Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Εξάμηνο 9ο: Προχωρημένα Θέματα Βάσεων Δεδομένων *(2021-2022)*
Αναφορά Εξαμηνιαίας Εργασίας

el17088 - Στυλιανός Κανδυλάκης
el17025 - Χρήστος (Κίτσος) Ορφανόπουλος
el17176 - Χρήστος Τσούφης

Μέρος 1ο

Ζητούμενο 1

1. Καταρχάς κατεβάζουμε και αποσυμπιέζουμε τα απαιτούμενα αρχεία στο VM μας.

```
user@master: $wget --no-check-certificate \
'http://www.cslab.ntua.gr/courses/atds/movie_data.tar.gz' -O data.tar.gz
user@master: tar -xzf data.tar.gz
```

2. Στη συνέχεια δημιουργούμε τον φάκελο files εντός του hdfs:

```
user@master: $hadoop fs -mkdir hdfs://master:9000/files
```

3. Τέλος προσθέτουμε διαδοχικά τα τρία αρχεία:

```
user@master: $hadoop fs -put movie_genres.csv hdfs://master:9000/files
user@master: $hadoop fs -put movies.csv hdfs://master:9000/files
user@master: $hadoop fs -put ratings.csv hdfs://master:9000/files
```

Τα αρχεία μας τώρα βρίσκονται στο hdfs: http://83.212.76.41:50070/explorer.html#/files

Ζητούμενο 2

Χρησιμοποιώντας τον κώδικα **csv_to_parquet.py** παράγουμε τα απαιτούμενα parquet αρχεία. Σημείωση: η διαδικασία αργεί λίγο.

Ζητούμενο 3

Για καθένα query υλοποιήσαμε τρεις λύσεις, μία με χρήση του RDD API και δύο με χρήση του SPARK SQL, για csv και parquet αρχεία αντίστοιχα. Παρακάτω για κάθε ερώτημα παρουσιάζουμε τον αντίστοιχο ψευδοκώδικα MapReduce:

Q1.

```
map(key, value):
    # key = movies
    # value = details

    for movie in movies:
        values = value.split(',')
        title = values[1]
        year = extractYear(values[3])
        cost = values[5]
        revenue = values[6]
        if((year>=2000) and cost!=0 and revenue!=0):
            name = values[1]
            year = values[3].split('-')[0]
            emit(year, (name, ((revenue-cost)/cost)*100))

reduce(key, values):
    maxProfit = values[0][0] # take first of the list
    maxMovie = values[0][1]
    for tuple in values:
        if (tuple[1] > maxProfit)
            maxMovie = tuple[0]
            maxProfit = tuple[1]
    emit(year, (maxMovie, maxProfit))
```

Q2.

```
# 1. Split comma
# 2. create (userID, (rating, 1))
map(key, value):
    # key = rating
    for line in key:
        userID = line.split(',')[0]
        rating = line.split(',')[2]
        emit(userID, (rating, 1))

# 3. aggregate (userID, totalRatingValue, totalRatings)
reduce(userID, values):
    # key = userID
    # values = list((rating, 1),...)
    totalRatingValue = count = 0
    for tuple in values:
        totalRatingValue += tuple[0]
        totalRatings += tuple[1]
    emit(userID, (souma, count))

# 4. create (1,(1,1)) for "good", (1,(0,0)) for "bad"
# (userID in now considered same for all)
map(key, value):
    # key = userID
    # value = (totalRatingValue, totalRatings)
    if (totalRatingValue/totalRatings)>3):
        emit(1, (1, 1))
    else:
        emit(1, (0, 1))

# 5. aggregate "good"s and "bad"s
reduce(key, list((kalos, 1)...)):
    # key = 1
    # values = list((0,1),(1,1)...) where (1,1) "good" (0,1) "bad"
    goodUsers=0
```

```
    allUsers=0
    for tuple in values:
        goodUsers += tuple[0]
        allUsers += tuple[1]
    emit(key, (goodUsers, allUsers))

# 6. calculate percentage
map(key,value):
    # key = 1
    # value = (goodUsers/allUsers)
    emit(1, goodUsers*100/allUsers)
```

