Προχωρημένα Θέματα Βάσεων Δεδομένων – Ακ. Έτος 2021-22

Γεωργούτσος Αθανάσιος (03117151)

Βαλουξής Σπυρίδων (03117003)

Εξαμηνιαία Εργασία

Χρήση του Apache Spark στις Βάσεις Δεδομένων

Περιγραφή

Σε αυτήν την άσκηση, θα κάνουμε χρήση του Apache Spark για τον υπολογισμό αναλυτικών ερωτημάτων πάνω σε αρχεία που περιγράφουν σύνολα δεδομένων, τα οποία αφορούν movies, genres και ratings. Χρησιμοποιείται ένα υποσύνολο μίας εκδοχής του Full MovieLens Dataset, η οποία παρέχεται και στο Kaggle. Για την υλοποίηση των αναλυτικών ερωτημάτων, κάνουμε χρήση δύο APIs, που μας παρέχονται:

- ***RDD API***
- Dataframe API / ***Spark SQL***

Μέρος 1^{ο} : Υπολογισμός αναλυτικών ερωτημάτων με τα APIs του Apache Spark

Ζητούμενο 1) Φόρτωση των 3 CSV αρχείων στο hdfs σε ένα φάκελο ***files***.

Πρώτα, κατεβάζουμε το συμπιεσμένο αρχείο των δεδομένων στον *master* και το αποσυμπιέζουμε.

wget --no-check-certificate 'http://www.cslab.ntua.gr/courses/atds/movie_data.tar.gz' -O movie_data.tar.gz

tar -zxf movie_data.tar.gz

Έπειτα, δημιουργούμε το directory ***files*** στο hdfs.

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

Τέλος, μεταφέρουμε στο ***files*** του hdfs τα αρχεία της εργασίας, που αποσυμπιέσαμε νωρίτερα.

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/.

Ζητούμενο 2) Ο κώδικας που χρησιμοποιήσαμε για τη μετατροπή των αρχείων εισόδου από csv σε parquet format βρίσκεται στον φάκελο ***code/part1***, στο αρχείο ***parquet.py***. Αυτός περιλαμβάνει απλό διάβασμα των δεδομένων από το csv αρχείο και, έπειτα, γράψιμό τους σε parquet format σε νέο αρχείο στο hdfs. Για να εκτελέσουμε τον κώδικα μετατροπής, απλά εκτελούμε την εντολή:

spark-submit parquet.py <csv_file_path> <parquet_file_path>

Ζητούμενο 3) Παρακάτω, ακολουθεί ***ψευδοκώδικας Map-Reduce*** για τα 5 ερωτήματα που κληθήκαμε να υλοποιήσουμε με RDD API και Spark SQL. Οι υλοποιήσεις με τα παραπάνω APIs βρίσκονται στον φάκελο ***code/part1***.

- ***Query 1***: Από το 2000 και μετά, να βρεθεί για κάθε χρονιά η ταινία με το μεγαλύτερο κέρδος, αγνοώντας εγγραφές χωρίς ημερομηνία κυκλοφορίας, ή με μηδενικά έσοδα ή προϋπολογισμό. Το κέρδος υπολογίζεται όπως φαίνεται:

$$\frac{Έσοδα - Κόστος}{Κόστος} \cdot 100$$

map(key, line):

```
# we use the movies.csv file

# line is in typical csv format, and we process it that way

year = line.split( “,” )[3][0:4]

budget = int( line.split( “,” )[5] )

earnings = int( line.split( “,” )[6] )

if ( year != ‘’ and int( year ) > 1999 and budget > 0 and earnings > 0):

        movie = line.split( “,” )[1]

        profit = ((earnings-budget)/budget)*100

        emit( year, ( movie, profit ) )
```

reduce(year, list):

```
# list contains tuples of (movie, profit) for each year

max_profit = 0

max_movie = “None”

for item in list:

        if ( item[1] > max_profit ):
```

```
                max_profit = item[1]
                max_movie = item[0]
    emit( year, max_movie, max_profit )
```

- ***Query 2***: Να βρεθεί το ποσοστό των χρηστών (%) που έχουν δώσει σε ταινίες μέση βαθμολογία μεγαλύτερη από 3.

map(key, line):

```
    # we use the ratings.csv file
    # line is in typical csv format, and we process it that way
    user = line.split( “,” )[0]
    rating = float( line.split( “,” )[2] )
    emit( user, rating )
```

reduce(user, list):

```
    # list contains all the ratings of a user
    number_ratings = 0
    sum_ratings = 0
    for item in list:
        sum_ratings += item
        number_ratings += 1
    avg_rating = sum_ratings / number_ratings
    if ( avg_rating > 3.0 ):
        emit( “same”, “true” )
    else:
```

emit(“same”, “false”)

map(key, value):

emit(key, value)

reduce(key, list):

there is only one key: “same”

list contains booleans: “true” for avg_rating > 3, else “false”

total_users = 0

above3_users = 0

for item in list:

total_users += 1

if (item == “true”):

above3_users += 1

percentage = (above3_users / total_users) * 100

emit(“none“, percentage)

