ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Αναφορά Εργασίας του μαθήματος Προχωρημένα Θέματα Βάσεων Δεδομένων

Εξαμηνιαία Εργασία
2020-2021

Επιμέλεια:

Περόγαμβρος Γεώργιος, ΑΜ 03117024
Πουλίδης Στέφανος, ΑΜ 03114059

Μέρος 1ο: Υπολογισμός Αναλυτικών Ερωτημάτων με τα APIs του Apache Spark

Ζητούμενο 1

Αρχικά κατεβάσαμε το συμπιεσμένο αρχείο από τον σύνδεσμο http://www.cslab.ntua.gr/courses/atds/movie_data.tar.gz μέσω της εντολής:
wget --no-check-certificate 'http://www.cslab.ntua.gr/courses/atds/movie_data.tar.gz' -O movie_data.tar.gz

Αφού το κάναμε unzip με την εντολή: **tar -zxf movie_data.tar.gz**
και εκκινήσαμε το hdfs εκτελώντας: **start-dfs.sh**, δημιουργήσαμε τον φάκελο movie_data στο hdfs μέσω της εντολής: **hadoop fs -mkdir hdfs://master:9000/movie_data**. Σε αυτόν τον φάκελο ανεβάσαμε τα 3 csv αρχεία με τις εντολές:
hadoop fs -put movies.csv hdfs://master:9000/movie_data,
hadoop fs -put movie_genres.csv hdfs://master:9000/movie_data και
hadoop fs -put ratings.csv hdfs://master:9000/movie_data.

Ζητούμενο 2

Προκειμένου να μετατρέψουμε τα csv αρχεία σε parquet χρησιμοποιήσαμε ένα script στο οποίο αφού δημιουργήσουμε ένα spark session για να έχουμε πρόσβαση στο csv reader του spark, διαβάζουμε τα αρχεία csv δημιουργώντας ένα dataframe για το καθένα και έπειτα τα αποθηκεύουμε ως αρχεία parquet.

Ζητούμενο 3

Παραθέτουμε τους ψευδοκώδικες σε Map Reduce για τις υλοποιήσεις με RDD API του πρώτου μέρους

Q1:

```
map(key, value) {
        record = String(value)
        parts = record.split(“,”)
        year = int(get_year(parts[3]))
        income = int(parts[6])
```

```
        outcome = int(parts[5])
        title = parts[1]
        if (year != Null and year >= 2000 and income != 0 and outcome != 0)
                profit = (income-outcome)*100/outcome
                emit(year, (title, profit))
}

reduce(key, values) {
        max = -1
        for each value in values
                if value[1] > max
                max = value[1]
                title = value[0]
        emit(key, (title, max))
}

map(key,value) {
        emit(1, (key, value))
}

reduce(key, values) {
        mergesort(values[0])
        for each value in values
                emit(value[0], value[1])
}
```

Q2:

```
map(key, value) {
        record = String(value)
        parts = record.split(“,”)
        userid = int(parts[0])
        rating = float(parts[2])
        emit(userid, (rating, 1))
}

reduce(key, values) {
        rating_sum = 0
        count = 0
        for each value in values
                rating_sum +=  value[0]
                count += value[1]
        emit(key, (rating_sum, count))
```

```
}
reduce(key, value) {
        emit(key, value[0]/value[1])
}

map(key,value) {
        emit(1, (key, value))
}

reduce(key, values) {
        mergesort(values[0])
        for each value in values
                emit(value[0], value[1])
}

map(key,value) {
        emit(1, (key, value))
}

reduce(key, values) {
        count = 0
        for each value in values
                count += 1
        for each value in values
                emit(value[0], (value[1], count))
}

map(key, value) {
        if (value[1] >= 3.0)
                emit(1, (key, value))
}

reduce(key, values) {
        count = 0
        for each value in values
                count += 1
        emit(count*100/value[1][1]))
}
```

Q3:

```
//ratings
map(key, value) {
        record = String(value)
```

```
        parts = record.split(“,”)
        movieid = int(parts[1])
        rating = float(parts[2])
        emit(movieid, (rating, 1)
}

//movie_genres
map(key, value) {
        record = String(value)
        parts = record.split(“,”)
        movieid = int(parts[0])
        genre = parts[1]
        emit(movieid, genre)
}

//ratings
reduce(key, values) {
        rating_sum = 0
        count = 0
        for each value in values
                rating_sum +=  value[0]
                count += value[1]
        emit(key, rating_sum/count))
}

broadcast_join(movie_genres, ratings)

map(key, value) {
        emit(value[0], (value[1], 1) // value[0] is genre and value[1] is the average
}

reduce(key, values) {
        rating_sum = 0
        count = 0
        for each value in values
                rating_sum +=  value[0]
                count += value[1]
        emit(key, (rating_sum/count, count))
}

map(key,value) {
        emit(1, (key, value))
}
```

```
reduce(key, values) {
        mergesort(values[0])
        for each value in values
                emit(value[0], value[1])
}
```

