ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

$9^{ο}$ Εξάμηνο – Ροή Λ – Προχωρημένα Θέματα Βάσεων Δεδομένων

Εξαμηνιαία Εργασία – Χρήση του Apache Spark στις Βάσεις Δεδομένων

Η παρακάτω εργασία έγινε με την επιμέλεια των φοιτητών Κριθαρούλα Αναστασίας με αριθμό μητρώου 03117073 και Κριθαρούλα Διονύση με αριθμό μητρώου 03117875.

Αθήνα, Μάρτιος 2022

Πίνακας περιεχομένων

Μέρος 1^{ο}: Υπολογισμός Αναλυτικών Ερωτημάτων με τα API του Apache Spark

Ζητούμενο 1:

Για την φόρτωση των τριών csv αρχείων που μας δόθηκαν (movies.csv, movie_genres.csv, ratings.csv) στο hdfs σε έναν φάκελο files εκτελούμε τις ακόλουθες εντολές:

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

Με την πρώτη από τις παραπάνω τέσσερις εντολές δημιουργούμε το φάκελο *files* στο hdfs και με τις επόμενες τρεις εντολές φορτώνουμε καθένα από τα csv αρχεία (movies.csv, movie_genres.csv και ratings.csv αντίστοιχα) στο φάκελο αυτόν του hdfs.

Ζητούμενο 2:

Στο ζητούμενο αυτό καλούμαστε να μετατρέψουμε καθένα από τα csv αρχεία που υπάρχει στο hdfs σε Parquet μορφή. Για το σκοπό αυτό, διαβάζουμε κάθε csv αρχείο που υπάρχει στο hdfs σε dataframe μέσω της ακόλουθης εντολής:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('csv_to_parquet').getOrCreate()

df = spark.read.format(“csv”). \
     options(header = “false”, InferSchema = “true”) . \
     load(“hdfs://master:9000/files/<file_name>.csv”)
```

και στη συνέχεια αποθηκεύουμε αυτό το dataframe πίσω στο hdfs σε μορφή parquet, μέσω της εντολής:

```
df.write.parquet("hdfs://master:9000/files/<file_name>.parquet")
```

Για τη διαδικασία που μόλις περιγράψαμε έχουμε υλοποιήσει το script *csv_parquet.py*. Μετά την εκτέλεση του script αυτού, βλέπουμε πλέον με την εντολή *hadoop fs –ls hdfs://master:9000/files/.* ότι στο hdfs και συγκεκριμένα στο φάκελο files υπάρχουν έξι αρχεία, τα τρία csv αρχεία (movies.csv, movie_genres.csv, ratings.csv) καθώς και τα τρία αντίστοιχα parquet αρχεία (movies.parquet, movie_genres.parquet, ratings.parquet).

```
user@master:~$ hadoop fs -ls hdfs://master:9000/files/.
Found 6 items
-rw-r--r--   2 user supergroup    1264187 2022-03-04 12:49 hdfs://master:9000/files/movie_genres.csv
drwxr-xr-x   - user supergroup          0 2022-03-04 13:07 hdfs://master:9000/files/movie_genres.parquet
-rw-r--r--   2 user supergroup   17466695 2022-03-04 12:45 hdfs://master:9000/files/movies.csv
drwxr-xr-x   - user supergroup          0 2022-03-04 13:07 hdfs://master:9000/files/movies.parquet
-rw-r--r--   2 user supergroup  709550294 2022-03-04 12:50 hdfs://master:9000/files/ratings.csv
drwxr-xr-x   - user supergroup          0 2022-03-04 13:08 hdfs://master:9000/files/ratings.parquet
```

Ζητούμενο 3:

Στο συγκεκριμένο ζητούμενο υλοποιούμε για καθένα από τα ερωτήματα του ακόλουθου πίνακα:

# Ερωτήματος	Λεκτική Περιγραφή
Q1	Από το 2000 και μετά, να βρεθεί για κάθε χρονιά η ταινία με το μεγαλύτερο κέρδος, Αγνοείστε εγγραφές που δεν έχουν τιμή στην ημερομηνία κυκλοφορίας ή μηδενική τιμή στα έσοδα ή στον προϋπολογισμό.
Q2	Να βρεθεί το ποσοστό των χρηστών (%) που έχουν δώσει σε ταινίες μέση βαθμολογία μεγαλύτερη από 3.
Q3	Για κάθε είδος ταινίας, να βρεθεί η μέση βαθμολογία του είδους και το πλήθος των ταινιών που υπάγονται σε αυτό το είδος. Αν μία ταινία αντιστοιχεί σε περισσότερα του ενός είδη, θεωρούμε ότι μετριέται σε κάθε είδος.
Q4	Για τις ταινίες του είδους "Drama", να βρεθεί το μέσο μήκος περίληψης ταινίας σε λέξεις ανά 5ετία από το 2000 και μετά (1η 5ετία: 2000-2004, 2η 2005-2009, 3η 2010 – 2014, 4η 2015 - 2019). Αγνοείστε ταινίες στις οποίες απουσιάζει η περίληψη.
Q5	Για κάθε είδος ταινίας, να βρεθεί ο χρήστης με τις περισσότερες κριτικές, μαζί με την περισσότερο και την λιγότερο αγαπημένη του ταινία σύμφωνα με τις αξιολογήσεις του. Για την περίπτωση που ο χρήστης έχει την ίδια ψηλότερη / χαμηλότερη βαθμολογία σε περισσότερες από 1 ταινίες επιλέξτε την πιο δημοφιλή ταινία από αυτές της κατηγορίας που συμπίπτει η βέλτιστη / χείριστη βαθμολογία του χρήστη. Τα αποτελέσματα να είναι σε αλφαβητική σειρά ως προς την κατηγορία ταινίας και να παρουσιαστούν σε ένα πίνακα με τις ακόλουθες στήλες. • είδος • χρήστης με περισσότερες κριτικές • πλήθος κριτικών, • περισσότερο αγαπημένη ταινία • βαθμολογία περισσότερο αγαπημένης ταινίας • λιγότερο αγαπημένη ταινία • βαθμολογία λιγότερο αγαπημένης ταινίας

δύο λύσεις, μια λύση με το RDD API και μια με το Spark SQL. Για κάθε λύση κατασκευάζουμε και ένα script, οπότε τελικά δημιουργούμε δέκα scripts, πέντε της μορφής *rdd_query<i>.py* και πέντε *sparksql-query<i>.py*, με i = 1, 2, 3, 4, 5 για καθένα από τα πέντε ερωτήματα. Μάλιστα, η λύση σε Spark SQL μπορεί να διαβάζει είτε csv αρχεία χρησιμοποιώντας το option InferSchema, είτε αρχεία parquet. Ο τύπος των αρχείων που θα διαβαστούν κάθε φορά δίνεται ως παράμετρος κατά την εκτέλεση του εκάστοτε script (π.χ. *python sparksql-query<i>.py csv*). Πιο αναλυτικά, δίνοντας ως παράμετρο τη συμβολοσειρά "csv" σημαίνει ότι τα αρχεία που πρόκειται να διαβαστούν από το εκάστοτε script θα είναι σε csv μορφή ενώ δίνοντας ως παράμετρο τη συμβολοσειρά "parquet" σημαίνει ότι τα αρχεία που πρόκειται να διαβαστούν είναι σε parquet μορφή.

Όσον αφορά τις λύσεις με RDD API, παραθέτουμε ψευδοκώδικα σε Map-Reduce, ο οποίος περιγράφει τις υλοποιήσεις μας για καθένα από τα ερωτήματα του παραπάνω πίνακα.

rdd_query1

Map(key, value):

//key: None

//value: each row of movies.csv in string format

list_values = value.split(‘,’)

title = list_values[1]

year = list_values[3].split(‘-’)[0]

cost = int(list_values[5])

income = int(list_values[6])

if year != ‘’ and cost != 0 and income != 0 and int(year) > 1999:

profit = $100 \cdot \frac{\text{income}-\text{cost}}{\text{income}}$

emit(year, (title, profit)**)**

Reduce(key, value):

//key: year

//value: list of tuples (title, profit) correspond to same key: year

max_profit = $-\infty$

max_movie_title = “”

for item in value:

If item[1] > max_profit:

max_profit = item[1]

max_movie_value = item[0]

emit(key, (max_movie_title, max_profit)**)**

rdd_query2

Map(key, value):