- ***Query 3***: Για κάθε είδος ταινίας, να βρεθεί η μέση βαθμολογία του είδους και το πλήθος των ταινιών που υπάγονται σε αυτό το είδος. Αν μια ταινία αντιστοιχεί σε περισσότερα του ενός είδη, μετριέται σε κάθε είδος. Η μέση βαθμολογία υπολογίζεται όπως φαίνεται:

$$\frac{\sum_{ταινίες}\left(\frac{1}{\#αξιολόγησεων\ ταινίας}\cdot\sum_{αξιολογήσεις} αξιολόγηση\right)}{\#ταινιών\ στην\ κατηγορία}$$

Note!!!: Επιλέγουμε μία map συνάρτηση στην αρχή, που ανάλογα με το key-αρχείο εισόδου (ratings.csv ή movie_genres.csv), κάνει τις ανάλογες ενέργειες. (αντίστοιχα και σε επόμενα queries)

map(file, line):

```
# file: either “ratings.csv” or “movie_genres.csv”
# line is in typical csv format, and we process it that way
if ( file == “ratings.csv”):
        movie_id = line.split( “,” )[1]
        rating = float( line.split( “,” )[2] )
        emit( movie_id, ( “rating”, rating) )
else if ( file == “movie_genres.csv” ):
        movie_id = line.split( “,” )[0]
        genre = line.split( “,” )[1]
        emit( movie_id, ( “genre”, genre ) )
```

reduce(movie_id, list):

```
# list contains tuples, each tuple contains either a genre or a rating for the movie
sum_ratings = 0
number_ratings = 0
genres = [ ]
for item in list:
        if ( item[0] == “genre” ):
                genres.add( item[1] )
        else:
                sum_ratings += item[1]
                number_ratings += 1
avg_rating = sum_ratings / number_ratings
```

```
for genre in genres:
        emit( ( genre, movie_id ), avg_rating )
```

map((genre, movie_id), avg_rating):

```
# avg_rating: average rating for movie with this id
emit( genre, avg_rating )
```

reduce(genre, list):

```
# list contains the average ratings for the movies of this genre
sum_avg_ratings = 0
number_genre_movies = 0
for avg_rating in list:
        sum_avg_ratings += avg_rating
        number_genre_movies += 1
avg_genre_rating = sum_avg_ratings / number_genre_movies
emit( genre, (avg_genre_rating, number_genre_movies) )
```

- ***Query 4***: Για τις ταινίες του είδους “Drama”, να βρεθεί το μέσο μήκος περίληψης ταινίας σε λέξεις ανά 5ετία από το 2000 και μετά (ως και την 5ετία 2014-2019). Αγνοούνται οι ταινίες χωρίς περίληψη.

  map(file, line):

```
# file: either “movies.csv” or “movie_genres.csv”
# line is in typical csv format, and we process it that way
if ( file == “movies.csv”):
        movie_id = line.split( “,” )[0]
        words = len( line.split( “,” )[2] )
        year = line.split( “,” )[3][0:4]
        if ( year != ‘’ and int( year ) > 1999 and words > 0 ):
```

```
                    if ( int( year ) <= 2004 ):
                        emit( movie_id, ( “movie”, “2000-2004”, words ) )
                    else if ( int( year ) <= 2009 ):
                        emit( movie_id, ( “movie”, “2005-2009”, words ) )
                    else if ( int( year ) <= 2014 ):
                        emit( movie_id, ( “movie”, “2010-2014”, words ) )
                    else if ( int( year ) <= 2019 ):
                        emit( movie_id, ( “movie”, “2015-2019”, words ) )
        else if ( file == “movie_genres.csv” ):
            movie_id = line.split( “,” )[0]
            genre = line.split( “,” )[1]
            if ( genre == “Drama” ):
                emit( movie_id, ( “Drama” ) )
```

reduce(movie_id, list):

```
    # list: contains tuple either with half-decade and synopsis lenth, or with the word Drama
    words = 0
    is_drama = false
    for item in list:
        if ( item[0] == “Drama” ):
            is_drama = true
        else if ( item[0] == “movie” ):
            half_decade = item[1]
            words = item[2]
    if ( is_drama == true and words > 0):
        emit( movie_id, ( half_decade, words ) )
```

map(movie_id, (half_decade, words)):

```
    emit( half_decade, words )
```

reduce(half_decade, list):

```
    # list contains synopsis lengths for the Drama movies of this half_decade
    sum_words = 0
    number_movies = 0
    for item in list:
        sum_words += item
        number_movies += 1
    avg_length = sum_words / number_movies
    emit( half_decade, avg_length )
```

