζητούμενο 4

Για την μέτρηση των χρόνων χρησιμοποιούμε την μέθοδο που δίνεται στο παράδειγμα κώδικα της εκφώνησης (time.time()). Οι χρόνοι κρατιούνται όταν τα αποτελέσματα γράφονται στον φάκελο outputs του hdfs, στο αρχείο times.txt για rdd, csv και parquet αντιστοιχα. Για τον σχεδιασμό των barplots χρησιμοποιούμε τον κώδικα barplotQueries.py που βρίσκεται στον φάκελο code του παραδοτέου. Ακολουθεί το διάγαμμα:



Συμπεράσματα:

Παρατηρούμε οτι σε γενικές γραμμές το RDD API είναι πιο αργό απο το SPARK SQL και πως η επεξεργασία Parquet αρχείων είναι σημαντικά γρηγορότερη απο ta csv.
Εξαίρεση αποτελούν τα ερωτήματα Q1, Q4 όπου το rdd είναι γρηγορότερο.
Με μια πιο προσεκτική ματιά μπορούμε να δούμε οτι αυτό συμβαίνει επειδή σε αυτά τα ερωτήματα δεν γίνεται χρήση του αρχείου ratings το οποίο είναι κατα πολύ μεγαλύτερο σε μέγεθος απο τα υπόλοιπα.
Αντίθετα στα Q2,Q3,Q5 όπου χρησιμοποιούμε τα ratings, η Spark SQL υλοποίηση (ειδικά με χρηση parquet) είναι αρκετά ταχύτερη.
Με βάση αυτό μπορούμε να συμπεράνουμε οτι το RDD api είναι πιο αποδοτικό όταν έχουμε να διαχειριστούμε αρχεία μικρότερου μεγέθους.

Αν εξαιρέσουμε το ερώτημα 1 το οποίο έχει γενικά πολύ μικρό χρόνο εκτέλεσης και πιθανώς περιθώριο λάθους, η χρήση Spark SQL με αρχεία Parquet είναι η βέλτιστη επιλογή.

Παρατηρήσεις για την χρήση Parquet.

Οι λόγοι για τους οποίου τα συγκεκριμένα αρχεία είναι καλύτερα, αναφέρθηκαν και στο ζητούμενο 2 όπου κληθήκαμε να τα δημιουργήσουμε. Οστώσο, συνοπτικά, τα αρχεία Parquet έχουν τα εξής πλεονεκτήματα:

Σε αντίθεση με τους περισσότεορους τύπους αρχείων που είναι row based, τα parquet einai columnar based. Το format αυτό τα κάνει πολύ αποδοτικά σε χρήση queries αφού κάθε κολόνα τέινει να εχει παρόμοια δεδομένα αφού

Μας δίνουν την δυνατότητα να skipparoume γρήγορα irrelevant data κάνοντας aggregation queries γρηγορότερα

Κάνουν πολύ πιο αποδοτικό το compression.

Εχουν σημαντικά μικρότερο αποτύπωμα στην μνήμη κάνοντας το read/write πολύ πιο πιο γρήγορο

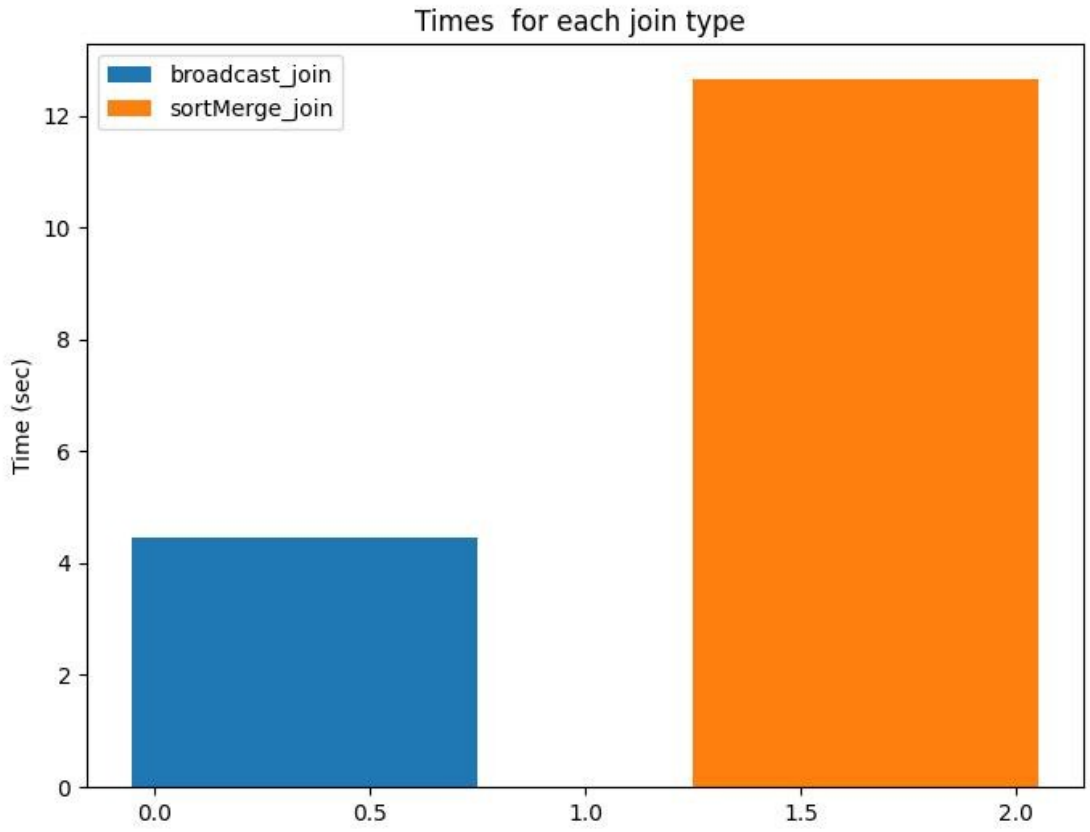
Ακόμη δεν χρειάζονται infer schema καθως τα parquet έχουν ενα “metadata αρχειο” το οποίο κρατάει τις πληροφορίες του σχήματος για όλα τα δεδομένα του αρχείου.

Η αποφυγή του inferschma είναι πολύ σημαντική στον χρόνο αφού μας γλυτώνει ενα πέρασμα.

ΜΕΡΟΣ 2ο.

Ζητούμενο 4

Χρησιμοποιούμε το script της εκφώνησης με τις κατάλληλες τροποποιήσεις που βρίσκεται στην θέση code/2/optimizeJoin.py και το τρέχουμε 2 φορές, με τον βελτισοποιητή ενεργοποιημένο και απενεργοποιημένο αντίστοιχα.



Τα αποτελέσματα απο το explain είναι τα εξής:

Y

```
== 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 4.4712 sec.

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 12.6570 sec.
```

Βλέπουμε πως το broadcast join με ενεργοποιημένο τον βελτιστοποιητή ειναι αισθητά πιο γρήγορο απο το sort merge join που συμβαίνει οταν απενεργοποιούμε τον βελτιστοποιητή με το πρώτο να είναι 3 φορές γρηγορότερο απο το δεύτερο.
Αυτό συμβαίνει γιατί έκτός από τα πλεονεκτήματα του broadcast, όπως βλέπουμε στον πλάνο εκτέλεσης, η sort merge join κάνει ενα sorting το οποίο προσθέτει αισθητά σε χρόνο και πολυπλοκότητα.