```
//key: None
//value: each row of ratings.csv in string format
list_values = value.split(‘,’)
user_id  = list_values[0]
rating = list_values[2]
emit(user_id, (rating, 1))
```

Reduce(key, value):

```
//key: user_id
//value: list of tuples (rating, 1) correspond to same key: user_id
sum = 0
count =  0
for item in value:
        sum = sum + item[0]
        count = count + item[1]
mean_rating =  sum/count
if mean_rating > 3:
        emit(null, ‘higher than 3’)
else:
        emit(null, ‘lower than 3’)
```

Map(key, value):

```
//key: null
//value: ‘higher than 3’ or ‘lower than 3’
emit(key, value)
```

Reduce(key, value):

```
//key: null
//value: list of strings ‘higher than 3’ or ‘lower than 3’
all_users =  len(value)
correct_users = len([item for item in value if item == ‘higher than 3’])
percentage =  correct_users /all_users
emit(‘percentage’, percentage)
```

rdd_query3

Map(key, value):

```
//key: file type (ratings.csv or movie_genres.csv)
//value: each row of file
list_values = value.split(',')
if key == 'ratings.csv':
        movie_id = list_values[1]
        movie_rating = list_values[2]
        emit(movie_id, (movie_rating, 'ratings'))
else:
        movie_id = list_values[0]
        category = list_values[1]
        emit(movie_id, (category, 'genres'))
```

Reduce(key, value):

```
sum = 0 ; count =  0 ; list_of_categories = []
for item in value:
        If item[1] == 'ratings': //tuple came from ratings dataframe
                sum = sum + item[0]
                count = count + 1
        else:  //tuple came from movie_genres dataframe
                list_of_categories.append(item[0])
mean_movie_rating = sum/count
for item in list_of_categories:
                emit(item, mean_movie_rating)
```

Map(key, value):

```
emit(key, value)
```

Reduce(key, value):

```
sum = 0 ; count = 0
for item in value:
        sum = sum + item ; count = count + 1
mean_rating_of_genre = sum / count
emit(key, mean_rating_of_genre, count)
```

rdd_query4 -help functions

```
def find_years(x):
        if x < 2005:
                return ‘2000-2004’
        elif 2005 <= x <= 2009:
                return ‘2005-2009’
        elif 2010 <= x <= 2014:
                return ‘2010-2014’
        elif 2015 <= x <= 2019:
                return ‘2015-2019‘
        else:
                print(‘it can ‘t be happening… ’)
                return 0

def keep_only_words(input_list):
        output_list = [re.sub( r'[^A-Za-z] + ',  ' ' ,  item)
                        for item in input_list if item != ""]

        output_list = [item.replace(" ", "") for item in output_list

                        if item.replace(" ", "") !=""]

        return output_list
```

rdd_query4

Map(key, value):

```
//key: file type (movies.csv or movie_genres.csv)
//value: each row of file
list_values = value.split(‘,’)
if key == ‘movies.csv’:
        movie_id = list_values[0]
        summary_list = keep_only_words(list_values[2].split(‘,’))
        year = list_values[3].split(‘-’)[0]
        pentaetia = find_years(year)
        if summary_list != []:
                emit(movie_id, (summary_list, pentaetia))
else:
        movie_id = list_values[0]
        category = list_values[1]
        if category == ‘Drama’:
                emit(movie_id, category)
```

Reduce(key, value):

