



**ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ**

**ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ  
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ**

## ***Προχωρημένα Θέματα Βάσεων Δεδομένων***

***Αναφορά για την Εξαμηνιαία Εργασία:***

### ***Χρήση του Apache Spark στις Βάσεις Δεδομένων***

Ον/μο : Μπερέτσος Θεόδωρος  
Α.Μ.: : 03111612

Ον/μο : Ντόκος Χρήστος  
Α.Μ.: : 03117171

Ον/μο : Στάβαρης Δημοσθένης  
Α.Μ.: : 03117404

Ημερομηνία παράδοσης: 24/03/2022

## Μέρος 1<sup>ο</sup>: Υπολογισμός Αναλυτικών Ερωτημάτων με τα APIs του Apache Spark

### Ζητούμενο 1

Έγινε λήψη του dataset `movie_data.tar.gz`. Αποσυμπίεστηκε. Δημιουργήθηκαν τα directories **files** και **outputs** στο Hadoop file system. Τέλος, φορτώθηκαν τα 3 CSV αρχεία που μας δόθηκαν στο `hdfs` στο φάκελο **files** εκτελώντας τις παρακάτω εντολές στον **master** (βλ. Εικόνα 1).

```
user@master:~$ wget 'http://www.cslab.ntua.gr/courses/atds/movie_data.tar.gz'
--2022-03-11 19:41:19-- http://www.cslab.ntua.gr/courses/atds/movie_data.tar.gz
Resolving www.cslab.ntua.gr (www.cslab.ntua.gr)... 147.102.3.238
Connecting to www.cslab.ntua.gr (www.cslab.ntua.gr)|147.102.3.238|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 184259305 (176M) [application/x-gzip]
Saving to: 'movie_data.tar.gz'

movie_data.tar.gz      100%[=====] 175.72M  109MB/s   in 1.6s

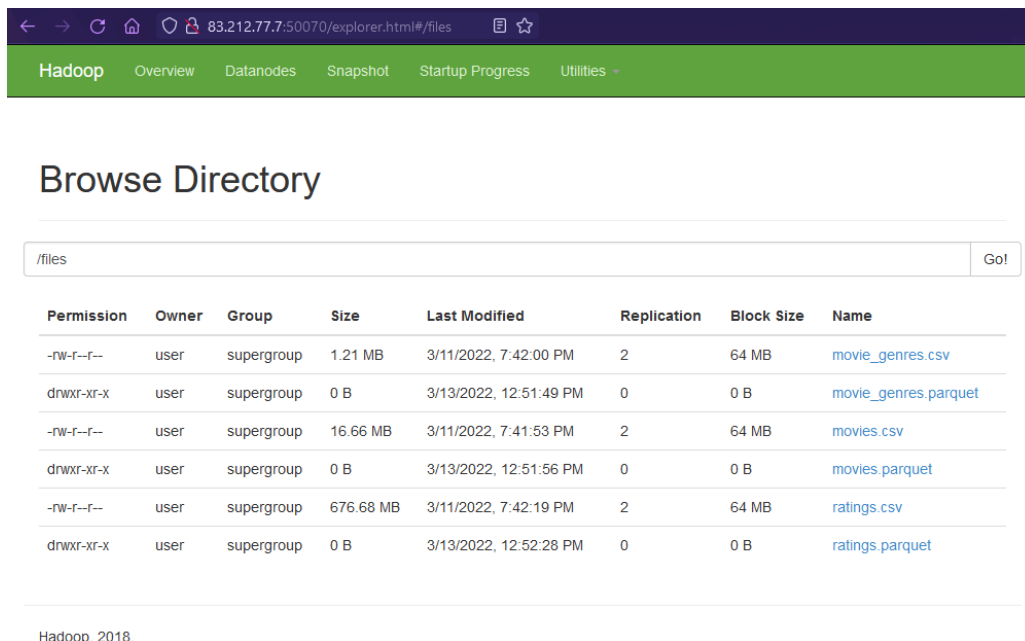
2022-03-11 19:41:21 (109 MB/s) - 'movie_data.tar.gz' saved [184259305/184259305]

user@master:~$ tar -xzf movie_data.tar.gz
user@master:~$ hadoop fs -mkdir hdfs://master:9000/files
user@master:~$ hadoop fs -put movies.csv hdfs://master:9000/files/.
user@master:~$ hadoop fs -put movie_genres.csv hdfs://master:9000/files/.
user@master:~$ hadoop fs -put ratings.csv hdfs://master:9000/files/.
user@master:~$ hadoop fs -mkdir hdfs://master:9000/outputs
user@master:~$ hadoop fs -ls hdfs://master:9000/
Found 2 items
drwxr-xr-x - user supergroup          0 2022-03-11 19:42 hdfs://master:9000/files
drwxr-xr-x - user supergroup          0 2022-03-11 19:42 hdfs://master:9000/outputs
user@master:~$
```

