Χρήση του Apache Spark στις Βάσεις Δεδομένων

Γιωργουλάκης, Νικόλαος 03117138
Καπαρού, Αλεξάνδρα 03117100

18 Μαρτίου 2022

Περίληψη

Η παρούσα αναφορά αποτελεί την εργασία της ομάδας μας για το μάθημα Προχωρημένα θέματα Βάσεων Δεδομένων για το ακαδημαϊκό έτος 2021-2022. Σε αυτή την εργασία γίνεται χρήση του Apache Spark για τον υπολογισμό ερωτημάτων πάνω σε ένα σύνολο δεδομένων από ταινίες.

1 Μέρος 1

1.1 Ζητούμενο 1

Αρχικά δημιουργήσαμε έναν φάκελο files και έναν φάκελο outputs που θα μας χρειαστεί στην συνέχεια, στο hdfs με χρήση των εντολών

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

1.2 Ζητούμενο 2

Στην συνέχεια μας ζητήθηκε να μετατρέψουμε τα csv αρχεία μας σε μορφή parquet με σκοπό να βελτιστοποιήσουμε το I/O μειώνοντας τον χρόνο εκτέλεσης και να γίνεται πιο αποτελεσματική επεξεργασία της πληροφορίας. Αυτό επιτεύχθηκε με χρήση του python script csv_to_parquet.py το οποίο διαβάζει τα csv αρχεία και μετά χρησιμοποιώντας την παρακάτω εντολή τα αποθηκεύσαμε στον αντίστοιχο φάκελο στο hadoop.

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

1.3 Ζητούμενο 3

Σε αυτό το σημείο υλοποιήσαμε μία RDD API και μία Spark SQL λύση για κάθε ένα από τα ερωτήματα της εκφώνησης.

Σημείωση: Για την υλοποίηση των ερωτημάτων με Spark SQL δημιουργήσαμε με ένα if statement την επιλογή να διαβάζονται αρχεία τόσο CSV (χρησιμοποιώντας το option inferSchema) όσο και Parquet.Προκειμένου να εκτελέσουμε τα συγκεκριμένο queries γράφουμε:

```
spark-submit query.py csv
```

ή

```
spark-submit query.py parquet
```

αντίστοιχα.

Σημείωση 2: Για την υλοποίηση των ερωτημάτων με RDD API παραθέτουμε τον ψευδοκώδικα που περιγράφει τις υλοποιήσεις σε Map Reduce.

- Ερώτημα 1

  Για το ερώτημα 1 αρχικά διαβάζουμε το αρχείο movies.csv και οργανώνουμε τα δεδομένα. Στην συνέχεια κρατάμε μόνο τα στοιχεία που μας ενδιαφέρουν (δηλαδή η χρονολογία να είναι μετά το 2000 και τα cost, incomes να είναι θετικά). Έπειτα, υπολογίζουμε για κάθε ταινία το profit. Γνωρίζουμε ότι στην φάση του shuffling θα γίνει μία πράξη τύπου group by με κλειδί την χρονολογία και τέλος κρατάμε για κάθε χρονολογία την ταινία με το μεγαλύτερο κέρδος.

```
Map(key:movie_file,values:line_of_file)
    movie_id = line_of_file.split(",")[0];
    movie_title = line_of_file.split(",")[1];
    date = line_of_file.split(",")[3];
    cost = line_of_file.split(",")[5];
    incomes = line_of_file.split(",")[6];
emit(movie_id,movie_title,date,cost,incomes)
/* returns for example [1,"Title",2022-11-12,20,100]                    */

Reduce(key:movie_id,values)
    if (values[2]>0) and (values[3]>0) and (values[1].split("-")[0]>=2000) then
        emit(movie_id,values)
    end
end

Map(key:movie_id,values)
    movie_title = values[0];
    year = values[1].split("-")[0];
    cost = values[2];
    income = values[3];
    profit = 100*(income-cost)/cost
emit(date,movie_id,movie_title,cost,income,profit)
/* returns for example [2022,1,"Title",20,100,8]                        */
/* After sorting and shuffling phase we have
   [2011,[(1,"Title",10,100,9),(2,"Title2",20,100,8)]                   */

Reduce(key:date,values)
    max = 0;
    foreach value in values do
        if value[4]>max then
            max = value[4];
            name = value[1];
            id = value[0];
        end
    end
emit(date,id,name,value[2],value[3],value[4])
/* returns for example [2022,1,"Title",10,100,9]                        */
```

Algorithm 1: Query Q1

- Ερώτημα 2

  Για το ερώτημα 2 αρχικά διαβάζουμε τα περιεχόμενα του αρχείου ratings.csv. Γνωρίζουμε ότι στην φάση του shuffling θα γίνει μία πράξη τύπου group by με κλειδί το user_id του κάθε χρήστη. Στην συνέχεια, στην φάση του ρεδυςε κάνουμε την κατάλληλη πράξη για να βρούμε όλους τους χρήστες καθώς και τους χρήστες που έχουν δώσει σε ταινίες βαθμολογία μεγαλύτερη από 3 και παράγουμε το κατάλληλο αποτέλεσμα.

```
Map(key:ratings_file,values:line_of_file)
    user = line_of_file.split(",")[0];
    rating = line_of_file.split(",")[2];
emit(user;user,rating;rating)
/* returns for example [user 1,rating 4]                                    */
/* in emit the ; acts like a space in latex                                 */
/* After sorting and shuffling phase we have [user 1,[(rating 4),(rating 6)]]
   */

Reduce(key:user,ratings)
    all_users = 0;
    some_users = 0;
    sum = 0;
    count = 0;
    mean = 0;
    tokens = ratings.split(",");
    if tokens[0].equals("user") then
        all_users += 1
    end
    foreach rating in ratings do
        sum += rating[1];
        count += 1;
    end
    mean = sum/count;
    if mean > 3 then
        some_users += 1
    end
emit(some_users*100/all_users)
```

