Γεώργιος Καλλίτσης , Α.Μ.: el17051

Ηλιάνα-Μαρία Ξύγκου, Α.Μ.: el17059

Προχωρημένα Θέματα Βάσεων Δεδομένων

Εξαμηνιαία Εργασία

Μέρος 1

Ζητούμενο 1

Αρχικά, φορτώθηκαν τα 3 δοθέντα αρχεία στο HDFS και συγκεκριμένα σε ένα φάκελο files, εκτελώντας τις ακόλουθες 4 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/

Ζητούμενο 2

Εν συνεχεία, τα *csv* αρχεία μετατράπηκαν σε *parquet*, προκειμένου να ελαχιστοποιηθεί ο χρόνος εκτέλεσης και να βελτιστοποιηθεί η διαδικασία I/O (καθώς έχει μικρότερο αποτύπωμα στη μνήμη και στο δίσκο), όπως αναφέρεται χαρακτηριστικά και στην εκφώνηση της άσκησης. Η μετατροπή έγινε εκτελώντας το αρχείο *convert.py* που βρίσκεται στο φάκελο της εργασίας. Στο αρχείο αυτό, αρχικά διαβάστηκαν τα 3 *csv* αρχεία και εν συνεχεία αποθηκεύτηκαν σε μορφή *parquet* στο HDFS.

Ζητούμενο 3

Για καθένα από τα 5 ερωτήματα που περιγράφονται στην εκφώνηση της άσκησης, παρακάτω παρατίθεται ψευδοκώδικας Map-Reduce που εξηγεί τη λογική με την οποία επιλύθηκε το εκάστοτε ερώτημα:

Q1

```
# line from movies.csv

map(_, line):
        line = line.split(“,”)                  # works as given split_complex function
        year = line[3][:4] if line[3] else None
        movie_name = line[1]
        cost = float(x[5])
        gain = float(x[6])
        if year != None and cost != 0 and gain != 0 and int(year) >= 2000:
                emit(year, (movie_name, (gain - cost) / cost * 100))
```

```
reduce(year, list_values):
        movie_name, max_profit = list_values[0]
        for x in list_values[1:]:
                profit = x[1]
                if profit > max_profit:
                        movie_name, max_profit = x
        emit(year, (movie_name, max_profit))
```

Q2

```
# line from ratings.csv
map(_, line):
        line = line.split(“,”)
        user = line[0]
        rating = float(line[2])
        emit(user, (rating, 1))

reduce(user, list_values):
        sum = 0
        ratings = 0
        for x in list_values:
                sum += x[0]
                ratings += x[1]
        emit(user, (sum, ratings))

map(user, (sum, ratings)):
        mean = sum / ratings
        if mean > 3.0:
                emit(“result”, (1, 1))
        else:
                emit(“result”, (1, 0))
```

```
reduce(key, list_values):
        # key = “result”
        users = 0
        above_3_users = 0
        for x in list_values:
                users += x[0]
                above_3_users += x[1]
        emit(None, above_3_users / users * 100)
```

Q3

Σε αυτό το ερώτημα, εκτελέστηκαν 3 αλληλουχίες Map-Reduce.

1η αλληλουχία:

```
# key = “g” for movie_genres.csv
map(key, line):
        movie_id, genre = line.split(“,”)
        emit(movie_id, genre)

reduce(movie_id, list_values):
        emit(“g”, (movie_id, list_values))
```

2η αλληλουχία:

```
# key = “r” for ratings.csv
map(key, line):
        line = line.split(“,”)
        movie_id = line[1]
        rating = float(line[2])
        emit(movie_id, (rating, 1))

reduce(movie_id, list_values):
        sum = 0
        ratings = 0
        for x in list_values:
```

```
            sum += x[0]
            ratings += x[1]
        emit("r", (movie_id, sum / ratings))
```

3η αλληλουχία:

```
# join results of first 2 sequences of Map-Reduce
map(key, value):
    if key = "g":
        movie_id, list_genres = value
        emit(movie_id, ("g", list_genres))
    else if key = "r":
        movie_id, mean = value
        emit(movie_id, ("r", mean))

reduce(movie_id, list_values):
    list_g = [list_genres for (x, list_genres) in list_values if x = "g"]
    # above list will consist of one item
    list_r = [mean for (x, mean) in list_values if x = "r"]
    # above list will consist of one item
    for list_genres in list_g:
        for mean in list_r:
            emit(movie_id, (list_genres, mean))

map(movie_id, (list_genres, mean)):
    for genre in list_genres:
        emit(genre, (mean, 1))

reduce(genre, list_values):
    sum = 0
    ratings = 0
    for x in list_values:
        sum += x[0]
        ratings += x[1]
```

```
        emit(genre, (sum / ratings, ratings))
```

