# Αναφορά Εξαμηνιαίας Εργασίας

**Μάθημα:** Προχωρημένα Θέματα Βάσεων Δεδομένων, **Θέμα:** Χρήση του Apache Spark στις Βάσεις Δεδομένων

## Ομάδα

Καναρόπουλος Ιωάννης Γεράσιμος 03116016

Κουτρούμπας Αθανάσιος 03116073

## Μέρος 1ο

### Ζητούμενο 1

Μέσα από το Master VM κατεβάζουμε τα `movie_data`:

```
wget www.cslab.ntua.gr/courses/atds/movie_data.tar.gz
```

Κάνουμε untar τα 3 `.csv` αρχεία που περιέχονται:

```
tar -xvzf movie_data.tar.gz
```

Δημιουργούμε ένα φάκελο `movie_data` στο hdfs:

```
hadoop fs -mkdir hdfs://master:9000/movie_data
```

Μεταφέρουμε τα 3 `.csv` αρχεία σε αυτό τον φάκελο στο hdfs:

```
hadoop fs -put movie_genres.csv movies.csv ratings.csv hdfs://master:9000/movie_data
```

### Ζητούμενο 2

Για να μετατρέψουμε τα `.csv` αρχεία σε `.parquet` τρέχουμε το script `convert_csv_to_parquet.py` που γράψαμε πάνω στο Spark, το οποίο ορίζει το schema των δεδομένω και φορτώνει τα `.csv` σε dataframe μορφή κρατώντας το δωσμένο schema. Έπειτα τα dataframe γράφονται σε `.parquet` αρχεία μέσα στον φάκελο που δημιουργήσαμε στο hdfs:

```
spark-submit convert_csv_to_parquet.py > log_convert_csv_to_parquet.txt 2>&1
```

### Ζητούμενο 3

#### Ερώτημα Q1

Ψευδοκώδικας Map Reduce για την υλοποίηση με RDD API:

```
# Read record from movies.csv
map(line_no, record):
   if year != '' and cost != '' and revenue != '' and cost != 0 and revenue != 0 and year >= 2000:
      profit = ((revenue-cost)/cost)*100
      emit(year, (profit, title))

reduce(year, [tuple]):
   max_profit = 0
   max_title = ''
   for profit, title in [tuple]:
```

```
      if profit > max_profit:
         max_profit = profit
         max_title = title
   emit(year, title)
```

**Ερώτημα Q2**

Ψευδοκώδικας Map Reduce για την υλοποίηση με RDD API:

```
# Global Variables
total_users = 0
over_3_users = 0

# Read record from ratings.csv
map(line_no, record):
   emit(userId, (rating, 1))

reduce(userId, [tuple]):
   total_users += 1
   sum = 0
   count = 0
   for rating, cnt in [tuple]:
      sum += rating
      count += cnt
   avg = sum/count

   if avg > 3.0:
      over_3_users += 1

   emit(null)

result = (over_3_users/total_users)*100
```

**Ερώτημα Q3**

Ψευδοκώδικας Map Reduce για την υλοποίηση με RDD API:

```
# Read record from ratings.csv
map_ratings(line_no, record):
   emit(movieId, rating)

# Read record from movie_genres.csv
map_movie_genres(line_no, record):
   emit(movieId, genre)

# Implements groupByKey
reduce_movie_genres(movieId, [genre]):
   emit(movieId, [genre])

join (movieId, rating) and (movieId, [genre]) on movieId
# Result: (movieId, (rating, [genre]))

# Reads from joined dataset
map_join_1(movieId, (rating, [genre])):
   emit(movieId, (rating, [genre], 1))

reduce_join_1(movieId, [tuple]):
   sum = 0
   count = 0
   for rating, [genre], cnt in [tuple]:
      sum += rating
      count += cnt
   avg_rating = sum/count
   emit(movieId, (avg_rating, [genre]))

map_join_2(movieId, (avg_rating, [genre])):
   for genre in [genre]:
      emit(genre, (avg_rating, 1))
```

```
reduce_join_2(genre, [tuple]):
   sum = 0
   count = 0
   for avg_rating, cnt in [tuple]:
      sum += avg_rating
      count += cnt
   avg_rating_by_genre = avg_rating/count
   emit(genre, (avg_rating_by_genre, count))
```