```
drama_flag = False
for item in value:
        if item == ‘Drama’:
                drama_flag = =True ; break

if drama_flag:  //only movies belong to Drama category
        for item in value:
                if item.is_tuple: //skip category value,
                                        //keep only (summary_list, penaetia)
                        summary_list = item[0]
                        pentaetia = item[1]
                        emit(pentaetia, len(summary_list))
```

rdd_query4

Map(key, value):

```
    emit(key, value)
```

Reduce(key, value):

```
    sum = 0
    count = 0
    for item in value:
        sum = sum + item
        count = count + 1
    mean_summary = sum / count
    emit(key, mean_summary)
```

rdd_query5

Map(key, value):

```
//key: file type (movies.csv or movie_genres.csv or ratings.csv)
//value: each row of file
list_values = value.split(',')
if key == 'ratings.csv':
        user_id = list_values[0]
        movie_id = list_values[1]
        rating = list_values[2]
        emit(movie_id, (user_id, rating, 'R'))
elif key == 'movie_genres.csv':
        movie_id = list_values[0]
        category = list_values[1]
        emit(movie_id, (category, 'G'))
else:
        movie_id = list_values[0]
        popularity = list_values[7]
        emit(movie_id, (popularity, 'M'))
```

query5

Reduce(key, value):

```
list_of_categories = [] ; popularity = 0
users_rating = dict()  //dictionary with key: user_id and
                       value: [max_rating, min_rating, count_ratings]
                       for specific movie_id
for item in value:
    if item[-1] == 'R': //value from ratings.csv
        user_id = item[0]
        rating = item[1]
        if user_id not in users_rating:
            users_rating[user_id] = [rating, rating, 1]
        else:
            max_rating = max(users_rating[user_id][0], rating)
            min_rating = min(users_rating[user_id][0], rating)
            count_ratings = user_rating[user_id][2] + 1
            users_rating[user_id] = [max_rating, min_rating,
                                     count_ratings]
    elif item[-1] == 'G':  //value from movie_genres.csv
        category = item[0]
        list_of_categories.append(category)
    else:  //value from movies.csv – only one time I will be here per
                                                        movie_id
        popularity = item[0]
    for category in list_of_categories:
        for user_id, list_rating in users_ratings.items():
            max_rating = list_rating[0]
            min_rating = list_rating[1]
            count_rating = list_rating[2]
            emit(category, (user_id, max_rating, movie_id,
            min_rating, movie_id, count_rating, popularity))
```

query5

Map(key, value):

emit(key, value**)**

Reduce(key, value):

```
dictionary = dict()
for item in value:
        user_id = item[0]
        if user_id not in dictionary:
                dictionary[user_id] = item[1:6] //without popularity
        else:
                dictionary[user_id][5] += item[5] //number of ratings per user
                if dictionary[user_id][1] < item[1] or (dictionary[user_id][1] ==
                item[1] and dictionary[user_id][6] < item[6]):
                        dictionary[user_id][1]= item[1]
                        dictionary[user_id][2] = item[2]

