Προχωρημένα Θέματα Βάσεων Δεδομένων

Εξαμηνιαία Εργασία

Γεζεκελιάν Βικέν ΑΜ: 03116112
Μπακιρτζή Χριστίνα ΑΜ: 03116013
Ακ. έτος: 2020 – 2021

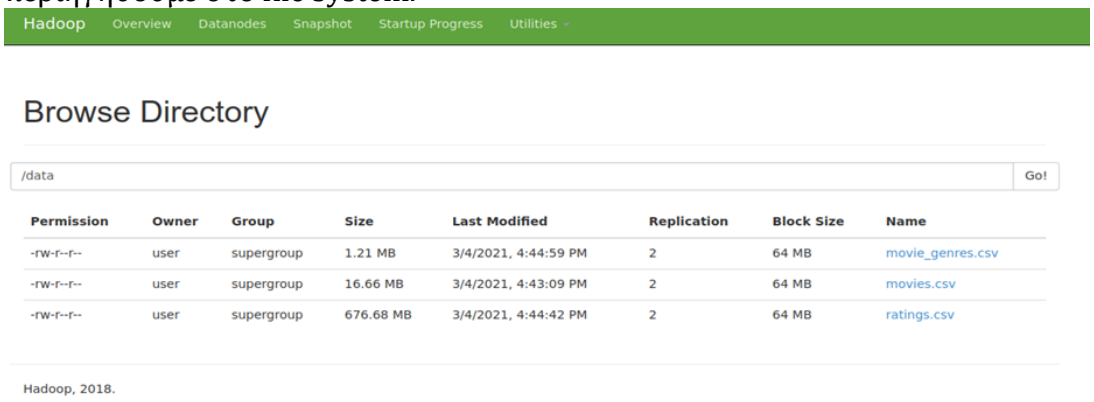
Μέρος Α

Ζητούμενο 1

Αρχικά φορτώσαμε τα CSV αρχεία movies.csv, movie_genres.csv και ratings.csv στο Hadoop file system, δημιουργώντας πρώτα ένα φάκελο "data" στο home directory:

```
user@master:~$ hadoop fs -mkdir hdfs://master:9000/data
user@master:~$ hadoop fs -ls hdfs://master:9000/.
Found 2 items
drwxr-xr-x   - user supergroup          0 2021-03-04 16:41 hdfs://master:9000/data
drwxr-xr-x   - user supergroup          0 2020-12-14 14:56 hdfs://master:9000/examples
user@master:~$ hadoop fs -put movies.csv hdfs://master:9000/data/
user@master:~$ hadoop fs -ls hdfs://master:9000/data/.
Found 1 items
-rw-r--r--   2 user supergroup   17466695 2021-03-04 16:43 hdfs://master:9000/data/movies.csv
user@master:~$ hadoop fs -put ratings.csv hdfs://master:9000/data/.
user@master:~$ hadoop fs -put movie_genres.csv hdfs://master:9000/data/.
user@master:~$ hadoop fs -ls hdfs://master:9000/data/.
Found 3 items
-rw-r--r--   2 user supergroup    1264187 2021-03-04 16:44 hdfs://master:9000/data/movie_genres.csv
-rw-r--r--   2 user supergroup   17466695 2021-03-04 16:43 hdfs://master:9000/data/movies.csv
-rw-r--r--   2 user supergroup  709550294 2021-03-04 16:44 hdfs://master:9000/data/ratings.csv
```

Αυτό επιβεβαιώνεται αν επισκεφθούμε τη σελίδα 83.212.74.74:50070 και περιηγηθούμε στο file system.



Ζητούμενο 2

Χρησιμοποιήσαμε το αρχείο csv2parquet.py για την μετατροπή των αρχείων CSV σε Parquet, έχοντας πλέον 6 αρχεία στο hdfs, 3 CSV και 3 Parquet.

Ζητούμενο 3

Υλοποιήσαμε διαφορετικές λύσεις για τα ερωτήματα που ζητήθηκαν χρησιμοποιώντας το RDD API και τη Spark SQL (η οποία διαβάζει είτε αρχεία Parquet είτε αρχεία CSV χρησιμοποιώντας το option inferSchema).

Όσον αφορά την υλοποίηση σε MapReduce, έχουν χρησιμοποιηθεί διάφορες λειτουργικότητες που μας δίνονται έτοιμος από το API. Συγκεκριμένα αυτές είναι οι:

Join: ο ψευδοκώδικας για την υλοποίηση της οποίας μας δίνεται έτοιμος στην εκφώνηση της εργασίας και σε αυτόν βασιστήκαμε για τις υλοποιήσεις τους μέρους 2ου.