Q3.

```
# A1. Split ratings
# A2. create (movieID, (rating, 1))
map(key, value):
    # key = line number in ratings
    # value = the line
    for line in value:
        movieID = line.split(',')[1]
        rating = line.split(',')[2]
        emit(movieID, (rating, 1))

# A3. aggregate (totalRatingValue, totalNumberOfRatings)
# A4. compute meanRatingPerMovie = totalRatingValue/totalNumberOfRatings
reduce(key, values):
    # key = movieID
    # values = list((rating, 1),...)
    totalRatingValue = 0
    totalNumberOfRatings =0
    for tuple in values:
        totalRatingValue += tuple[0]
        totalNumberOfRatings += tuple[1]
    emit(movieID, (totalRatingValue/totalNumberOfRatings, totalNumberOfRatings))


# B1. Split movies genres
# B2. create (movieID, genre)
map(key, value):
    # key = line number in movies_genre
    # value = the line
    for line in movie_genres:
        movieID = line.split(',')[0]
        genre = line.split(',')[1]
        emit(movieID, genre)

# C1. join meanRatingPerMovie with movieGenres
join(ratings, movie_genres)

# C2. create (genre, (meanRatingPerMovie, 1))
map(key, value):
    # key =  movieID
    # value = ((meanRatingPerMovie, totalNumberOfMovies), genre)
    genre = value[1]
    meanRatingPerMovie = value[0][0]
    emit(genre, (meanRatingPerMovie, 1))

# C3. aggregate (totalRatingPerGenre, totalMoviesPerGenre)
```

```
# C4. create (genre, meanRatingPerGenre =
#      round(totalRatingPerGenre/totalMoviesPerGenre), totalMoviesPerGenre)
reduce(genre, values):
    # value = genre
    # values = list((meanRatingPerMovie, 1))
    totalRatingPerGenre = 0
    totalMoviesPerGenre = 0
    for tuple in values:
        totalRatingPerGenre += tuple[0]
        totalMoviesPerGenre += tuple[1]
    emit(genre, (totalRatingPerGenre/totalMoviesPerGenre, totalMoviesPerGenre))
```

Q4.

```
# A1. split movies
# A2. create (movieID, (timePeriod, countWords))
map(key, value):
    # value line of movies
    for line in value:
        movidID = value[0]
        description = value[2]
        year = findYear(value[3])
        if (year>2000) and value[2]!="":
            if (2000<=year<2005):
                timePeriod="2000-2004"
            elif (2005<=year<2010):
                timePeriod="2005-2010"
            elif (2010<=year<2015):
                timePeriod="2010-2015"
            elif (2015<=year<2020):
                timePeriod="2015-2020"
        emit(movieID, (timePeriod, len(description)))

# B1. split movie_genre
# B2. create (movieID, genre) if drama, else ()
map(key, value):
    # value line of genres
    for line in value:
        movieID=value[0]
        genre = value[1]
        if (genre=="drama"):
            emit(movieID,genre)

# C1. join moviePeriodWords with movieDrama
join(moviesWordsPeriods,genreDrama).on(movieID)

# C2. create (timePeriod, (countWords, 1))
map(key, value):
    # value = lines of movies+genres
    for line in value:
        if isDrama(takeGenre(line)):
            emit(timePeriod, (countWords, 1))

# C3. aggregate (totalWordsPerPeriod, totalMoviesPerPeriod)
# C4. calculate meanDescriptionWordsPerPeriod = totalWordsPerPeriod/totalMoviesPerPeriod
reduce(key, values):
    # key = timePeriod
    # values = list((countWords, 1),...)
    for value in values:
        totalWordsPerPeriod+=value[1][0]
        totalMoviesPerPeriod++
    emit(key, totalWordsPerPeriod/totalMoviesPerPeriod)
```