Algorithm 2: Query Q2

- Ερώτημα 3

  Για το ερώτημα 3 αρχικά διαβάζουμε τα κατάλληλα αρχεία genres.csv και ratings.csv και κρατάμε τις στήλες που μας ενδιαφέρουν. Κατά την φάση του Shuffling η λίστα με τα ratings θα κάνει group by με κλειδί το movie_id. Στην συνέχεια υπολογίζουμε την μέση βαθμολογία κάθε ταινίας και έπειτα κατά τα γνωστά στο shuffling γίνεται group by με κλειδί το movie_id. Μετά χρησιμοποιούμε άλλο ένα map ώστε να κρατήσουμε μόνο το είδος και την βαθμολογία από κάθε λίστα και τέλος υπολογίζουμε ανά είδος την μέση βαθμολογία του είδους και το πλήθος των ταινιών που υπάγονται σε αυτό το είδος.

```
Map(key:genres_file,values:line_of_file)
    movie = line_of_file.split(",")[0];
    genre = line_of_file.split(",")[1];
emit(movie,"genre";genre)
/* returns for example [1,Comedy]                                          */

Map(key:ratings_file,values:line_of_file)
    movie = line_of_file.split(",")[1];
    rating = line_of_file.split(",")[2];
emit(movie,"rating";rating)
/* returns for example [1,3],[1,5]                                         */
/* After sorting and shuffling phase we have [1,[(rating 4),(rating 6)]]   */

Reduce(key:movie,values:ratings)
    groupByKey();
    foreach rating in ratings do
        sum += rating[0];
        count += rating[1]
    end
    mean = sum/count;
emit(movie,("rating";mean))
/* returns for example [1,4]                                               */
/* After sorting and shuffling phase we have [1,[(genre Comedy),(rating 4)]]
   */

Map(key:movie,values:list)
emit(list[0],list[1])
/* returns for example [(genre Comedy),(rating 4)],[(genre Comedy),(rating 6)]
   */
/* After sorting and shuffling phase we have [(genre Comedy),(rating
   4),(rating 6)]                                                          */

Reduce(key:genre,values:ratings)
    len = 0;
    sum = 0;
    average = 0;
    tokens = ratings.split(",");
    if tokens[0].equals("rating") then
        len += 1;
        sum += tokens[1];
    end
    else if tokens[0].equals("genre") then
        name = tokens[1]
    end
    average = sum/len;
emit(name,average,len)
```

Algorithm 3: Query Q3

- Ερώτημα 4

  Στο ερώτημα αυτό αρχικά διαβάζουμε το αρχείο movies.csv, διασχίζοντάς το γραμμή-γραμμή υπολογίζουμε το πλήθος των λέξεων στην περιγραφή κάθε ταινίας και αντικαθιστούμε το πεδίο αυτό με την αριθμητική τιμή σε κάθε γραμμή. Κρατάμε όσες γραμμές έχουν ημερομηνία μεγαλύτερη του 2000 και όσες δεν έχουν κενή περιγραφή. Στη συνέχεια διαβάζουμε το αρχείο movie_genres.csv και κρατάμε όσες ταινίες ανήκουν στην κατηγορία Drama. Στη συνέχεια κάνουμε join των προηγούμενων δύο και διασχίζοντας τον νέο πίνακα στο πεδίο της ημερομηνίας υπολογίζουμε μέσω βοηθητικής συνάρτησης την τετραετία στην οποία ανήκει και αντικαθιστούμε το πεδίο με την τιμή αυτή. Ως κλειδί του νέου πίνακα θέτουμε την τετραετία στην οποία ανήκει. Στη συνέχεια, για κάθε τετραετία αθροίζουμε το μήκος των περιγραφών αλλά και υπολογίζουμε το πλήθος των ταινιών που ανήκουν σε αυτή. Από αυτές τις δύο τιμές παίρνουμε τον μέσο όρο για κάθε τετραετία τον οποίο και επιστρέφουμε.

```
Map(key:movie_file,values:line_of_file)
    movie_id = line_of_file.split(",")[0];
    movie_description = line_of_file.split(",")[2];
    movie_date = line_of_file.split(",")[3];
emit(movie_id,movie_description,movie_date)
Map(key:movie_id,values:(movie_description,movie_date))
    movie_description = count_udf(movie_description);
    if movie_description ≠ 0 and movie_date ≠ "" and year(movie_date) > 2000 then
        emit(movie_id,(movie_description,movie_date))
    end
end
/* returns for example [1,("a description",2022-11-12)]                    */

Map(key:movie_id,values:movie_category)
    if movie_category.equals("Drama") then
        emit(movie_id,movie_category)
    end
end
/* returns for example [1,Drama]                                            */
/* After sorting and shuffling phase we have [1,(Drama,"a
   description",2022-11-12)]                                                */

Map(key:movie_id,values:(movie_title,movie_description,movie_date))
    year = year_culc(movie_date);
emit(year,movie_description,1)
/* year_culc computes in which group of 5 years does a movie belong to      */
/* returns for example [2000-2004,("a description",1))]                    */

Reduce(key:year,values)
    count_desc = 0;
    count_ap = 0;
    foreach v in values do
        count_desc += values[0];
        count_ap += 1
    end
emit(year,(count_desc,count_ap))

Map(key:year,(count_desc,count_ap))
emit(year,count_desc/count_up)
```