SortBy: Πρακτικά πρόκειται για μια συνάρτηση η χρήση της οποίας δεν είναι απαραίτητη, καθώς στην πραγματικότητα γίνεται αυτόματα ανάμεσα στα Map και Reduce στάδια της διαδικασίας, στα στάδια Shuffle και Sorting του Hadoop.

Filter: Πρόκειται για ακόμη μια συνάρτηση η οποία στην πραγματικότητα δεν θα μας ήταν απαραίτητη, καθώς θα μπορούσαμε να θέσουμε τους περιορισμούς που επιθυμούμε εντός του Map σταδίου.

Παρακάτω παραθέτουμε τον ψευδοκώδικα σε MapReduce για κάθε υλοποίηση με RDD API, μαζί με τον κώδικα που χρησιμοποιήσαμε:

Query 1:

```
map(movies, value):
    for line in movies:
        data = line.split(',')
        if((data[3].split('-')[0]>=2000)
        and data[5]!=0 and data[6]!=0):
            name = data[1]
            year = data[3].split('-')[0]
            profit = ((data[6]-data[5])/data[5])*100
            emit(year, (name, profit))

reduce(year, list((name, profit)...)):
    maxname = '0'
    maxprofit = 0
    for pair in values:
        if (pair[1] > maximum)
            maxname = pair[0]
            maxprofit = pair[1]
    emit(year, (maxname, maxprofit))
```

Query 2:

```
map(ratings, value):
    for line in ratings:
        userID = line.split(',')[0]
        rating = line.split(',')[2]
        emit(userID, (rating, 1))

reduce(userID, list((rating, 1)...)):
    souma = count = 0
    for pair in values:
        souma += pair[0]
        count += pair[1]
    emit(userID, (souma, count))

map(userID, (souma, count)):
    if ((souma/count)>3):
        emit(1, (1, 1))
    else:
        emit(1, (0, 1))

reduce(key, list((kalos, 1)...)):
    souma = count = 0
    for pair in values:
        souma += pair[0]
        count += pair[1]
    emit(key, (souma, count))
#kalos παιρνει τιμες 0 ή 1 αναλογως με το εαν
#εχουμε user με avgRating>3

map(key, (souma, count)):
    emit(1, souma*100/count)
```

Query 3:

```
map(ratings, value):
    for line in ratings:
        movieID = line.split(',')[1]
        rating = line.split(',')[2]
        emit(movieID, (rating, 1))

reduce(movieID, list((rating, 1)...)):
    souma = count = 0
    for pair in values:
        souma += pair[0]
        count += pair[1]
    emit(movieID, (souma/count, count))
#souma/count = AvgMovieRating

map(movie_genres, value):
    for distinct(line) in movie_genres:
        movieID = line.split(',')[0]
        genre = line.split(',')[1]
        emit(movieID, genre)

Join(ratings, movie_genres)

map(movieID, ((AvgMovieRating, count), genre)):
    genre = values[1]
    AvgMovieRating = values[0][0]
    emit(genre, (AvgMovieRating, 1))

reduce(genre, list((AvgMovieRating, 1))):
    for pair in values:
        souma += pair[0]
        count += pair[1]
    emit(genre, (souma/count, count))
```

Query 4:

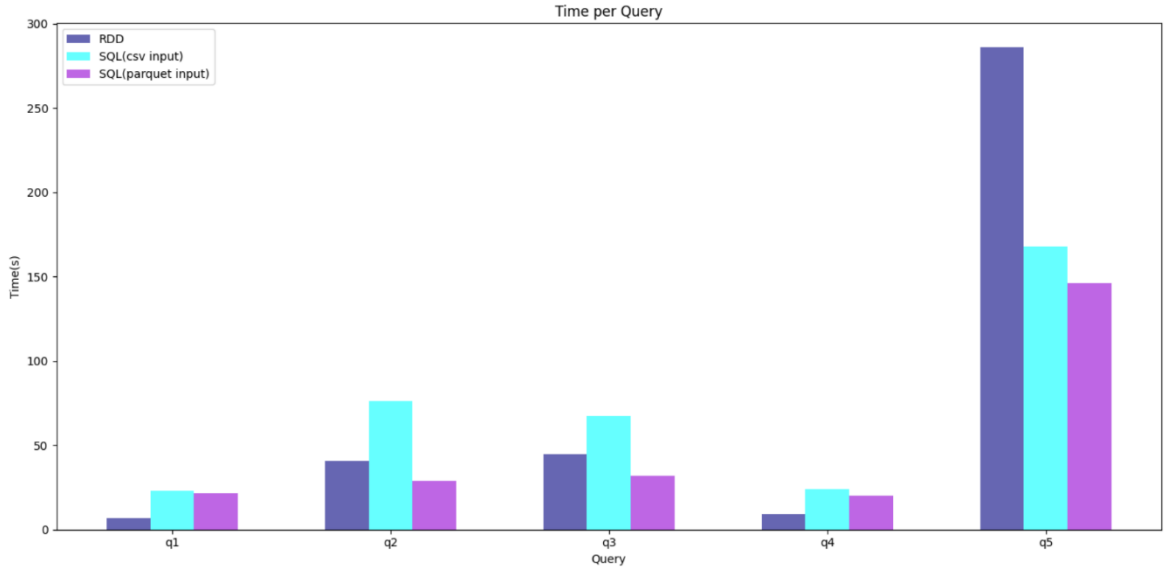
```
map(movies, value):
    for line in movies:
        data = line.split(',')
        if(2000<=(data[3].split('-')[0])<=2019):
            movieID = data[0]
            year = data[3].split('-')[0]
            summary = data[2]
            for word in summary.split(' '):
                emit((movieID, year), 1)

reduce((movieID, year), list(1)):
    for i in values:
        count += i
    emit(movieID, (year, count))
#(movieID, (year, #ofWords))

map(movie_genres, value):
    for line in movie_genres:
        if (line.split(',')[1] == 'Drama'):
            movieID = line.split(',')[0]
            genre = line.split(',')[1]
            emit(movieID, genre)

Join(movies, movie_genres)

map(movieID, ((year, numofWords), genre)):
    if (2000<=values[0][0]<=2004):
        period = "2000-2004"
    elif (2005<=values[0][0]<=2009):
        period = "2005-2009"
    elif (2010<=values[0][0]<=2014):
        period = "2010-2014"
    elif (2015<=values[0][0]<=2019):
        period = "2015-2019"
    numofWords = values[0][1]
    emit(period, (numofWords, 1))

reduce(period, list((numofWords, 1))):
    souma = count = 0
    for pair in values:
        souma += values[0]
        count += values[1]
    avgRatingLength = souma/count
    emit(period, avgRatingLength)
```

Query 5:

```
map(movie_genres, values):
    for distinct(line) in movie_genres:
        movieID = line.split(',')[0]
        genre = line.split(',')[1]
        emit(movieID, genre)

map(movies, values):
    for line in movies:
        movieID = data[0]
        name = data[3].split('-')[0]
        popularity = data[7]
        emit(movieID, (name, popularity))

movie_info = Join(movie_genres, movies)

map(movieID, (genre, (name, popularity))):
    emit(movieID, (genre, name, popularity))

map(ratings, values):
    for line in movie_genres:
        movieID = line.split(',')[1]
        userID = line.split(',')[0]
        rating = line.split(',')[2]
        emit(movieID, (userID, rating))

Join(movie_info, ratings)

map(movieID, ((genre, name, popularity), (userID, rating))):
    MaxName=MinName=name
    MaxRating=MinRating=rating
    MaxPopularity=MinPopularity=popularity
    emit((genre, userID), (1, MaxName, MaxRating, MaxPopularity, MinName, MinRating, MinPopularity))

reduce(key, list(values)):
    count = 0
    MaxRating=MaxPopularity=0
    MinRating=MinPopularity=inf
    for item in values:
        count += item[0]
        if ((item[2]>MaxRating) or ((item[2]==MaxRating) and (item[3]>MaxPopularity))):
            MaxName=item[1]
            MaxRating=item[2]
            MaxPopularity=item[3]
        if ((item[5]<MinRating) or ((item[5]==MinRating) and (item[6]>MinPopularity))):
            MinName=item[4]
            MinRating=item[5]
            MinPopularity=item[6]
    emit((genre, userID), (count, MaxName, MaxRating, MaxPopularity, MinName, MinRating, MinPopularity))

map((genre, userID), (count, MaxName, MaxRating, MaxPopularity, MinName, MinRating, MinPopularity)):
    emit(genre, (userID, count, MaxName, MaxRating, MinName, MinRating))

reduce(key, list(values)):
    MaxCount=0
    for item in values:
        if (item[1]>MaxCount):
            MaxCount = item[1]
            userID = item[0]
            MaxName = item[2]
            MaxRating = item[3]
            MinName = item[4]
            MinRating = item[5]
    emit(genre, (userID, MaxCount, MaxName, MaxRating, MinName, MinRating))
```