                if dictionary[user_id][3] > item[3] or (dictionary[user_id][3] ==
                item[3] and dictionary[user_id][6] < item[6]):
                        dictionary[user_id][3]= item[3]
                        dictionary[user_id][4] = item[4]
user_with_max_cnt = 0
for item in dictionary.items():
        user_id = item[0]
        number_of_ratings = item[4]
        if number_of_ratings > max_cnt:
                user_with_max_cnt = user_id
number_of_ratings = dictionary[user_with_max_cnt][4]
max_rating = dictionary[user_with_max_cnt][0]
min_rating = dictionary[user_with_max_cnt][1]
min_movie_title = dictionary[user_with_max_cnt][3]
```

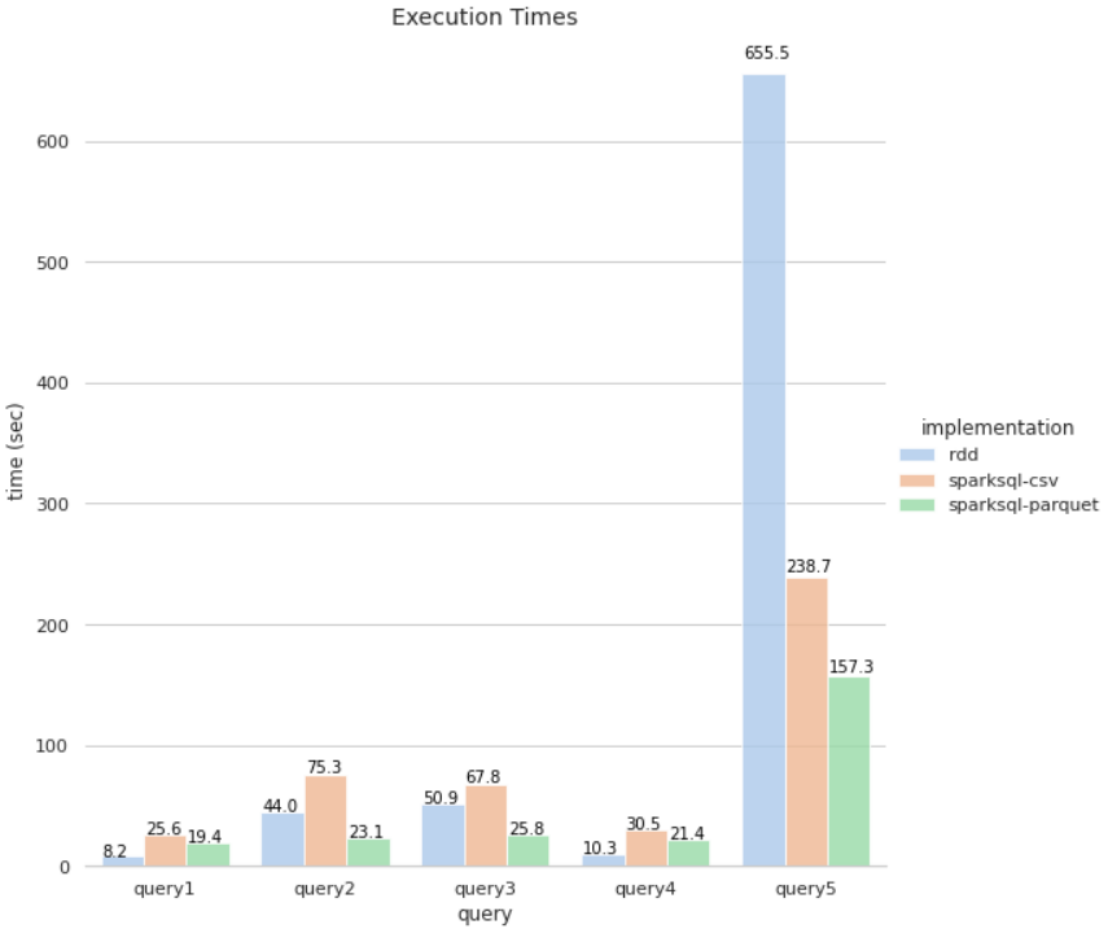
emit(key, (user_with_max_cnt, number_of_ratings, max_rating, min_rating, min_movie_title)**)**

Παρατηρήσεις:

- Στο query Q4 χρησιμοποιούμε τη βοηθητική συνάρτηση *keep_only_words*, η οποία φιλτράρει τα κείμενα όλων των περιλήψεων κρατώντας μόνο τις λέξεις αυτών, ενώ αφαιρεί όλα τα σημεία στίξης, ειδικούς χαρακτήρες καθώς και αριθμούς.
- Όσον αφορά το query Q5, παρατηρούμε ότι για τη κατηγορία History υπάρχουν δύο χρήστες με ίσο μέγιστο αριθμό κριτικών για ταινίες της κατηγορίας αυτής. Για το λόγο αυτό φροντίζουμε ώστε η υλοποίησή μας να εμφανίζει τελικά και τις δύο αυτές εγγραφές.

Ζητούμενο 4:

Αφού εκτελέσουμε όλες τις υλοποιήσεις (map reduce – rdd api, spark-sql με είσοδο csv αρχείο και spark-sql με είσοδο parquet αρχείο) για καθένα από τα πέντε ερωτήματα (query) που παρουσιάσαμε παραπάνω, αποθηκεύουμε τις εξόδους αυτών στο hdfs σε αντίστοιχα αρχεία στο φάκελο outputs (επισυνάπτεται στο παραδοτέο). Παράλληλα, παρουσιάζουμε τους χρόνους εκτέλεσης των υλοποιήσεων αυτών στο ακόλουθο διάγραμμα (στο χρόνο εκτέλεσης συμπεριλαμβάνουμε τόσο το χρόνο που απαιτείται για το διάβασμα των εκάστοτε αρχείων από το hdfs όσο και το χρόνο που απαιτείται για την αποθήκευση των αρχείων εξόδου σε αυτό).



Σύμφωνα με το παραπάνω ραβδόγραμμα έχουμε τις εξής παρατηρήσεις:

Αρχικά παρατηρούμε ότι για τα queries Q1 και Q4 την καλύτερη επίδοση εμφανίζει η υλοποίηση σε RDD API. Το γεγονός αυτό οφείλεται στο ότι και τα δύο αυτά ερωτήματα αποτελούν απλές περιπτώσεις στις οποίες οι πίνακες που επεξεργαζόμαστε έχουν σχετικά μικρό όγκο δεδομένων. Στο query Q1 χρησιμοποιείται μόνο ο πίνακας movies.csv (17MB) ενώ στο query Q4 οι πίνακες movies.csv (17MB) και movie_genres.csv (1.3MB). Επιπλέον η εκτέλεση των ερωτημάτων αυτών είναι σχετικά απλή (στο Query Q1 επεξεργαζόμαστε μόνο τον πίνακα από το αρχείο movies.csv ενώ στο query Q4 εκτελούμε ένα join μεταξύ των πινάκων των αρχείων movies.csv και movie_genres.csv) γεγονός που φαίνεται και από τους χρόνους εκτέλεσης οι οποίοι είναι μικρότεροι σε σχέση με τα υπόλοιπα queries αφού και ο όγκος δεδομένων είναι αρκετά μικρότερος. Αντίθετα στο query Q2 όπου χρησιμοποιείται ο μεγάλος πίνακας από το αρχείο ratings.csv (677 MB)

βλέπουμε ότι η υλοποίηση σε RDD API χάνει την «πρωτιά» και αποκτά χειρότερη επίδοση συγκριτικά με την υλοποίηση σε SparkSQL με χρήση όμως parquet αρχείων. Παρόμοια αποτελέσματα παρατηρούνται και στο query Q3 όπου και πάλι χρησιμοποιείται ο πίνακας από το αρχείο ratings.csv. Μάλιστα στο query Q3 ενώ η υλοποίηση σε SparkSQL με csv αρχεία έχει μικρότερο χρόνο εκτέλεσης σε σχέση με την αντίστοιχη υλοποίηση στο ερώτημα Q2 και η υλοποίηση σε SparkSQL με parquet αρχεία έχει ελάχιστα μεγαλύτερο χρόνο εκτέλεσης , ο χρόνος εκτέλεσης της υλοποίησης σε RDD API αυξάνεται σε σχέση με το Query Q2 (θυμίζουμε ότι στο query Q3 πραγματοποιείται join μεταξύ των πινάκων από τα αρχεία ratings.csv και movie_genres.csv). Από τα παραπάνω λοιπόν μπορούμε