9ο Εξάμηνο, Ακαδημαϊκό Έτος 2021-2022

Προχωρημένα Θέματα Βάσεων Δεδομένων

Χρήση του Apache Spark στις Βάσεις Δεδομένων

Αλεξόπουλος Ιωάννης, 03117001
Ζάρα Στέλλα, 03117154
Λιάγκα Αικατερίνη, 03117208

Μέρος 1ο : Υπολογισμός Αναλυτικών Ερωτημάτων με τα APIs του Apache Spark

Ζητούμενο 3

Ψευδοκώδικας Map Reduce για τις υλοποιήσεις με το RDD API

- **Query 1**

```
map(line_id, line):
 data = line.split(',')
 if (data[3] != ‘’ && int(data[3][0:4]) >= 2000 &&
int(data[5]) != 0 && int(data[6]) != 0)
 date = int(data[3][0:4])
 name = data[1]
 cost = int(data[5])
 income = int(data[6])
 profit = 100*((income-cost)/cost)
 emit(date, (name, profit))

reduce(date, tuple_list):
 max = 0
 name = ‘None’
 for tuple in tuple_list:
   if (tuple[1] > max):
   max = tuple[1]
   name = tuple[0]
 emit(date, name, max)
```

- **Query 2**

```
map(line_id, line):
 line = line.split(',')
 user_id = line[0]
 rating = float(line[2])
 emit(user_id, (rating,1))

reduce(user_id, rating_tuples):
 rating_sum = 0
 rating_count = 0
 for rating in rating_tuples:
  rating_sum += rating[0]
  rating_count += rating[1]
 emit(rating_sum, rating_count)

map(rating_sum, rating_count)
 if (rating_sum/rating_count) > 3:
  emit(1, (1,1))
 else:
```

```
  emit(1, (1,0))

reduce(1, indicator_tuples):
 total_users = 0
 ratings_greater_than_three = 0
 for i in indicator_tuples:
  total_users += i[0]
  ratings_greater_than_three += i[1]
 emit(total_users, ratings_greater_than_three)

map(total_users, ratings_greater_than_three):
 percentage = (total_users/ratings_greater_than_three)*100
 emit(1,percentage)
```

- **Query 3**

```
# MR 1 -> vote_avg

# read and create dataset from ratings.csv
map(line_id, line):
    # value = (user_id, movie_id, rating, time) = line from
ratings.csv
    line = line.split(",")
    movie_id = line[1]
    rating = float(line[2])
    emit (movie_id, (rating, 1))

# for each movie get rating_sum and reating_count so we can then
calculate average
reduce(movie_id, values):
    # values = list((rating, 1), ...)
    ratingCount = 0
    ratingSum = 0
    for i in values:
        ratingSum += i[0]
        ratingCount += i[1]
    emit (movie_id, (ratingSum, ratingCount))

# calculate average
map(movie_id, (ratingSum, ratingCount)):
    movie_rating_avg = ratingSum/ratingCount
    emit (movie_id, movie_rating_avg)


# MR 2 -> genres

# read and create dataset from movie_genres.csv
map(line_id, line):
    # value = (movie_id, genre) = line from movie_genres.csv
    line = line.split(",")
    movie_id = line[0]
    genre = line[1]
```

```
    emit (movie_id, genre)


# MR -> result (res)

# join result = (movie_id, (genre, movie_average))
join(genres, vote_avg)

# make genre the key
map( null, (movie_id, (genre, movie_average)) ):
    emit (genre, (movie_average, 1))

# calculate count and sum for the movie_average rating for each genre
reduceByKey(genre, values):
    # values = list((movie_average, 1), ...)
    avgRatingCount = 0
    avgRatingSum = 0
    for i in values:
        avgRatingSum += i[0]
        avgRatingCount += i[1]
    emit (genre, (avgRatingSum, avgRatingCount))

# calculate genre rating average
map( null, (genre, (avgRatingSum, avgRatingCount)) ):
    average_rating_per_genre = avgRatingSum/avgRatingCount
    genre_movie_count = avgRatingCount  # because each movie has its
own vote_avg
    emit (genre, (average_rating_per_genre, genre_movie_count))
```