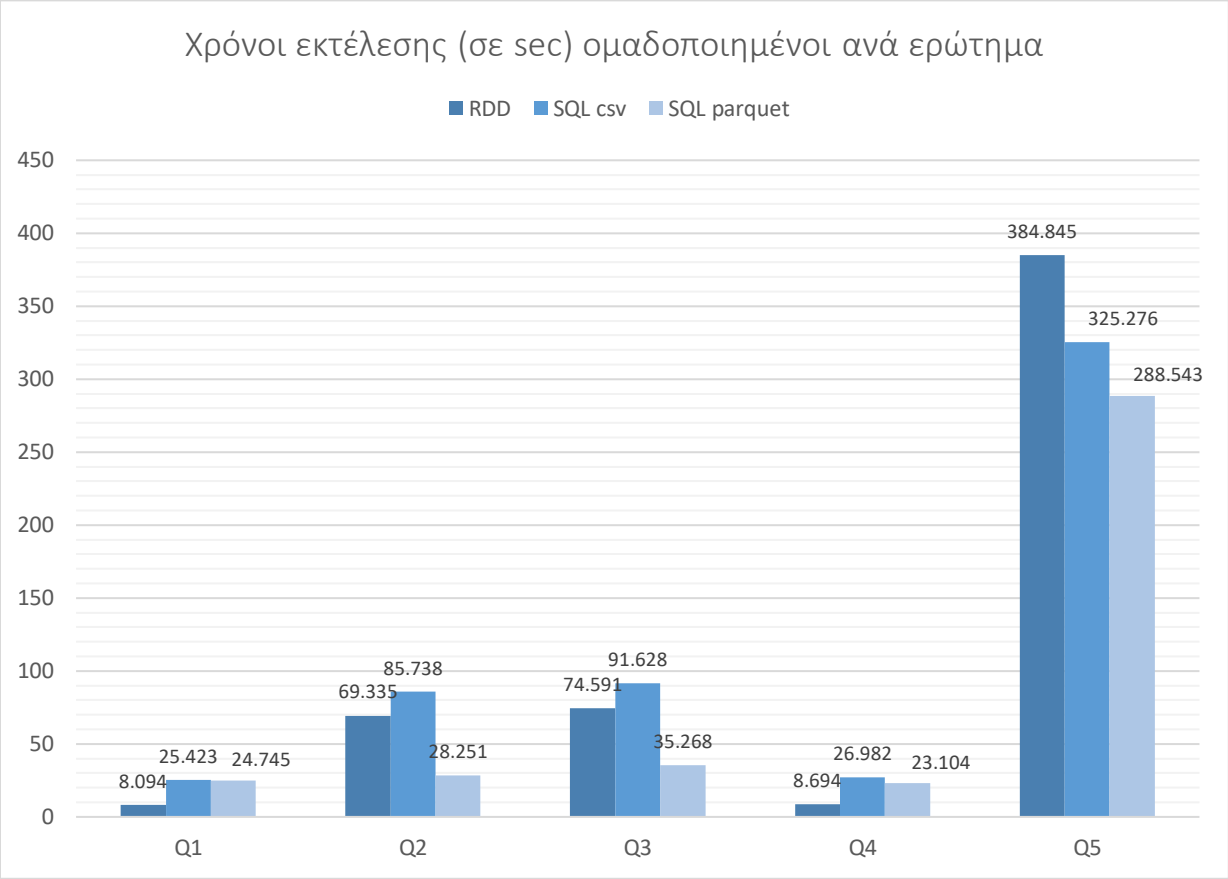
Q5. Αρχικά, εντοπίζεται ο χρήστης με τις περισσότερες κριτικές για κάθε genre. Από το πρώτο MapReduce προκύπτει ένας πίνακας με το id του χρήστη με τα περισσότερα ratings, καθώς και το πλήθος αυτών για κάθε genre. Στην συνέχεια, τα επόμενα reduce με input το αποτέλεσμα του προηγούμενου join παράγουν 2 πίνακες. Ο ένας πίνακας έχει key το genre, και values τη μέγιστη βαθμολογία ταινίας στο συγκεκριμένο genre, το αντίστοιχο popularity και τον τίτλο της ταινίας και ο άλλος έχει key το genre και values την ελάχιστη βαθμολογία ταινίας στο συγκεκριμένο genre, το αντίστοιχο popularity και τον τίτλο της ταινίας. Τέλος, το επιθυμητό αποτέλεσμα προκύπτει εντοπίζοντας σε ποιον χρήστη αντιστοιχεί το ζεύγος των κριτικών, το οποίο γίνεται με join του τελευταίου πίνακα με τον πίνακα C (βλ. σχόλια κάτω).

```
# A1. split ratings
# A2. create (userID, (movieID, rating))
# A3. create (movieID, userID)
map(key, value):
        # value line of rating
        for line in value:
                # userID = value[0]
                # movieID = value[1]
                # rating = value[2]
                emit(movieID, userID)

# ----------------------------

# B1. split movies_genres
# B2. create (movieID, genre)
map(key, value)
        # value line of movie_genres
        for line in value:
                # movieID = x[0]
                # genre = x[1]
                empit(movieID, genre)

# C1. Join A.movieID_userID with B.movieID_genre
join(ratings, movies_genre).on(movieID)

# C2. create (userID, genre)
map(key, value):
        # key = movieID
        emit(value.genre, value.user)

# C3. aggregate userID:genre total ratings
# C5. keep the userID, where totalRatings is max
# C6. create(userID, (genre, totalRatings)) genre is the key
reduce(key, values):
        dict = {}
        for user in values:
                if user in dict:
                        dict(user)++
                else:
                        dict(user)=1
        totalRatings = dict.max()
        userWithMaxTotalRatings = dict.max().value()