να συμπεράνουμε πως η υλοποίηση με RDD API είναι καλύτερη σε πιο απλές περιπτώσεις όπου ο όγκος δεδομένων είναι αρκετά μικρός αλλά και η επεξεργασία αυτών σχετικά απλή. Σε αυτές τις περιπτώσεις προσφέρει προγραμματιστική ευκολία και καλύτερη επίδοση. Αντίθετα όσο αυξάνεται ο όγκος των δεδομένων αλλά και η επεξεργασία γίνεται πιο σύνθετη η υλοποίηση σε SparkSQl με parquet αρχεία παρουσιάζει μία σταθερότητα στον χρόνο εκτέλεσης. Η υλοποίηση σε SparkSQL με csv αρχεία έχει μειονεκτήματα τα οποία θα σχολιάσουμε στην συνέχεια. Ο λόγος που κάτι τέτοιο παρατηρείται είναι το γεγονός ότι η υλοποίηση με SparkSQL παρέχει και βελτιστοποιητή ο οποίος αναλαμβάνει να εκτέλεση τα queries με τον πιο αποδοτικό τρόπο επιλέγοντας τον καλύτερο τρόπο εκτέλεσης των join ανά περίπτωση αλλά και αναδιατάσσοντας τους τελεστές προκειμένου το query να έχει καλύτερη απόδοση. Αντίθετα στο RDD API ο χρήστης έχει μεγαλύτερο μερίδιο ευθύνης αναλαμβάνοντας ο ίδιος να υποδείξει την σειρά εκτέλεσης των ενεργειών αλλά και να βελτιστοποιήσει την απόδοση του προγράμματος του χρησιμοποιώντας για παράδειγμα ενέργειες όπως φιλτράρισμα αλλά και άλλα. Επαλήθευση των παραπάνω συμπερασμάτων αποτελεί το query Q5 στο οποίο ο χρόνος εκτέλεσης με RDD API γίνεται πολύ μεγαλύτερος σε σχέση με την υλοποίηση σε SparkSQL τόσο σε csv όσο και σε parquet αρχεία. Στο query αυτό πραγματοποιείται τριπλό join μεταξύ και των τριών πινάκων (και διάφορα άλλα join μεταξύ των πινάκων) και σαφώς ο όγκος δεδομένων αλλά και η πολυπλοκότητα του ερωτήματος είναι πολύ μεγαλύτερα σε σχέση με τα υπόλοιπα ερωτήματα. Φαίνεται λοιπόν ξεκάθαρα ότι σε τέτοιου είδους ερωτήματα όπου ο όγκος είναι πολύ

μεγάλος και η επεξεργασία περίπλοκη η χρήση RDD API δίνει χειρότερα αποτελέσματα όσο αναφορά τον χρόνο εκτέλεσης.

Όσον αφορά την σύγκριση μεταξύ parquet και csv παρατηρούμε ότι η χρήση parquet αρχείων εμφανίζει σαφώς καλύτερη απόδοση σε όλα τα ερωτήματα. Μάλιστα στα πρώτα τέσσερα ερωτήματα παρά τα πλεονεκτήματα που αναφέραμε για την χρήση SparkSQL, η χρήση csv αρχείων οδηγεί την SparkSQL να έχει χειρότερη επίδοση συγκριτικά με αυτή με RDD API. Κάτι τέτοιο δείχνει τη σημασία που έχει ο τρόπος αποθήκευσης των αρχείων. Αντίθετα στο query Q5 όπου όπως αναφέραμε έχουμε αρκετά μεγάλο όγκο δεδομένων και πολυπλοκότητα στην επεξεργασία η SparkSQL ακόμα και με χρήση csv αρχείων παρουσιάζει καλύτερα αποτελέσματα από το RDD API. Ο λόγος της καλύτερης επίδοσης που εμφανίζει η χρήση parquet αρχείων συγκριτικά με τη χρήση csv αρχείων είναι αρχικά ο τρόπος αποθήκευσης των parquet αρχείων, όπως αυτός περιγράφεται και στην εκφώνηση της εργασίας (μικρότερο αποτύπωμα στην μνήμη και στον δίσκο που βελτιστοποιεί το I/O και άρα μείωση του χρόνου εκτέλεσης, διατήρηση επιπλέον πληροφορίας όπως στατιστικά πάνω στο Dataset τα οποία βοηθούν στην πιο αποτελεσματική επεξεργασία του). Μάλιστα καθώς πολλές φορές στα ερωτήματα που καλούμαστε να υλοποιήσουμε αναζητούμε τη μέγιστη, τη μέση τιμή κ.ο.κ., η χρήση στατιστικών πληροφοριών (μέγιστη τιμή ενός block, μέση τιμή κ.ο.κ.) παίζουν σημαντικό ρόλο στην βελτίωση της απόδοσης. Ένας επιπλέον λόγος της καλύτερης απόδοσης των parquet έναντι των csv είναι πως στα csv χρησιμοποιούμε το inferSchema (θέτουμε inferSchema = True) προκειμένου να αναγνωριστεί το σχήμα, δηλαδή ο τύπος της κάθε στήλης των δεδομένων. Κάτι τέτοιο για να πραγματοποιηθεί χρειάζεται να «περαστούν» τα δεδομένα μία επιπλέον φορά, γεγονός που προκαλεί σημαντικό overhead στην επίδοση, το οποίο μάλιστα μεγαλώνει όσο μεγαλύτερο είναι και το μέγεθος των εκάστοτε αρχείων (π.χ. ratings.csv). Αντίθετα, με την χρήση parquet αρχείων κάτι τέτοιο δεν απαιτείται καθώς η εξαγωγή των τύπων γίνεται αυτόματα κατά την δημιουργία του αρχείου και τα δεδομένα αυτά κρατούνται στο parquet αρχείο ως μεταδεδομένα του.

