ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΕΞΑΜΗΝΙΑΙΑ ΕΡΓΑΣΙΑ

ΡΟΗ Λ – ΕΞΑΜΗΝΟ 9ο
ΑΚ. ΕΤΟΣ 2021-2022
ΠΡΟΧΩΡΗΜΕΝΑ ΘΕΜΑΤΑ ΒΑΣΕΩΝ ΔΕΔΟΜΕΝΩΝ

Κύρου Μιχάλης – 03117710
Πεγειώτη Νάταλυ – 03117707

Χρήση του Apache Spark στις Βάσεις Δεδομένων

ΜΕΡΟΣ 1

Ψευδοκώδικας Map Reduce για τα ζητούμενα Queries

Q1

```
map(line_id, line):
        #line is in csv format
        #line comes from movies.csv file
        if (line.split(‘,’)[3] != ‘’ && int(line.split(‘,’)[3][0:4] >= 2000 && line.split(‘,’)[5] != ‘’ &&
        line.split(‘,’)[6] != ‘’)
        date = int(line.split(‘,’)[3][0:4]
        name = line.split(‘,’)[1]
        cost = int(line.split(‘,’)[5])
        income = int(line.split(‘,’)[6])
        profit = 100*((income-cost)/cost))
        emit(date, (name, profit))

reduce(date, tuple_list):
        max = 0
        name = ‘None’
        for tuple in tuple_list:
                if (tuple[1] > max):
                        max = tuple[1]
                        name = tuple[0]
        emit(date, name, max)
```

Q2

```
map(line_id, line):
        #line is in csv format
        #line comes from ratings.csv file
        user = line.split(‘,’)[0]
        rating = float(line.split(‘,’))[2]
        emit(user, rating)

reduce(date, rating_list):
        ratings = 0
        sum = 0
        for rat in rating_list:
                ratings += 1
                sum += rat
        avg_rating = sum/ratings
        if (avg_rating > 3.0):
                emit(‘same’, ‘high’)
        else:
                emit(‘same’, ‘all’)

map(key, value):
        emit(key, value)

reduce(_, value_list):
        all_users = value_list.length()
        users = 0
        for value in value_list:
                if (value == ‘high’):
                        users += 1
        percentage = 100*users/all_users
        emit(‘none’, percentage)
```

Q3

```
map(file_type, line):
        #file type is either ‘rating’ for ratings.csv or ‘genres’ for movie_genres.csv
        #line is in csv format
        if (file_type == ‘genres’):
                genre = line.split(‘,’)[0]
                movie_id = line.split(‘,’)[1]
                emit(movie_id, (‘g’, genre))
        else:
                movie_id = line.split(‘,’)[1]
                rating = float(line.split(‘,’)[2])
                emit(movie_id, (‘r’, rating))

reduce(movie_id, tuple_list):
        genres_list = []
        for tuple in tuple_list:
                if (tuple[0] == ‘g’):
                        genres_list.add(tuple[1]) // we think that l.add(x) adds x in list l
                        tuple_list.remove(tuple) // we think that l.remove(x) removes x from list l
        for genre in genres_list:
                for tuple in tuple_list:
                        emit((genre, movie_id), tuple[1])

map(key,value):
        emit(key,value)

reduce((genre, movie_id), rating_list):
        sum = 0
        ratings = 0
        for rat in rating_list:
                sum += rat
                ratings += 1
        avg_rating = sum/ratings
        emit(genre, avg_rating)

map(key, value):
        emit(key, value)

reduce(genre, rating_list):
        sum = 0
        ratings = 0
        for rat in rating_list:
                sum += rat
                ratings += 1
        avg_rating = sum/ratings
        emit(genre, avg_ratings)
```