        emit(key,(totalRatings, userWithMaxTotalRatings))
# talbe C. (genre, TotalRatingsNumber, userID)

# D1. Join C.userID_genre_MaxTotalRatings with A.movieID_userID_rating
movieID_userID_rating_genre =
        join(ratings, userID_genre_MaxTotalRatings).on(userID)
```

```
# F1. split movies
# F2. create (movieID, (title, popularity))
map(key, value):
        # value line of movies
        for line in value:
                # movieID = value[0]
                # title = value[1]
                # popularity = value[7]
                emit(movieID, (title, popularity))

# I2 create (title, (rating, popularity))
title_rating_popularity =
        join(movies, movieID_userID_rating_genre).on(movieID)


# I3. create (genre max(title, rating, popularity))
# L1. create (genre, (userID, maxTotalRatings))
# maxMovie
reduce(key, values):
        maxRating = values.rating
        maxPopularity = values.popularity
        index = 0
        for i, value in len(values):
                if maxTuple(value.rating, maxRating):
                        maxRating = value.rating
                        maxPopularity = value.popularity
                        index = i
        title = values[i].title
        emit(key, (title, maxRating, maxPopularity))

# I4. create (genre min(title, rating, popularity))
# L1. create (genre, (userID, maxTotalRatings))
# minMovie
reduce(key, values):
        minRating = values.rating
        minPopularity = values.popularity
        index = 0
        for i, value in len(values):
                if minTuple(value.rating, minRating):
                        minRating = value.rating
                        minPopularity = value.popularity
                        index = i
        title = values[i].title
        emit(key, (title, minRating, minPopularity))

# L2. create (genre, ((max_movie_ratings), (min_movie_ratings)))
join(maxMovie, minMovie).on(genre)


# N1. join L1 and L2
# N2. create final tuple (see print below) and sort it
join(max_min_movie_ratings, genre_user_maxTotalRatings).on(genre)
```