Εικόνα 1: Εντολές φόρτωσης .csv αρχείων στο `hdfs`

### Ζητούμενο 2

Χρησιμοποιήθηκε το script με όνομα `csntoparquet.py` για την μετατροπή των αρχείων CSV σε Parquet. Εποπτικά η εικόνα των 6 αρχείων (3 CSV και 3 Parquet) στο directory `files` μέσα από το Web UI του Hadoop είναι η εξής (βλ. Εικόνα 2):



The screenshot shows the Hadoop Web UI interface. At the top, there's a navigation bar with 'Hadoop' selected. Below it, a 'Browse Directory' section shows the contents of the '/files' directory. A table lists the files and their metadata.

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	user	supergroup	1.21 MB	3/11/2022, 7:42:00 PM	2	64 MB	<a href="#">movie_genres.csv</a>
drwxr-xr-x	user	supergroup	0 B	3/13/2022, 12:51:49 PM	0	0 B	<a href="#">movie_genres.parquet</a>
-rw-r--r--	user	supergroup	16.66 MB	3/11/2022, 7:41:53 PM	2	64 MB	<a href="#">movies.csv</a>
drwxr-xr-x	user	supergroup	0 B	3/13/2022, 12:51:56 PM	0	0 B	<a href="#">movies.parquet</a>
-rw-r--r--	user	supergroup	676.68 MB	3/11/2022, 7:42:19 PM	2	64 MB	<a href="#">ratings.csv</a>
drwxr-xr-x	user	supergroup	0 B	3/13/2022, 12:52:28 PM	0	0 B	<a href="#">ratings.parquet</a>

Hadoop, 2018.

Εικόνα 2: Web UI Hadoop

### Ζητούμενο 3

Υλοποιήθηκαν διαφορετικές λύσεις για κάθε ένα από τα ερωτήματα Q1-Q5, χρησιμοποιώντας το RDD API και τη Spark SQL. Συγκεκριμένα στην υλοποίηση της τελευταίας η είσοδος έγινε με τους εξής δύο τρόπους: αρχεία CSV και αρχεία Parquet. Παρακάτω παρουσιάζονται οι ψευδοκώδικες σε Map Reduce που εφαρμόστηκαν στις υλοποιήσεις με το RDD API.

Χρησιμοποιήθηκαν οι μέθοδοι `map()`, `reduceByKey()`, `filter()`, `count()`, `join()` και `sortByKey()` από το RDD API.

### Ερώτημα Q1

```
# movies = movies.csv

map(movies, value):
    for movie in movies:
        line = movie.split(',')
        # movie = (movie_id, title, description, publish_date,
        #         duration, cost, income, favoured)
        title = line[1]
        year = line[3].split('-')[0]
        publish_date = line[3]
        cost = line[5]
        income = line[6]
        if ((publish_date != '') and (cost != 0) and
            (income != 0) and (year > 2000)):
            profit = ((income - cost) / cost) * 100
            emit(year, (title, profit))

reduce(year, (title, profit)):
    max_profit = 0
    max_title = ''
    for title, profit in (title, profit):
        if (max_profit > profit):
            max_profit = profit
            max_title = title
    emit(year, (max_title, max_profit))

map(year, (max_title, max_profit)):
    emit(year, title)
```

### Ερώτημα Q2

```
# ratings = ratings.csv

map(ratings, value):
```

```

for movie in movies:
    line = movie.split(',')
    # ratings = (user_id, movie_id, rating, timestamp))
    user_id = line[0]
    rating = line[2]
    emit(user_id, (rating, 1))

reduce(user_id, (rating, 1)):
    sum_of_ratings = 0
    cnt_of_movies = 0
    for rating, cnt in (rating, 1):
        sum_of_ratings += rating
        cnt_of_movies++
    emit(user_id, (sum_of_ratings, cnt_of_movies))

map(user_id, (sum_of_ratings, cnt_of_movies)):
    avg_rating = sum_of_ratings / cnt_of_movies
    emit(user_id, avg_rating)

all_users = (user_id, avg_rating).count()
users_above_3 = (user_id, avg_rating).filter(avg_rating >= 3.0)

percentage = users_above_3 / all_users * 100

```

### Ερώτημα Q3

```

# ratings = ratings.csv
# genres = genres.csv

map(ratings, value):
    for movie in movies:
        line = movie.split(',')
        # ratings = (user_id, movie_id, rating, timestamp))
        movie_id = line[1]
        rating = line[2]
        emit(movie_id, (rating, 1))

reduce(movie_id, (rating, 1)):
    sum_of_ratings = 0
    cnt_of_ratings = 0
    for rating, cnt in (rating, 1):
        sum_of_ratings += rating
        cnt_of_ratings++
    emit(movie_id, (sum_of_ratings, cnt_of_ratings))