Q4

```
map(file_type, line):
	#file type is either ‘moves’ for movies.csv or ‘genres’ for movie_genres.csv
	#line is in csv format
	#we think that .split() function works like given split_complex()
	if (file_type == ‘genres’):
		genre = line.split(‘,’)[0]
		movie_id = line.split(‘,’)[1]
		if(genre == ‘Drama’):
			emit(movie_id, (‘g’, genre))
	else:
		movie_id = line.split(‘,’)[0]
		length = len(line.split(‘,’)[2])
		if (line.split(‘,’)[3] != ‘’):
			date = int(line.split(‘,’)[3])
			if (date >= 2000 && date <= 2004):
				emit(movie_id, (‘m’, length, ‘2020-2004’))
			else if (date >= 2005 && date <= 2009):
				emit(movie_id, (‘m’, length, ‘2005-2009’))
			else if (date >= 2010 && date <= 2014):
				emit(movie_id, (‘m’, length, ‘2010-2014’))
			else if (date >= 2015 && date <= 2019):
				emit(movie_id, (‘m’, length, ‘2015-2019’))

reduce(movie_id, tuple_list):
	genres_list = []
	flag = false
	for tuple in tuple_list:
		if (tuple[0] == ‘g’):
			tuple_list.remove(tuple)
			flag = true
	if (flag == true):
	for tuple in tuple_list:
		emit(tuple[2], length)

map(key, value):
	emit(key, value)

reduce(date_stamp, length_list):
	sum = 0
	lengths = 0
	for len in length_list:
		sum += len
		lengths += 1
	avg_length = sum/lengths
	emit(date_stamp, avg_length)
```