Q4:

```
//movies
map(key, value) {
        record = String(value)
        parts = record.split(“,”)
        movieid = int(parts[0])
        len_of_sum = wordcount(part[2])
        if (parts[3] != Null and parts[2] != Null)
                year = int(get_year(parts[3]))
                if year >= 2000 and year <= 2004
                        quinquennium = 2000
                elif year >= 2005 and year <= 2009
                        quinquennium = 2005
                elif year >= 2010 and year <= 2014
                        quinquennium = 2010
                elif year >= 2015 and year <= 2019
                        quinquennium = 2015
                else
                        quinquennium = 0
        if (quinquennium >= 2000)
                emit(movieid, (len_of_sum, quinquennium))
}

//movie_genres
map(key, value) {
        record = String(value)
        parts = record.split(“,”)
        movieid = int(parts[0])
        genre = parts[1]
        if (genre == “Drama”)
                emit(movieid, genre)
}

repartition_join(movie_genres, movies)
//value[0] is genre, value[1] is len_of_sum and value[2] is quinquennium

map(key, value) {
```

```
        emit(value[2], (value[1], 1))
}

reduce(key, values) {
        length_sum = 0
        count = 0
        for each value in values
                length_sum +=  value[0]
                count += value[1]
        emit(key, length_sum/count)
}

map(key,value) {
        emit(1, (key, value))
}

reduce(key, values) {
        mergesort(values[0])
        for each value in values
                emit(value[0], value[1])
}
```

Q5:

```
//movies
map(key, value) {
        record = String(value)
        parts = record.split(“,”)
        movieid = int(parts[0])
        title = parts[1]
        popularity = float(parts[7])
        emit(movieid, (title, popularity))
}

//ratings
map(key, value) {
        record = String(value)
        parts = record.split(“,”)
        movieid = int(parts[1])
        userid = int(parts[0])
        rating = float(parts[2])
        emit(movieid, (userid, rating))
}
```

```
//genres
map(key, value) {
	record = String(value)
	parts = record.split(“,”)
	movieid = int(parts[0])
	genre = parts[1]
	emit(movieid, genre)
}

repartition_join(broadcast_join(ratings, movies), genres)

map(key, value) {
	emit((value[4], value[0]), (((key, value[2], value[1], value[3]), (key,
value[2], value[1], value[3])), 1))
}// value[0] is userid, value[4] is genre, value[2] is title, value[1] is rating,
value[3] is popularity

reduce(key, values) {
	count = 0
	for each value in values
		count += value[1]
	y = values.top
	for each x in values
		if (x[0][0][2] > y[0][0][2] or (x[0][0][2] == y[0][0][2] and x[0][0][3] >
y[0][0][3])) and (x[0][1][2] < y[0][1][2] or (x[0][1][2] == y[0][1][2] and x[0][1][3] >
y[0][1][3])))
			res = ((x[0][0][0], x[0][0][1], x[0][0][2], x[0][0][3]), (x[0][1][0],
x[0][1][1], x[0][1][2], x[0][1][3])), count))
		elif ((x[0][0][2] > y[0][0][2] or (x[0][0][2] == y[0][0][2] and x[0][0][3] >
y[0][0][3])) and (y[0][1][2] < x[0][1][2] or (x[0][1][2] == y[0][1][2] and y[0][1][3] >
x[0][1][3])))
			res = ((x[0][0][0], x[0][0][1], x[0][0][2], x[0][0][3]), (y[0][1][0],
y[0][1][1], y[0][1][2], y[0][1][3]))
		elif((y[0][0][2] > x[0][0][2] or (x[0][0][2] == y[0][0][2] and y[0][0][3] >
x[0][0][3])) and (x[0][1][2] < y[0][1][2] or (x[0][1][2] == y[0][1][2] and x[0][1][3] >
y[0][1][3])))
			res = ((y[0][0][0], y[0][0][1], y[0][0][2], y[0][0][3]), (x[0][1][0],
x[0][1][1], x[0][1][2], x[0][1][3]))
		else
			res = ((y[0][0][0], y[0][0][1], y[0][0][2], y[0][0][3]), (y[0][1][0],
y[0][1][1], y[0][1][2], y[0][1][3]))
	emit (key, (res, 1))
}
```

```
map(key, value) {
	emit(key[0], (key[1], value[1], (value[0][0][1], value[0][0][2]),
(value[0][1][1], value[0][1][2])))
}

reduce(key, values) {
	y = values.top
	for x in values
		if (x[1]>y[1])
			res = (x[0], x[1], (x[2][0], x[2][1]), (x[3][0], x[3][1]))
		else
			res = (y[0], y[1], (y[2][0], y[2][1]), (y[3][0], y[3][1]))
	emit(key, res)
}

map(key,value) {
	emit(1, (key, value))
}

reduce(key, values) {
	mergesort(values[0])
	for each value in values
		emit(value[0], value[1])
}
```