Ζητούμενο 4

Αφού εκτελέσαμε τις υλοποιήσεις του παραπάνω ζητούμενου για κάθε query, συγκεντρώσαμε τους χρόνους εκτέλεσης για όλες τις περιπτώσεις (MapReduce Query - RDD API, Spark SQL με είσοδο CSV αρχείο με χρήση του infer schema, Spark SQL με είσοδο parquet αρχείο) στο εξής ραβδόγραμμα:



Όπως φαίνεται και στο διάγραμμα, ως προς την SparkSQL για όλα τα queries, η χρήση Parquet αρχείων επιφέρει καλύτερα αποτελέσματα, όπως ήταν αναμενόμενο, καθώς οι Parquet πίνακες είναι αποθηκευμένοι σε ένα columnar format που βελτιστοποιεί το I/O και τη χρήση μνήμης, μειώνοντας έτσι το χρόνο εκτέλεσης, ενώ ταυτόχρονα διατηρεί παραπάνω πληροφορία για το dataset (π.χ. στατιστικά). Αντίθετα, ο υπολογισμός ερωτημάτων αναλυτικής επεξεργασίας απευθείας πάνω σε αρχεία csv δεν είναι ιδιαίτερα αποδοτικός. Χρησιμοποιεί το infer schema, το οποίο καθυστερεί παραπάνω, προσπαθώντας να μαντέψει αυτόματα τους τύπους δεδομένων για κάθε πεδίο. Ενεργοποιώντας το InferSchema ουσιαστικά το API θα διαβάσει κάποιες ενδεικτικές εγγραφές από το αρχείο για να βγάλει συμπεράσματα ως προς το σχήμα.

Συγκρίνοντας τις υλοποιήσεις με SQL και RDD API, παρατηρούμε ότι στα πιο απλά queries που χρειαζόμαστε μετασχηματισμούς και ενέργειες χαμηλού επιπέδου, καλύτερα αποτελέσματα δίνει η χρήση RDD API (με την αποθήκευση των data να μοιάζει με μνήμη cache και το Spark να μπορεί να έχει άμεση πρόσβαση σε αυτά) , ενώ για πιο περίπλοκα ερωτήματα στα οποία χρησιμοποιούμε περισσότερες από μία MapReduce διεργασίες, η βέλτιστη επιλογή είναι η χρήση της SQL. Η τελευταία διαθέτει αυτόματο βελτιστοποιητή σε αντίθεση με την λύση RDD, όπου η βελτιστοποίηση αφήνεται στα χέρια του προγραμματιστή. Γενικότερα, επιλέγουμε RDDs για καλύτερη λειτουργικότητα χαμηλού επιπέδου και έλεγχο, ενώ SQL όταν θέλουμε υψηλού επιπέδου και συγκεκριμένου τομέα λειτουργίες, λιγότερη δέσμευση χώρου και καλύτερους χρόνους σε πιο περίπλοκα ερωτήματα.

Πιο συγκεκριμένα, για το ερώτημα Q1, βλέπουμε ότι ταχύτερη εκτέλεση ήταν αυτή με RDD API, καθώς χρησιμοποιείται μόνο μια διεργασία map-reduce και το query είναι σχετικά απλό. Σαφώς ως προς την SparkSQL, η χρήση parquet αρχείων έναντι των csv επιφέρει καλύτερα αποτελέσματα, αν και η διαφορά είναι μικρή.

Ως προς το ερώτημα Q2, και πάλι η λύση που χρησιμοποιεί RDD API φαίνεται να είναι ταχύτερη από αυτή της SQL όταν χρησιμοποιείται ως είσοδος αρχείο CSV για τους παραπάνω λόγους, με την SQL - Parquet να αποτελεί βέβαια την ταχύτερη υλοποίηση.