- **Query 4**

```
map(line_id, line):
    # value = (user_id, movie_id, rating, time) = line from
ratings.csv
    line = line.split(",")
    movie_id = line[0]
    genre = line[1]
    emit (movie_id, genre)

filter(movie_id, genre):
    if genre == "Drama":
        emit(movie_id, genre)

map(line_id, line):
    line = split_complex(line)
    emit(movie_id,(Id, Title, Summary, Date, Duration, Cost, Profit,
Popularity))

filter(movie_id, (Id, Title, Summary, Date, Duration, Cost, Profit,
Popularity)):
     summary = Summary.split("-")
     date = Date.split("-")
```

```
     if summary != " " and date[3] != " " and date[3] >= 2000 and
date[3] <= 2019:
        emit(Id, Title, Summary, Date, Duration, Cost, Profit,
Popularity)

map(info):
     #info -> (Id, Title, Summary, Date, Duration, Cost, Profit,
Popularity)
     (movie_id, (5_year_period, number_of_summary_words)) =
mapMovies(info)
     emit(movie_id, (5_year_period, number_of_summary_words))

join(movies, genres)

map(movie_id, ((5_year_period, number_of_summary_words), genre)):
     emit(5_year_period, (number_of_summary_words,1))

reduce(5_year_period, summary_words_tuples):
     word_sum = 0
     word_count = 0
     for i in summary_words_tuples:
        word_sum += number_of_summary_words
        word_count += 1
     emit(5_year_period, (word_sum, word_count))

map(5_year_period, (word_sum, word_count)):
    emit(5_year_period, word_sum/word_count)
```