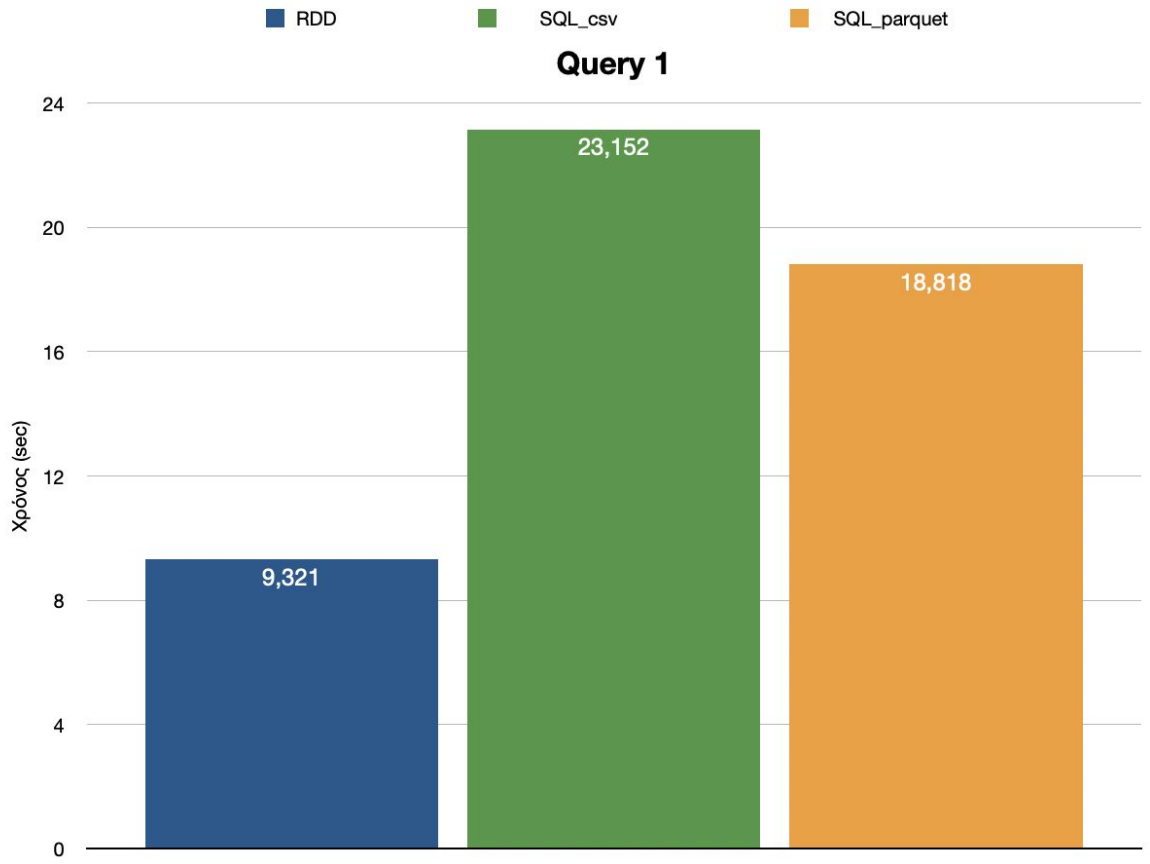
Παρατηρήσεις:

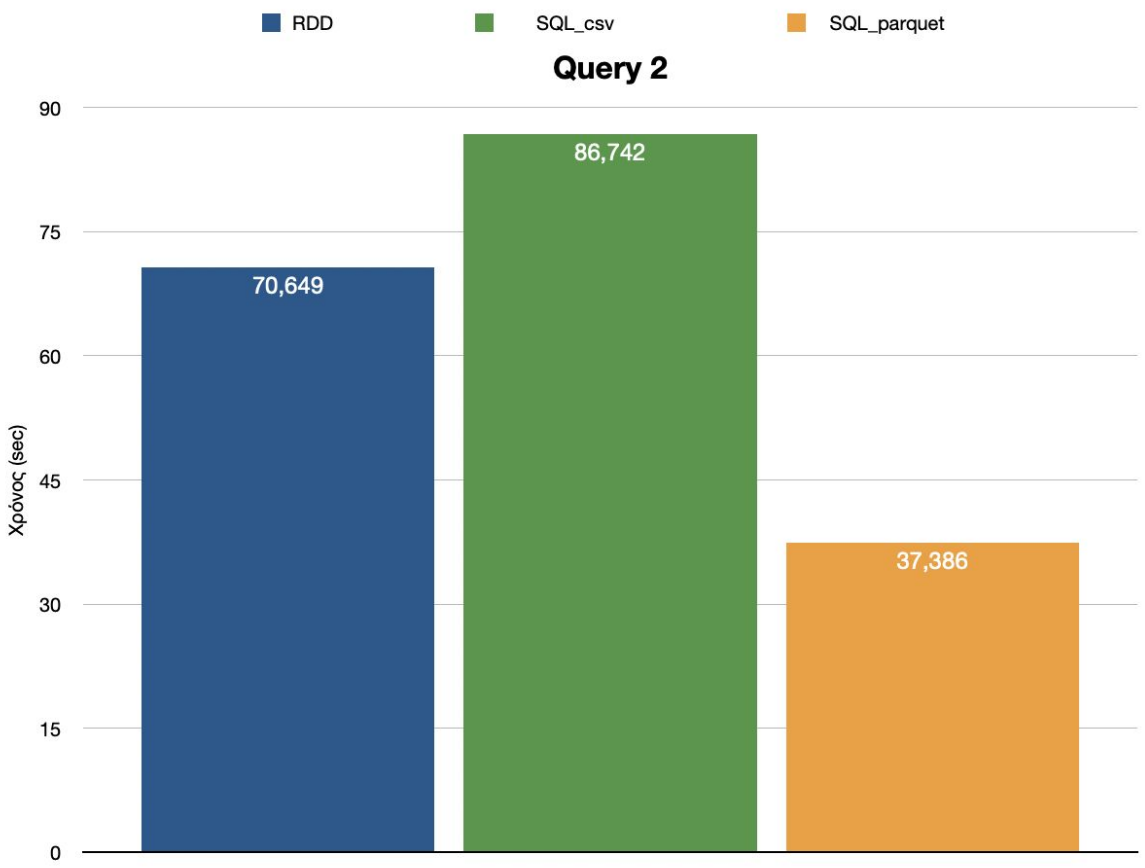
- Στην υλοποίηση του q1 με sparksql χρησιμοποιήσαμε ως κλειδί τον συνδυασμό έτους-κέρδους, καθώς θεωρήσαμε πως είναι πολύ δύσκολο δύο ταινίες να έχουν ακριβώς το ίδιο κέρδος και την ίδια χρονιά κυκλοφορίας
- Στην υλοποίηση του q4 δεν συμπεριλάβαμε τις κενές περιλήψεις
- Στην υλοποίηση του q5 στην περίπτωση που 2 χρήστες είχαν το ίδιο πλήθος κριτικών σε μια κατηγορία κρατήσαμε τις ταινίες του χρήστη με την υψηλότερη κριτική στην καλύτερη ταινία