```

```

map(movie_id, (sum_of_ratings, cnt_of_ratings)):
    avg_rating = sum_of_ratings / cnt_of_ratings
    emit(movie_id, (avg_rating, cnt_of_ratings))

map(genres, value):
    for genre in genres:
        line = genre.split(',')
        # genres = (movie_id, genre)
        movie_id = line[0]
        genre = line[1]
        emit(movie_id, genre)

join((movie_id, genre), (movie_id, (avg_rating, cnt_of_ratings))):
    emit(movie_id, (genre, (avg_rating, cnt_of_ratings)))

map(movie_id, (genre, (avg_rating, cnt_of_ratings))):
    for line in lines:
        emit(genre, (avg_rating, 1))

reduce(genre, (avg_rating, 1)):
    sum_of_avg_ratings = 0
    cnt_of_avg_ratings = 0
    for avg_rating, cnt in (avg_rating, 1):
        sum_of_avg_ratings += avg_rating
        cnt_of_avg_ratings++
    emit(movie_id, (sum_of_avg_ratings, cnt_of_avg_ratings))

map(movie_id, (sum_of_avg_ratings, cnt_of_avg_ratings)):
    avg_rating_per_genre = sum_of_avg_ratings / cnt_of_avg_ratings
    emit(genre, (avg_rating_per_genre, cnt_of_avg_rating_per_genre))

map(genre, (avg_rating_per_genre, cnt_of_avg_rating_per_genre))
    emit(genre, avg_rating_per_genre, cnt_of_avg_rating_per_genre)

```

#### Ερώτημα Q4

```

# movies = movies.csv
# genres = genres.csv

map(movies, value):
    for movie in movies:
        line = movie.split(',')
        # movie = (movie_id, title, description, publish_date,

```

```

        # duration, cost, income, favoured)
        movie_id = line[0]
        title = line[1]
        description = line[2]
        publish_date = line[3]
        year = line[3].split('-')[0]
        if ((title != '') and (publish_date != '') and
            (year != '') and (year >= 2000) and (year <= 2019)):
            word_count_in_description = 0
            for word in description.split():
                word_count_in_description++
            emit(movie_id, (year, word_count_in_description))

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

join((movie_id, (year, word_count_in_description)), (movie_id, 'Drama')):
    emit(movie_id, ((year, word_count_in_description), 'Drama'))

map(movie_id, ((year, word_count_in_description), 'Drama')):
    if (year >= 2000) and (year <= 2004):
        period = '2000-2004'
    elif (year >= 2005) and (year <= 2009):
        period = '2005-2009'
    elif (year >= 2010) and (year <= 2014):
        period = '2010-2014'
    else:
        period = '2015-2019'
    emit(period, (word_count_in_description, 1))

reduce(period, (word_count_in_description, 1)):
    sum_of_words = 0
    cnt_of_movies = 0
    for words, cnt in (word_count_in_description, 1):
        sum_of_words += words
        cnt_of_movies++
    emit(period, (sum_of_words, cnt_of_movies))

map(period, (sum_of_words, cnt_of_movies)):

```

```
avg_cnt_of_words = sum_of_words / cnt_of_movies  
emit(period, avg_cnt_of_words)
```