- ***Query 5***: Για κάθε είδος ταινίας, να βρεθεί ο χρήστης με τις περισσότερες κριτικές, μαζί με την περισσότερο και τη λιγότερο αγαπημένη του ταινία, σύμφωνα με τις αξιολογήσεις του. Σε περίπτωση ίσης βαθμολογίας σε ταινίες, επιλέγεται η πιο δημοφιλής ταινία εξ αυτών. Παράδειγμα αποτελέσματος:

('Action', 8659, 588, Hulk, 5.0, The Day After Tomorrow, 2.0)

Note!!!: Σε σημεία που φαίνεται με bold, κρατάμε σε csv αρχεία ενδιάμεσα αποτελέσματα, όπου τα χρησιμοποιούμε στη συνέχεια.

map(file, line):

```
# file: either “ratings.csv” or “movie_genres.csv”

# line is in typical csv format, and we process it that way

if ( file == “ratings.csv”):

        user_id = line.split( “,” )[0]

        movie_id = line.split( “,” )[1]

        emit( movie_id, ( “user”, user_id) )

else if ( file == “movie_genres.csv” ):

        movie_id = line.split( “,” )[0]

        genre = line.split( “,” )[1]

        emit( movie_id, ( “genre”, genre ) )
```

reduce(movie_id, list):

```
# list contains tuples, each tuple contains either a genre or a user who rated the movie

genres = [ ]

users = [ ]

for item in list:

        if ( item[0] == “genre” ):

                genres.add( item[1] )
```

```
            else:
                users.add( item[1] )
    avg_rating = sum_ratings / number_ratings
    for genre in genres:
        for user in users:
            emit( ( genre, movie_id, user ), 1 )
```

map((genre, movie_id, user), 1):

```
    emit( ( genre, user ), movie_id )
```

reduce((genre, user), list):

```
    # list: contains the movies that belong to this genre and the user has rated.
    number_ratings = 0
    for movie in list:
        number_ratings += 1
    emit( genre, ( user, number_ratings ) )
```

map(genre, (user, number_ratings)):

```
    emit( genre, ( user, number_ratings ) )
```

reduce(genre, list):

```
    # list: contains tuples, user and how many ratings he has for movies of this genre
    max_user = 0
    max_ratings = 0
    for item in list:
```

```
            if ( item[1] > max_ratings ):
                max_user = item[0]
                max_ratings = item[1]
    write_to_file( “genre_top_user.csv”, ( genre, max_user,  max_ratings ) )
```

map(file, line):

```
    # file: either “ratings.csv” or “movie_genres.csv” or “movies.csv”
    # line is in typical csv format, and we process it that way
    if ( file == “ratings.csv”):
            user_id = line.split( “,” )[0]
            movie_id = line.split( “,” )[1]
            rating = float( line.split( “,” )[2] )
            emit( movie_id, ( “rating”, user_id, rating) )
    else if ( file == “movie_genres.csv” ):
            movie_id = line.split( “,” )[0]
            genre = line.split( “,” )[1]
            emit( movie_id, ( “genre”, genre ) )
    else if ( file == “movies.csv” ):
            movie_id = line.split( “,” )[0]
            name = line.split( “,” )[1]
            popularity = float( line.split( “,” )[7] )
            emit( movie_id, ( “movie”, name, popularity ) )
```

reduce(movie_id, list):

```
    # list: contains either genre, or user with rating, or name and popularity.
    genres = [ ]
```

```
users = [ ]
for item in list:
        if ( item[0] == “genre”):
                genres.add( item[1] )
        else if ( item[0] == “movie” ):
                name = item[1]
                popularity = item[2]
        else if ( item[0] == “rating” ):
                users.add( (item[1], item[2]) )
for genre in genres:
        for ( user, rating ) in users:
                emit( ( genre, movie_id, user ), ( name, popularity, rating ) )
```

map((genre, movie_id, user), (name, popularity, rating)):

```
emit( ( genre, user ), ( name, popularity, rating ) )
```

reduce((genre, user), list):

```
# list: contains the movies of this genre that this user has rated with name, popularity, rating
max_rating = 0
max_name = “none”
max_popularity  = 0
min_rating = 0
min_name = “none”
min_popularity  = 0
for item in list:
        if( item[2] > max_rating or ( item[2] == max_rating and item[1] > max_popularity ) ):
```

max_name = item[0]

max_popularity = item[1]

max_rating = item[2]

if(item[2] < min_rating or (item[2] == min_rating and item[1] < min_popularity)):

min_name = item[0]

min_popularity = item[1]

min_rating = item[2]