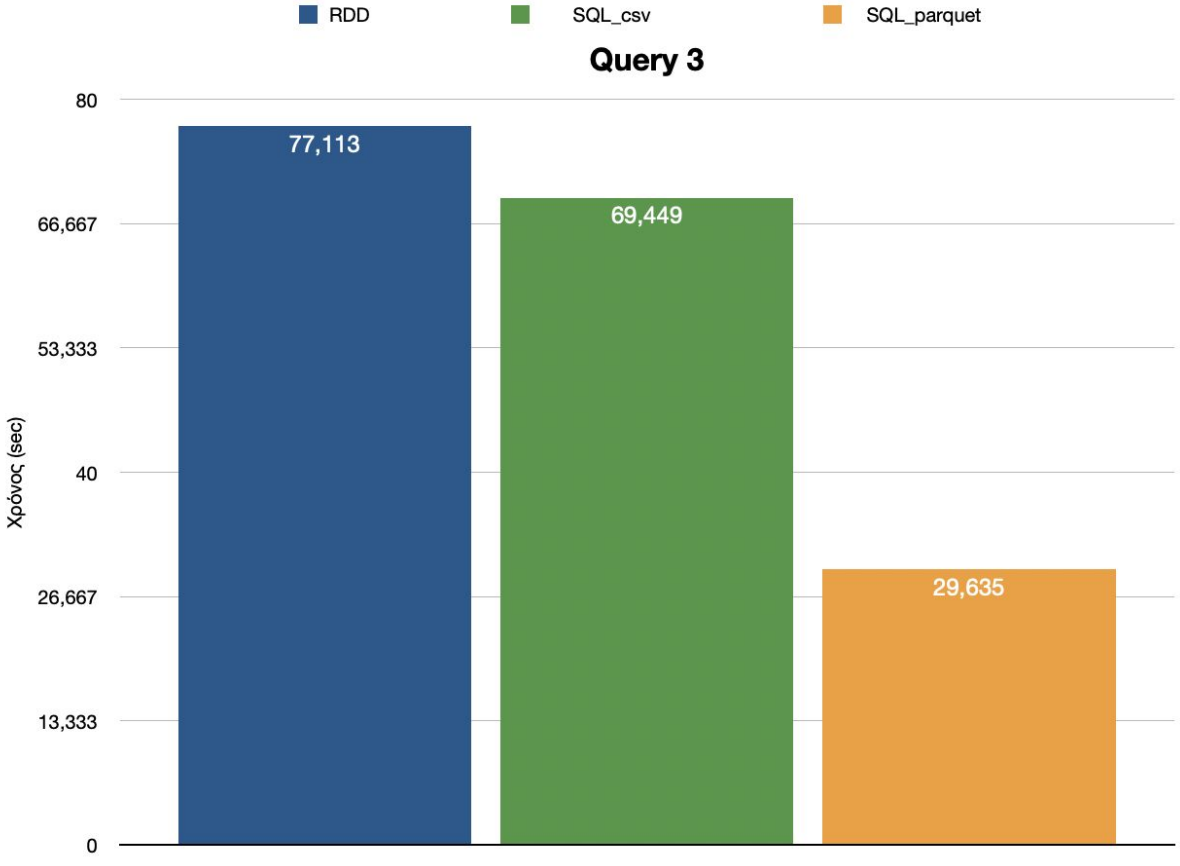
Ζητούμενο 4

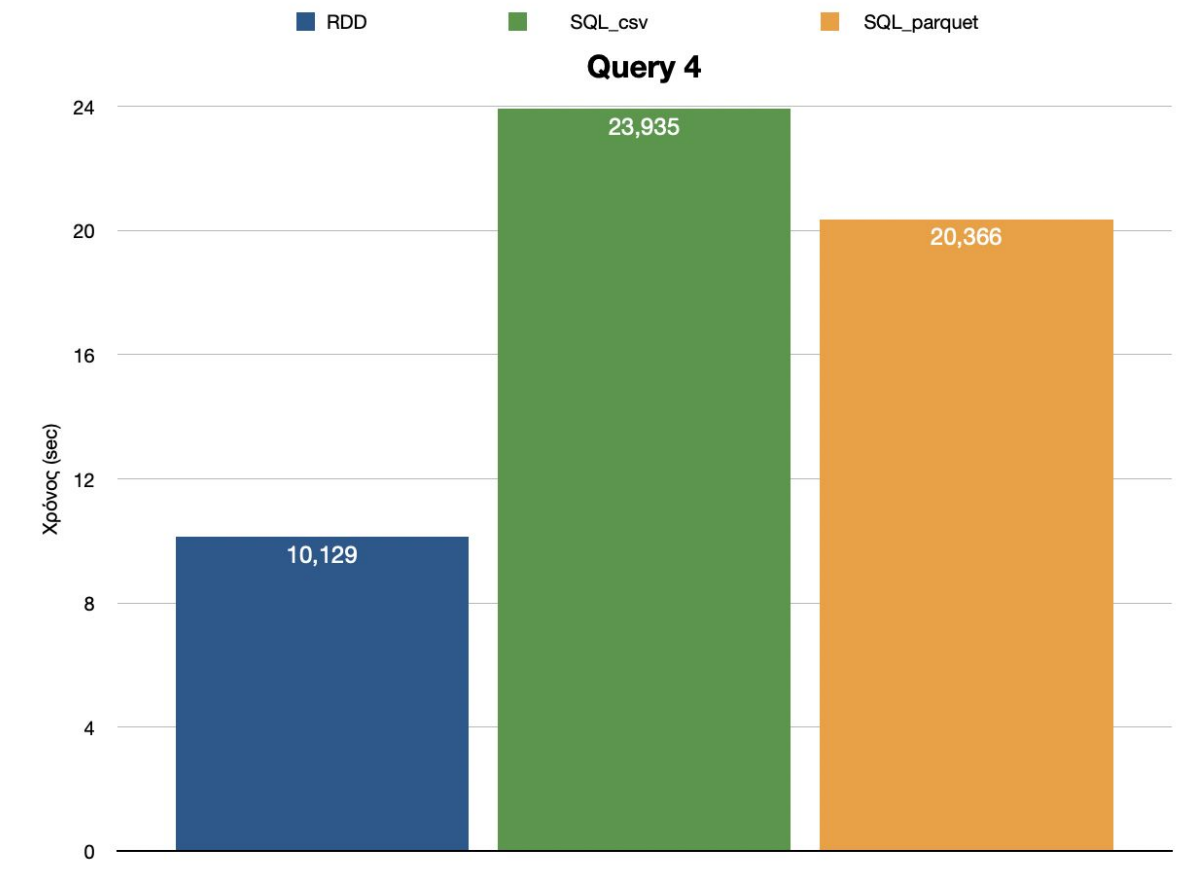
Όπως βλέπουμε από τους χρόνους εκτέλεσης των queries η υλοποίηση σε sparksql είναι πάντα πιο γρήγορη όταν έχουμε ως είσοδο αρχείο parquet σε σχέση με όταν έχουμε είσοδο αρχείο csv. Αυτό συμβαίνει, πρώτον, επειδή τα parquet αρχεία έχουν μικρότερο αποτύπωμα στη μνήμη και άρα βελτιστοποιούν τις I/O πράξεις. Ακόμα τα parquet κρατούν στατιστικές πληροφορίες πάνω στα dataset οπότε σε πράξεις που