### Ερώτημα Q5

```
# movies = movies.csv  
# genres = genres.csv  
# ratings = ratings.csv  
  
map(movies, value):  
    for movie in movies:  
        line = movie.split(',')  
        # movie = (movie_id, title, description, publish_date,  
        #         duration, cost, income, favoured)  
        movie_id = line[0]  
        title = line[1]  
        favoured = line[7]  
        emit(movie_id, (title, favoured))  
  
map(genres, value):  
    for genre in genres:  
        line = genre.split(',')  
        # genres = (movie_id, genre)  
        movie_id = line[0]  
        genre = line[1]  
        emit(movie_id, genre)  
  
map(ratings, value):  
    for movie in movies:  
        line = movie.split(',')  
        # ratings = (user_id, movie_id, rating, timestamp)  
        user_id = line[0]  
        movie_id = line[1]  
        rating = line[2]  
        emit(movie_id, (user_id, rating))  
  
join((movie_id, genre), (movie_id, (title, favoured))):  
    emit(movie_id, (genre, (title, favoured)))  
  
map(movie_id, (genre, (title, favoured))):  
    emit(movie_id, (genre, title, favoured))  
  
join((movie_id, (genre, title, favoured)), (movie_id, (user_id, rating))):
```

```

    emit(movie_id, ((genre, title, favoured), (user_id, rating)))

map(movie_id, ((genre, title, favoured), (user_id, rating))):
    max_title = min_title = title
    max_rating = min_rating = rating
    max_favoured = min_favoured = favoured
    emit((user_id, genre), (1, max_title, max_rating, max_favoured, min_title,
min_rating, min_favoured))

reduce((user_id, genre), (1, max_title, max_rating, max_favoured, min_title,
min_rating, min_favoured)):
    count = 0
    max_rating = max_favoured = 0
    min_rating = min_favoured = large_number
    for datum in data:
        count++
        if ((datum[2] > max_rating) or ((datum[2] == max_rating) and (datum[3] >
max_favoured))):
            max_title = datum[1]
            max_rating = datum[2]
            max_favoured = datum[3]
        if ((datum[5] < min_rating) or ((datum[5] == min_rating) and (datum[6] >
min_favoured))):
            min_title = datum[4]
            min_rating = datum[5]
            min_favoured = datum[6]
    emit((genre, userID), (count, max_title, max_rating, max_favoured, min_title,
min_rating, min_favoured))

map((user_id, genre), (count, max_title, max_rating, max_favoured, min_title,
min_rating, min_favoured)):
    emit(genre, (user_id, count, max_title, max_rating, max_favoured, min_title,
min_rating, min_favoured))

reduce(genre, (user_id, count, max_title, max_rating, max_favoured, min_title,
min_rating, min_favoured)):
    max_count = 0
    for line in lines:
        if (count > max_count):
            max_count = count
        assign max accordingly to every attribute
    emit(genre, (user_id, max_count, max_title, max_rating, min_title,
min_rating))

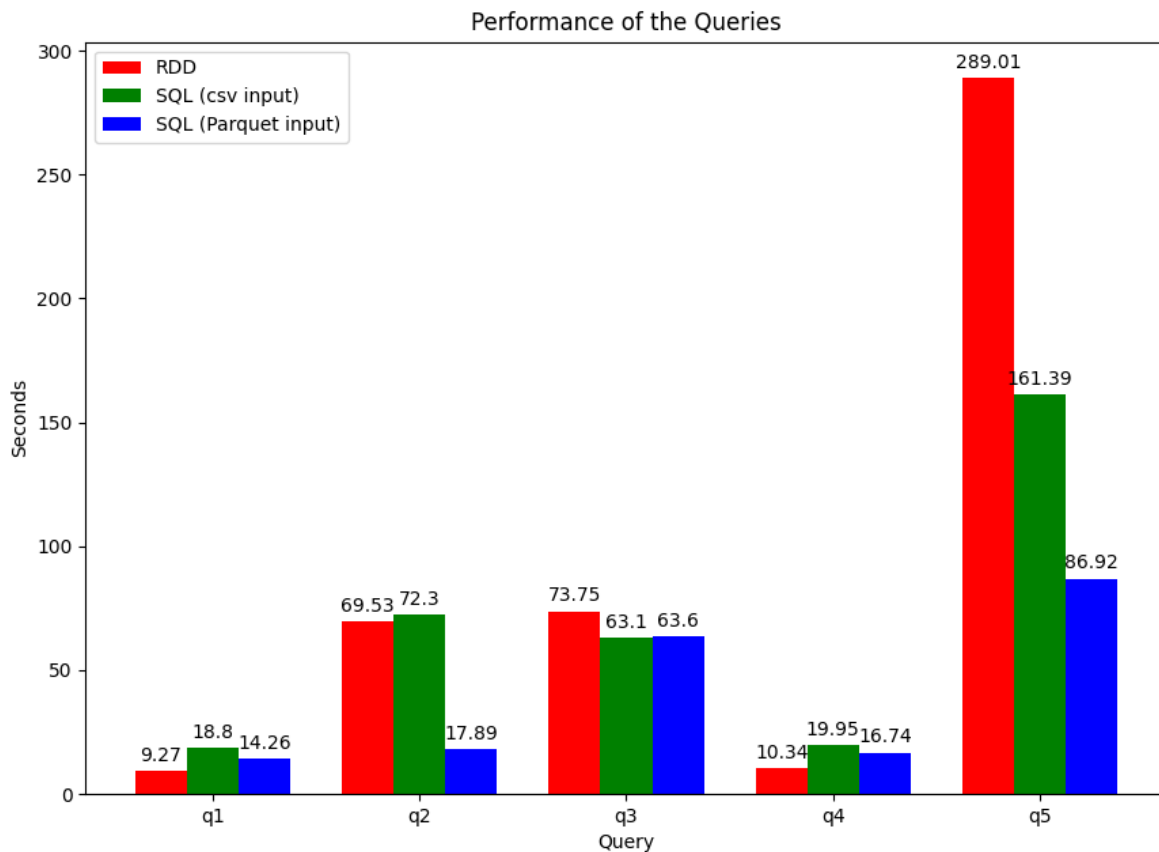
```



```
map(genre, (user_id, max_count, max_title, max_rating, max_favoured, min_title,
min_rating, min_favoured)):
    emit(genre, user_id, max_count, max_title, max_rating, max_favoured,
min_title, min_rating, min_favoured))
```

#### Ζητούμενο 4

Εκτελέστηκαν οι υλοποιήσεις του Ζητούμενου 3 για κάθε query και στο παρακάτω ραβδόγραμμα (βλ. Εικόνα 3) παρουσιάζονται οι χρόνοι εκτέλεσης ανά περίπτωση.