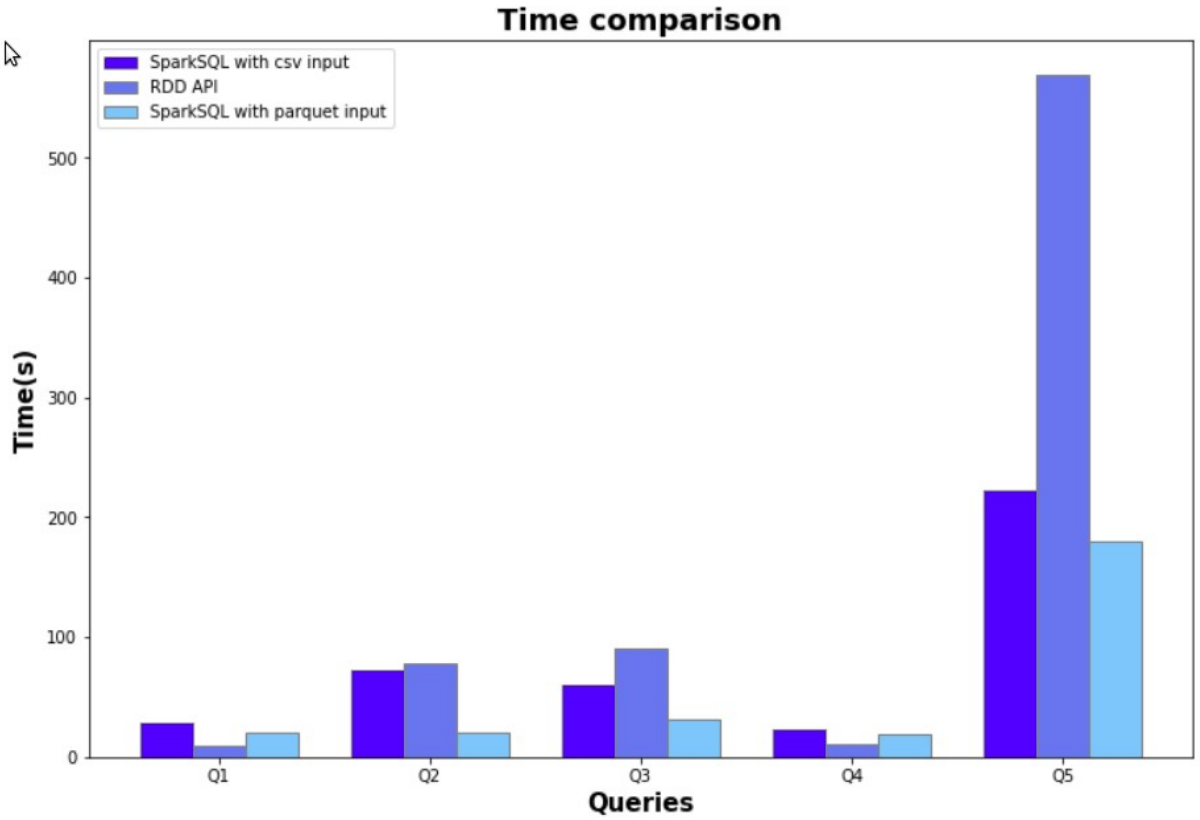
Q5

```
map(file_type, line):
        #file type is either ‘rating’ for ratings.csv or ‘genres’ for movie_genres.csv
        #line is in csv format
        if (file_type == ‘genres’):
                genre = line.split(‘,’)[0]
                movie_id = line.split(‘,’)[1]
                emit(movie_id, (‘g’, genre))
        else:
                movie_id = line.split(‘,’)[1]
                user_id = line.split(‘,’)[0]
                emit(movie_id, (‘r’, user_id))

reduce(movie_id, tuple_list):
        genres_list = []
        for tuple in tuple_list:
                if (tuple[0] == ‘g’):
                        genres_list.add(tuple[1]) // we think that l.add(x) adds x in list l
                        tuple_list.remove(tuple) // we think that l.remove(x) removes x from list l
        for genre in genres_list:
                for tuple in tuple_list:
                        emit((genre, tuple[1]), 1)


map(key,value):
        emit(key,value)

reduce((genre, user_id), user_id_list):
        ratings = 0
        for user in user_id_list:
                ratings += 1
        emit(genre, (user_id, ratings))


map(key, value):
        emit(key, value)

reduce(genre, tuple_list):
        max = 0
        user = ‘none’
        for tuple in tuple_list:
                if (tuple[1] > max):
                        max = tuple[1]
                        user = tuple[0]
        write_to_file(‘category_ratings.csv’, (genre, user, max) # we use write to file to keep the
                                                                 result
```

```
map(file_type, line):
	#file type is ‘movies’ for movies.csv, ‘genres’ for movie_genres.csv, or ‘ratings’ for
                                                                                ratings.csv
	if (file_type == ‘genres’):
		genre = line.split(‘,’)[0]
		movie_id = line.split(‘,’)[1]
		emit(movie_id, (‘g’, genre))
	else if(file_type == ‘ratings’):
		movie_id = line.split(‘,’)[1]
		rating = float(line.split(‘,’)[2])
		user_id = line.split(‘,’)[0]
		emit(movie_id, (‘r’, user_id, rating))
	else:
		movie_id = line.split(‘,’)[1]
		user_id = line.split(‘,’)[0]