Q4

Σε αυτό το ερώτημα, εκτελέστηκαν 3 αλληλουχίες Map-Reduce.

```
def five_year_period(date):
    if date:
        date = int(date[:4])
        if  2000 <= date <= 2004:
            return '2000-2004'
        elif 2005 <= date <= 2009:
            return '2005-2009'
        elif 2010 <= date <= 2014:
            return '2010-2014'
        elif 2015 <= date <= 2019:
            return '2015-2019'
        else:
            return None
    else:
        return None
```

1η αλληλουχία:

```
# key = “g” for movie_genres.csv
map(key, line):
        movie_id, genre = line.split(“,”)
        if genre = “Drama”:
                emit(movie_id, genre)

reduce(movie_id, list_values):
        # list_values contains one item: “Drama”
        emit(“g”, (movie_id, list_values[0]))
```

2η αλληλουχία:

```
# key = “m” for movies.csv
map(key, line):
        line = line.split(“,”)                    # works as given split_complex function
        description = line[2]
        if description != “”:
                movie_id = line[0]
                words = description.split(“ ”).length()
                datetime = line[3]
                period = five_year_period(datetime)
                if period != None:
                        emit(movie_id, (words, period))

reduce(movie_id, list_values):
        # list_values contrains one item
        words, period = list_values[0]
        emit(“m”, (movie_id, (words, period)))
```

3η αλληλουχία:

```
# join results of first 2 sequences of Map-Reduce
map(key, value):
        if key = “g”:
                movie_id, genre = value
                emit(movie_id, (“g”, genre))
        else if key = “m”:
                movie_id, (words, period) = value
                emit(movie_id, (“m”, words, period))

reduce(movie_id, list_values):
        list_g = [genre for (x, genre) in list_values if x = “g”]
        list_m = [(words, period) for (x, words, period) in list_values if x = “m”]
        for genre in list_g:
                for (words, period) in list_m:
```

```
emit(movie_id, (genre, (words, period)))
```

```
map(movie_id, value):
    words, period = value[1]
    emit(period, (words, 1))


reduce(period, list_values):
    sum = 0
    count = 0
    for x in list_values:
        sum += x[0]
        count += x[1]
    emit(period, sum / count)
```

Q5

Σε αυτό το ερώτημα, εκτελέστηκαν 7 αλληλουχίες Map-Reduce. Επίσης, θεωρείται ότι προκειμένου για κάθε είδος ταινίας να εμφανίζεται μόνο ένας χρήστης, σε περίπτωση ισοπαλίας στο πλήθος των κριτικών επιλέγεται ο χρήστης με το μεγαλύτερο user_id.

1η αλληλουχία:

```
#key = “g” for movie_genres.csv
map(key, line):
    movie_id, genre = line.split(“,”)
    emit((movie_id, genre), None)


reduce((movie_id, genre), list_values):
    emit(“g”, (movie_id, genre))
```

2η αλληλουχία:

```
#key = “m” for movies.csv
map(key, line):
    line = line.split(“,”)                    # works as given split_complex function
    movie_id = line[0]
    movie_name = line[1]
```

```
        popularity = float(line[7])
        emit(movie_id, (movie_name, popularity))

reduce(movie_id, list_values):
        # list_values contains only one item
        movie_name, popularity = list_values[0]
        emit("m", (movie_id, movie_name, popularity))
```

3η αλληλουχία:

```
#key = "r" for ratings.csv
map(key, line):
        line = line.split(",")
        movie_id = line[1]
        user = int(line[0])
        rating = float(line[2])
        emit((movie_id, user), rating)

reduce((movie_id, user), list_values):
        # list_values contains only one item
        rating = list_values[0]
        emit("r", (movie_id, user, rating))
```