Στο Q3, η υλοποίηση map-reduce γίνεται πιο περίπλοκη (όπως περιγράφηκε στον ψευδοκώδικα παραπάνω), επομένως αναμέναμε ότι η SparkSQL με χρήση Parquet αρχείων θα ήταν ταχύτερη στην εμφάνιση αποτελεσμάτων. Σε αυτό το ερώτημα, όπως και στα προηγούμενα, η πιο αργή λύση δείχνει να είναι αυτή της SparkSQL με χρήση csv.

Για το Q4 , η λύση που συντάξαμε με map-reduce συναρτήσεις φαίνεται να είναι η βέλτιστη. Ωστόσο το είδος των input αρχείων στη SparkSQL δεν φαίνεται να επηρεάζει σε μεγάλο βαθμό τα αποτελέσματα στο συγκεκριμένο ερώτημα, με την SQL - Parquet input να είναι ελάχιστα πιο γρήγορη. Εν γένει πρόκειται για ένα query που εκτελείται γρήγορα και με τις 3 υλοποιήσεις.

Ως προς το Q5, η λύση με RDD API είναι σαφέστατα πιο αργή από την χρήση SQL. Αυτό δικαιολογείται απόλυτα, καθώς πρόκειται για ένα ιδιαίτερα πολύπλοκο ερώτημα που απαιτεί τη χρήση αρκετών map-reduce διεργασιών και δέσμευση περισσότερης μνήμης. Η αυτόματη βελτιστοποίηση του SQL query στην περίπτωση των dataframes τους δίνει ένα σημαντικό προβάδισμα.

Επιβεβαιώνεται, λοιπόν, το γεγονός ότι είναι προτιμότερο να χρησιμοποιούμε RDD σε πιο απλά ερωτήματα, όταν αποσκοπούμε σε χαμηλού επιπέδου λειτουργίες. Τα RDDs επικεντρώνονται περισσότερο στο “τι” παρά στο “πώς” μιας λύσης - για query optimization χρησιμοποιούμε SQL.

Μέρος Β

Ζητούμενο 1

Υλοποιήσαμε το broadcast join στο RDD API όπως φαίνεται στον παρακάτω κώδικα:

```
from pyspark.sql import SparkSession
import csv
from io import StringIO
import time

start_time = time.time()

spark=SparkSession.builder.appName("broadcast_join").getOrCreate()
sc=spark.sparkContext

R_PATH = "hdfs://master:9000/data/movie_genres_100.csv"
L_PATH = "hdfs://master:9000/data/ratings.csv"

def split_complex(x):
        return list(csv.reader(StringIO(x), delimiter=','))[0]

#takes a key-value pair from L-data and searches the HashMap using that key.
#Makes tuples of (key, (r,val)) for every r-value stored in Hashmap under that key.
def combines(key, val):
        combined = []
        if BrMap.value.get(key, "-") == "-":
                return combined
        for r in BrMap.value.get(key):
                combined.append((key, (r, val)))
        return combined

#reads input file rows, makes (key, (values)) pairs,
#makes a list of the values for each key and creates a HashMap.
broadcast_data = sc.textFile(R_PATH). \
        map(lambda row: split_complex(row)). \
        map(lambda x : (x[0], (x[1]) ) ). \
        groupByKey(). \
        map(lambda x : (x[0], list(x[1]))). \
        collectAsMap()

#Broadcasting the small dataset
BrMap = sc.broadcast(broadcast_data)

#reads input file rows, makes (key,(values)) pairs, uses the "combines" function on them,
# adding each returned result to one list as join operation would.
result = \
        sc.textFile(L_PATH). \
        map(lambda row: split_complex(row)). \
        map(lambda x: ( x[1], (x[0], x[2], x[3]) )). \
        flatMap(lambda x: combines(x[0],x[1]))

print(result.collect())
print("Total time: {} sec".format(time.time()-start_time))
```

Ο κώδικας είναι προσαρμοσμένος στα δεδομένα του Ζητούμενου 3, αλλά γενικεύεται αλλάζοντας τις μεταβλητές L_PATH, R_PATH, τη θέση του κλειδιού και το πλήθος των υπόλοιπων attributes, και είναι βασισμένος στον ψευδοκώδικα του paper που μας δόθηκε, όπως αναφέρθηκε και προηγουμένως.