		emit(movie_id, (‘r’, user_id))

reduce(movie_id, tuple_list):
	genres_list = []
	for tuple in tuple_list:
		if (tuple[0] == ‘g’):
			genres_list.add(tuple[1]) // we think that l.add(x) adds x in list l
			tuple_list.remove(tuple) // we think that l.remove(x) removes x from list l
		if (tuple[0] == ‘m’):
			movie_name = tuple[1]
			movie_popularity = tuple[2]
			tuple_list.remove(tuple) // we think that l.remove(x) removes x from list l
	for genre in genres_list:
		for tuple in tuple_list:
			emit((genre, tuple[1]), (movie_name, tuple[2], movie_popularity))

map(key, value):
	emit(key, value)

reduce((genre, user_id), tuple_list):
	max_rat = min_rat =0
	max_pop = 0
	min_pop = 0
	max_name = ‘none’
	min_name = ‘none’
	for tuple in tuple_list:
		if (max_rat < tuple[1] || (max_rat == tuple[1] && max_pop < tuple[2]):
			max_name = tuple[0]
			max_pop = tuple[2]
			max_rat = tuple[1]
		if (min_rat > tuple[1] || (min_rat == tuple[1] && min_pop < tuple[2]):
			min_name = tuple[0]
			min_pop = tuple[2]
			min_rat = tuple[1]
write_to_file(‘fav_worst_movie.csv’, (genre, user_id, max_name, max_rat, min_name, min_rat)
```

```
map(file_type, line):
	#file type is either ‘f_w_movie’ for fav_worst_movie.csv or ‘cat_rating’ for
                                                                category_rating.csv
	#line is in csv format
	if (file_type == ‘f_w_movie’):
		genre = line.split(‘,’)[0]
		user_id = line.split(‘,’)[1]
		fav_name = line.split(‘,’)[2]
		fav_rat = line.split(‘,’)[3]
		worst_name = line.split(‘,’)[4]
		worst_rat = line.split(‘,’)[5]
		emit((genre, user_id), (‘f_w’, fav_name, fav_rat, worst_name, worst_rat))
	else:
		genre = line.split(‘,’)[0]
		user_id = line.split(‘,’)[1]
		ratings = line.split(‘,’)[2]
		emit((genre, user_id), (‘c_r’, ratings))

reduce((genre, user_id), tuple_list):
	c_r_list = []
	for tuple in tuple_list:
		if (tuple[0] == ‘c_r’):
			c_r_list.add(tuple[1]) // we think that l.add(x) adds x in list l
			tuple_list.remove(tuple) // we think that l.remove(x) removes x from list l
	for ratings in c_r_list:
		for tuple in tuple_list:
			emit(genre, (user_id, ratings, tuple[1], tuple[2], tuple[3], tuple[4]))
```