4η αλληλουχία:

```
# join results of 1st and 3rd sequence of Map-Reduce
map(key, value):
        if key = "g":
                movie_id, genre = value
                emit(movie_id, ("g", genre))
        else if key = "r":
                movie_id, user, rating = value
                emit(movie_id, ("r", user, rating))
```

```
reduce(movie_id, list_values):
        list_g = [genre for (x, genre) in list_values if x = “g”]
        list_r = [(user, rating) for (x, user, rating) in list_values if x = “r”]
        for genre in list_g:
                for user, rating in list_r:
                        emit(movie_id, (genre, (user, rating)))

map(movie_id, (genre, (user, rating))):
        emit((genre, user), (rating, movie_id))

reduce((genre, user), list_values):
        list_ratings = list_values
        emit((genre, user), list_ratings)

map((genre, user), list_ratings):
        ratings = list_ratings.length()
        emit(genre, (user, list_ratings, ratings))

reduce(genre, list_values):
        max_user, list_ratings, max_ratings = list_values[0]
        for x in list_values[1:]:
                ratings = x[2]
                user = x[0]
                if (ratings, user) > (max_ratings, max_user):
                        max_user, list_ratings, max_ratings = x
        emit(genre, (max_user, list_ratings, max_ratings))

map(genre, (max_user, list_ratings, max_ratings)):
        for rating, movie_id in list_ratings:
                emit((movie_id, rating, genre, max_user), max_ratings)

reduce((movie_id, rating, genre, max_user), list_values):
        # list_values contains only one item
```

```
        max_ratings = list_values[0]
        emit("gr",(movie_id, rating, genre, max_user, max_ratings))

# join results of 2nd and current sequence of Map-Reduce
map(key, value):
        if key = "gr":
                movie_id, rating, genre, max_user, max_ratings = value
                emit(movie_id, ("gr", rating, genre, max_user, max_ratings))
        else if key = "m":
                movie_id, movie_name, popularity = value
                emit(movie_id, ("m", movie_name, popularity))

reduce(movie_id, list_values):
        list_gr = [(rating, genre, max_user, max_ratings) for (x, rating, genre, max_user,
        max_ratings) in list_values if x = "gr"]
        list_m = [ (movie_name, popularity) for (x, movie_name, popularity) in list_values if
        x = "m"]
        for rating, genre, max_user, max_ratings in list_gr:
                for movie_name, popularity in list_m:
                        emit((genre, max_user, max_ratings), (movie_name, rating, popularity))
```

5η αλληλουχία:

```
# on the result of 4th sequence
map(key, value):
        emit(key, value)

reduce((genre, max_user, max_ratings), list_values):
        movie_name, best_rating, best_popularity = list_values[0]
        for x in list_values[1:]:
                rating = x[1]
                popularity = x[2]
                if (rating, popularity) > (best_rating, best_popularity):
                        movie_name, best_rating, best_popularity = x
```

emit(“b”, ((genre, max_user, max_ratings), (movie_name, best_rating, best_popularity)))

6η αλληλουχία:

on the result of 4th sequence

map(key, value):

emit(key, value)

reduce((genre, max_user, max_ratings), list_values):

movie_name, worst_rating, worst_popularity = list_values[0]

for x in list_values[1:]:

rating = x[1]

popularity = x[2]

if rating < worst_rating or (rating == worst_rating and popularity > worst_popularity):

movie_name, worst_rating, worst_popularity = x

emit(“w”, ((genre, max_user, max_ratings), (movie_name, worst_rating, worst_popularity)))

7η αλληλουχία:

join results of 5th and 6th sequence of Map-Reduce

map(key, value):

if key = “b”:

(genre, max_user, max_ratings), (movie_name, best_rating, best_popularity) = value

emit((genre, max_user, max_ratings), (“b”, movie_name, best_rating, best_popularity))

else if key = “w”:

(genre, max_user, max_ratings), (movie_name, worst_rating, worst_popularity) = value

emit((genre, max_user, max_ratings), (“w”, movie_name, worst_rating, worst_popularity))

reduce((genre, max_user, max_ratings), list_values):

list_b = [(movie_name, best_rating, best_popularity) for (x, movie_name, best_rating, best_popularity) if x = “b”]

#list will consist of one item

```
        list_w = [(movie_name, worst_rating, worst_popularity) for (x, movie_name,
        worst_rating, worst_popularity) if x = "w"]

        #list will consist of one item

        for a in list_b:

                for b in list_w:

                        emit((genre, max_user, max_ratings), (a, b))


map(key, value):

        emit("result", (key, value))

reduce(key, list_values):

        # key = "result"

        list_values = sort(list_values, key=x[0]) # sort according to genre

        for (genre, max_user, max_ratings), ((best_movie, best_rating, _), (worst_movie,
        worst_rating, _)) in list_values:

                emit(None, (genre, max_user, max_ratings, best_movie, best_rating,
                worst_movie, worst_rating))
```