Ζητούμενο 4

Για καθένα query παρουσιάζονται οι χρόνοι εκτέλεσης για τις διαφορετικές υλοποιήσεις:



Σύγκριση των RDD API & Spark SQL API

Τα ερωτήματα Q1, Q4 φαίνεται σαφώς να έχουν καλύτερη απόδοση στην υλοποίηση RDD API, γεγονός που πιθανώς να οφείλεται σε μία μη βέλτιστη αντίστοιχη υλοποίηση σε Spark SQL. Αντίθετα τα ερωτήματα Q2, Q3, Q5 δείχνουν να έχουν καλύτερη απόδοση σε Spark SQL, σε αρχεία parquet, και παρόμοιους χρόνους σε αρχεία csv.

Πιο συγκεκριμένα, από την σύγκριση των RDD με των αντίστοιχων SQL με csv input υλοποιήσεων, όπου και οι δύο έχουν csv file ως input, παρατηρείται ότι στα queries στα οποία δεν πραγματοποιείται κανένα join πινάκων (Q1, Q2) αλλά και σε εκείνα που πραγματοποιούνται σχετικά λίγα joins μεταξύ μικρών πινάκων (Q4) ο χρόνος εκτέλεσης είναι μειωμένος στο RDD API.

Ειδικότερα,

- Στο ερωτήματα Q1, Q4 ο χρόνος εκτέλεσης έχει μειωθεί αισθητά. Αυτό οφείλεται στο γεγονός ότι δε γίνονται πολλά joins, οπότε δεν υπάρχει καθυστέρηση στο δίκτυο λόγω της μεταφοράς μικρού όγκου δεδομένων. Επίσης, αυτό συμβαίνει και λόγω της αντιστοίχισης σε πολλά keys οπότε βελτιώνεται η παραλληλοποίηση μέσω Map-Reduce. Πιο αναλυτικά, στο Q1, η εκτέλεση είναι γρήγορη τόσο λόγω της ύπαρξης μόνο μιας διεργασίας Map-Reduce

αλλά και λόγω της απλότητας του ερωτήματος. Στο Q4, το είδος των input files στην Spark SQL, δεν φαίνεται να επηρεάζει σε μεγάλο βαθμό τα αποτελέσματα.

- Στο ερώτημα Q2 φαίνεται ο χρόνος εκτέλεσης να είναι μειωμένος, αλλά η διαφορά δεν είναι πολύ σημαντική. Αυτό γίνεται εξαιτίας των δυο Map-Reduce, και μάλιστα στο δεύτερο η αντιστοίχιση όλων των δεδομένων γίνεται σε ένα key, άρα όλα καταλήγουν στον ίδιο reducer. Ως αποτέλεσμα, το πρώτο Map-Reduce βελτιώνει τον χρόνο, ενώ το δεύτερο στην πράξη δεν εκτελεί παραλληλοποίηση, αλλά αποτελεί bottleneck.

- Στο ερώτημα Q3 ο χρόνος εκτέλεσης είναι συγκρίσιμος καθώς η υλοποίηση Map-Reduce είναι πιο πολύπλοκη.

- Στο ερώτημα Q5 ο χρόνος εκτέλεσης είναι μεγαλύτερος κατά την Map-Reduce εκτέλεση. Αυτό δικαιολογείται από τα πολλά joins και την δέσμευση περισσότερης μνήμης, τα οποία έχουν ως αποτέλεσμα να μεταφέρεται μεγάλος όγκος δεδομένων στο δίκτυο, πράγμα το οποίο δεν είναι αποδοτικό με Map-Reduce. Η αυτόματη βελτιστοποίηση του SQL ερωτήματος στην περίπτωση των Dataframes οδηγεί σε καλύτερη επίδοση.

Εν γένει πάντως, η Spark SQL φαίνεται να είναι καλύτερη, αφού ενσωματώνει έναν βελτιστοποιητή, ο οποίος επιταχύνει τους χρόνους εκτέλεσης των ερωτημάτων. Μάλιστα, αυτός μπορεί και να τροποποιεί κατάλληλα τα ερωτήματα, μέσω τεχνικών φιλτραρίσματος και εύρεσης του καλύτερου δυνατού τρόπου εκτέλεσης των joins. Αντιθέτως, στο RDD API το group και το aggregate επί των δεδομένων είναι «βαριές» υπολογιστικά διεργασίες, διότι παρεμβάλλεται και το στάδιο της μεταφοράς των δεδομένων μέσω του δικτύου, το οποίο πολλές φορές οδηγεί σε χρονική καθυστέρηση, ιδίως για μεγάλα αρχεία.