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

Ζητούμενα 1,2:

Στο ζητούμενο αυτό καλούμαστε να υλοποιήσουμε τόσο το broadcast όσο και το repartition join στο RDD API (Map Reduce).

Broadcast Join

Σε πολλές εφαρμογές join μεταξύ δύο πινάκων L και R, ο ένας εκ των δύο πινάκων είναι αρκετά μικρότερος από τον άλλο, έστω |R| << |L|, οπότε αντί να μετακινούμε και τους δύο πίνακες L, R διαμέσου του δικτύου κατά τη διαδικασία του shuffle and sort, μπορούμε απλά να κάνουμε broadcast τον μικρότερο πίνακα R. Με τον τρόπο αυτό αποφεύγουμε την ταξινόμηση και στους δύο πίνακες και κυρίως αποφεύγουμε το overhead που προκαλείται στο δίκτυο μετακινώντας το μεγαλύτερο πίνακα L, καθώς με τον τρόπο αυτό το broadcast join υλοποιείται πλέον μόνο με map διεργασίες. Αρχικά, τροποποιούμε τα records καθενός από τους πίνακες L και R έτσι ώστε να έχουν ως key το κοινό join key και στη συνέχεια δημιουργούμε ένα hash table για τα record του πίνακα R με κοινό key, το οποίο και κάνουμε broadcast. Στη συνέχεια κάθε map διεργασία, για κάθε record του πίνακα L αναζητά τοπικά στο hash table (που είναι πλέον αποθηκευμένο στο local file system), τα records του πίνακα R με το ίδιο key και συνενώνει καθένα από αυτά με το τρέχον record του R.

Repartition Join

Το βασικό Repartition Join μεταξύ δύο πινάκων L και R, υλοποιείται με ένα MapReduce job. Στη φάση του Map, κάθε map task εργάζεται πάνω σε ένα κομμάτι του L ή του R. Προκειμένου, να αναγνωρίζουμε από ποιον πίνακα προέρχεται κάθε

record, κάθε map διεργασία «ταγκάρει» το record με τον πίνακα από τον οποίο προέρχεται, και δίνει ως έξοδο το join key που έχει εξάγει από το συγκεκριμένο record, μαζί με το υπόλοιπο ταγκαρισμένο record. Στη συνέχεια οι έξοδοι των map διεργασιών χωρίζονται, ταξινομούνται και κατανέμονται στους reducers μέσω του framework. Όλα τα records για ένα συγκεκριμένο join key συγκεντρώνονται μαζί και τελικά καταλήγουν στον ίδιο reducer. Για κάθε join key η συνάρτηση reduce αρχικά χωρίζει και αποθηκεύει τα records σε δύο διαφορετικούς buffers σύμφωνα με το tag τους (τον πίνακα από τον οποίο προέρχονται) και στη συνέχεια εκτελεί καρτεσιανό γινόμενο μεταξύ των records αυτών των buffers πάνω στο join key.

Οι κώδικες υλοποίησης των δύο παραπάνω joins βρίσκονται στα scripts rdd_repartitionjoin.py , rdd_broadcastjoin.py αντίστοιχα (<type_of_join>.py). Κατά την εκτέλεση του εκάστοτε script δίνονται ως παράμετροι τα δύο αρχεία στα οποία επιθυμούμε να πραγματοποιήσουμε συνένωση μαζί με τον δείκτη index της κολώνας η οποία αποτελεί το κλειδί της συνένωσης (join key) για το κάθε αρχείο. Έτσι έχουμε:

spark-submit <type of join>.py <path to first file> <index of join key of first file> <path to second file> <index of join key of second file>

Σχόλιο : *Στην υλοποίηση του broadcast join θεωρούμε ότι ως πρώτο αρχείο θα δοθεί το μικρότερο (αυτό που έχει τις λιγότερες εγγραφές) στο οποίο θα εκτελεστεί τη συνέχεια και το broadcast.*

Ζητούμενο 3:

Αφού υλοποιήσουμε τα παραπάνω δύο είδη join στην συνέχεια απομονώνουμε τις 100 πρώτες εγγραφές του αρχείου movie_genres.csv. Τις εγγραφές αυτές τις αποθηκεύουμε σε ένα νέο αρχείο με όνομα movie_genres_100.csv το οποίο

φορτώνουμε στο hdfs στο φάκελο files στο οποίο βρίσκονται ήδη φορτωμένα τα υπόλοιπα αρχεία με τα δεδομένα μας. Η παραπάνω διαδικασία υλοποιείται με το script part2_3.py, ο κώδικας του οποίου φαίνεται στην συνέχεια:

part2_3.py

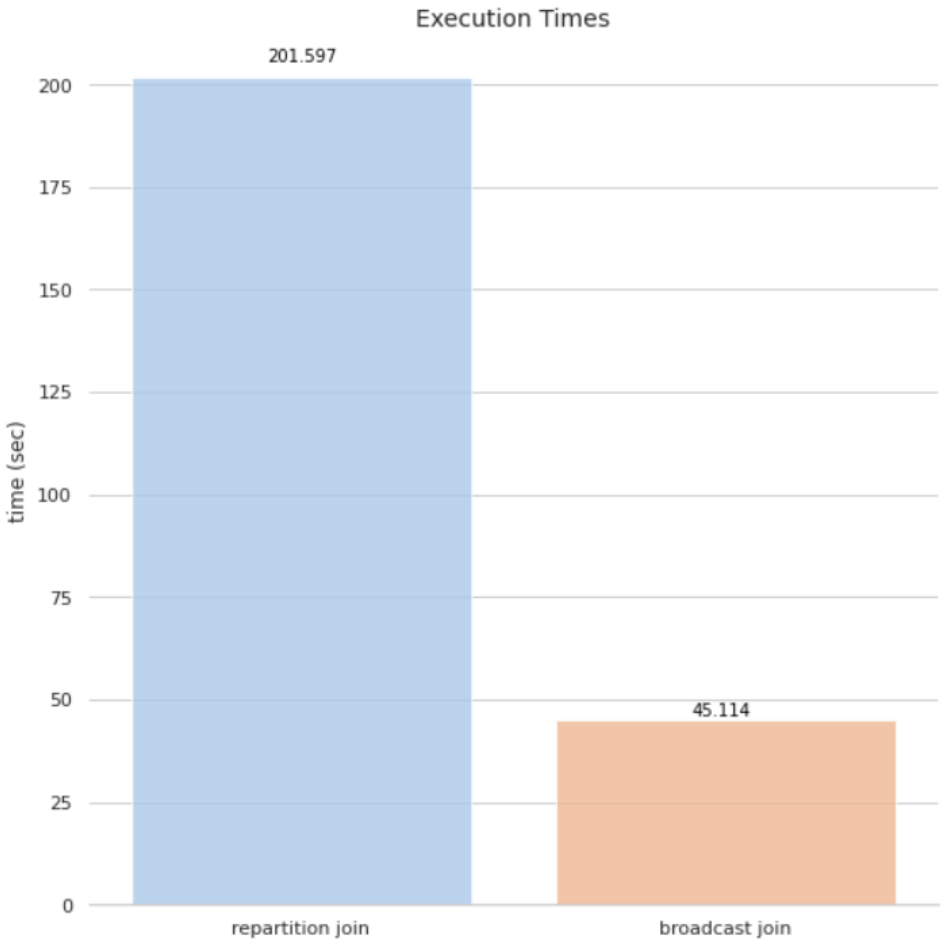
```
from pyspark.sql import SparkSession
import sys