Ζητούμενο 2

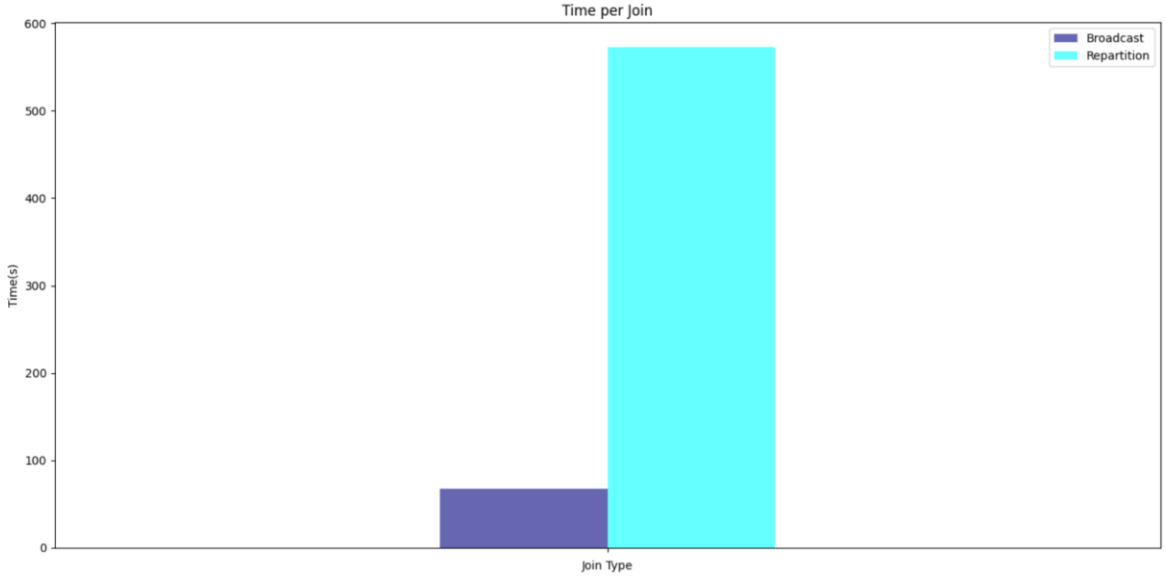
Υλοποιήσαμε το repartition join στο RDD API όπως φαίνεται στον παρακάτω κώδικα:

```
from pyspark.sql import SparkSession
import csv
from io import StringIO
import time
start_time = time.time()

spark=SparkSession.builder.appName("repartition_join").getOrCreate()
sc=spark.sparkContext

R_PATH = "hdfs://master:9000/data/movie_genres_100.csv"
L_PATH = "hdfs://master:9000/data/ratings.csv"

def split_complex(x):
        return list(csv.reader(StringIO(x), delimiter=','))[0]

def combines(key, list_of_values):
        length = int(len(list_of_values))
        b_r = []
        b_l = []
        for i in range(length):
                elem = list_of_values[i]
                if(elem[0] == 'R'):
                        b_r.append(elem[1:])
                elif(elem[0] == 'L'):
                        b_l.append(elem[1:])
        combined=[]
        for r in b_r:
                for l in b_l:
                        combined.append((key,(r+l)))
        return combined

#reads R input file, creates (key, ('R', values)) pairs (tagged with 'R')
R_data = \
        sc.textFile(R_PATH). \
        map(lambda row: split_complex(row)). \
        map(lambda x : (x[0],[('R', x[1])] ) )

#reads L input file, creates (key, ('L', values)) pairs (tagged with 'L')
L_data = \
        sc.textFile(L_PATH). \
        map(lambda row: split_complex(row)). \
        map(lambda x: (x[1],[('L', (x[0], x[2], x[3]) )]))

#unions both datasets, reduces by key to (key, list of values) pairs,
#uses "combines" function on them, adding each returned result to one list as join operation would
res = L_data.union(R_data). \
        reduceByKey(lambda x,y: x + y ). \
        flatMap(lambda x: combines(x[0], x[1]))

print(res.collect())
print("Total time: {} sec".format(time.time()-start_time))
```

Και αυτός ο κώδικας είναι προσαρμοσμένος στα δεδομένα του Ζητούμενου 3, αλλά γενικεύεται αλλάζοντας τις μεταβλητές L_PATH, R_PATH, τη θέση του κλειδιού και το πλήθος των υπόλοιπων attributes, και είναι βασισμένος στον ψευδοκώδικα του paper που μας δόθηκε.