Σύγκριση csv & parquet

Κατά την εκτέλεση σε Spark SQL παρατηρούμε σημαντικές διαφορές κατά τη χρήση των csv και των parquet files ως input. Τα parquet παρουσιάζουν σημαντικά μειωμένο χρόνο εκτέλεσης, κάτι το οποίο είναι φυσικά αναμενόμενο, δεδομένων των βελτιστοποιήσεων που έγιναν κατά την δημιουργία τους, δηλαδή του μικρότερου αποτυπώματος αυτών των files στην μνήμη και στον δίσκο (με αποτέλεσμα την ταχύτερη εκτέλεση read/write). Αναλυτικότερα, τα metadata που καταγράφονται, παρέχουν δυνατότητες βελτιστοποίησης της επεξεργασίας των ερωτημάτων, αφού το πλήθος των δεδομένων προς σκανάρισμα μειώνεται, οπότε τελικά αυξάνεται η ταχύτητα εκτέλεσής τους. Οι parquet πίνακες είναι αποθηκευμένοι σε ένα columnar format που βελτιστοποιεί το I/O και τη χρήση μνήμης, ενώ διατηρεί την απαραίτητη πληροφορία για το Dataset και ταυτόχρονα μειώνει τον χρόνο εκτέλεσης.

Τεχνική inferSchema

Η τεχνική inferSchema κατά το διάβασμα των αρχείων σε csv format συμβάλλει στην αναγνώριση του είδους κάθε στήλης. Αυτό επιτυγχάνεται με ένα επιπλέον πέρασμα του αρχείου κατά το διάβασμα. Αυτό βέβαια έχεις ως επίπτωση το μεγαλύτερος χρόνος εκτέλεσης. Από την άλλη,

στα parquet δε χρειάζεται να χρησιμοποιηθεί, αφού τα δεδομένα είναι ήδη μορφοποιημένα και βελτιστοποιημένα κατά την δημιουργία του αρχείου. Επομένως η επιλογή του inferSchema αφενός επιβαρύνει την ανάγνωση, διότι απαιτεί επιπλέον σκανάρισμα του input file, ώστε να αναγνωρίσει τον τύπο δεδομένων κάθε column, αφετέρου όμως, προσφέρει ένα Dataframe που έχει σωστά ορισμένους τύπους στα columns. Ενεργοποιώντας το InferSchema ουσιαστικά το API θα διαβάσει κάποιες ενδεικτικές εγγραφές από το αρχείο για να βγάλει συμπεράσματα ως προς το σχήμα.

Μέρος 2ο

Ζητούμενο 1

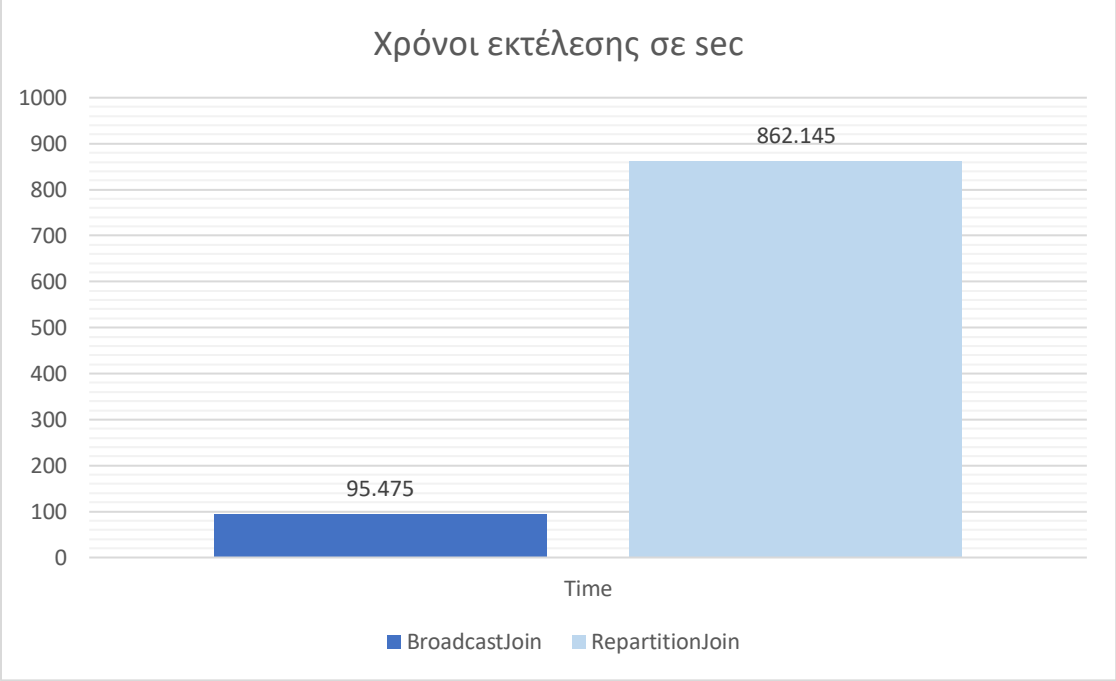
Η υλοποίηση του Broadcast Join στο RDD API βρίσκεται στο αρχείο **broadcastJoin.py**.