spark = SparkSession.builder.appName('sparksql-part2_3').getOrCreate()
genres = spark.read.format("csv").\ options(header = "false" , inferSchema = "true").\
        load("hdfs://master:9000/files/movie_genres.csv")
genres_list = genres.head(100)
genres_100 = spark.createDataFrame(genres_list)
genres_100.write.csv("hdfs://master:9000/files/movie_genres_100.csv")
```

Έπειτα εκτελούμε τη συνένωση των δεδομένων του αρχείου movie_genres_100.csv με το αρχείο ratings.csv και με τα δύο είδη join. Το αρχείο movie_genres_100.csv έχει το join key movie_id στην θέση (index) 0 και το αρχείο ratings.csv έχει το join key movie_id στην θέση (index) 1. Οπότε η εντολή για την εκτέλεση της συνένωσης είναι η ακόλουθη:

spark-submit <type of join>.py hdfs://master:9000/files/movie_genres_100.csv 0 hdfs://master:9000/files/ratings.csv 1

Σχόλιο: Παρατηρούμε πως το αρχείο movie_genres_100.csv δίνεται ως πρώτο όρισμα, όπως και θα έπρεπε για να εκτελεστεί σωστά το broadcast join (να γίνει broadcast ο πίνακας του αρχείου αυτού).

Στην συνέχεια παραθέτουμε σε ένα ραβδόγραμμα τον χρόνο εκτέλεσης της συνένωσης αυτής για τις δύο υλοποιήσεις των joins:



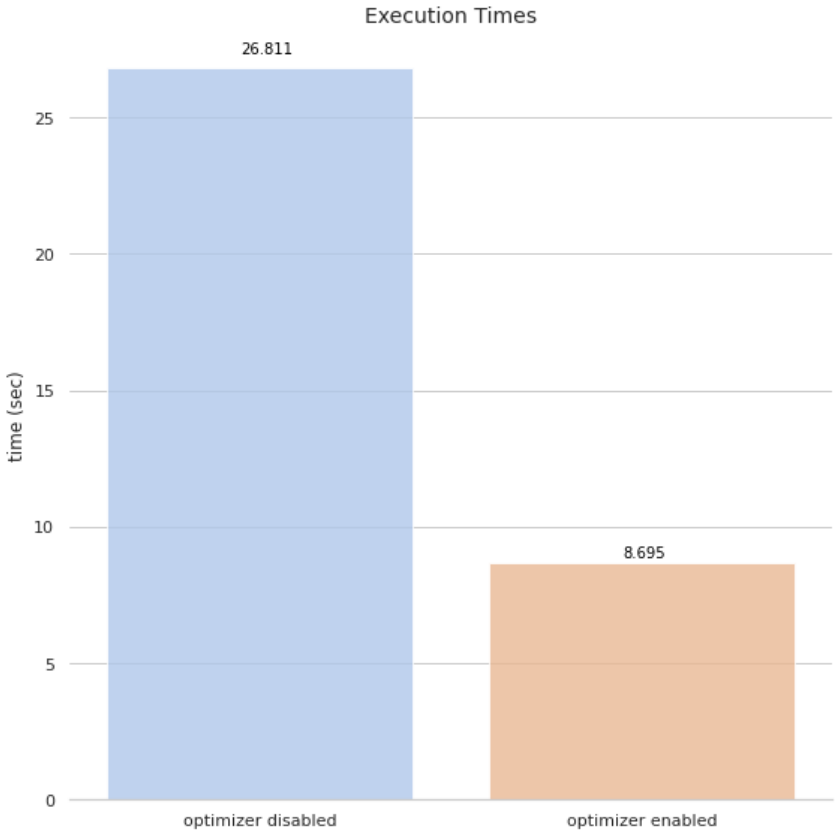
Παρατηρούμε, λοιπόν, ότι το broadcast join είναι αρκετά ταχύτερο σε σχέση με το repartition join. Κάτι τέτοιο είναι λογικό αφού στην περίπτωση μας συνενώνουμε ένα αρκετά μικρό πίνακα R με έναν πολύ μεγαλύτερο πίνακα L. Ο λόγος που το broadcast join είναι αποδοτικότερο σε τέτοιες περιπτώσεις είναι πως αντί να μεταφέρουμε και τους δύο πίνακες μέσω δικτύου (κάτι που γίνεται στο repartition join καθώς μεταβαίνουμε από την map στην reduce) κάνουμε broadcast τον μικρό πίνακα R ο οποίος αποθηκεύεται τοπικά σε όλα τα μηχανήματα. Έτσι αποφεύγουμε την ταξινόμηση και των δύο πινάκων καθώς και την μεταφορά του πολύ μεγαλύτερου πίνακα L στο δίκτυο. Στο συγκεκριμένο παράδειγμα συνένωσης λοιπόν, και τα δύο joins, προφανώς δίνουν τα ίδια αποτελέσματα (με διαφορετική βέβαια σειρά) αλλά διαφέρουν αρκετά ως προς το χρόνο εκτέλεσης.

Ζητούμενο 4:

Σκοπός του ζητούμενου αυτού είναι να κατανοήσουμε την σημασία που έχει ο βελτιστοποιητής ερωτημάτων (query optimizer) ο οποίος παρέχεται από την SparkSQL. Ο βελτιστοποιητής αυτός όπως μας υποδεικνύει και η εκφώνηση της εργασίας έχει την δυνατότητα να επιλέγει αυτόματα την υλοποίηση που θα χρησιμοποιήσει για ένα συγκεκριμένο join λαμβάνοντας υπόψη το μέγεθος των δεδομένων. Για παράδειγμα εάν ένας πίνακας είναι αρκετά μικρός (με βάση ένα όριο που ρυθμίζει ο χρήστης) θα χρησιμοποιήσει το broadcast join αλλιώς θα κάνει ένα repartition join. Επιπλέον έχει την δυνατότητα να αλλάξει την σειρά ορισμένων τελεστών προκειμένου να μειώσει τον συνολικό χρόνο εκτέλεσης του ερωτήματος.

Για το ζητούμενο αυτό μας δίνεται ένα έτοιμο script στο οποίο εκτελείται ένα συγκεκριμένο query. Στο query αυτό εκτελείται ένα join μεταξύ 100 εγγραφών του πίνακα movie_genres και του πίνακα ratings πάνω στο κοινό τους join key movie_id και επιστρέφονται όλα τα αποτελέσματα του συγκεκριμένου join (ουσιαστικά είναι η ίδια εκτέλεση που πραγματοποιήσαμε και στο ζητούμενο 3). Το script αυτό μας δίνει την δυνατότητα να αφήσουμε ενεργοποιημένο τον βελτιστοποιητή ερωτημάτων που παρέχεται από το SparkSQL ή να τον απενεργοποιήσουμε δίνοντας ως παράμετρο είτε N (optimizer enabled) είτε Y (optimizer disabled) αντίστοιχα. Η απενεργοποίηση του βελτιστοποιητή γίνεται με την εντολή spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1) όπου θέτουμε στην μεταβλητή περιβάλλοντος spark.sql.autoBroadcastJoinThreshold την τιμή -1.

Στην συνέχεια εκτελούμε το παραπάνω script τόσο με παράμετρο Y όσο και με N και παραθέτουμε τους χρόνους εκτέλεσης αλλά και το πλάνο εκτέλεσης της κάθε περίπτωσης:



Physical plan – optimizer disabled