write_to_file(“favourite_worst_movies.csv”, (genre, user, max_name, max_rating, min_name, min_rating))

map(file, line):

file: either “favourite_worst_movies.csv” or “genre_top_user.csv”

line is in typical csv format, and we process it that way

if (file == “favourite_worst_movies.csv”):

genre = line.split(“,”)[0]

user_id = line.split(“,”)[1]

favourite_movie = line.split(“,”)[2]

favourite_rating = line.split(“,”)[3]

worst_movie = line.split(“,”)[4]

worst_rating = line.split(“,”)[5]

emit((genre, user_id), (“favourites”, favourite_movie, favourite_rating, worst_movie, worst_rating))

else if (file == “genre_top_user.csv”):

genre = line.split(“,”)[0]

user_id = line.split(“,”)[1]

```
        rating = line.split( “,” )[2]

        emit( ( genre, user_id ), ( “max_rating”, rating ) )
```

reduce((genre, user_id), list):

```
    # list: contains either max rating of genre-user, or favourite-worst movie/rating of user for genre

    is_top_user = false

    max_rating = 0

    for item in list:

        if ( item[0] == “max_rating” ):

            is_top_user = true

            max_rating = item[1]
        else if ( item[0] == “favourites” ):
            top_movie = item[1]
            top_rating = item[2]
            worst_movie = item[3]
            worst_rating = item[4]
    if ( is_top_user == true ):
        emit( genre, ( user_id, max_rating, top_movie, top_rating, worst_movie, worst_rating ) )
```

Ζητούμενο 4) Παρακάτω παρουσιάζονται οι χρόνοι εκτέλεσης των υλοποιήσεων του προηγούμενου ζητούμενου για κάθε query. Τα αποτελέσματα των εκτελέσεων φαίνονται στο φάκελο ***output/part1***. Υπενθυμίζουμε τις 3 περιπτώσεις που εξετάσαμε:

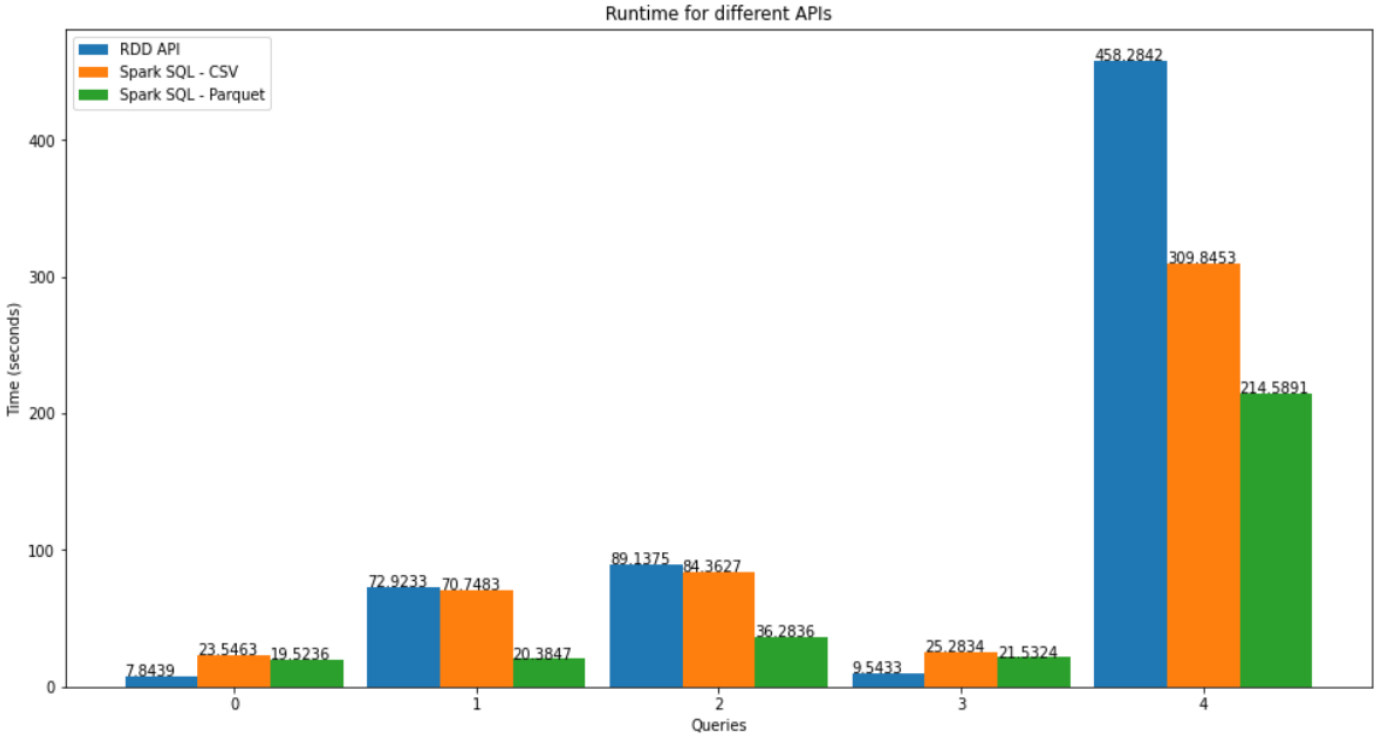
- Map Reduce – RDD API
- Spark SQL με εισόδους csv αρχεία και infer schema
- Spark SQL με εισόδους parquet αρχεία