Algorithm 4: Query Q4

- Ερώτημα 5

  Στο ερώτημα αυτό αρχικά ενώνουμε με join τα αρχεία movies.csv,ratings.csv και movie_genres.csv και καταλήγουμε σε έναν πίνακα με κλειδιά κατηγορία και id χρήστη και τιμές τίτλο ταινίας, βαθμολογία και δημοφιλία της ταινίας. Στη συνέχεια με ένα reduce σε αυτόν τον πίνακα βρίσκουμε για κάθε χρήστη την αγαπημένη του και τη λιγότερο αγαπημένη του ταινία για κάθε κατηγορία. Έπειτα, ενώνουμε τον πίνακα categories με τον πίνακα ratings και αφού αθροίσουμε τις βαθμολογίες κάθε χρήστη, με ένα reduce βρίσκουμε τον χρήστη με τις περισσότερες βαθμολογίες ανά κατηγορία. Φροντίζουμε ώστε σε περίπτωση που δύο χρήστες ισοβαθμούν να κρατάμε και τους δύο, γι'αυτό αντί για την custom reduceByKey του pyspark , χρησιμοποιούμε την combineByKey με συναρτήσεις που ορίζουμε ώστε να χειριστούμε τις ισοβαθμίες μαζί με την flatMapValues.Τέλος κάνουμε join τον πίνακα με τις αγαπημένες ταινίες κάθε χρήστη με τον πίνακα με τις λιγότερο αγαπημένες και στη συνέχεια με τον πίνακα με τον χρήστη με τις περισσότερες βαθμολογίες ανά κατηγορία και παίρνουμε το τελικό αποτέλεσμα.

```
/* The below map is called "movies" in the source code                    */
Map(key:movies_file,values:line_of_file)
    movie = line_of_file.split(",")[0];
    title = line_of_file.split(",")[1];
    popularity = line_of_file.split(",")[7];
emit(movie,title,popularity)
/* returns for example [1,"Title",5]                                      */

/* The below map is called "categories" in the source code                */
Map(key:genre_file,values:line_of_file)
    movie = line_of_file.split(",")[0];
    category = line_of_file.split(",")[1];
emit(movie,category)
/* returns for example [1,Comedy]                                         */

/* The below map is called "ratings" in the source code                   */
Map(key:ratings_file,values:line_of_file)
    user = line_of_file.split(",")[0];
    movie = line_of_file.split(",")[1];
    rating = line_of_file.split(",")[2];
emit(movie,user,rating)
/* returns for example [1,10,3]                                           */

/* The below map is called "rati" in the source code                      */
Map(key:ratings_file,values:line_of_file)
    user = line_of_file.split(",")[0];
    movie = line_of_file.split(",")[1];
emit(movie,user)
/* returns for example [1,10]                                             */
/* After sorting and shuffling phase we have [1,[(10,3),("Title",5)] (ratings
   and movies are joined)                                                 */
/* The below map is called "user_rat" in the source code                  */
Map(key:movie,values
    user = values[0][0];
    title = values[1][0];
    rating = values[0][1];
    popularity = values[1][1];
emit(movie,(user,title,rating,popularity))
/* returns for example [1,(10,"Title",3,5)]                               */
```

Algorithm 5: Query Q5

```
/* After sorting and shuffling phase we have [1,[(10,"Title",3,5),Comedy]]
   (user_rat and categories are joined)                                    */
Map(key:movie,values
    category = values[1];
    user = values[0][0];
    title = values[0][1];
    rating = values[0][2];
    popularity = values[0][3];
emit((category,user),(title,rating,popularity))
/* returns for example [(Comedy,1),("Title",3,5)]                          */

/* The below map is called "best_user" in the source code                  */
Reduce(key:(category,user),values:(title,rating,popularity))
    final = [];
    foreach v1,v2 in values do
        if v1.rating > v2.rating then
            final.append(v1)
        end
        else if v2.rating > v1.rating then
            final.append(v2)
        end
        else if v2.rating.equals(v1.rating) then
            if v1.popularity > v2.popularity then
                final.append(v1)
            end
            else
                final.append(v2)
        end
    end
emit(final)
/* keeps the max from every two elements                                   */

/* The below map is called "worst_user" in the source code                 */
Reduce(key:(category,user),values:(title,rating,popularity))
    final = [];
    foreach v1,v2 in values do
        if v1.rating < v2.rating then
            final.append(v1)
        end
        else if v2.rating < v1.rating then
            final.append(v2)
        end
        else if v2.rating.equals(v1.rating) then
            if v1.popularity > v2.popularity then
                final.append(v1)
            end
            else
                final.append(v2)
        end
    end
emit(final)
/* keeps the min from every two elements                                   */
/* After sorting and shuffling phase we have [1,[(10,3,Comedy)]] (categories
   and ratings are joined)                                                 */
```