```
== 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>
```

Physical plan – optimizer enabled

```
== 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>
```

Παρατηρώντας το πλάνο εκτέλεσης στην περίπτωση που έχουμε ενεργοποιημένο τον βελτιστοποιητή βλέπουμε ότι επιλέγεται να εκτελεστεί το BroadcastHash join. Κάτι τέτοιο είναι λογικό καθώς ο βελτιστοποιητής καταλαβαίνει ότι από τον πίνακα movie_genres θα χρησιμοποιηθούν μόνο 100 εγγραφές για την συνένωση και έτσι επιλέγει την χρήση του broadcast join καθώς ο ένας πίνακας είναι αρκετά μικρότερος σε σχέση με τον άλλον. Αντίθετα, απενεργοποιώντας τον βελτιστοποιητή βλέπουμε ότι πραγματοποιείται το SortMerge join. Παρατηρώντας τον χρόνο εκτέλεσης της κάθε περίπτωσης βλέπουμε όπως ήταν αναμενόμενο ότι το SortMerge Join παρουσιάζει αρκετά χειρότερο χρόνο εκτέλεσης (περίπου τριπλάσιο) σε σχέση με το Broadcast Join. Κάτι τέτοιο είναι λογικό καθώς όπως γνωρίζουμε στην περίπτωση που ο ένας πίνακας είναι αρκετά μικρότερος από τον άλλον η χρήση του Broadcast Join είναι η καλύτερη επιλογή κάνοντας αρχικά broadcast τον μικρό πίνακα σε όλους τους workers χωρίς να μεταφέρει στην συνέχεια δεδομένα πάνω στο δίκτυο. Αντίθετα στο SortMerge Join απαιτείται αρχικά οι πίνακες να ταξινομηθούν ως προς τα join keys ώστε να διαμοιραστούν κατάλληλα και να γίνουν οι εργασίες παράλληλα. Επιπλέον απαιτείται η μεταφορά δεδομένων και από τους δύο πίνακες πάνω στο δίκτυο ώστε εγγραφές με το ίδιο key να καταλήξουν στους ίδιους workers.

Βλέπουμε λοιπόν πως η χρήση του βελτιστοποιητή του SparkSQL μπορεί να οδηγήσει σε μεγάλη μείωση του χρόνου εκτέλεσης γεγονός που επαληθεύει τη χρησιμότητά του.