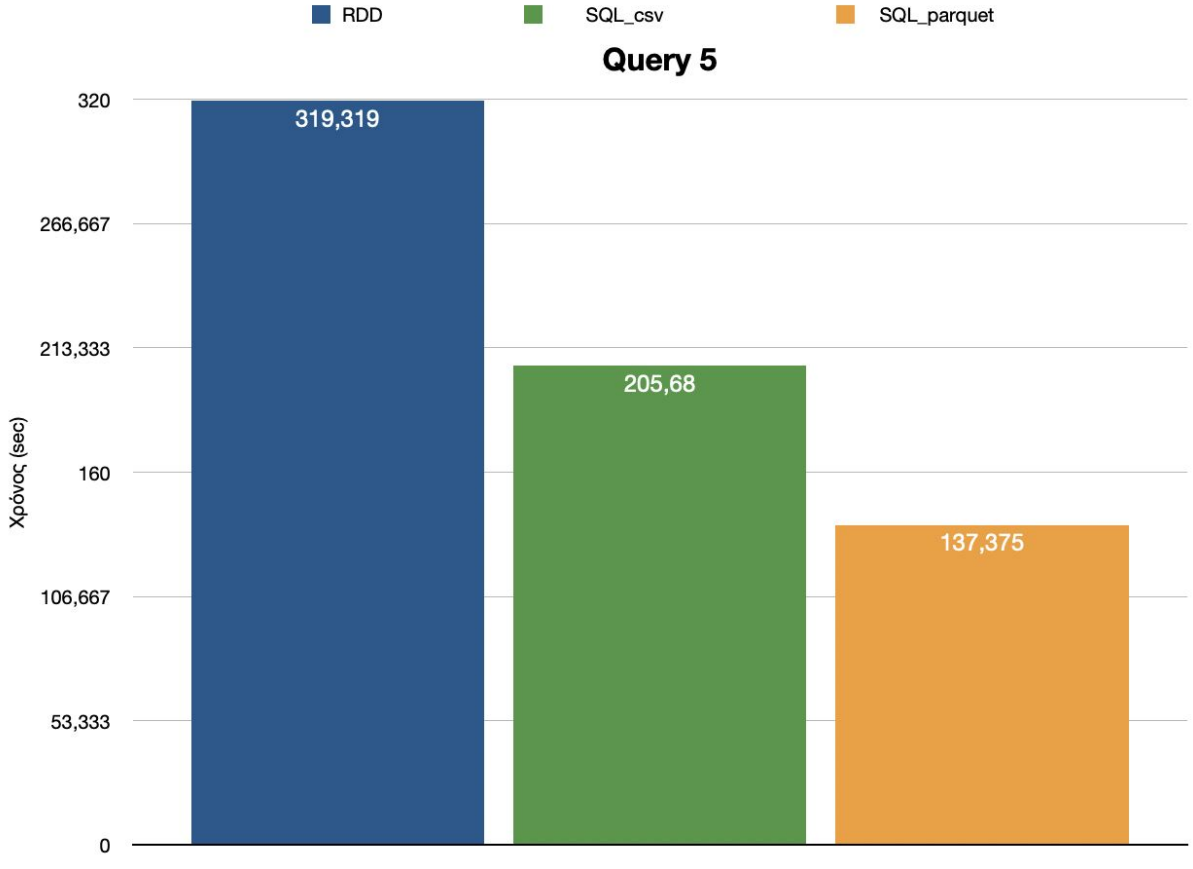
έχουν να κάνουν με συγκρίσεις, μέγιστα κτλ είναι πολύ πιθανό να ελεγχθούν ολόκληρα blocks του dataset πολύ γρήγορα. Επίσης, ο λόγος που δεν χρησιμοποιούμε inferSchema στα parquet αρχεία είναι πως αυτά έχουν από μόνα τους αποθηκευμένο τον τύπο των data για κάθε στήλη, σε αντίθεση με τα csv αρχεία που χρειάζονται το inferSchema κατά το διάβασμα για να γίνει ο συμπερασμός τύπων. Όσον αφορά τους χρόνους των rdd υλοποιήσεων σε σύγκριση με αυτές των sparksql, βλέπουμε πως στα περισσότερα queries, με εξαίρεση το q5 και το q3, η υλοποίηση με rdd είναι πιο γρήγορη από αυτή της sql με είσοδο αρχείο csv και στις περιπτώσεις των q1, q4 σημαντικά πιο γρήγορη και από της sql με είσοδο αρχείο parquet. Στην περίπτωση του q5, βέβαια, η υλοποίηση με rdd είναι σημαντικά πιο αργή από τις 2 άλλες υλοποιήσεις. Φυσικά σε κάποιες από αυτές τις περιπτώσεις μπορεί να παίζει ρόλο η χειρότερη υλοποίηση από μεριάς μας της sql ή του rdd.