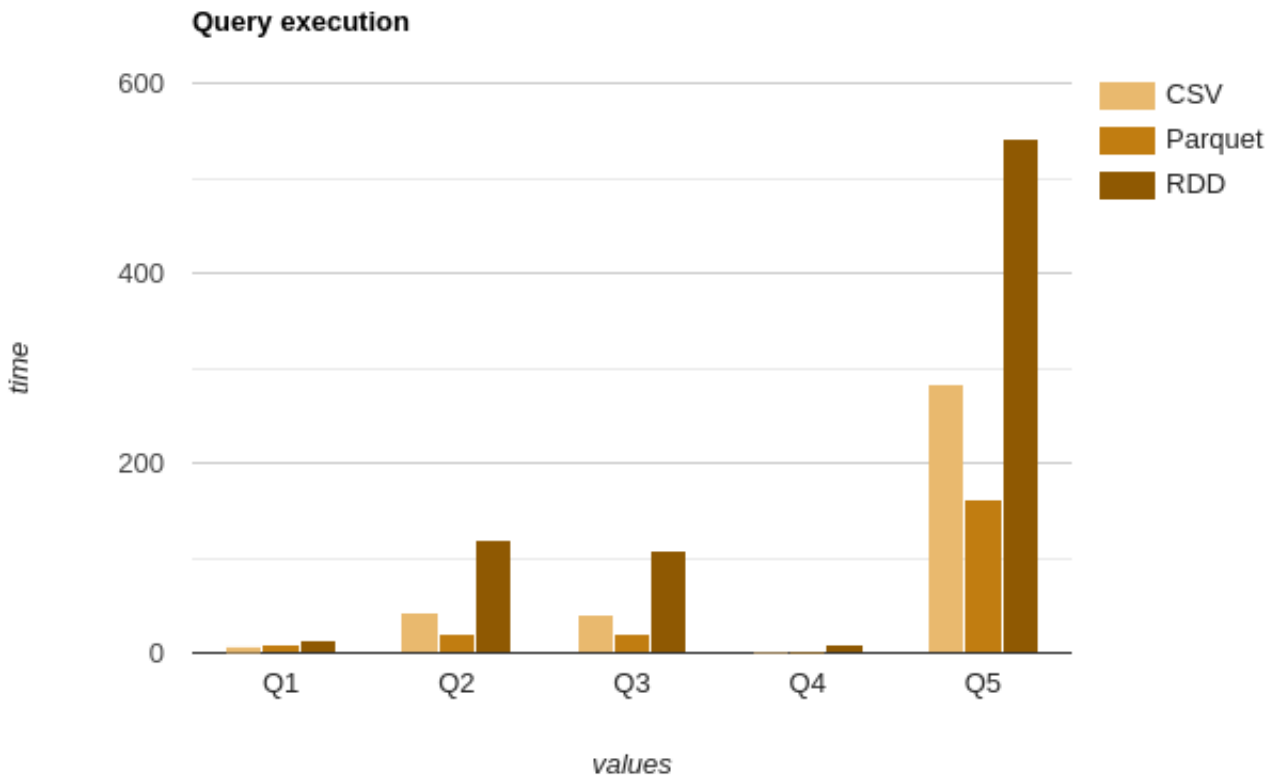
Algorithm 6: Query Q5 (Cont.)

```
/* The below map is called "user_per" in the source code                    */
Map(key:movie,values:(category,(user,rating)))
    category = values[0];
    user = values[1][0];
emit(category,user,1)
/* returns for example [Comedy,10,1]                                        */

Reduce(key:null,values:(category,user,1))
    counter = 0;
    foreach v in values do
        counter += values[2]
    end
emit(category,user,counter)
/* returns for example [Comedy,10,42]                                       */

Map(key:null,values:(category,user,counter))
    category = values[0];
    user = values[1];
    counter = values[2];
emit(category,(user,counter))
/* returns for example [Comedy,(10,42)]                                     */

Reduce(key:category,values:(user,counter))
    keep = [];
    foreach v1,v2 in values do
        if v1.counter < v2.counter then
            keep.append(v2.counter)
        end
        if v1.counter > v2.counter then
            keep.append(v1.counter)
        end
        if v1.counter.equals(v2.counter) then
            keep.append(v1.counter);
            keep.append(v2.counter)
        end
    end
emit(keep)
/* returns for example [Comedy,(10,54)]                                     */
/* with the above operation we keep the element with the max counter        */

Map(key:category,values:(user,counter))
emit((category,user),counter)
/* returns for example [(Comedy,10),54]                                     */
/* After sorting and shuffling phase we have
   [(Comedy,10),[("Best_title",9,100),("Worst_title",1,2]] (best_user and
   worst_user are joined)                                                   */

/* The below map is called "res" in the source code                         */
Map(key:(category,user),values:((best_title,best_rating,best_popularity),
 (worst_title,worst_rating,worst_popularity)))
emit((category,user),best_title,best_rating,
/* returns for example [(Comedy,10),[("Best_title",9,"Worst_title",1]]      */
/* After sorting and shuffling phase we have
   [Comedy,10,67,"Best_title",9,"Worst_title",1] (res and user_per are joined)
   */
```

Algorithm 7: Query Q5 (Cont.)

1.4 Ζητούμενο 4

Στο ερώτημα αυτό μας ζητήθηκε να μετρήσουμε τους χρόνους εκτέλεσης των queries και να τους παρουσιάσουμε σε ένα ραβδόγραμμα όπως φαίνεται παρακάτω:



Σχήμα 1: Χρόνοι εκτέλεσης ερωτημάτων

Οι χρόνοι σε μορφή πίνακα φαίνονται παρακάτω:

Χρόνοι Εκτέλεσης			
Query number/- Type of query	CSV	Parquet	RDD
Q1	7.325	9.506	13.010
Q2	42.020	20.520	119.105
Q3	40.334	19.629	107.094
Q4	0.515	0.606	8.477
Q5	283.701	161.903	541.688

Σχολιασμός:

Με βάση τα παραπάνω αποτελέσματα βλέπουμε ότι σε κάθε περίπτωση το RDD Api παράγει τα χειρότερα αποτελέσματα από άποψη χρόνου. Ο λόγος για τον οποίο συμβαίνει αυτό είναι το γεγονός ότι αντίθετα με την Spark SQL, το RDD Api δεν διαθέτει κάποια inbuilt optimization engine [1] (σε αντίθεση με την Spark SQL που διαθέτει τον catalyst optimizer) η οποία θα βελτιστοποιούσε τα αποτελέσματα και κατα συνέπεια τους χρόνους εκτέλεσης των ερωτημάτων. Επίσης, όποτε το Spark χρειάζεται να γράψει δεδομένα στον δίσκο χρησιμοποιεί Java serialization, διαδικασία που είναι χρονοβόρα μιας και στέλνει τόσο

τα δεδομένα όσο και την δομή τους ανάμεσα στους κόμβους. Τέλος, το RDD Api είναι γενικά πιο αργό κατά των υπολογισμό απλών πράξεων ομαδοποίησης και συνάθροισης μιας και χρειάζεται να μεταφερθούν δεδομένα μέσω του δικτύου, κάτι το οποίο αυξάνει τον χρόνο εκτέλεσης.