Οι χρόνοι εκτέλεσης προέκυψαν ως ο μέσος όρος πολλαπλών ξεχωριστών εκτελέσεων, ώστε να εξασφαλίσουμε μεγαλύτερη αξιοπιστία για τα αποτελέσματά μας.

Για το Spark SQL, λαμβάνουμε υπόψη στον χρόνο και το διάβασμα των αρχείων και την καταχώρηση των αντίστοιχων tables. Επίσης, γράφουμε τα αποτελέσματα στο hdfs σε csv αρχεία, καθώς το spark έχει lazy evaluation και σε άλλη περίπτωση οι χρόνοι που προέκυπταν θα ήταν σημαντικά μικρότεροι και ανακριβείς.

Note!!!: Σε κάποιες υλοποιήσεις για αριθμητικά αποτελέσματα, υπάρχουν διαφορές σε κάθε εκτέλεση στα μικρότερα ($> 9^{o}$)δεκαδικά ψηφία. Ο λόγος είναι πως χρησιμοποιούμε διάφορες έτοιμες συναρτήσεις, όπως π.χ. την ***avg*** στο Spark SQL. Καθώς τα αποτελέσματα δεν έχουν σημαντική αλλοίωση, και για λόγους απλότητας, η χρήση αυτών των συναρτήσεων κρίνεται ωφέλιμη.

Στο παρακάτω ραβδόγραμμα φαίνονται οι ζητούμενοι χρόνοι εκτέλεσης, ομαδοποιημένοι ανά ερώτημα.



Παραπάνω, φαίνεται πως ***στα πιο "χρονοβόρα" ερωτήματα, το Spark SQL αποδίδει καλύτερα***, ιδιαίτερα αν χρησιμοποιείται ***με αρχεία εισόδου σε parquet format***. Στα ερωτήματα 1 και 4 (που είναι σαφώς λιγότερο απαιτητικά, μιας και δεν χρησιμοποιείται το σημαντικά μεγαλύτερο εκ των αρχείων εισόδου ratings.csv), παρατηρούμε πως το RDD API δίνει τον καλύτερο χρόνο, με τις άλλες υλοποιήσεις να είναι ελαφρώς πιο αργές. Αυτό συμβαίνει και λόγω των απαιτήσεων των ερωτημάτων και, πιθανώς,

σε ζητήματα υλοποίησης. Συνεπώς, είναι πιο συνετό τα πορίσματά μας να αφορούν τα υπόλοιπα 3 ερωτήματα, όπου οι διαφορές είναι πιο ξεκάθαρες. Γενικά, το Spark SQL αναμέναμε να αποδίδει καλύτερα, λόγω του ενσωματωμένου βελτιστοποιητή που διαθέτει. Για παράδειγμα, υλοποιεί ζητούμενα joins με τον καλύτερο δυνατό τρόπο, ελαχιστοποιώντας τον χρόνο που απαιτείται, όπου σε ερωτήματα όπως το 3 και το 5 οδηγούν και σε σημαντική μείωση του χρόνου εκτέλεσης. Αντιθέτως, το RDD API, έχει ένα σημαντικό overhead μεταφοράς δεδομένων, για λειτουργίες της SQL που γίνονται απλούστερα με το Spark SQL, όπως για παράδειγμα η “GROUP BY”.

- Παρατηρούμε πως όσα διαβάσαμε θεωρητικά σχετικά με το parquet format, ισχύουν και στην πράξη. Τα πλεονεκτήματά του είναι το μικρότερο αποτύπωμα στη μνήμη και τον δίσκο (I/O optimization -> μείωση χρόνου εκτέλεσης) και η διατήρηση στατιστικών επί του dataset, που επιτρέπει να αποφεύγουμε blocks δεδομένων, τα οποία δεν μας είναι αναγκαία για τη λειτουργία που επιτελούμε. Αυτή η τελευταία λειτουργία είναι εκείνη που πιστεύουμε πως οδήγησε κυρίως στον καλύτερο χρόνο για τα “χρονοβόρα” ερωτήματα. Ο όγκος επεξεργασίας πληροφορίας που αποφεύγουμε είναι αξιοσημείωτος, ιδιαίτερα δεδομένου ότι τα αρχικά μας δεδομένα έχουν ιδιαίτερα μεγάλο μέγεθος. ***Συνεπώς, η χρήση του parquet format πραγματικά οδηγεί στις περισσότερες περιπτώσεις, στον καλύτερο χρόνο εκτέλεσης***.