Εικόνα 3: Χρόνοι εκτέλεσης των queries ανά api και αντίστοιχο input

Παρατηρείται ότι σε όλα τα ερωτήματα η υλοποίηση της SQL που χρησιμοποιεί ως είσοδο αρχεία τύπου Parquet παρουσιάζει μικρότερους χρόνους εκτέλεσης.

Πιο συγκεκριμένα, για το ερώτημα Q1, παρατηρείται ότι μικρότερο χρόνο εκτέλεσης είχε η υλοποίηση με RDD API, καθώς χρησιμοποιείται μόνο μια διεργασία map-reduce. Επίσης παρατηρούνται παρόμοιοι χρόνοι για την υλοποίηση με SQL και είσοδο με csv και Parquet αρχεία.

Ακόμη συμπεραίνεται ότι τα queries Q1 (δεν απαιτεί κάποιο join) και Q4 (επεξεργάζεται και κάνει join τα δύο σχετικά μικρά datasets που δίνονται – movies.csv και genres.csv - ) δεν παρουσιάζουν ουσιαστική διαφορά ως προς το χρόνο εκτέλεσης και στις τρεις υλοποιήσεις.

Αντίθετα τα queries Q2 και Q5 παρουσιάζουν πενταπλάσιο και τριπλάσιο χρόνο εκτέλεσης αντίστοιχα στην υλοποίηση με RDD API από ότι στην υλοποίηση με SQL και είσοδο με αρχεία τύπου Parquet.

Γενικότερα τα ερωτήματα τα Q2, Q3 και Q5 είναι πιο χρονοβόρα από τα Q1, Q4 ανεξάρτητα της υλοποίησης διότι επεξεργάζονται το μεγαλύτερο από τα τρία datasets (ratings.csv) που δίνονται ως είσοδος.

Παρατηρείται επιπλέον ότι το Q5 είναι το πιο χρονοβόρο από όλα, καθώς γίνονται πολλά joins (RDD API) και group by (SQL), και με τα τρία datasets.

Συμπεραίνεται ότι είναι προτιμότερο να χρησιμοποιείται υλοποίηση με RDD API σε πιο απλά ερωτήματα και μικρού μεγέθους data. Ενώ για πιο σύνθετα ερωτήματα και μεγαλύτερα datasets ενδείκνυται η υλοποίηση με SQL με είσοδο αρχεία Parquet.

Με την χρήση του αρχείου Parquet αντί του csv για εκτέλεση ερωτημάτων SQL παρατηρούμε πως για μεγάλα αρχεία ο χρόνος εκτέλεσης μειώνεται κατά πολύ όπως φαίνεται και από το ραβδόγραμμα. Λόγω του ειδικού columnar format των αρχείων Parquet βελτιστοποιείται η χρήση της μνήμης και το I/O καθώς έχει μικρότερο αποτύπωμα στη μνήμη. Επίσης τα στατιστικά (min, max) που διατηρούνται από το dataset βοηθούν στην επεξεργασία, παρακάμπτοντας block κατά την διαδικασία αυτή.

Τέλος στην περίπτωση των Parquet αρχείων δεν χρησιμοποιήθηκε το Infer Schema. Αυτό συνέβη γιατί τα αρχεία Parquet διατηρούν το schema των αρχικών δεδομένων που το πήραν κατά την μετατροπή τους από csv. Έτσι δεν χρειάζεται να ορίσουμε το schema κατά την εκτέλεση και να χάσουμε χρόνο τότε, όπως γίνεται με τα csv αρχεία.

## **Μέρος 2<sup>ο</sup>: Υλοποίηση και μελέτη συνένωσης σε ερωτήματα και Μελέτη του βελτιστοποιητή του Spark**

Σε αυτό το μέρος θα μελετήσουμε και θα αξιολογήσουμε τις διαφορετικές υλοποιήσεις των συνενώσεων broadcast και repartitions στο περιβάλλον Map-Reduce του Spark.