Από την άλλη πλευρά, όσον αφορά το Spark SQL παρατηρούμε ότι στις περισσότερες περιπτώσεις οι υλοποιήσεις με το διάβασμα των αρχείων parquet εκτελούνται πιο γρήγορα από αυτές στις οποίες διαβάζονται αρχεία CSV . Οι περιπτώσεις στις οποίες συμβαίνει το αντίθετο είναι αυτές των Q1, Q4 όπου όπως βλέπουμε οι χρόνοι είναι πολύ μικροί σε σχέση με αυτούς των υπολοίπων ερωτημάτων και παρόμοιοι μεταξύ τους οπότε δεν αποτελούν κατάλληλες μετρικές σύγκρισης. Όπως γνωρίζουμε, τα parquet files έχουν μικρότερο αποτύπωμα στην μνήμη και τον δίσκο, κάτι το οποίο καθιστά ταχύτερη την εγγραφή και την ανάγνωση τους. Ακόμα, τα αρχεία parquet αποθηκεύουν τα metadata των αρχείων κάτι το οποίο δεν συμβαίνει στα αρχεία CSV . Το γεγονός αυτό σημαίνει ότι στα CSV αρχεία για να εκτελεστεί το πρόγραμμα θα πρέπει είτε να παρέχουμε τα μεταδεδομένα είτε να γίνει inferred το schema [2] .

Όσον αφορά το τελευταίο, γιάυτό τον λόγο ενεργοποιήσαμε στα CSV αρχεία την επιλογή inferSchema για τον συμπερασμόυ του σχήματος των δεδομένων, δηλαδή του τύπου της κάθε στήλης δεδομένων. Προκειμένου να γίνει αυτό ωστόσο, απαιτείται ένα επιπλέον πέρασμα όλου του αρχείου κάτι το οποίο καθυστερεί αισθητά την διαδικασία εκτέλεσης του αρχείου αν το μέγεθος του είναι μεγάλο,γιάυτό τον λόγο δεν προτιμάται σαν τακτική. Βέβαια, στο τέλος της εξαγωγής σχήματος το dataframe μας θα έχει σίγουρα ένα σωστό σχήμα [3].

2 Μέρος 2

Σε αυτό το σημείο μας ζητήθηκε η μελέτη και αξιολόγηση των broadcast join και repartition join με βάση την δημοσίευση που μας δόθηκε [4].

2.1 Ζητούμενο 1

Υλοποιήσαμε τον αλγόριθμο broadcast join στο RDD Api με βάση τον παρακάτων ψευδοκώδικα.Δεδομένου ότι στην εκφώνηση μας ζητείται να θεωρήσουμε ότι ο ένας πίνακας είναι πάντα αρκετά μικρός ώστε να μπορεί να γίνει broadcast ολόκληρος, αγνοήσαμε την συνάρτηση Close() του ψευδοκώδικα.

Init ()
  if R not exist in local storage **then**
    remotely retrieve R
    partition R into p chunks $R_1..R_p$
    save $R_1..R_p$ to local storage

  if $R <$ a split of L **then**
    $H_R \leftarrow$ build a hash table from $R_1..R_p$
  else
    $H_{L_1}..H_{L_p} \leftarrow$ initialize p hash tables for L

Map (K: null, V: a record from an L split)
  if H_R exist **then**
    probe H_R with the join column extracted from V
    for each match r from H_R **do**
      emit (null, $new\_record(r, V)$)
  else
    add V to an H_{L_i} hashing its join column

Σχήμα 2: Ψευδοκώδικας Broadcast Join

Όσον αφορά την δική μας υλοποίηση, αρχικά διαβάζουμε τα κατάλληλα αρχεία (τον πίνακα ratings και τις 100 πρώτες γραμμές του πίνακα movie genres). Στην συνέχεια ορίσαμε την συνάρτηση broadcast join η οποία πρώτα δημιουργεί ένα hash table με χρήση της εντολής broadcast. Εφόσον το κλειδί μας είναι το movie_id , για κάθε εγγραφή του πίνακα L την αναζητούμε στο hash table και στο τέλος επιστρέφουμε μία λίστα από τούπλες που περιέχει το κλειδί του L , μία εγγραφή του R και μία εγγραφή του L .

2.2 Ζητούμενο 2

Ακολουθούμε σε μεγάλο βαθμό τον ψευδοκώδικα της δημοσίευσης. Αρχικά διαβάζουμε το αρχείο ratings.csv καθώς και το αρχείο με τις 100 πρώτες γραμμές από το movie_genres. Φροντίζουμε ώστε στους δύο πίνακες να έρθει πρώτο πεδίο το κλειδί δηλαδή το movie id, ενώ ταυτόχρονα προσθέτουμε ένα χαρακτηριστικό αναγνωριστικό "ratings" στον πίνακα με τα ratings και "genres" στον πίνακα με τα movie_genres. Ενώνουμε με union τους δύο πίνακες και μέχρι στιγμής έχουμε πραγματοποιήσει τη φάση map του ψευδοκώδικα της δημοσίευσης. Στη συνέχεια, για τη φάση reduce,γκρουπάρουμε με κλειδί και εφαρμόζουμε την flatMapValues με μία δική μας βοηθητική συνάρτηση η οποία κάνει τη διαδικασία με τα buckets όπως περιγράφεται στη δημοσίευση. Επομένως το αποτέλεσμα θα είναι για κάθε ταινία όλοι οι συνδυασμοί από κατηγορία-βαθμολογίες(χρήστης,βαθμολογία,ημερομηνία).