- Το inferSchema, για τις εκτελέσεις του Spark SQL με τα csv αρχεία εισόδου, απαιτεί ένα επιπλέον πέρασμα του αρχείου, ώστε να εξάγει τον τύπο των δεδομένων κάθε στήλης. Η απουσία του inferSchema σημαίνει πως, αρχικά, όλες οι στήλες θεωρούμε πως έχουν δεδομένα τύπου string, και έτσι πρέπει να τα διαχειριστούμε. Συνεπώς, ***το tradeoff εδώ είναι πιο αργή εκτέλεση λόγω του επιπλέον περάσματος με τη σωστή εξαγωγή των τύπων δεδομένων κάθε στήλης***. Σε περίπλοκα αρχεία, για τα οποία δεν έχουμε εξαρχής πληροφορίες, το inferSchema θα μπορούσε να φανεί εξαιρετικά χρήσιμο. Στην περίπτωσή μας, τα οφέλη του inferSchema δεν υπερτερούν του runtime overhead, καθώς γνωρίζαμε εξαρχής τη δομή των csv αρχείων. Θα ήταν πιο αποδοτικό στην περίπτωσή μας να ορίσουμε το Schema του csv αρχείου, καθώς το διαβάζουμε/ επεξεργαζόμαστε. Αυτή η διαδικασία θα ήταν ταχύτερη από το inferSchema και θα οδηγούσε πάλι σε ορθό ορισμό των τύπων δεδομένων των στηλών του csv αρχείου.

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

Ζητούμενο 1) Υλοποίηση του broadcast join στο RDD API (βρίσκεται στο αρχείο ***broadcast_join.py*** στο φάκελο ***code/part2***).

Η ιδέα του ***broadcast join***, σύμφωνα με την δημοσίευση "A Comparison of Join Algorithms for Log Processing in MapReduce", Blanas et al , in Sigmod 2010", περιλαμβάνει μόνο ένα μέρος map. Ξεκινάει με την υπόθεση πως ***ο ένας εκ των δύο πινάκων είναι σημαντικά μικρότερος*** του άλλου. Αυτός ο πίνακας γίνεται ***broadcast σε όλους τους workers, γίνεται partition σύμφωνα με το join key και αποθηκεύεται στο local storage***. Έπειτα, ***δημιουργείται hash table από το partition του μικρότερου ή αρχικοποιούνται hash tables από τον μεγαλύτερο πίνακα***, αν το split του είναι μικρότερο του μικρότερου πίνακα. Τέλος, περνάμε στο κομμάτι της map. Εδώ, γίνεται το ***join κατάλληλα, ανάλογα με ποιο/ποια hash tables δημιουργήθηκαν***. Παραθέτουμε και μια συντόμευση του ψευδοκώδικα της map της δημοσίευσης.

Init():

We want to join the small table R and the significantly larger table L

Retrieve R, partition and store it locally. Then, if R < split of L, create hash table from partition (p chunks) of R, else initialize p hash tables for L.

map(K: null, V: record from split of L):

If there is a hash table from R, then emit matches of the record with the hash table.

Else add record to corresponding hash table of L.

Close():

If there are hash tables of L, then for each of them, for each record of R, emit matches of the record with the hash table.

Ζητούμενο 2) Υλοποίηση του repartition join στο RDD API (βρίσκεται στο αρχείο ***repartition_join.py*** στο φάκελο ***code/part2***).

Η ιδέα του ***standard repartition join***, σύμφωνα με την δημοσίευση "A Comparison of Join Algorithms for Log Processing in MapReduce", Blanas et al , in Sigmod 2010", περιλαμβάνει ένα μέρος map και ένα μέρος reduce. Στο ***map εξετάζουμε records και από τους δύο πίνακες*** που θέλουμε να συνενώσουμε και, ***προσθέτοντας στο value ένα αναγνωριστικό σχετικά με τον πίνακά του***, το κάνουμε emit με το key που θα χρησιμοποιηθεί συνέχεια για τη συνένωση. Έπειτα, στο ***reduce***, λαμβάνουμε λίστα με τα values για ένα join key. ***Εξετάζουμε το αναγνωριστικό κάθε value της λίστας και το τοποθετούμε στον κατάλληλο buffer***, ανάλογα με το πίνακα που άνηκε. Τέλος, ***κάνουμε emit όλα τα ζεύγη των values, όπου το $1^{ο}$ θα ανήκει στον buffer του $1^{ου}$ table και το $2^{ο}$ στου $2^{ου}$ table***. Παραθέτουμε και τον ψευδοκώδικα map-reduce της δημοσίευσης.