Ζητούμενο 2

Η υλοποίηση του Repartition Join στο RDD API βρίσκεται στο αρχείο **repartitionJoin.py**.

Ζητούμενο 3

Καταρχάς δημιουργούμε το αρχείο movie_genre_first100lines.py και εκτελούμε τους απαιτούμενους κώδικες. Παρακάτω φαίνονται οι χρόνοι εκτέλεσης για το repartition join και για το broadcast join:



Παρατηρείται ότι ο χρόνος εκτέλεσης για το repartition join έχει αισθητά μεγαλύτερη καθυστέρηση από το χρόνο εκτέλεσης του broadcast join. Αυτό το γεγονός είναι αναμενόμενο εάν λάβει κανείς υπόψιν ότι το Broadcast Join αποφεύγει την διαρκή και επαναλαμβανόμενη αποστολή μεγάλων πινάκων στο δίκτυο. Στο Repartition Join, από την άλλη, μετά την ολοκλήρωση της διαδικασίας από τον mapper, εκτελείται ένα reduceByKey (ως reducer) και ύστερα ένα flatMap (ως mapper) ώστε να προκύψει το τελικό αποτέλεσμα.

Το Broadcast Join είναι αποδοτικότερο όταν γίνεται join ενός μεγάλου logTable και ενός σχετικά μικρότερου referenceTable. Το Spark στέλνει ένα αντίγραφο του μικρού πίνακα σε όλους τους executor nodes και έτσι, δεν απαιτείται η επικοινωνία many-to-many διότι κάθε τέτοιος κόμβος είναι self-sufficient στο να κάνει join τα records του μεγάλου Dataset με το μικρό broadcasted table.

Το Repartition Join υλοποιεί μια Map-Reduce διεργασία. Στο map, κάθε record λαμβάνει μια ετικέτα που υποδεικνύει την προέλευσή του και εξάγεται το (key, value) pair που γίνεται partitioned, sorted και merged από το framework. Στο reduce, για κάθε key, η συνάρτηση κάνει πρώτα split και τοποθετεί σε buffers τα input records ανάλογα με την ετικέτα τους και εξάγεται κάθε δυνατός συνδυασμός μεταξύ των records των δυο sets.

Εν προκειμένω, η λύση του Broadcast Join είναι κατά πολύ καλύτερη από το Repartition Join διότι όλο το μικρό table μπορεί να αποθηκευτεί στην RAM κάθε worker και να αποφευχθεί η επιβάρυνση του δικτύου χωρίς προβλήματα μνήμης. Επίσης έχουμε μόνο 2 executors. Ωστόσο, εάν αυξηθούν οι executors που απαιτείται να λάβουν ένα αντίγραφο του μικρού table, τότε θα αυξηθεί και το κόστος του broadcasting και μπορεί να ξεπεράσει και το κόστους του ίδιου του join. Ακόμη, λόγω της δέσμευσης μνήμης από τους executors, μπορεί να προκληθεί και out of memory exception με tables μετρίου μεγέθους, αφού το table που αποστέλλεται στους κόμβους μπορεί να μεγαλώσει.

Τέλος, επισημαίνεται ότι η αλυσίδα του RDD είναι πολυπλοκότερη και μακρύτερη και για αυτό το λόγο η διαδικασία χρειάζεται παραπάνω χρόνο, σε συνδυασμό και με το στάδιο της μεταφοράς των δεδομένων στο δίκτυο.