Map (K: null, V: a record from a split of either R or L)
  $join\_key \leftarrow$ extract the join column from V
  $tagged\_record \leftarrow$ add a tag of either R or L to V
  emit ($join\_key, tagged\_record$)

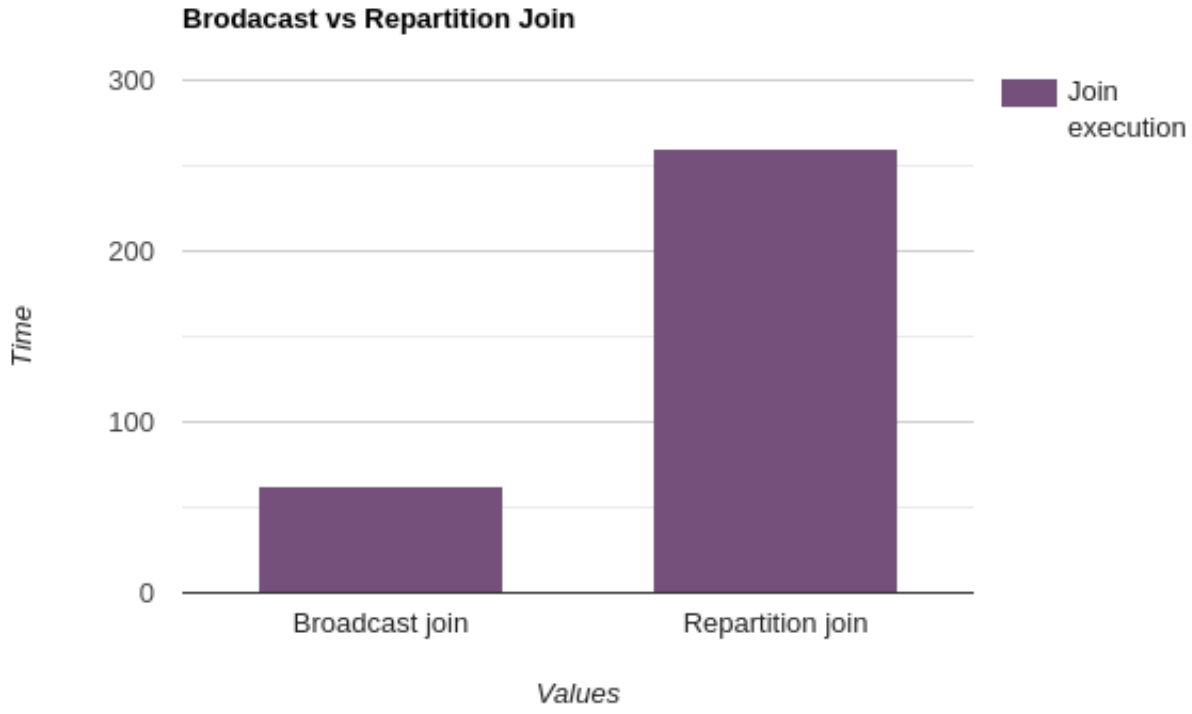
Reduce (K': a join key,
    $LIST\_V'$: records from R and L with join key K')
  create buffers B_R and B_L for R and L, respectively
  for each record t in $LIST\_V'$ **do**
    append t to one of the buffers according to its tag
  for each pair of records (r, l) in $B_R \times B_L$ **do**
    emit (null, $new\_record(r, l)$)

Σχήμα 3: Ψευδοκώδικας Repartition Join

2.3 Ζητούμενο 3

Σε αυτό το σημείο απομονώσαμε 100 γραμμές του πίνακα movie genres (με χρήση του αρχείου collect_100_rows_of_genres.py) και τρέξαμε τους δύο παραπάνω αλγορίθμους για συνένωση των 100 αυτών γραμμών με τον πίνακα ratings . Μετρήσαμε τους χρόνους εκτέλεσης και τα αποτελέσματα που πήραμε φαίνονται στο παρακάτω ραβδόγραμμα.

Όπως παρατηρούμε παρακάτω, το broadcast join θέλει σχεδον τον υποτετραπλάσιο χρόνο εκτέλεσης από ότι το repartition join. Κάτι τέτοιο ήταν αναμενόμενο μιας και όπως γνωρίζουμε το broadcast join (aka Map-Side Join) είναι πολύ αποδοτικός αλγόριθμος για συνενώσεις μεταξύ ενός μεγάλου πίνακα και ενός μικρού, όπως συμβαίνει στην περίπτωση μας. Αυτό συμβαίνει επειδή σε αυτόν τον τύπο συνένωσης αποφεύγεται η αποστολή όλων των δεδομένων του μεγάλου πίνακα κατά μήκος του δικτύου [5]. Πιο συγκεκριμένα, η Spark στέλνει ένα αντίγραφο του μικρού πίνακα σε όλους τους κόμβους εκτέλεσης. Με αυτόν τον τρόπο, κάθε executor καθίσταται επαρκής για την συνένωση εγγραφών ενός μεγάλου πίνακα με τον μικρό πίνακα, σε κάθε κόμβο [6]. Αντίθετα, στην περίπτωση του repartition join (aka reduce-side join) μεταφέρουμε και τους δύο πίνακες σε ολόκληρο το δίκτυο κάτι το οποίο οδηγεί στην κατακόρυφη αύξηση του χρόνου εκτέλεσης της συνένωσης. Πιο αναλυτικά, στο repartition join χρησιμοποιείται η sort-merge του MapReduce για την ομαδοποίηση εγγραφών. Το Hadoop στέλνει τα ίδια κλειδιά στον



Σχήμα 4: Χρόνοι εκτέλεσης των Joins

ίδιο reducer επομένως προκειμένου να εκτελεστεί η συνένωση πρέπει κάθε φορά να συγκρίνουμε ένα κλειδί με όλα τα υπόλοιπα, κάτι το οποιό απαιτεί την μεταφορά και την ανάμειξη των δεδομένων κατά μήκος του δικτύου [7].