Για τους κώδικες των ζητούμενων 1 και 2 βασιζόμαστε στην δημοσίευση “A Comparison of Join Algorithms for Log Processing in MapReduce”, Blanas et al , in Sigmod 2010”.

Όπως αναφέρει η δημοσίευση, το broadcast είναι πιο αποδοτικό join όταν έχουμε ένα μεγάλο fact table (ratings) και ένα μικρότερο dimension table (movie\_genre\_100), κάτι που θα αποδείξουμε και θα αναλύσουμε στην συνέχεια.

### **Ζητούμενο 1**

Η υλοποίηση του broadcast join στο RDD API βρίσκεται στον φάκελο project\_codes με όνομα broadcast\_join.py.

### **Ζητούμενο 2**

Η υλοποίηση του repartition join στο RDD API βρίσκεται στον φάκελο project\_codes με όνομα repartition\_join.py.

### Ζητούμενο 3

Αρχικά μειώνουμε το μέγεθος του αρχείου `movie_genres` στις 100 πρώτες καταχωρήσεις και τις αποθηκεύουμε στο αρχείο `movie_genres_100`. Στην συνέχεια ανεβάζουμε το αρχείο αυτό στο `hdfs` και ενημερώνουμε τις δύο υλοποιήσεις `join` να χρησιμοποιούν το εν λόγω αρχείο.

Για να συγκρίνουμε τις δύο μεθόδους συνένωσης πινάκων, `repartition join` και `broadcast join`, χρησιμοποιούμε τα παραπάνω αρχεία (`repartition_join.py` και `broadcast_join.py`). Υλοποιούμε την συνένωση των πινάκων αρκετές φορές έτσι ώστε να έχουμε πιο σίγουρα αποτελέσματα.

Οι χρόνοι κάθε συνένωσης αποθηκεύονται στο αρχείο `execution_times.txt`.

Παρατηρούμε πως ο χρόνος που απαιτείται για να γίνει το `broadcast join` είναι σημαντικά μικρότερος από τον χρόνο του `repartition join`. Κατά μέσο όρο το `broadcast join` είναι έως και 13 φορές πιο γρήγορο από το `repartition`.

Πιο συγκεκριμένα, επειδή ο πίνακας `movie_genres` έχει μέγεθος 100 καταχωρήσεων, είναι πιο αποτελεσματικό να κάνουμε αυτόν τον πίνακα `broadcast` σε κάθε `node` από τον να σταλούν και οι δύο πίνακες σε όλο το δίκτυο. Στην συνέχεια κάθε `node` αποθηκεύει τον μικρό πίνακα `locally` και στην συνέχεια να υλοποιεί τα `joins` με μία μόνο `map` διεργασία. Με αυτό τον τρόπο αποφεύγεται η εκτεταμένη χρήση του δικτύου και του χρονικού κόστους που επιφέρει αυτό. Προϋπόθεση αποτελεί να χωράει ο μικρός πίνακας `movie_genres` στους `executor nodes`.

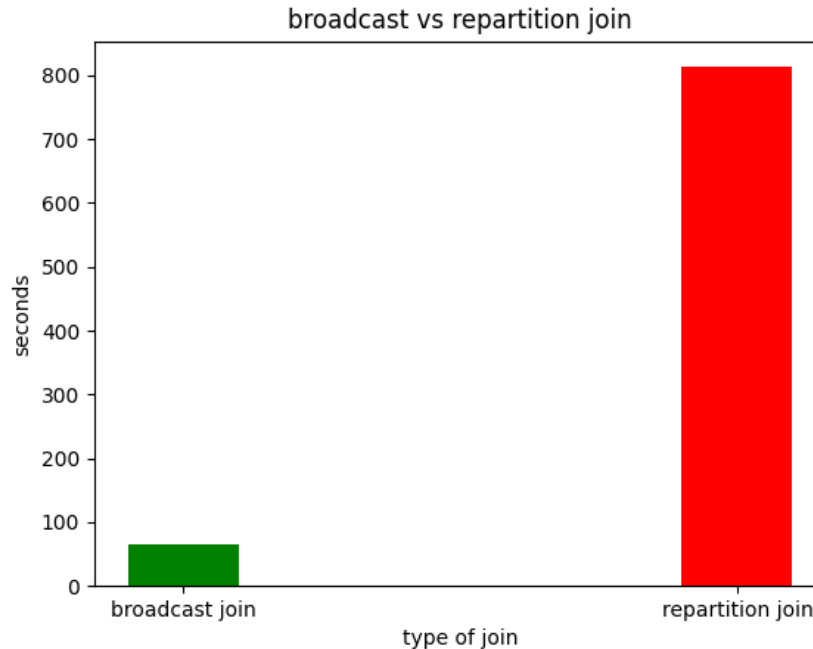
Στην περίπτωση μας, έχουμε δύο `workers` οπότε το κόστος του `broadcasting` είναι αρκετά μικρό.