Ζητούμενο 4

Στο παρόν ερώτημα, εκτελέστηκαν οι παραπάνω υλοποιήσεις για τα 5 ερωτήματα με 3 διαφορετικούς τρόπους:

1. Map-Reduce Queries – RDD API
2. Spark SQL με είσοδο το *csv* αρχείο (συμπεριλαμβάνεται το *infer schema*)
3. Spark SQL με είσοδο το *parquet* αρχείο

Τα αποτελέσματα συμπεριλαμβάνονται στο φάκελο outputs εντός του φακέλου της εργασίας.

Σημειώνεται ότι η μορφή των εξόδων των υλοποιήσεων με το RDD API είναι η εξής:

1ο ερώτημα: (έτος, (ταινία με το μεγαλύτερο κέρδος, κέρδος))

2ο ερώτημα: το επιθυμητό ποσοστό

3ο ερώτημα: (είδος ταινίας, (μέση βαθμολογία είδους, πλήθος ταινιών είδους))

4ο ερώτημα: (πενταετία, μέσο μήκος περίληψης ταινιών)

5ο ερώτημα: (είδος ταινίας, χρήστης με τις περισσότερες κριτικές, πλήθος κριτικών, περισσότερο αγαπημένη ταινία, βαθμολογία περισσότερο αγαπημένης ταινίας, λιγότερο αγαπημένη ταινία, βαθμολογία λιγότερο αγαπημένης ταινίας)

Επίσης, η μορφή των εξόδων των υλοποιήσεων με το Spark SQL είναι η κάτωθι:

1ο ερώτημα: έτος, ταινία με το μεγαλύτερο κέρδος, κέρδος

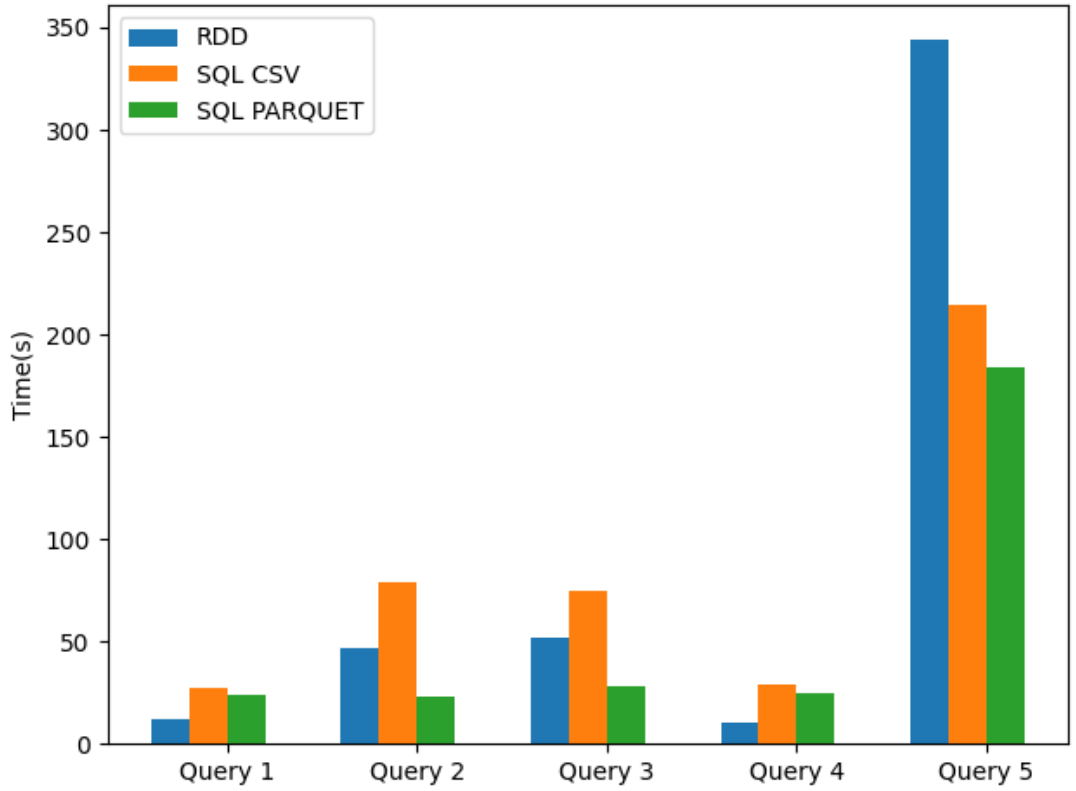
2ο ερώτημα: το επιθυμητό ποσοστό

3ο ερώτημα: είδος ταινίας, μέση βαθμολογία είδους, πλήθος ταινιών είδους

4ο ερώτημα: πενταετία, μέσο μήκος περίληψης ταινιών

5ο ερώτημα: είδος ταινίας, χρήστης με τις περισσότερες κριτικές, πλήθος κριτικών, περισσότερο αγαπημένη ταινία, βαθμολογία περισσότερο αγαπημένης ταινίας, λιγότερο αγαπημένη ταινία, βαθμολογία λιγότερο αγαπημένης ταινίας

Επιπλέον, για κάθε ερώτημα και για κάθε υλοποίηση, μετρήθηκε ο χρόνος που απαιτήθηκε για την εξαγωγή του εκάστοτε αποτελέσματος και με βάση αυτούς τους χρόνους, υλοποιήθηκε το ραβδόγραμμα στο οποίο οι χρόνοι είναι ομαδοποιημένοι ανά ερώτημα, όπως απεικονίζεται παρακάτω.



Το ραβδόγραμμα υλοποιήθηκε εκτελώντας το αρχείο plot.py που βρίσκεται στο φάκελο της εργασίας.

<u>Παρατηρήσεις-Σχόλια</u>

Η πρώτη εύλογη παρατήρηση που μπορεί να γίνει είναι η σημαντική διαφορά που παρουσιάζεται ως προς την επίδοση όσον αφορά τις υλοποιήσεις σε Spark SQL. Πιο συγκεκριμένα, είναι φανερό πως η υλοποίηση με είσοδο το *parquet* αρχείο υπερτερεί σε όλα τα ερωτήματα και ιδιαίτερα στα ερωτήματα 2,3 και 5 όπου η χρονική ψαλίδα είναι εντονότερη. Στα ερωτήματα αυτά, χρησιμοποιείται ο πίνακας *ratings* ο οποίος είναι κατά πολύ μεγαλύτερος από τους άλλους 2 πίνακες, με συνέπεια η διαφορά στην επίδοση για τα ερωτήματα αυτά να είναι καταφανής, καθώς όπως αναφέρθηκε και στην αρχή, η ανάγνωση από *parquet* αρχεία βελτιστοποιεί τη διαδικασία I/O και τη χρήση της μνήμης.

Μια επόμενη παρατήρηση αφορά τη χρήση ή όχι infer schema στις υλοποιήσεις σε Spark SQL. Αυτή αφορά την αναγνώριση του σχήματος, δηλαδή του τύπου δεδομένων κάθε στήλης των

πινάκων. Στην περίπτωση των *csv* αρχείων, χρειάστηκε η ενεργοποίησή της, η οποία όμως επιφέρει σημαντικές επιπτώσεις στο χρονικό κομμάτι και εν γένει στο κομμάτι της επίδοσης, καθώς απαιτεί μία επιπλέον ανάγνωση όλων των δεδομένων, κάτι το οποίο παίρνει χρόνο, ειδικά αν ο πίνακας είναι πολύ μεγάλος (όπως ο πίνακας *ratings*). Από την άλλη πλευρά, κατά την ανάγνωση των δεδομένων από αρχείο *parquet* δεν απαιτείται ανάλογη ρητή ενεργοποίηση του infer schema, καθώς αυτόματα κατά τη δημιουργία του αρχείου εξάγεται το σχήμα για τις διάφορες στήλες του πίνακα και συμπεριλαμβάνεται στα μεταδεδομένα του. Επομένως, μια δεύτερη υπερίσχυση της ανάγνωσης από *parquet* αρχεία έναντι της ανάγνωσης από *csv* αρχεία είναι εμφανής.