Ζητούμενο 3

Αφού απομονώσαμε 100 γραμμές του πίνακα movie_genres σε ένα άλλο αρχείο csv, movie_genres_100, συνενώσαμε τον πίνακα με τον πίνακα ratings πρώτα με broadcast join και έπειτα με repartition join, όπως τα υλοποιήσαμε παραπάνω. Τα αποτελέσματά μας φαίνονται στο παρακάτω ραβδόγραμμα:



Παρατηρούμε μεγάλη διαφορά στο χρόνο εκτέλεσης των δύο υλοποιήσεων. Συγκεκριμένα, το broadcast join φαίνεται να είναι εξαιρετικά πιο γρήγορο από το repartition. Κάτι τέτοιο ήταν αναμενόμενο, δεδομένου του μικρού μεγέθους του πίνακα movie_genres. Το broadcast join θεωρείται πιο 2 αποδοτικό σε περίπτωση join ενός μεγάλου fact table και ενός σχετικά μικρότερου dimension table . Ουσιαστικά, το Spark στέλνει ένα αντίγραφο του μικρού πίνακα σε όλους τους executor nodes. Έτσι δεν χρειάζεται πλέον η στρατηγική επικοινωνίας όλων με όλους, καθώς κάθε executor node είναι self-sufficient στο να συνενώσει τα records του μεγάλου dataset που του έχουν ανατεθεί με το μικρό broadcasted table. Αντίθετα, στο repartition join, υλοποιείται μια map και μια reduce διαδικασία. Στη φάση map, κάθε record λαμβάνει μια ετικέτα που υποδεικνύει από ποιο σύνολο δεδομένων προέρχεται και εξάγεται το (key,value) ζευγάρι. Οι έξοδοι γίνονται partitioned, sorted και merged από το framework. Στη φάση reduce, για κάθε κλειδί, η συνάρτηση αρχικά χωρίζει και τοποθετεί σε buffers τις εγγραφές εισόδου σε δύο sets ανάλογα με την ετικέτα προέλευσής τους και έπειτα εξάγει κάθε δυνατό συνδυασμό μεταξύ των εγγραφών από τα δύο sets. Αυτό σημαίνει βέβαια ότι όλες οι εγγραφές για ένα δεδομένο join θα πρέπει να τοποθετηθούν σε buffers, γεγονός που μπορεί να προκαλέσει πρόβλημα μνήμης.

Συνεπώς, εφόσον στην προκειμένη περίπτωση έχουμε έναν μικρό πίνακα (movie_genres) ενώ ταυτόχρονα έχουμε μόνο δύο worker nodes, η λύση που προσφέρει ένα broadcast join σε σύγκριση με το repartition είναι προφανώς πιο συμφέρουσα και έτσι δεν θα χρειαστεί να τοποθετήσουμε σε buffers και όλες τις εγγραφές του εξαιρετικά μεγάλου table “records”. Όλο το μικρό table μπορεί να αποθηκευτεί στην RAM κάθε worker και να αποφύγουμε την επιβάρυνση του δικτύου, ολοκληρώνοντας τη διαδικασία του join από τη φάση map χωρίς τη χρήση reducers (narrow dependecy).

Ωστόσο αξίζει να σημειωθεί ότι αν αυξήσουμε τον αριθμό των executors που χρειάζεται να λάβουν ένα αντίγραφο του μικρού πίνακα, θα αυξηθεί και το κόστος του broadcasting, το οποίο σε ορισμένες περιπτώσεις θα μπορούσε να ξεπεράσει το κόστος και του ίδιου του join. Επιπλέον, στο broadcast join δεσμεύουμε διαθέσιμη μνήμη των executor nodes, το οποίο μπορεί πολλές φορές να οδηγήσει σε out of memory

exceptions με πίνακες μετρίου μεγέθους, είτε τώρα είτε στο μέλλον, καθώς ο πίνακας που αποστέλλεται σε όλους τους κόμβους μπορεί να μεγαλώσει και να προκύψουν εξαιρέσεις ελλιπούς μνήμης.

Ζητούμενο 4

Συμπληρώνοντας το script που μας δόθηκε, εκτελέσαμε το παρακάτω query σε SparkSQL με και χωρίς βελτιστοποιητή.

```
from pyspark.sql import SparkSession
import sys, time

disabled = sys.argv[1]
spark = SparkSession.builder.appName('join-sql').getOrCreate()

if disabled == "Y" :
        spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1 )
elif disabled =='N':
        pass
else:
        raiseException ("This setting is not available.")

df = spark.read.format("parquet")

df1 = df.load("hdfs://master:9000/data/ratings.parquet")
df2 = df.load("hdfs://master:9000/data/movie_genres.parquet")

df1.registerTempTable("ratings")
df2.registerTempTable("movie_genres")

sqlString =\
        "SELECT * "+\
        "FROM "+\
        "       (SELECT * FROM movie_genres LIMIT 100) as g, "+\
        "       ratings as r "+\
        "WHERE "+\
        "       r._c1 = g._c0"

t1 = time.time()
spark.sql(sqlString).show()

t2 =time.time()

spark.sql(sqlString).explain()

print("Time with choosing join type %s is %.4f sec."%("enabled" if disabled =='N' else "disabled", t2-t1))
```