Όσον αφορά το `repartition join`, χρησιμοποιείται μία `map-reduce` διαδικασία όπου στο `map` κάθε `record` παίρνει ένα `tag` προέλευσης και στην συνέχεια ενώνουμε τα `records` με βάση το `key` τους (`movie_id`) και δημιουργούμε τις `value lists` με τούπλες από τα `records` των δύο πινάκων. Έπειτα στο `reduce` στάδιο ξεχωρίζουμε τις λίστες με βάση το `tag` προέλευσης και υπολογίζουμε το εξωτερικό γινόμενο των δύο λιστών. Έτσι διαθέτουμε όλους τους πιθανούς συνδυασμούς καταχωρήσεων. Σημαντική προϋπόθεση του `repartition join` είναι να χωρούν όλα τα δεδομένα για κάθε `key` στον `buffer` του εκάστοτε `worker node`. Η παραπάνω υλοποίηση είναι χρονικά πιο ακριβή (από το `broadcast join`) καθώς έχουμε ένα επίπεδο `reduce` το οποίο χρειάζεται `shuffling` και `sorting` στα δεδομένα.

Χρόνοι εκτέλεσης των `joins`:

```
broadcast_join.py: 65.8225507736206 seconds
broadcast_join.py: 66.94003105163574 seconds
broadcast_join.py: 67.53642249107361 seconds
broadcast_join.py: 66.5188615322113 seconds
repartition_join.py: 856.2306735515594 seconds
repartition_join.py: 792.8050441741943 seconds
repartition_join.py: 761.0503301620483 seconds
repartition_join.py: 845.5805656909943 seconds
```

Παίρνουμε την μέση τιμή των παραπάνω και έχουμε:



Εικόνα 4: broadcast vs repartition join

Όπως έχουμε αναφέρει, στην περίπτωση που έχουμε ένα σχετικά πιο μικρό πίνακα (movie\_genres\_100) και ένα πιο μεγάλο (ratings) καλύτερη λύση είναι broadcast join.

#### Ζητούμενο 4

Σε αυτό το κομμάτι θα χρησιμοποιήσουμε τις έτοιμες υλοποιήσεις των joins στο DataFrame API. Οι έτοιμες υλοποιήσεις περιέχουν βελτιστοποιήσεις που αυτόματα επιλέγουν την υλοποίηση join που θα χρησιμοποιηθεί. Για να μπορέσουμε να συγκρίνουμε την περίπτωση των συνενώσεων με optimizer και χωρίς, συμπληρώνουμε το κομμάτι κώδικα που μας δόθηκε και βρίσκεται πλέον στον φάκελο project\_code με όνομα join\_optimizer.py.

Με optimizer disabled έχουμε:

```
== Physical Plan ==
*(6) SortMergeJoin [_c0#8], [_cl#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,_cl#9] Batched: true, Format: Parquet, Location: InMemoryFileIndex[
hdfs://master:9000/files/movie_genres.parquet], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<_c0:int,_
cl:string>
+- *(5) Sort [_cl#1 ASC NULLS FIRST], false, 0
  +- Exchange hashpartitioning(_cl#1, 200)
    +- *(4) Project [_c0#0, _cl#1, _c2#2, _c3#3]
      +- *(4) Filter isnotnull(_cl#1)
        +- *(4) FileScan parquet [_c0#0,_cl#1,_c2#2,_c3#3] Batched: true, Format: Parquet, Location: InMemoryFile
Index[hdfs://master:9000/files/ratings.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(_cl)], ReadSchema: s
truct<_c0:int,_cl:int,_c2:double,_c3:int>
Time with choosing join type disabled is 15.3829 sec.
```

Με optimizer enabled έχουμε:

```

== 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/files/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/files/ratings.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(_c1)], ReadSchema: struct<_c0:int,_c1:int,_c2:double,_c3:int>
Time with choosing join type enabled is 6.5676 sec.

```

Παρατηρούμε πως χωρίς optimizer επιλέγεται sortMerge ενώ με optimizer επιλέγεται το broadcastHashJoin. Τα παραπάνω αποτελέσματα επιβεβαιώνουν το πόρισμα μας από το ζητούμενο 3, ότι το broadcast join είναι πιο αποτελεσματικό όταν έχουμε μικρό πίνακα movie\_genres (το query στο script μας επιλέγει τα πρώτα 100 records του movie\_genres.csv).

Για να ελέγξουμε περαιτέρω τα λεγόμενα μας θα επιλέξουμε τα 1000 πρώτα records από τον πίνακα movie\_genres και μετά τα 2000 πρώτα.