Επιπλέον, όσον αφορά την υλοποίηση με τη χρήση RDD API, για τα ερωτήματα που χρησιμοποιούνται οι 2 μικροί πίνακες και όχι ο μεγάλος πίνακας *ratings*, η χρήση RDD φαίνεται πως υπερέχει έναντι των 2 υλοποιήσεων με Spark SQL. Η υπεροχή όμως αυτή φαίνεται πως εξαφανίζεται όταν «μπαίνει στο παιχνίδι» και ο πίνακας *ratings* και γενικά όταν αυξάνεται ο όγκος των δεδομένων (εκτελώντας πολλά διαδοχικά join), αφού σε αυτές τις περιπτώσεις η χρήση της Spark SQL αναδεικνύεται ανώτερη λόγω του γεγονότος ότι με στόχο τη βέλτιστη επίδοση επιστρατεύει τον Catalyst Optimizer, το βελτιστοποιητή ερωτημάτων που διαθέτει (π.χ. επιλέγει την καλύτερη υλοποίηση join ανάλογα με το πλήθος των δεδομένων προς συνένωση).

Συμπερασματικά, θα μπορούσε να ειπωθεί πως την καλύτερη συμπεριφορά ως προς το κομμάτι της επίδοσης έχει η υλοποίηση Spark SQL με ανάγνωση από *parquet* αρχεία, η οποία παρουσιάζει και μια σταθερή και συγκρατημένη αύξηση στο χρονικό κομμάτι, εν αντιθέσει με την υλοποίηση με τη χρήση RDD, η οποία στο ερώτημα 5 ξεφεύγει σημαντικά στον τομέα του χρόνου, εξαιτίας της πολύ απότομης αύξησης του όγκου των δεδομένων.

Μέρος 2

Ζητούμενο 1

Στο παρόν ερώτημα, υλοποιήθηκε το broadcast join στο RDD API (Map-Reduce). Κατά την υλοποίησή του, όπως αναφέρεται και στην εκφώνηση της άσκησης, θεωρήθηκε πως ο ένας πίνακας είναι πάντα αρκετά μικρός ώστε να μπορεί να γίνει broadcast ολόκληρος, επομένως αγνοήθηκε το κομμάτι του αλγορίθμου το οποίο διασπάει τον πίνακα σε μικρότερα μέρη και εκτελεί διαδοχικά broadcast. Η υλοποίηση του broadcast join διατίθεται στο αρχείο *broadcast_join.py* εντός του φακέλου *joins* που εμπεριέχεται στο φάκελο της εργασίας και βασίζεται στον ψευδοκώδικα Α.4 της δημοσίευσης *"A Comparison of Join Algorithms for Log Processing in MapReduce", Blanas et al , in Sigmod 2010*.

Ζητούμενο 2

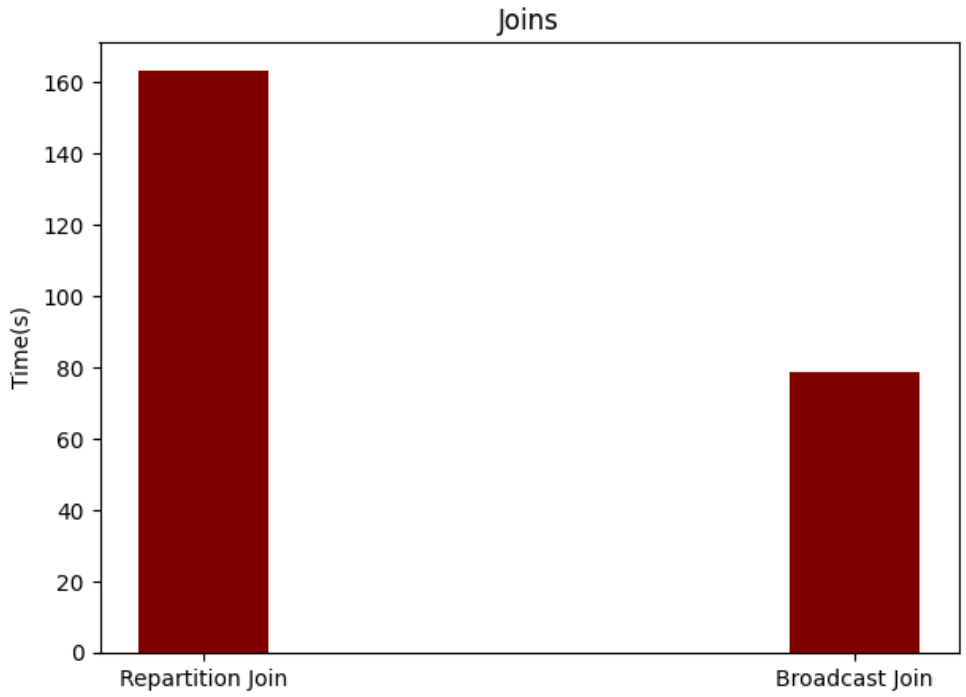
Έπειτα, υλοποιήθηκε το repartition join στο RDD API (Map-Reduce). Η υλοποίηση του διατίθεται στο αρχείο *repartition_join.py* εντός του φακέλου *joins* που εμπεριέχεται στο φάκελο της εργασίας και βασίζεται στον ψευδοκώδικα Α.1 της δημοσίευσης *"A Comparison of Join Algorithms for Log Processing in MapReduce", Blanas et al , in Sigmod 2010*.