❖ Query 5

```
# MR 1 -> movies

# read and create dataset from movies.csv
from q5.q5_rdd_solution import split_complex


map(null, value):
    # value = (movie_id, title, summary, date, duration, cost, profit,
popularity) = line from movies.csv
    movie_id = split_complex(value)[0]
    title = split_complex(value)[0]
    poplularity = float(split_complex(value)[7])
    emit (movie_id, (title, popularity))


# MR 2 -> ratings

# read and create dataset from ratings.csv
map(null, value):
    # value = (user_id, movie_id, rating, time) = line from
ratings.csv
    movie_id = value.split(",")[1]
```

```
    user_id = value.split(",")[0]
    rating = float(value.split(",")[2])
    emit (movie_id, (user_id, rating))


# MR 3 -> genres

# read and create dataset from movie_genres.csv
map(null, value):
    # value = (movie_id, genre) = line from movie_genres.csv
    movie_id = value.split(",")[0]
    genre = value.split(",")[1]
    emit (movie_id, genre)


# MR 4 -> helperRatings

# From ratings (MR 2) rearrange so : key = user_id
map(movie_id, (user_id, rating)):
    emit (user_id, (movie_id, rating))


# MR 5 -> ratingsFormated

# join result = (movie_id, ( (user_id, rating), genre) )
join(ratings, genres)

# key = (genre, user_id) and value = 1 -> counter for ratings per
genre per user
map( null, (movie_id, ((user_id, rating), genre)) ):
    emit ((genre, user_id), 1)

# count ratings per genre per user
reduceByKey((genre, user_id), values):
    # values = list(1, 1, 1, ...)
    counter = 0
    for i in values:
        counter += i
    emit ((genre, user_id), ratingCount_perUser_perGenre)

# key = genre
map( null, ((genre, user_id), ratingCount_perUser_perGenre) ):
    emit (genre,  ( ratingCount_perUser_perGenre, (userId) ))

# findMaxRatingUser
# res = (maxRatingCount_perGenre, (userId, ...)) -> res[1]= ALL users
that have max rating count in same genre
reduceByKey(genre, values):
    # values = ( (ratingCount_perUser_perGenre, (userId)), ... )
    res = tuple()     # tuple, in here all users have same
ratingCount_perUser_perGenre
    for i in values:
        current_ratingCount = i[0]
```

```
        current_userId = i[1]   # tuple
        if (users==() or current_ratingCount >
users[0].ratingCount_perUser_perGenre):
            res = tuple(i)
        elif (current_ratingCount <
users[0].ratingCount_perUser_perGenre):
            continue    # res stay the same
        else:  # if current_ratingCount ==
users[0].ratingCount_perUser_perGenre
            res[1] += current_userId   # add to userIds_tuple the new
userId

    emit ( genre, (maxRatingCount_perGenre, list(userId, ...)) )

# mapMultipleUsers
# if multiple users for 1 maxRatingCount create different row in map
for each user
flatMap(null, values):
    # values = ( genre, (maxRatingCount_perGenre, list(userId, ...)) )
-> one line
    genre = value[0]
    ratingCount = value[1][0]
    userIdList = value[1][1]
    res = []
    for userId in userIdList:
        output_item = ( userId, (ratingCount, genre) )
        res.append(output_item)
    emit [ (user_id, (maxRatingCount_perGenre, genre)), ...]    # each
array position will become one line

# currMR == result of the flatMap above
# join result =  (user_id, ( (maxRatingCount_perGenre, genre),
(movie_id, rating)) )
join(currMR, helperRatings)

map( null, (user_id, ( (maxRatingCount_perGenre, genre), (movie_id,
rating))) ):
    ratingCount = maxRatingCount_perGenre
    wantedgenre = genre
    emit (movie_id, (user_id, ratingCount, wantedgenre, rating))

# currMR == result of the map above
# join result = ( movie_id, ( (userId, ratingCount, wantedgenre,
rating), genre) )
join(currMR, genres)

# wantedGenre == genre, so we can search the ratings for the
approriate category (genre)
filter( movie_id, ((userId, ratingCount, wantedgenre, rating), genre)
):
    if (wantedgenre == genre):
        emit(movie_id, ((userId, ratingCount, wantedgenre, rating),
genre))
```

```
# currMR == result of the filter above
# join result = ( movie_id, (    ( (userId, ratingCount, wantedgenre,
rating), genre),   (title, popularity) ) )
join(currMR, movies)

# key = (genre, user_id, ratingCount) and values = (rating,
popularity, title) so then we can search for min, max rating of these
users for this genre
# ratingsFormated result = MR 5 result
map(null, (movie_id, (    ( (user_id, ratingCount, wantedgenre,
rating), genre),   (title, poplularity) )) ):
    emit ( (genre, user_id, ratingCount), (rating, popularity, title)
)


# MR 6a -> minRatings -> use ratingsFormated result

reduceByKey((genre, user_id, ratingCount), values):
    # values = list((rating, popularity, title), ...)
    res = tuple()
    for i in values:
        curr_rating = i[0]
        minRating = res [0]
        if res==() or curr_rating < minRating:
            res = i
        elif minRating < curr_rating:
            continue    # dont change res
        else:   # if curr_rating == minRating
            minPopularity = res[1]
            curr_popularity = i[1]
            if minPopularity < curr_popularity:
                res = i
            else:
                continue    # res stays the same
    # res contains min rating row
    emit ((genre, user_id, ratingCount), res)


# MR 6b -> maxRatings -> use ratingsFormated result
reduceByKey((genre, user_id, ratingCount), values):
    # values = list((rating, popularity, title), ...)
    res = tuple()
    for i in values:
        curr_rating = i[0]
        maxRating = res [0]
        if res==() or curr_rating > maxRating:
            res = i
        elif maxRating > curr_rating:
            continue    # dont change res
        else:   # if curr_rating == minRating
            maxPopularity = res[1]
            curr_popularity = i[1]
```

```
            if maxPopularity < curr_popularity:
                res = i
            else:
                continue    # res stays the same
    # res contains max rating row
    emit ((genre, user_id, ratingCount), res)


# MR -> res

# join result -> ( (genre, user_id, ratingCount), ((ratingMin,
populMin, titleMin) , (ratingMax, populMax, titleMax)) )
join(minRatings, maxRatings)

# map to make genre key so we can sort be genre
map(  null, ( (genre, user_id, ratingCount), ((ratingMin, populMin,
titleMin) , (ratingMax, populMax, titleMax)) )  ):
    emit (genre , (user_id, ratingCount, titleMax, ratingMax,
titleMin, ratingMin))


sortByKey()

# RESULT
map(  null, ( (genre, user_id, ratingCount), ((ratingMin, populMin,
titleMin) , (ratingMax, populMax, titleMax)) )  ):
    emit ( genre, user_id, ratingCount, titleMax, ratingMax, titleMin,
ratingMin )
```