Με 1000 records από movie\_genres και optimizer disabled έχουμε:

```

== 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 1000
:       +- Exchange SinglePartition
:         +- *(1) LocalLimit 1000
:           +- *(1) FileScan parquet [_c0#8,_c1#9] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://master:9000/files/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 26.5668 sec.

```

Με 1000 records από movie\_genres και optimizer enabled έχουμε:

```

== 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 1000
:     +- Exchange SinglePartition
:       +- *(1) LocalLimit 1000
:         +- *(1) FileScan parquet [_c0#8,_c1#9] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://master:9000/files/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/files/ratings.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(_c1)], ReadSchema: struct<_c0:int,_c1:int,_c2:double,_c3:int>
Time with choosing join type enabled is 13.3960 sec.

```

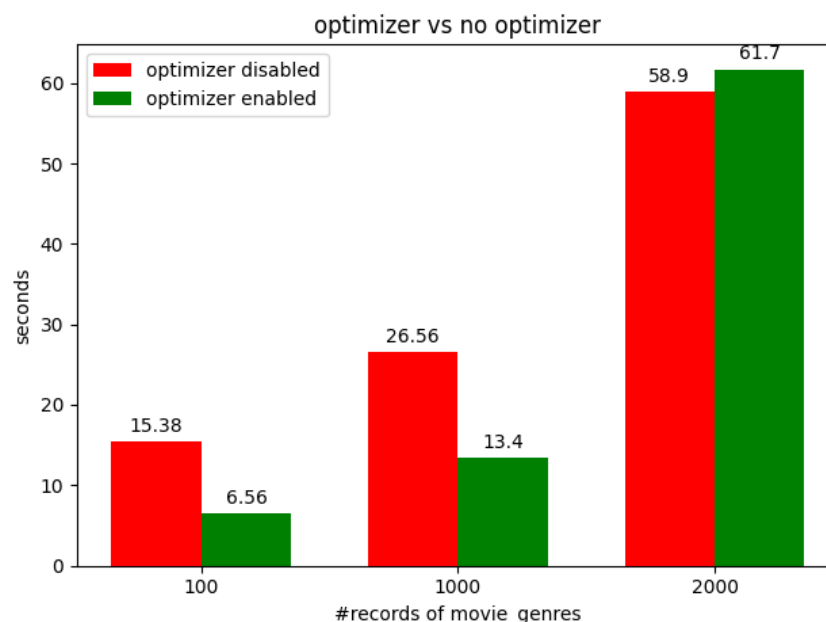
Με 2000 records από movie\_genres και optimizer disabled έχουμε:

```
== 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 2000
:       +- Exchange SinglePartition
:         +- *(1) LocalLimit 2000
:           +- *(1) FileScan parquet [_c0#8,_c1#9] Batched: true, Format: Parquet, Location: InMemoryFileIndex[
hdfs://master:9000/files/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: InMemoryFile
Index[hdfs://master:9000/files/ratings.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(_c1)], ReadSchema: s
truct<_c0:int,_c1:int,_c2:double,_c3:int>
Time with choosing join type disabled is 58.9411 sec.
```

Με 2000 records από movie\_genres και optimizer enabled έχουμε:

```
== 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 2000
:     +- Exchange SinglePartition
:       +- *(1) LocalLimit 2000
:         +- *(1) FileScan parquet [_c0#8,_c1#9] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdf
s://master:9000/files/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/files/ratings.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(_c1)], ReadSchema: struct<
_c0:int,_c1:int,_c2:double,_c3:int>
Time with choosing join type enabled is 61.6898 sec.
```

Τοποθετούμε τα εξαγόμενα δεδομένα σε ένα ραβδόγραμμα:



### *Εικόνα 5: optimizer vs no optimizer*

Με πράσινο έχει αναπαρασταθεί η περίπτωση που έχουμε enabled optimizer, λογικό είναι σε αυτή την περίπτωση να έχουμε μικρότερους χρόνους στα 100 και 1000 records του movie\_genres. Επιλέγεται ορθά από τον optimizer το broadcast join όπως βλέπουμε και στα παραπάνω screenshots. Στην περίπτωση που δεν έχουμε optimizer επιλέγεται το sort merge που όπως λέει και το όνομα του θα ταξινομήσεις τα records, κάτι που είναι χρονικά ακριβό.

Τελικά παρατηρούμε πως όταν το πλήθος των records φτάνει τα 2000 επιλέγεται ξανά από τον optimizer το sort merge με αποτέλεσμα τα joins και στις δύο περιπτώσεις να παίρνουν τον ίδιο χρόνο.

Θα μπορούσαμε να αυξήσουμε και άλλο το μέγεθος του πίνακα movie\_genres αλλά το σύστημα μας δεν μπορεί να χειριστεί αρχεία τέτοιου μεγέθους.