map(K: null, V: record from split of either R or L):

```
# R and L are the 2 tables we want to join
join_key <- extracted the join column from V
tagged_record <- add tag of R or L to V
emit( join_key, tagged_record )
```

reduce(K': join key, LIST_V': list of records from R and L):

```
# using buffers Br and Bl for records from R or L respectively
for record t in LIST_V':
        append t to Br or Bl, according to tag
for each pair of records (r,l) in Br X Bl:
        emit( null, new_record(r,l) )
```

Ζητούμενο 3) Χρήση των custom υλοποιήσεων για join 100 γραμμών του movie_genres.csv με το ratings.csv.

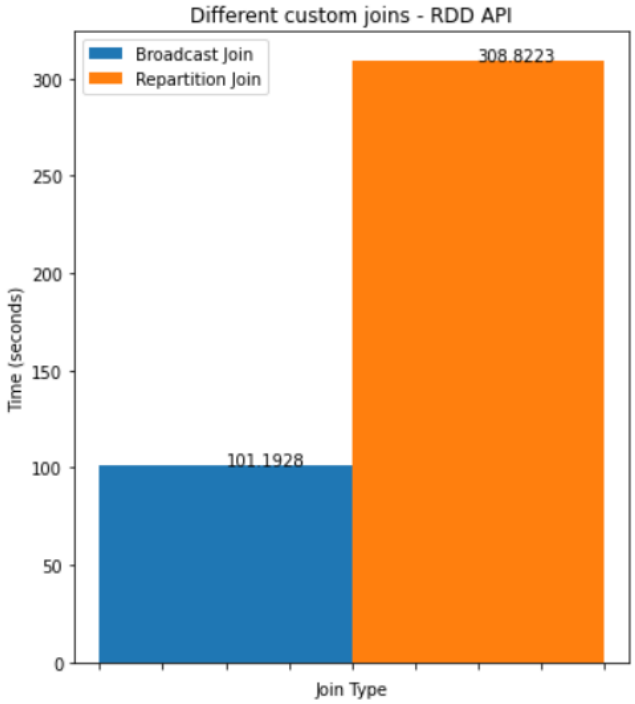
Για να κρατήσουμε μόνο 100 γραμμές από το αρχείο movie_genres.csv, ώστε να δημιουργήσουμε το μικρού μεγέθους table, εκτελούμε την παρακάτω εντολή:

head -100 movie_genres.csv > genres_100_rows.csv

Ύστερα, απλά το μεταφέρουμε στο ***files*** του hdfs:

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

Οι χρόνοι εκτέλεσης των δύο joins φαίνονται παρακάτω:

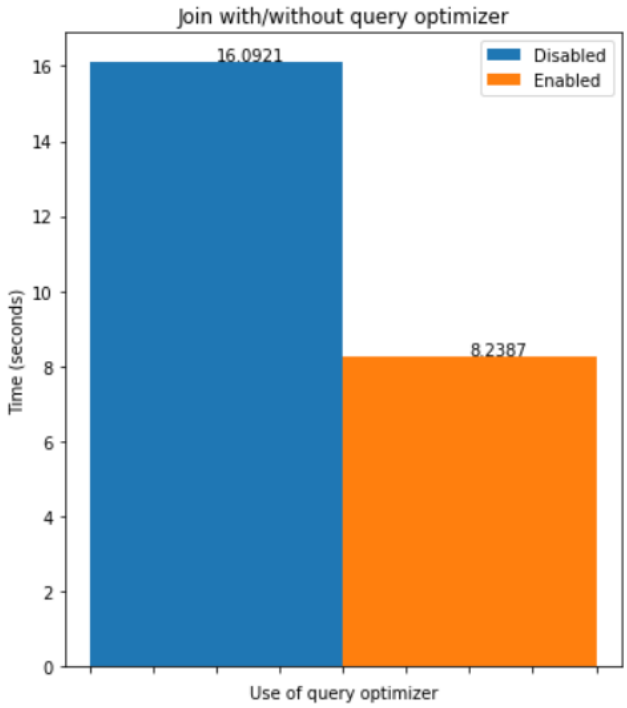


Note!!!: Όπως μπορείτε να δείτε και εσείς, έχουμε υλοποιήσει το ***reduce μέρος του repartition join μας***, χρησιμοποιώντας ένα ***groupByKey()*** και, έπειτα, ένα ***flatMap(…)*** για την παραγωγή των τελικών records του join_table. Αυτό είναι μια σχεδιαστική επιλογή που καθιστά ευκολότερη τη δουλειά μας, στην υλοποίηση του repartition join. Όμως, πρέπει να αναφέρουμε πως η χρήση του ***groupByKey()*** προσθέτει ένα χρονικό overhead, το οποίο οδηγεί στους παραπάνω χρόνους. Φυσικά, αυτό το overhead δεν είναι τόσο σημαντικό που να αναιρεί τις παρατηρήσεις μας που ακολουθούν.