Ζητούμενο 3

Αρχικά, με τη χρήση των εντολών:

- tail -100 movie_genres.csv > genres_100.csv
- hadoop fs -put genres_100.csv hdfs://master:9000/files/

απομονώθηκαν οι τελευταίες 100 γραμμές του πίνακα *movie_genres* σε ένα άλλο *csv* αρχείο με όνομα *genres_100.csv*. Το αρχείο αυτό γίνεται join με το αρχείο *ratings.csv*, χρησιμοποιώντας τις 2 μορφές join που υλοποιήθηκαν στα παραπάνω ερωτήματα (broadcast join και repartition join). Για τις 2 διαφορετικές μορφές join, συγκρίνεται ο χρόνος εκτέλεσής τους και υλοποιείται το σχετικό ραβδόγραμμα, το οποίο απεικονίζεται παρακάτω:



Παρατηρήσεις-Σχόλια:

Από το παραπάνω διάγραμμα, είναι προφανής η χρονική απόκλιση που παρατηρείται ανάμεσα στις 2 μορφές join για τη συνένωση που υλοποιήθηκε στο παρόν ερώτημα. Πιο συγκεκριμένα, όπως αναμενόταν άλλωστε, το broadcast join παρουσιάζει πολύ καλύτερη επίδοση από το repartition join, καθώς ο πίνακας *movie_genres100* είναι πολύ μικρότερος από τον πίνακα *ratings*, κι επομένως μπορεί να γίνει broadcast σε όλα τα μηχανήματα, αποφεύγοντας τη μεταφορά και των 2 πινάκων σε ολόκληρο το δίκτυο (όπως κάνει το repartition join) κάτι το οποίο προκαλεί σημαντική επιδείνωση στο χρονικό κομμάτι.