Χρόνοι εκτέλεσης ομαδοποιημένοι ανά ερώτημα



** Οι χρόνοι λήφθηκαν από τον ακόλουθο σύνδεσμο http://83.212.79.75:8080 όπου μπορείτε να τους δείτε και εσείς.*

Όσον αφορά την σύγκριση με χρήση του RDD API, έναντι της SparkSQL λαμβάνουμε υπόψην τα συμπεράσματα που προκύπτουν από τα Queries που τρέχουν για κάποιο εύλογο χρονικό διάστημα (Q2, Q3, Q5). Ο λόγος είναι πως σε ερωτήματα με πολύ μικρή χρονική εκτέλεση, μια κακή υλοποίηση μπορεί να επηρεάσει σημαντικά το αποτέλεσμα, όσων αφορά τον χρόνο εκτέλεσης της υλοποίησης τους με την μια ή την άλλη μέθοδο. Αναγνωρίζουμε, λοιπόν, πως για τα Q1 και Q4 ενδεχομένως η ταχύτερη εκτέλεση της υλοποίησης με RDD API να οφείλεται, απλώς, σε μια όχι βέλτιστη υλοποίηση με το SparkSQL.

Είναι εμφανές πως η SparkSQL στα ερωτήματα Q2, Q3, Q5 εκτελείται σε πολύ λιγότερο χρόνο από το RDD API.

Η καλύτερη απόδοση της SparkSQL ήταν αναμενόμενη, αφού η ίδια ενσωματώνει έναν βελτιστοποιητή (Catalyst Optimizer), την καλύτερη απόδοση των queries. Συγκεκριμένα, ο βελτιστοποιητής είναι υπεύθυνος για την παραλλαγή των queries προκειμένου αυτά να τρέχουν με τον καλύτερο δυνατό τρόπο χρησιμοποιώντας τεχνικές όπως το φιλτράρισμα και την εύρεση του καλύτερου δυνατού τρόπου εκτέλεσης των Joins.
Επίσης, στη διαφορά ανάμεσα στη χρήση SparkSQL και RDD API συμβάλλει το γεγονός πως, διαδικασίες όπως το group ή το aggregate επί των δεδομένων είναι πολύ πιο βαριές όταν με χρήση RDD API, καθώς σε αυτή την περίπτωση υπάρχει το στάδιο της μεταφοράς δεδομένων μέσω του δικτύου που οδηγεί σε επιπλέον χρονική καθυστέρηση.