Σημείωση: Στον ψευδοκώδικα που παρατέθηκε για τα queries έχουν χρησιμοποιηθεί τα ονόματα κάποιον συναρτήσεων οι οποίες ορίστηκαν για δική μας διευκόλυνση και φαίνοντια και στον αντίστοιχο κώδικα στον φάκελο code. Αυτές είναι οι εξής:

- Η split_complex, η οποία ορίστηκε ώστε να γίνεται σωστά το split στο αρχείο movies.csv σε περίπτωση που ο τίτλος κάποιας ταινίας περιέχει κόμμα. Ο κώδικας φαίνεται παρακάτω:

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

- Η mapMovies, η οποία μετράει τον αριθμό των λέξεων για την περιγραφή κάθε ταινίας που κυκλοφόρησε εντός συγκεκριμένου χρονικού διαστήματος. Ο κώδικας φαίνεται παρακάτω:

```
def mapMovies(x):
    movie_id = x[0]

    SummaryWords = x[2].split(" ")
    count = 0
    for i in SummaryWords:
```

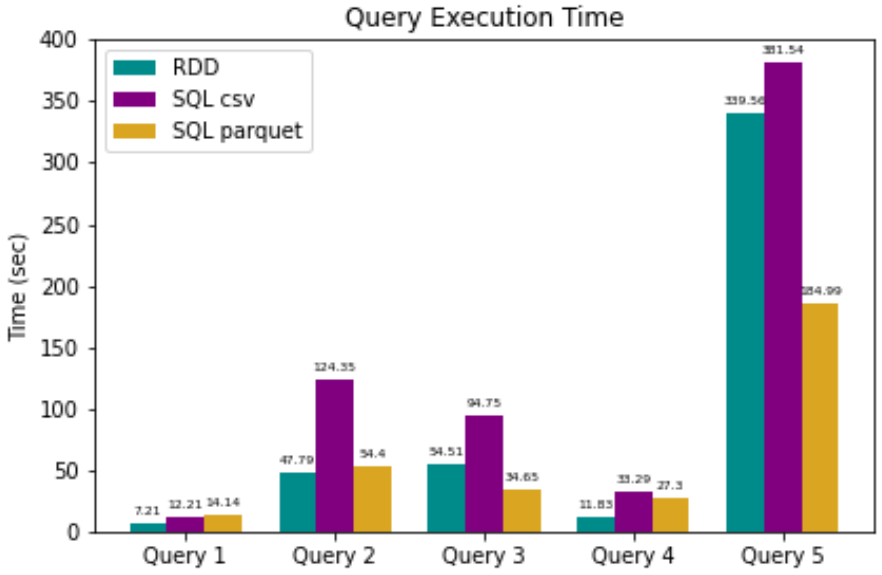
```
        count += 1

    y = int(x[3].split("-")[0])
    if  y >= 2000 and y <= 2004:
        year = "2000-2004"
    elif y >= 2005 and y <= 2009:
        year = "2005-2009"
    elif y >= 2010 and y <= 2014:
        year = "2010-2014"
    else:
        year = "2015-2019"

    return (movie_id, (year, count))
```

Ζητούμενο 4

Παρακάτω παρατίθεται το ραβδόγραμμα με τους χρόνους εκτέλεσης των queries, για κάθε υλοποίηση:



Ραβδόγραμμα με χρόνους εκτέλεσης ερωτημάτων για τις διαφορετικές υλοποιήσεις

Από το ραβδόγραμμα παρατηρούμε ότι υπάρχει ένα μοτίβο στη συμπεριφορά ανάλογα με την υλοποίηση. Συγκεκριμένα, η υλοποίηση σε Spark SQL που διαβάζει από το csv αρχείο παρουσιάζει σταθερά την χειρότερη επίδοση. Όπως ήταν αναμενόμενο, η αντίστοιχη υλοποίηση που διαβάζει από αρχείο parquet παρουσιάζει σημαντική βελτίωση. Αυτό οφείλεται στο τρόπο με τον οποίο διαχειρίζεται τα δεδομένα το parquet, αφού έχει μικρότερο αποτύπωμα στη μνήμη, μειώνοντας έτσι τους χρόνους ανάγνωσης και εγγραφής ενώ ακόμα κρατάει επιπρόσθετα δεδομένα (metadata) με τα οποία προσφέρει βελτιστοποιημένη πρόσβαση σε αυτά. Ειδικότερα, στα πιο απαιτητικά και χρονοβόρα queries, βλέπουμε ότι η ανάγνωση από αρχείο parquet κόβει το χρόνο εκτέλεσης σχεδόν στο μισό. Οι μόνες περιπτώσεις στις οποίες τα δύο format δεν διαφέρουν ιδιαίτερα στην επίδοση είναι τα πιο σύντομα queries οπού παρατηρούμε πως το parquet μπορεί να είναι ακόμα και πιο αργό (Query 1). Το γεγονός αυτό μπορεί να οφείλεται σε κάποιο bottleneck το οποίο προκύπτει

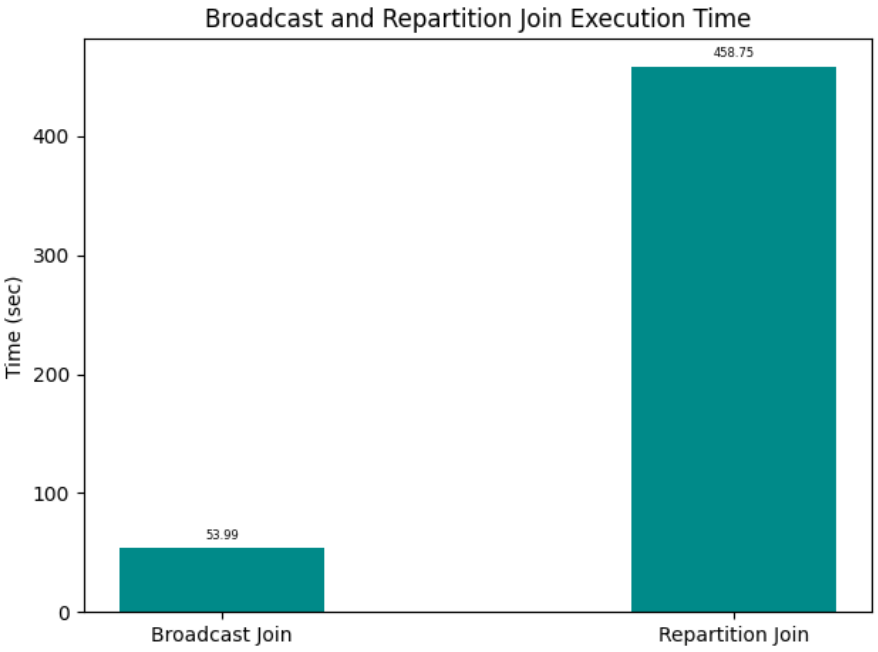
από την αρχική ανάγνωση, το οποίο όμως στα πιο χρονοβόρα queries αντισταθμίζεται από τη καλύτερη συνολική διαχείριση δεδομένων από το parquet.