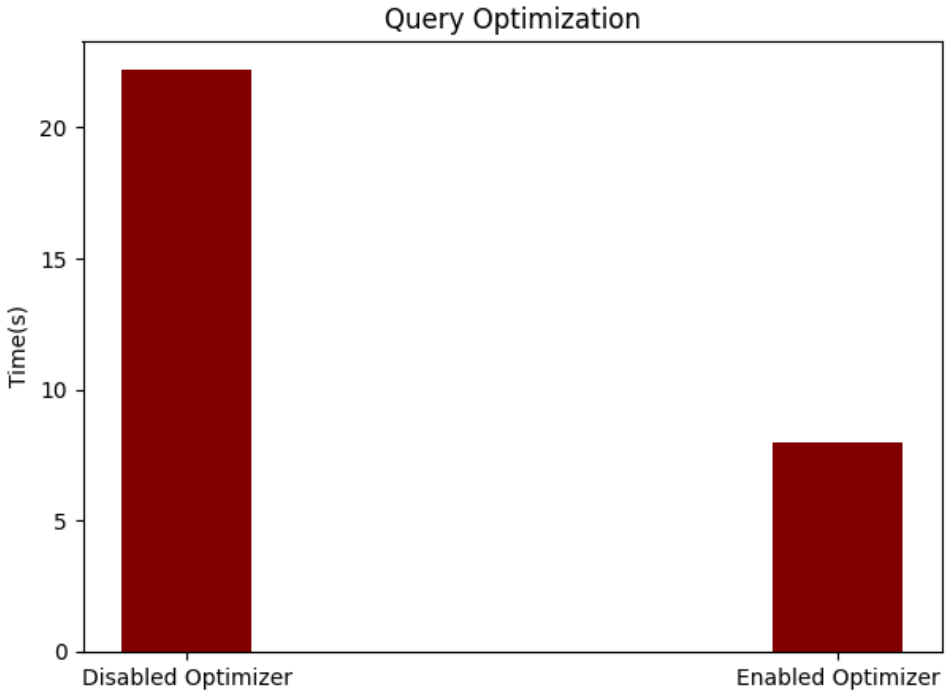
Ζητούμενο 4

Στο τελευταίο ερώτημα της εργασίας, έγιναν πειραματισμοί με τον query optimizer που παρέχει η Spark SQL για το κομμάτι των joins. Πιο συγκεκριμένα, ο βελτιστοποιητής ερωτημάτων της Spark SQL επιλέγει αυτόματα την υλοποίηση που θα χρησιμοποιήσει για ένα ερώτημα join, λαμβάνοντας υπόψιν το μέγεθος των δεδομένων. Επιπλέον, πολλές φορές αλλάζει και τη σειρά ορισμένων τελεστών προσπαθώντας να μειώσει τον συνολικό χρόνο εκτέλεσης του ερωτήματος. Στην περίπτωσή μας, η βελτιστοποίηση που εκτελεί είναι πως αν ο ένας πίνακας είναι αρκετά μικρός (με βάση ένα όριο που ορίζει ο χρήστης), θα χρησιμοποιήσει το broadcast join, αλλιώς θα χρησιμοποιήσει το repartition join.

Με βάση το *script* που δόθηκε στην εκφώνηση της άσκησης, συμπληρώθηκαν τα κενά σημεία με σκοπό ο χρήστης να έχει τη δυνατότητα να απενεργοποιεί την επιλογή του join από το

βελτιστοποιητή. Πρακτικά, αυτό επιτυγχάνεται, θέτοντας στη μεταβλητή spark.sql.autoBroadcastJoinThreshold την τιμή -1. Το *script* παρέχεται στο αρχείο *optimizer.py* που βρίσκεται στο φάκελο *joins* εντός του φακέλου της εργασίας.

Τέλος, εκτελείται το παραπάνω query join με και χωρίς βελτιστοποιητή, καταγράφοντας τον αντίστοιχο χρόνο, τον οποίο παρουσιάζουμε σε ραβδόγραμμα παρακάτω:



Τα 2 πλάνα εκτέλεσης που χρησιμοποιούνται με και χωρίς τη δυνατότητα βελτιστοποίησης ερωτημάτων join απεικονίζονται παρακάτω:

“Yes” (disabled query optimizer)

```
== 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/f
iles/movie_genres.parquet], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<_c0:int,_c1:string>
+- *(5) Sort [_c1#1 ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(_c1#1, 200)
      +- *(4) Project [_c0#0, _c1#1, _c2#2, _c3#3]
         +- *(4) Filter isnotnull(_c1#1)
            +- *(4) FileScan parquet [_c0#0,_c1#1,_c2#2,_c3#3] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://master:
9000/files/ratings.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(_c1)], ReadSchema: struct<_c0:int,_c1:int,_c2:double,_c3:int
>
Time with choosing join type disabled is 22.1806 sec.
```

“No” (enabled query optimizer)

```
== 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/file
s/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/f
iles/ratings.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(_c1)], ReadSchema: struct<_c0:int,_c1:int,_c2:double,_c3:int>
Time with choosing join type enabled is 7.9827 sec.
```

Σχόλια-Παρατηρήσεις:

Το εύλογο συμπέρασμα που προκύπτει από το παραπάνω ραβδόγραμμα είναι πως η εκτέλεση του ερωτήματος με ενεργοποιημένη τη χρήση βελτιστοποιητή είναι σαφώς αποδοτικότερη από την εκτέλεση του ερωτήματος με απενεργοποιημένη τη χρήση βελτιστοποιητή. Αυτό είναι προφανές, καθώς στην πρώτη περίπτωση ο βελτιστοποιητής αντιλαμβάνεται πως ο πίνακας *movie_genres100* είναι πολύ μικρός, επομένως επιλέγει να κάνει broadcast hash join, το οποίο φυσικά αποδίδει εξαιρετικά σε αυτήν την περίπτωση. Αντιθέτως, κατά την απενεργοποιημένη χρήση του βελτιστοποιητή, το σύστημα χρησιμοποιεί by default sort merge join, το οποίο απαιτεί φυσικά και ταξινόμηση των κλειδιών ώστε να γίνει το join ανάμεσα στα ζεύγη που έχουν ίδια κλειδιά.