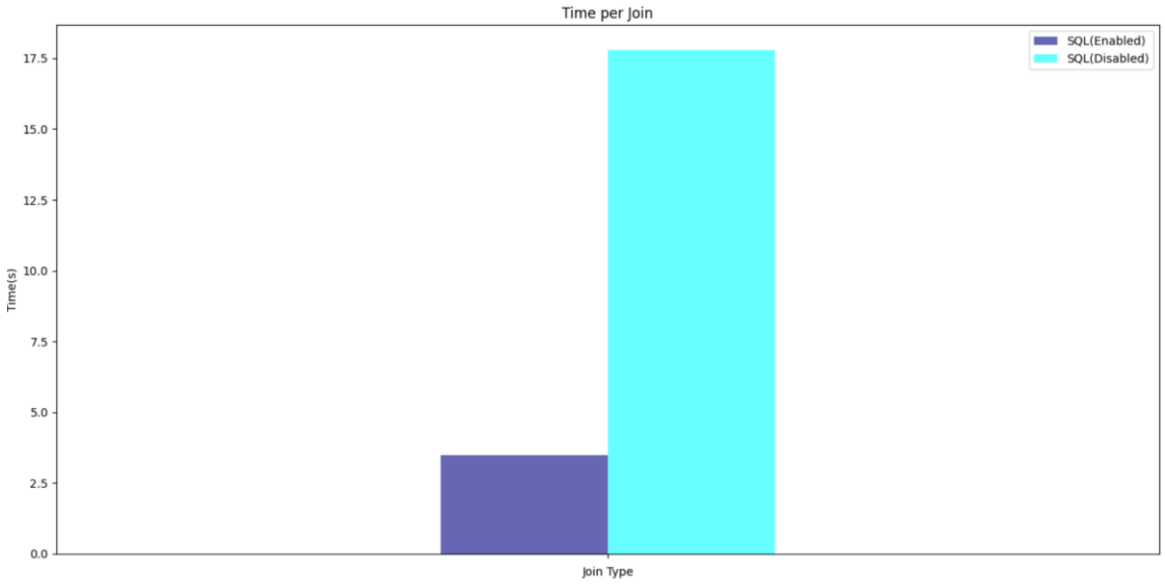
Εκτελώντας το αρχείο με ενεργοποιημένο βελτιστοποιητή, το πλάνο εκτέλεσης ήταν:

```
== 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/data/movie_genres.parquet], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<_c0:string,_c1:st
ring>
+- *(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[hd
fs://master:9000/data/ratings.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(_c1)], ReadSchema: struct<_c0:s
tring,_c1:string,_c2:string,_c3:string>
Time with choosing join type enabled is 5.3946 sec.
```

Ενώ χωρίς βελτιστοποιητή:

```
== 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[hd
fs://master:9000/data/movie_genres.parquet], PartitionFilters: [], PushedFilters: [], 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, Location: InMemoryFileIn
dex[hdfs://master:9000/data/ratings.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(_c1)], ReadSchema: struct
<_c0:string,_c1:string,_c2:string,_c3:string>
Time with choosing join type disabled is 16.3261 sec.
```

Παρατηρούμε ότι με χρήση αυτόματου βελτιστοποιητή, επιλέγεται το broadcast join ενώ χωρίς, γίνεται ένα Sort Merge Join. Αυτό επιβεβαιώνει τα παραπάνω, ότι η βέλτιστη επιλογή είναι το broadcast join στη δεδομένη περίπτωση.
Τα αποτελέσματα που λάβαμε ως προς τους χρόνους εκτέλεσης παρουσιάζονται στο εξής ραβδόγραμμα:



Είναι σαφές ότι το broadcast join, όπως προαναφέρθηκε, αποτελεί τη βέλτιστη υλοποίηση του join στη δεδομένη περίπτωση συνένωσης μικρού πίνακα με έναν αρκετά μεγαλύτερο.