Οι εκτελέσεις με είσοδο σε csv μορφή είναι αρκετά κοντά σε αυτές με χρήση του RDD API, με εξαίρεση το Q5 για το οποίο παρατηρούμε περίπου τον τριπλάσιο χρόνο εκτέλεσης.
Οι εκτελέσεις με είσοδο parquet είναι οι βέλτιστες σε κάθε περίπτωση.
Αυτό συμβαίνει λόγω του μικρότερου αποτυπώματος αυτών των αρχείων στη μνήμη και στο δίσκο, με αποτέλεσμα την πολύ ταχύτερη ανάγνωση και εγγραφή τους, αλλά και λόγω των ιδιοτήτων αυτών των αρχείων.
Συγκεκριμένα, λόγω της μεταπληροφορίας που καταγράφουν παρέχουν πολλαπλές δυνατότητες βελτιστοποίησης της επεξεργασίας των queries, καθώς σε αρκετές περιπτώσεις το πλήθος των δεδομένων που χρειάζεται να σκαναριστούν μειώνεται σε πολύ μεγάλο βαθμό κι, έτσι, η ταχύτητα της εκτέλεσης των ερωτημάτων μειώνεται.

Όσον αφορά τη χρήση του option inferSchema κατά το διάβασμα αρχείων σε csv μορφή, αυτή επιβαρύνει ακόμα περισσότερο το χρόνο εκτέλεσης, καθώς απαιτεί ένα επιπλέον σκανάρισμα του αρχείου εισόδου, προκειμένου να αναγνωρίσει τον τύπο δεδομένων κάθε κολώνας και, έτσι, επιβραδύνει την ανάγνωση.
Παρ’όλ’αυτά προσφέρει ένα dataframe που έχει ορθά ορισμένους τους τύπους στις κολώνες, ωστόσο, στην περίπτωση δεδομένων, όπως αυτά που διαχειριζόμαστε, για τα οποία είμαστε σε θέση να εξάγουμε με ευκολία αυτή την πληροφορία χωρίς να χρειαστεί η διαδικασία να αυτοματοποιηθεί, είναι προτιμότερο να γλιτώσουμε την χρονική αυτή καθυστέρηση.

ΜΕΡΟΣ 2

Ψευδοκώδικας Map Reduce για τα ζητούμενα Joins

Broadcast Join
Preprocess(small_table): broadcast small table to all machines

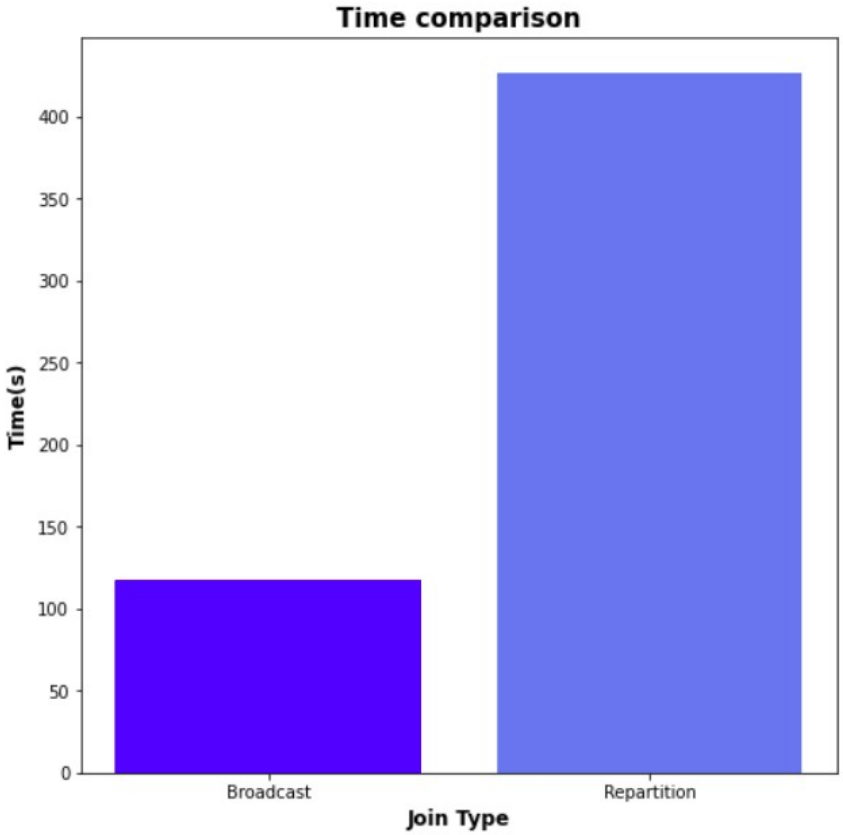
```
map(key, value):
    # key is the join key and value is a tuple with all the other data
    # small_table is cached
    for tuple in small_table:
        # tuple.join_key returns the join_key of the tuple
        # tuple.values returns the tuple without join_key
        if tuple.join_key == key:
            emit(key, value, tuple.values)
```