Ζητούμενο 4

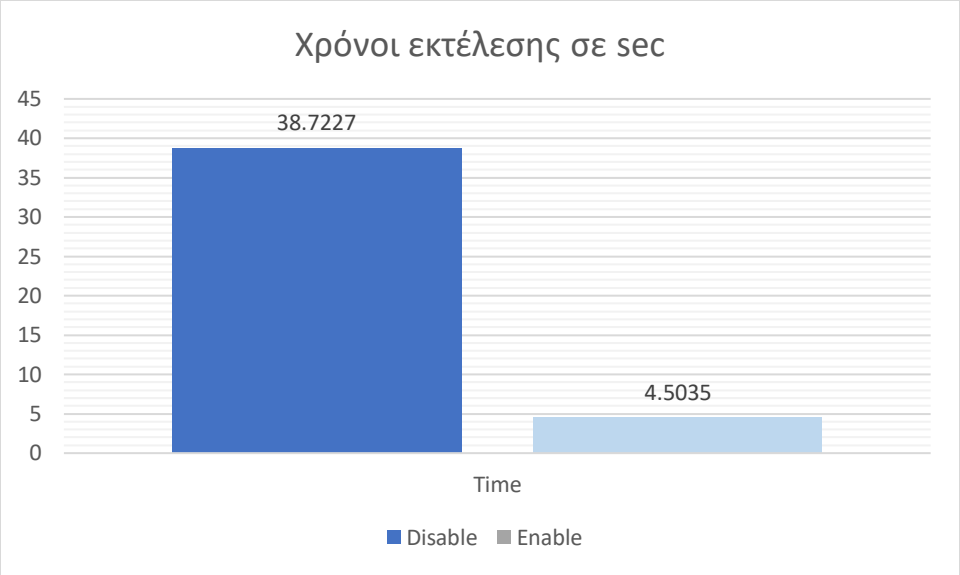
Η τροποποιημένη έκδοση του δοθέντος κώδικα βρίσκεται στο αρχείο **givenCode.py**.

Η πιο σημαντική γραμμή που συμπληρώθηκε στο δοσμένο script είναι η ακόλουθη.

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)
```

Με αυτή την εντολή απενεργοποιείται η βελτιστοποίηση για το broadcast join.

Εκτελούμε το πρόγραμμα δύο φορές, με απενεργοποιημένη και ενεργοποιημένη την βελτιστοποίηση και λαμβάνουμε τους εξής χρόνους εκτέλεσης:



Παρατηρείται ότι ο χρόνος εκτέλεσης είναι μεγαλύτερος όταν δεν χρησιμοποιείται ο βελτιστοποιητής, κάτι το οποίο είναι αναμενόμενο.

Πράγματι, απενεργοποιώντας την δυνατότητα του βελτιστοποιητή να κάνει optimization τα join με BroadcastHash Join, τελικά γίνεται εκτέλεση με SortMerge Join, το οποίο αποτελεί την επόμενη καλύτερη επιλογή. Κατά συνέπεια, η απόδοση είναι χειρότερη και ο χρόνος εκτέλεσης πιο αυξημένος. Αυτό συμβαίνει διότι, από την μια εκτελείται ένα Map-Side Join που αποφεύγει την μεταφορά δεδομένων πάνω από το δίκτυο, ενώ από την άλλη εκτελείται ένα Map-Side Join κατά το οποίο δεν αποφεύγεται μεταφορά δεδομένων πάνω από το δίκτυο. Αυτό συμβαίνει αφού για να ολοκληρωθεί το join θα πρέπει τα ίδια keys από κάθε Dataset να φτάσουν στους ίδιους mappers και να ταξινομηθούν, έτσι ώστε να γίνονται parse παράλληλα και τελικά το join να γίνει στις tuples με τα ίδια keys.

Πάντως πρέπει να τονίσουμε ότι η διαφορά στους χρόνους εκτέλεσης είναι τόσο μεγάλη γιατί ο ένας πίνακας που χρησιμοποιήσαμε, ο movie_genre_first100lines.py, είναι πολύ μικρότερος πό τον άλλο. Το Broadcast Join διαφοροποιείται όταν ένα από αυτούς που εμπλέκονται στο join είναι αισθητά μικρότερος από τον άλλο.