RDD
SQL_csv
SQL_parquet
Query 2
90
75
60
45
30
15
0
Χρόνος (sec)
70,649
86,742
37,386


RDD
SQL_csv
SQL_parquet
Query 3
80
66,667
53,333
40
26,667
13,333
0
Χρόνος (sec)
77,113
69,449
29,635


RDD
SQL_csv
SQL_parquet
Query 4
Χρόνος (sec)
24
20
16
12
8
4
0
23,935
20,366
10,129


RDD
SQL_csv
SQL_parquet
Query 5
Χρόνος (sec)
320
266,667
213,333
160
106,667
53,333
0
319,319
205,68
137,375

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

Ζητούμενο 1

Ο κώδικας που υλοποιεί το Broadcast join, βρίσκεται στον φάκελο “code” όπως ζητήθηκε στην εκφώνηση, και όπου βρίσκονται όλοι οι κώδικες της παρούσας εργασίας.

Ζητούμενο 2

Ο κώδικας που υλοποιεί το Repartition join, βρίσκεται στον φάκελο “code” όπως ζητήθηκε στην εκφώνηση, και όπου βρίσκονται όλοι οι κώδικες της παρούσας εργασίας.

Ζητούμενο 3

Ο κώδικας που συγκρίνουμε και αξιολογούμε τα δύο joins, βρίσκεται στον φάκελο “code” όπως ζητήθηκε στην εκφώνηση, και όπου βρίσκονται όλοι οι κώδικες της παρούσας εργασίας.

Αποτελέσματα:

- **24.61 sec με Broadcast Join**
- **84.1 sec με Repartition Join**

Παρατηρούμε ότι είμαστε πολύ κοντά στο τι προβλέπει η δημοσίευση “A Comparison of Join Algorithms for Log Processing in MapReduce”, Blanas et al , in Sigmod 2010”, που στα πειραματικά τους δεδομένα βρήκαν ότι το Broadcast Join είναι περίπου 3 φορές πιο γρήγορο σε σχέση με το Repartition Join.

Εδώ, βλέπουμε ότι το Broadcast Join είναι 3.4 φορές πιο γρήγορο σε σχέση με το Repartition Join.
Αυτό προφανώς και το περιμέναμε, αφού ο πίνακας των ratings είναι πολύ μεγάλος, ενώ ο πίνακας των genres πολύ μικρός, αφού είναι μόνο 100 γραμμές ενός πίνακα-στήλη. Άρα, τα αποτελέσματα μας ταιριάζουν απόλυτα με τις θεωρητικές προβλέψεις.

Ζητούμενο 4

Ο τροποποιημένος κώδικας που μας δόθηκε βρίσκεται στον φάκελο “code” όπως ζητήθηκε στην εκφώνηση, και όπου βρίσκονται όλοι οι κώδικες της παρούσας εργασίας.