Repartition Join

```
map(file_type, tuple):
    # file_type is either t1 or t2
    # tuple has the key and all the other data and supports the two “methods” explained in
                                                                    broadcast join
    if file_type == ‘t1’:
        emit(tuple.join_key, (‘t1’, tuple.values))
    else:
        emit(tuple.join_key, (‘t2’, tuple.values))

reduce(join_key, tuple_list):
    emit(join_key, tuple)

map(join_key, tuple_list):
    t1 = []
    for tuple in tuple_list:
        if (tuple[1][0] == ‘t1’):
            t1.append(tuple[1][1])
        else:
            t2.appent(tuple[1][1])
    for tuple1 in t1:
        for tuple2 it t2:
            emit(join_key, tuple1, tuple2)
```

Ζητούμενο 3

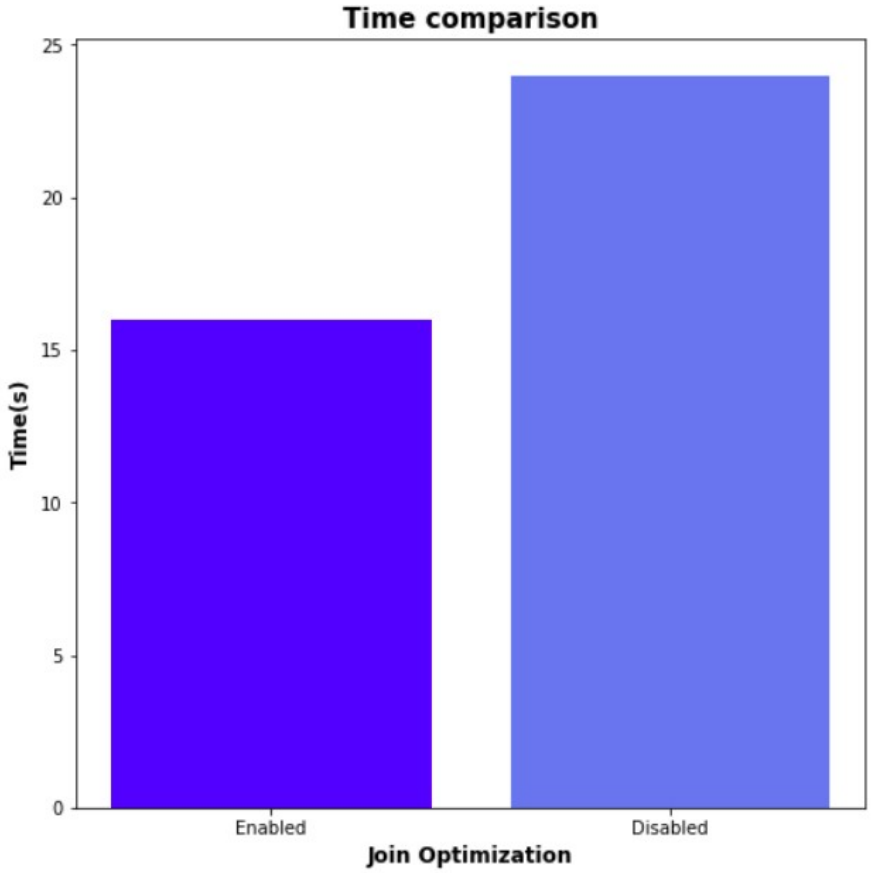
Χρόνοι εκτέλεσης των ζητούμενων Joins



Οπως φαίνεται πιο πάνω διάγραμμα οι δύο μέθοδοι παρουσιάζουν σημαντικά μεγάλη απόκλιση ως προς το χρόνο εκτέλεσης τους. Συγκεκριμένα, η εκτέλεση του Repartition Join απαιτεί σχεδόν τετραπλάσιο χρόνο σε σχέση με την εκτέλεση του Broadcast Join.
Το αποτέλεσμα αυτό είναι αναμενόμενο, καθώς στη μια περίπτωση πρόκειται για ένα Map Side Join, ενώ στην άλλη ένα Reduce Side Join.
Στο Repartition Join αφού ολοκληρώθεί η διαδικασία από την πλευρά του mapper, ακολουθεί η εκτέλεση ενός groupByKey (το οποίο λειτουργεί σαν reduce) για να προστεθεί στη συνέχεια ένα στάδιο flatMap (λειτουργεί σαν map), που θα οδηγήσει στο τελικό αποτέλεσμα.
Η αλυσίδα του RDD είναι σημαντικά πολυπλοκότερη και μακρύτερη γι’αυτό η διαδικασία απαιτεί πολύ περισσότερο χρόνο, ενώ υπάρχει και το στάδιο της μεταφοράς των δεδομένων πάνω από το δίκτυο (η οποία μεσολαβεί ανάμεσα στο αρχικό map και reduce) που οδηγεί σε περαιτέρω χρονική επιβάρυνση.
Πολύ πιθανό η απόκλιση να είναι μικρότερη, στην περίπτωση διαφορετικής υλοποίησης, καθώς η ίδια η επιλογή του groupByKey στο Repartition Join κάνει την καθυστέρηση μεγαλύτερη, αφού πρόκειται για μια αρκετά βαριά επιλογή σε σύγκριση με το ελαφρότερο reduceByKey, το οποίο κάνει ένα combine των δεδομένων που βρίσκονται στον ίδιο maper πριν το shuffle, κάτι που το groupByKey δεν κάνει. Επομένως, μια πιθανή υλοποίηση με reduceByKey ίσως να προκαλούσε μικρότερη χρονική διαφορά ανάμεσα στις δυο μεθόδους join που υλοποιήθηκαν.

Ζητούμενο 4

Χρόνοι εκτέλεσης με και χωρίς βελτιστοποιητή



Απενεργοποιώντας την δυνατότητα του βελτιστοποιητή να κάνει optimization τα join χρησιμοποιώντας το BroadcastHash Join, ο ίδιος οδηγείται σε ένα πλάνο εκτέλεσης με SortMerge Join, το οποίο συνιστά την αμέσως καλύτερη επιλογή. Όπως είναι αναμενόμενο, το πλάνο εκτέλεσης αυτό έχει σημαντικά χειρότερη απόδοση, με αρκετά μεγαλύτερο χρόνο εκτέλεσης. Άλλωστε στη μια περίπτωση έχουμε ένα Map-side Join που αποφεύγει τελείως τη μεταφορά δεδομένων πάνω από το δίκτυο (shuffle), ενώ στην άλλη περίπτωση αν και δουλεύουμε και πάλι με Map-side Join η μεταφορά των δεδομένων είναι αναπόφευκτη, καθώς για να ολοκληρωθεί το join πρέπει τα ίδια κλειδιά από κάθε dataset να φτάσουν στους ίδιους mapper.
Επιπλέον, χρειάζεται τα κλειδιά αυτά να ταξινομηθούν (sort) με τρόπο τέτοιο ώστε να μπορούν να γίνουν parse παράλληλα και να γίνει το join ανάμεσα στα tuples που έχουν τα ίδια κλειδιά. Δηλαδή, το SortMerge Join προσθέτει κάποια περαιτέρω στάδια επεξεργασίας για να υπάρχει εγγύηση ότι τα δεδομένα που αντιστοιχούν στα ίδια keys θα οδηγηθούν στα ίδια partitions και θα είναι ταξινομημένα με τον ίδιο τρόπο. Και μόνο η μεταφορά των δεδομένων είναι ένα εξαιρετικά χρονοβόρο και βαρύ στάδιο, το οποίο σε μεγάλο βαθμό εξηγεί τη χρονική διαφορά ανάμεσα στις δύο μεθόδους, όμως η ανάγκη για ταξινόμηση επιφέρει κι αυτή μια επιπρόσθετη χρονική καθυστέρηση που επιβαρύνει την κατάσταση.