2.4 Ζητούμενο 4

Το ζητούμενο αυτού του ερωτήματος ήταν να συμπληρώσουμε κατάλληλα την εντολή μεταξύ των < > προκειμένου να μπορούμε να ενεργοποίησουμε/απενεργοποιήσουμε την επιλογή του join από τον βελτιστοποιητή. Ο συγκεκριμένος βελτιστοποιητής ουσιαστικά λαμβάνει υπόψιν το μέγεθος των δεδομένων που πρόκειται να συνενωθούν και σε περίπτωση που ο ένας πίνακας είναι επαρκώς μικρός με βάση ένα ορισμένο από τον χρήστη threshold τότε εκτελείται broadcast join ενώ σε αντίθετη περίπτωση εκτελείται repartition join. Η εντολή που προσθέσαμε είναι η

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

η οποία θέτει ένα byte threshold με το οποίο ένας πίνακας μπορεί να γίνει broadcasted σε όλους τους worker nodesκατά την εκτέλεση του join . Όταν ωστόσο αυτή η εντολή πάρει την τιμή -1 όπως στην δική μας περίπτωση, απενεργοποιείται το broadcasting και άρα ο βελτιστοποιητής [8]. Μας ζητήθηκε να εκτελέσουμε το query με και χωρίς βελτιστοποιητή και να παρουσισάουμε τα αποτελέσματα.

Το πλάνο εκτέλεσης που παράγει ο βελτιστοποιητής κάθε φορά φαίνεται παρακάτω:

- Ενεργοποιημένος βελτιστοποιητής

```
== Physical Plan ==
*(3) BroadcastHashJoin [_c0#8], [_c1#1], Inner, BuildLeft
:- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, false]))
:  +- *(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: [], PushedFi
lters: [], ReadSchema: struct<_c0:string,_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, Locatio
n: InMemoryFileIndex[hdfs://master:9000/files/ratings.parquet], PartitionFilters: [], PushedFilt
ers: [IsNotNull(_c1)], ReadSchema: struct<_c0:string,_c1:string,_c2:string,_c3:string>
```

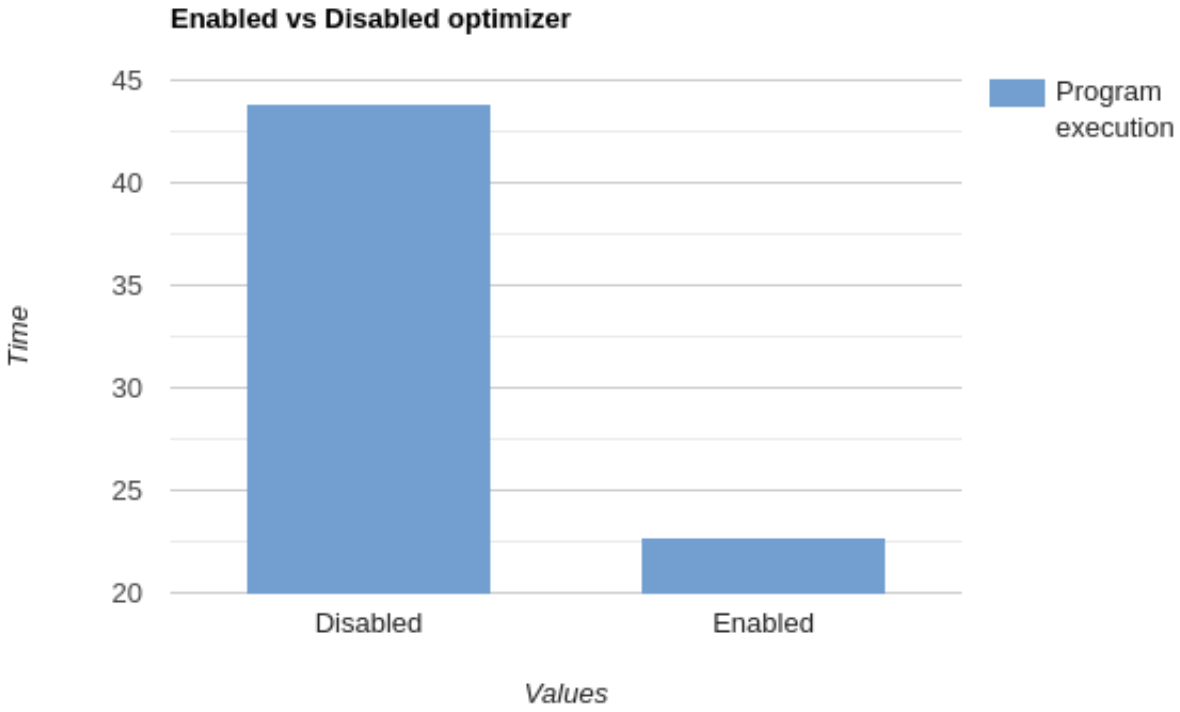
- Απενεργοποιημένος βελτιστοποιητής

```
== 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, Locatio
n: InMemoryFileIndex[hdfs://master:9000/files/movie_genres.parquet], PartitionFilters: [], Pushe
dFilters: [], ReadSchema: struct<_c0:string,_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, L
ocation: InMemoryFileIndex[hdfs://master:9000/files/ratings.parquet], PartitionFilters: [], Push
edFilters: [IsNotNull(_c1)], ReadSchema: struct<_c0:string,_c1:string,_c2:string,_c3:string>
```

Το ραβδόγραμμα με τους χρόνους εκτέλεσης φαίνεται παρακάτω στο σχήμα 5:

Σχολιασμός:

Με βάση το παραπάνω ραβδόγραμμα παρατηρούμε ότι με την απενεργοποίηση του βελτιστοποιητή, σχεδόν διπλασιάζεται ο χρόνος εκτέλεσης. Αυτό συμβαίνει επειδή όταν ο βελτιστοποιητής είναι ενεργοποιημένος τότε εκμεταλλεύεται το γεγονός ότι ο ένας πίνακας (movie_genres) είναι αρκετά μικρότερος από τον άλλο (ratings) επομένως μπορεί να χρησιμοποιηθεί το broadcast join το οποίο όπως αποδείξαμε παραπάνω είναι πολύ ταχύτερο με τις συγκεκριμένες συνθήκες. Στην περίπτωση που είναι απενεργοποιημένος ο βελτιστοποιητής, εκτελείται repartition join χρησιμοποιείται δηλαδή το Shuffle Sort Merge Join το οποίο απαιτεί την μεταφορά των δεδομένων κατά μήκος του δικτύου, διαδικασία που από μόνη της είναι αρκετά χρονοβόρα. Επιπρόσθετα, στο Shuffle Sort Merge απαιτούνται κάποιες επιπλέον πράξεις όπως είναι τα Sort και Exchange hashpartitioning τα οποία φαίνουνται και στην παραπάνω εικόνα. Πιο αναλυτικά, γνωρίζουμε ότι ο αλγόριθμος Shuffle Sort Merge Join αποτελείται από τις εξής τρεις φάσεις : Shuffle Phase, Sort Phase, Merge Phase. Στην πρώτη φάση τα δεδομένα διαβάζονται και ανακατεύονται μεταξύ τους. Μετά την πράξη της ανάμειξης, οι εγγραφές με τα ίδια κλειδιά από τα δύο σύνολα δεδομένων θα καταλήξουν στο ίδιο partition. Στην συνέχεια, στην φάση της ταξινόμησης, οι εγγραφές ταξινομούνται κατά κλειδί ενώ τέλος, στην φάση της συγχώνευσης εκτελείται μία συνένωση μέσω του iteration σε όλες τις εγγραφές του ταξινομημένου συνόλου δεδομένων. Εφόσον το σύνολο δεδομένων είναι πλέον ταξινομημένο, η πράξη της συνένωσης σταματάει για ένα στοιχείο μόλις συναντήσουμε μία ασυμφωνία κλειδιών [9]. Επομένως αντιλαμβανόμαστε ότι όταν ο βελτιστοποιητής είναι απενεργοποιημένος, η διαδικασία της



Σχήμα 5: Χρόνοι εκτέλεσης enabled/disabled optimizer

συνένωσης απαιτεί περισσότερα στάδια επεξεργασίας και άρα περισσότερο χρόνο.

3 Οργάνωση των αρχείων

Για καλύτερη εποπτεία του χώρου του virtual machine παραθέτουμε απο κάτω την δομή του:

- Στο master υπάρχει οι κώδικες *collect_100_rows_of_genres.py*, *csv_to_parquet.py* που λειτουργούν ως βοηθητικοί κώδικες όπως αναφέραμε παραπάνω. Επίσης υπάρχουν τα τρία αρχικά αρχεία movie_genres.csv, movies.csv, ratings.csv καθώς επίσης και ένας φάκελος code .
- Στον φάκελο code περιέχονται όλοι οι κώδικες που έχουμε υλοποιήσει για το 1ο και το 2ο μέρος. Επίσης υπάρχουν τα αρχεία create_times.py, times.json που χρησιμοποιήθηκαν προκειμένου να αποθηκευτούν οι χρόνοι εκτέλεσης κάθε κώδικα. Τέλος, για κάθε κώδικα υπάρχει το output του (αρχεία με την κατάληξη _output.txt , _output_parquet.txt) για καλύτερη εποπτεία. Τονίζουμε ότι στα ερωτήματα του 1ου μέρους, τα queries με RDD και SparkSQL παράγουν τα ίδια αποτελέσματα. Επίσης υπάρχει και ένα αρχείο checkout.py το οποίο αποθηκεύει τα αποτελέσματα από το hdfs τοπικά στο master.
- Στο hdfs υπάρχει ένας φάκελος files ο οποίος περιέχει τα αρχεία που μας δόθηκαν (movie_genres.csv, movies.csv, ratings.csv)καθώς και αυτά που δημιουργήσαμε κατά την διάρκεια της εργασίας (100_movie _genres.csv, movie_genres.parquet, movies.parquet, ratings.parquet).
- Στο hdfs υπάρχει ένας φάκελος outputs ο οποίος περιέχει τα αποτελέσματα των ερωτημάτων που υλοποιήσαμε.

  Οποιαδήποτε αρχεία δεν έχουν αναφερθεί παραπάνω αποτελούν αρχεία που κατέβηκαν για το στήσιμο της εργασίας ή παραδείγματα που κάναμε στο εργαστήριο και δεν θα πρέπει να ληφθούν υπόψιν κατά την αξιολόγηση της εργασίας.

References

[1] https://data-flair.training/blogs/apache-spark-rdd-vs-dataframe-vs-dataset/

[2] https://medium.com/ssense-tech/csv-vs-parquet-vs-avro-choosing-the-right-tool-for-the-right-job-79c9f56914a8

[3] https://stackoverflow.com/questions/56927329/spark-option-inferschema-vs-header-true

[4] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.644.9902rep=rep1type=pdf

[5] https://jaceklaskowski.gitbooks.io/mastering-spark-sql/content/spark-sql-joins-broadcast.html

[6] https://towardsdatascience.com/the-art-of-joining-in-spark-dcbd33d693c

[7] https://www.slideshare.net/udayaprasad9/repartition-join-in-mapreduce

[8] https://spark.apache.org/docs/latest/sql-performance-tuning.html

[9] https://www.hadoopinrealworld.com/how-does-shuffle-sort-merge-join-work-in-spark/