Παρατηρούμε πως ***τα δύο joins παρουσιάζουν σημαντική απόκλιση στον χρόνο εκτέλεσης***. ***Το broadcast join είναι σημαντικά ταχύτερο***, πάνω από 3 φορές, όπως φαίνεται στο παραπάνω διάγραμμα (παίρνοντας υπόψη μας και το overhead της υλοποίησής μας). Αυτό συμφωνεί με αυτό που αναμέναμε από τη θεωρητική ανάλυση των δύο joins. ***Το repartition join, ως Reduce Side Join***, απαιτεί μετά το ***αρχικό mapping***, ένα ***reduce των records*** (στην υλοποίηση μας μέσω της groupBykey()) και την ***παραγωγή των τελικών records του join table*** (στην υλοποίηση μας μέσω της flatMap(…)). Αντιθέτως, ***το broadcast join, ως Map Side Join***, εκμεταλλεύεται το γεγονός πως το ένα table είναι σημαντικά μικρότερο από το άλλο, κάνοντάς το ***broadcast σε όλους τους workers***, η οποίοι το αποθηκεύουν τοπικά. Έπειτα, καθένας εξ αυτών λαμβάνει ένα ***split του μεγαλύτερου πίνακα, και μέσω της Map, παράγει τα τελικά records του join table που του αντιστοιχούν***, έχοντας ήδη τοπικά το μικρότερο table. Οι παραπάνω διαδικασίες που περιλαμβάνει το repartition join, η ανάγκη για μεγαλύτερη μεταφορά δεδομένων πάνω από το δίκτυο και η υλοποίησή μας (με τη χρήση ReduceByKey(…), αναμέναμε μείωση του συνολικού χρόνου, καθώς κάνει combine των δεδομένων στον ίδιο mapper, πριν περάσουμε στο shuffle) καθιστούν το repartition join την χειρότερη επιλογή, για αυτά τα δύο tables.

Ζητούμενο 4) Το SparkSQL χρησιμοποιεί query optimizer, όπου επιλέγει αυτόματα την καλύτερη υλοποίηση για join query, σύμφωνα με τα δεδομένα που δίνονται. Καλούμαστε να εξετάσουμε την εκτέλεση ενός join query με και χωρίς την χρήση του query optimizer (συμπληρώνοντας τον σκελετό κώδικα που μας δόθηκε).

Τα αποτελέσματα φαίνονται παρακάτω:



Παρατηρούμε πως, ***χρησιμοποιώντας τον query optimizer, ο χρόνος εκτέλεσης του join υποδιπλασιάζεται, καθώς επιλέγεται το βέλτιστο πλάνο εκτέλεσης***.

Αρχικά, παρατηρούμε το ***πλάνο εκτέλεσης, όταν ο query optimizer είναι απενεργοποιημένος***. Εδώ, επιλέγεται η χρήση ενός ***SortMerge Join***.

```
== 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>
```

Έπειτα, παρατηρούμε το ***πλάνο εκτελέσης με τον query optimizer ενεργοποιημένο***. Εδώ, επιλέγεται η χρήση ενός ***BroadcastHash Join***.

```
== 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>
```

Συγκρίνοντας τα δύο πλάνα εκτέλεσης, το πρώτο που συμπεραίνουμε εύκολα είναι πως, ***χωρίς τη χρήση βελτιστοποιητή, προστίθενται περισσότερες διαδικασίες επεξεργασίας των δεδομένων και των ενδιάμεσων αποτελεσμάτων***, καθιστώντας αυτόματα την ολοκλήρωση του join πιο χρονοβόρα και περίπλοκη. Ένα εκ των σημαντικότερων παραγόντων καθυστέρησης είναι επίσης η ***ανάγκη για ταξινόμηση των join keys στο SortMerge Join***. Αυτό συμβαίνει ώστε έπειτα να πραγματοποιηθεί σωστά το join ανάμεσα σε records με το ίδιο join key. Ακόμα, υπάρχει και ***καθυστέρηση σχετικά με την κατάλληλη μεταφορά των δεδομένων στους mappers, έτσι ώστε να περιλαμβάνουν records από τα δύο tables με τα κοινά join keys***. Αντιθέτως, ο optimizer αναγνωρίζει πως το ένα εκ των δύο tables (genres) είναι σημαντικά μικρότερο του άλλου, οπότε λογικά επιλέγει το BroadcastHash Join, το οποίο λειτουργεί παρόμοια με το Broadcast Join που αναλύσαμε στα προηγούμενα ζητήματα.