Και τρέχοντας το και στις δύο περιπτώσεις (με είσοδο “Y” και είσοδο “N”) έχουμε:

Για την περίπτωση που έχουμε ενεργοποιημένο τον βελτιστοποιητή:

- **Πλάνο εκτέλεσης:**

```
== 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/movie_data/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/movie_data/ratings.parquet], PartitionFilters:
[], PushedFilters: [IsNotNull(_c1)], ReadSchema:
struct<_c0:int,_c1:int,_c2:double,_c3:int>
```

- **Χρόνος:**

Time with choosing join typedisabled is 3.8587 sec.

Για την περίπτωση που έχουμε απενεργοποιημένο τον βελτιστοποιητή:

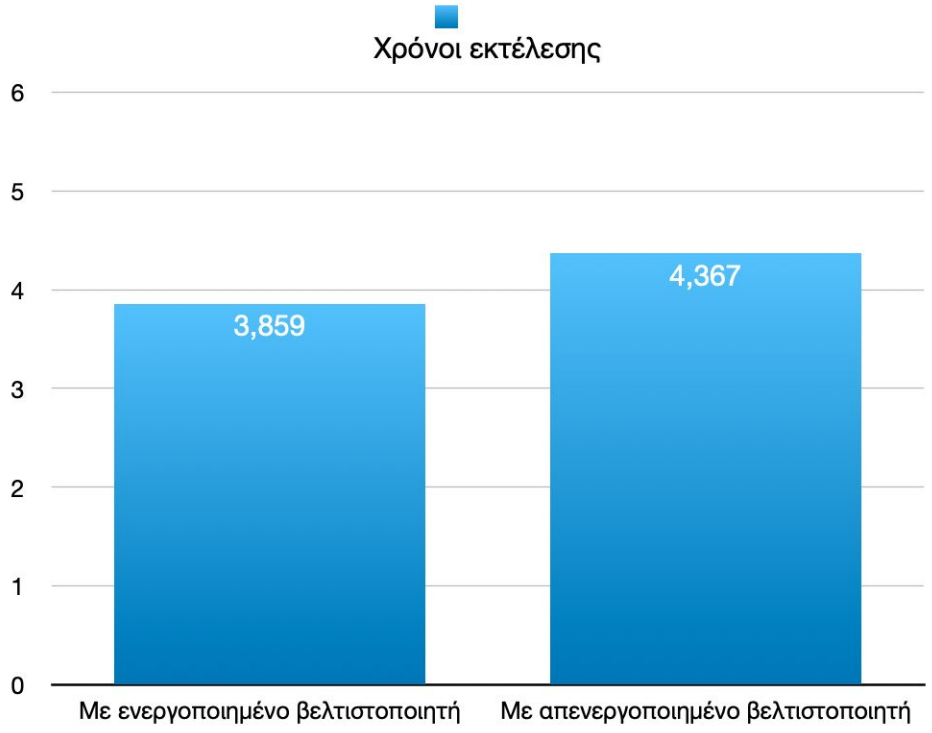
- **Πλάνο εκτέλεσης:**

```
== 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/movie_data/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/movie_data/ratings.parquet], PartitionFilters:
[], PushedFilters: [IsNotNull(_c1)], ReadSchema:
struct<_c0:int,_c1:int,_c2:double,_c3:int>
```

- **Χρόνος:**

Time with choosing join typeenabled is 4.3673 sec

Ραβδόγραμμα:



Παρατηρούμε ότι με ενεργοποιημένο τον βελτιστοποιητή προφανώς και έχουμε αρκετά καλύτερο χρόνο εκτέλεσης (περίπου 11% ταχύτερο χρόνο εκτέλεσης).

Αυτό είναι πολύ λογικό, αφού ο βελτιστοποιητής λαμβάνει υπόψιν του το μέγεθος των δεδομένων (που παίζουν σημαντικό ρόλο στην ταχύτητα εκτέλεσης) και πολλές φορές αλλάζει και την σειρά ορισμένων τελεστών προσπαθώντας να μειώσει τον συνολικό χρόνο εκτέλεσης του ερωτήματος. Αν ο ένας πίνακας είναι αρκετά μικρός θα χρησιμοποιήσει το broadcast join, αλλιώς θα κάνει ένα repartition join, όπως είδαμε και στη δημοσίευση “A Comparison of Join Algorithms for Log Processing in MapReduce”, Blanas et al , in Sigmod 2010”, που στα πειραματικά τους δεδομένα βρήκαν ότι το Broadcast Join είναι περίπου 3 φορές πιο γρήγορο σε σχέση με το Repartition Join, για join ενός μεγάλος log table L με ένα πολύ μικρότερο reference table (στήλη) R.

Ολοκλήρωση αναφοράς.

Επιμέλεια:

Περόγαμβρος Γεώργιος, ΑΜ 03117024
Πουλίδης Στέφανος, ΑΜ 03114059