Όσον αφορά τη σύγκριση του Spark SQL με το RDD API παρατηρούμε ότι στα χρονοβόρα queries το Spark SQL (η υλοποίηση που διαβάζει από τα αρχεία parquet) προσφέρει αισθητά βελτιωμένους χρόνους. Το γεγονός αυτό οφείλεται στο ότι αν και τα δύο API βασίζονται στο ίδιο core, το Spark SQL μας δίνει τη δυνατότητα να δώσουμε πολύ πιο συγκεκριμένες οδηγίες με τη χρήση των queries, σε σχέση με το RDD API το οποίο βασίζεται κυρίως στον κατάλληλο μετασχηματισμό των δεδομένων, ενώ προσφέρει και πλήθος βελτιστοποίησεων όσον αφορά την εκτέλεσή τους. Ακόμα, στα πιο σύντομα queries βλέπουμε ότι οι χρόνοι επεξεργασίας τείνουν να είναι παρόμοιοι ή και μερικές φορές το RDD API να είναι καλύτερο, αλλά αυτό θα μπορούσε να οφείλεται εν μέρει και στον τρόπο δόμησης των queries (μπορεί να μην είναι οι βέλτιστες λύσεις) και με αυτό τον τρόπο να υπονομεύεται η λειτουργία του Spark SQL σε σχέση με το RDD API. Εξάλλου, όπως είπαμε το Spark SQL προσφέρει πιο εξειδικευμένες εντολές σχετικά με τη διαχείριση των δεδομένων άρα είναι πιο “ευάλωτο” σε κακές υλοποίησεις σε σχέση με το RDD API.

Μέρος 2ο: Υλοποίηση και μελέτη συνένωσης σε ερωτήματα και Μελέτη του βελτιστοποιητή του Spark

Ζητούμενο 3

Παρακάτω παρατίθεται το ραβδόγραμμα με τους χρόνους εκτέλεσης για το Broadcast Join και το Repartition Join:

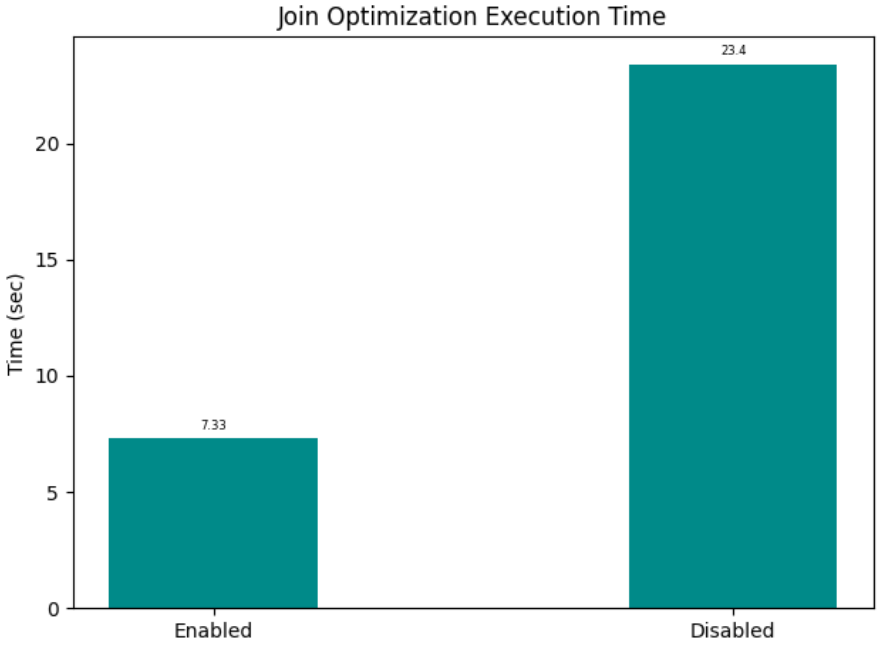


Ραβδόγραμμα με χρόνους εκτέλεσης join

Όπως φαίνεται και στο ανωτέρω διάγραμμα οι δύο τεχνικές join παρουσιάζουν σημαντική διαφοροποίηση ως προς το χρόνο εκτέλεσης τους. Συγκεκριμένα, η εκτέλεση του Repartition Join απαιτεί σχεδόν εννιά φορές περισσότερο χρόνο σε σχέση με την εκτέλεση του Broadcast Join. Το αποτέλεσμα αυτό είναι καθ' όλα αναμενόμενο, καθώς στη μια περίπτωση έχουμε ένα Map Side Join, ενώ στην άλλη ένα Reduce Side Join. Στο Repartition Join αφού ολοκληρώσουμε τη διαδικασία από την πλευρά του mapper, οδηγούμαστε στην εκτέλεση ενός groupByKey (το οποίο λειτουργεί σαν reduce) για να προσθέσουμε έπειτα ένα στάδιο flatMap (λειτουργεί σαν map), από το οποίο θα προκύψει το τελικό αποτέλεσμα. Το pipeline του RDD είναι σημαντικά πολυπλοκότερο με φυσικό επακόλουθο η διαδικασία να απαιτεί πολύ περισσότερο χρόνο, ενώ επιπλέον, τα δεδομένα μεταφέρονται πάνω από το δίκτυο μεταξύ του σταδίου Map και Reduce, κάτι που μας οδηγεί αναπόφευκτα σε περαιτέρω χρονική επιβάρυνση. Ο χρόνος βέβαια και των δύο υλοποιήσεων είναι αυξημένος καθώς γίνεται αποθήκευση του αποτελέσματος στο hdfs σε μορφή CSV προς αποφυγή λανθασμένων μετρήσεων λόγω lazy evaluation του Spark.

Ζητούμενο 4

Παρακάτω παρατίθεται το ραβδόγραμμα με τους χρόνους εκτέλεσης καθώς και τα πλάνα εκτέλεσης που προέκυψαν από κάθε βελτιστοποιητή:



Ραβδόγραμμα με χρόνους εκτέλεσης βελτιστοποιητών

```
== 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 23.4049 sec.
```

Πλάνο εκτέλεσης με απενεργοποιημένο το βελτιστοποιητή

```
== 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 7.3323 sec.
```

Πλάνο εκτέλεσης με ενεργοποιημένο το βελτιστοποιητή

Επεμβαίνοντας στον βελτιστοποιητή και απενεργοποιώντας την δυνατότητα του να κάνει optimize τα join χρησιμοποιώντας το BroadcastHash Join, ο ίδιος οδηγείται σε ένα πλάνο εκτέλεσης με SortMerge Join, το οποίο συνιστά την αμέσως καλύτερη επιλογή στην περίπτωση ενός equi-join με sortable κλειδιά. Όπως είναι αναμενόμενο, το πλάνο εκτέλεσης αυτό έχει σημαντικά χειρότερη απόδοση, οδηγώντας σε σχεδόν διπλάσιο χρόνο εκτέλεσης. Άλλωστε στη μια περίπτωση έχουμε ένα Map-side Join κατά το οποίο, το μικρότερο data set (εφόσον είναι αρκετά μικρό ώστε να χωράει στην μνήμη ενός worker node) γίνεται broadcast σε όλους τους worker nodes πριν την εφαρμογή του Hash Join ενώ στην άλλη περίπτωση αν και δουλεύουμε και πάλι με Map-side Join η μεταφορά των δεδομένων (shuffle) είναι πολυπλοκότερη, καθώς για να ολοκληρωθεί το Hash Join πρέπει οι εγγραφές με τα ίδια

κλειδιά, στα οποία γίνεται το join, από κάθε dataset να φτάσουν στους ίδιους workers. Επιπρόσθετα, χρειάζεται τα κλειδιά αυτά να ταξινομηθούν (sort) με τρόπο τέτοιο ώστε να μπορούν να γίνουν parse παράλληλα και να γίνει το join ανάμεσα στις τούπλες που μοιράζονται τα ίδια κλειδιά. Δηλαδή, το SortMerge Join προσθέτει κάποια περαιτέρω στάδια επεξεργασίας για να υπάρχει εγγύηση ότι τα δεδομένα που αντιστοιχούν στα ίδια keys θα οδηγηθούν στα ίδια partitions και θα είναι ταξινομημένα με τον ίδιο τρόπο.