**Ερώτημα Q4**

Ψευδοκώδικας Map Reduce για την υλοποίηση με RDD API:

```
# Read record from movies.csv
map_movies(line_no, record):
   if year != '' and 2000 <= year <= 2019:
      emit(movieId, (summary, year))

# Read record from movie_genres.csv
map_movie_genres(line_no, record):
   if genre == 'Drama':
      emit(movieId, 'Drama')

join (movieId, (summary, year)) and (movieId, 'Drama') on movieId
# Result: (movieId, ((summary, year), 'Drama'))

# Reads from joined dataset
map_join(movieId, ((summary, year), 'Drama')):
   if (year <= 2004):
      emit("2000-2004", (len(summary), 1))
   elif (year <= 2009):
      emit("2005-2009", (len(summary), 1))
   elif (year <= 2014):
      emit("2010-2014", (len(summary), 1))
   elif (year <= 2019):
      emit("2015-2019", (len(summary), 1))

reduce_join(range_5_years, [tuple]):
   sum = 0
   count = 0
   for length, cnt in [tuple]:
      sum += length
      count += cnt
   avg_summary = sum/count
   emit(range_5_years, avg_summary)
```

**Ερώτημα Q5**

Ψευδοκώδικας Map Reduce για την υλοποίηση με RDD API:

```
# Read record from movies.csv
map_movies(line_no, record):
   emit(movieId, (title, popularity))

# Read record from ratings.csv
map_ratings(line_no, record):
   emit(movieId, (userId, rating))

# Read record from movie_genres.csv
map_movie_genres(line_no, record):
   emit(movieId, genre)

# Implements groupByKey
reduce_movie_genres(movieId, [genre]):
   emit(movieId, [genre])

join (movieId, (title, popularity)) and (movieId, genre) on movieId
```

```
# Result: (movieId, ((title, popularity), [genre]))

join (movieId, ((title, popularity), [genre])) and (movieId, (userId, rating)) on movieId
# Result: (movieId, ((title, popularity, [genre]), (userId, rating)))

map_join_1(movieId, ((title, popularity, [genre]), (userId, rating))):
   user_dict = {}
   for genre in [genre]:
      user_dict[genre] = (1, popularity, popularity, title, title, rating, rating)
   emit(userId, user_dict)

# user_dict[genre] = (NumRatings, MinPopul, MaxPopul, MinTitle, MaxTitle, MinRating, MaxRating)

reduce_join_1(userId, [user_dict]):
   user1_dict = [user_dict][0]
   for user2_dict in [user_dict]:
      for genre, value1 in user1_dict.items():
         if genre in user2_dict:
            value2 = user2_dict[genre]
            NumRatings = value1[0] + value2[0]
            if value1[5] < value2[5] or (value1[5] == value2[5] and value1[1] > value2[1]):
               MinPopul = value1[1]
               MinTitle = value1[3]
               MinRating = value1[5]
            else:
               MinPopul = value2[1]
               MinTitle = value2[3]
               MinRating = value2[5]
            if value1[6] > value2[6] or (value1[6] == value2[6] and value1[2] > value2[2]):
               MaxPopul = value1[2]
               MaxTitle = value1[4]
               MaxRating = value1[6]
            else:
               MaxPopul = value2[2]
               MaxTitle = value2[4]
               MaxRating = value2[6]
            user1_dict[genre] = (NumRatings, MinPopul, MaxPopul, MinTitle, MaxTitle, MinRating, MaxRating)
            del user2_dict[genre]
      user1_dict = merge(user1_dict, user2_dict)
   emit(userId, user1_dict)

map_join_2(userId, user_dict):
   for genre in user_dict:
      emit(genre, (userId, user_dict[genre]))

reduce_join_2(genre, [tuple]):
   tuple1 = [tuple][0]
   max_NumRatings = [tuple][0].user_dict[genre][0]
   for userId, user_dict[genre] in [tuple]:
      NumRatings = user_dict[genre][0]
      if NumRatings > max_NumRatings:
         max_NumRatings = NumRatings
         tuple1 = (userId, user_dict[genre])

   userId = tuple1.userId
   MaxTitle = tuple1.user_dict[genre][4]
   MinTitle = tuple1.user_dict[genre][3]
   MaxRating = tuple1.user_dict[genre][6]
   MinRating = tuple1.user_dict[genre][5]

   emit(genre, (userId, max_NumRatings, MaxTitle, MaxRating, MinTitle, MinRating))
```

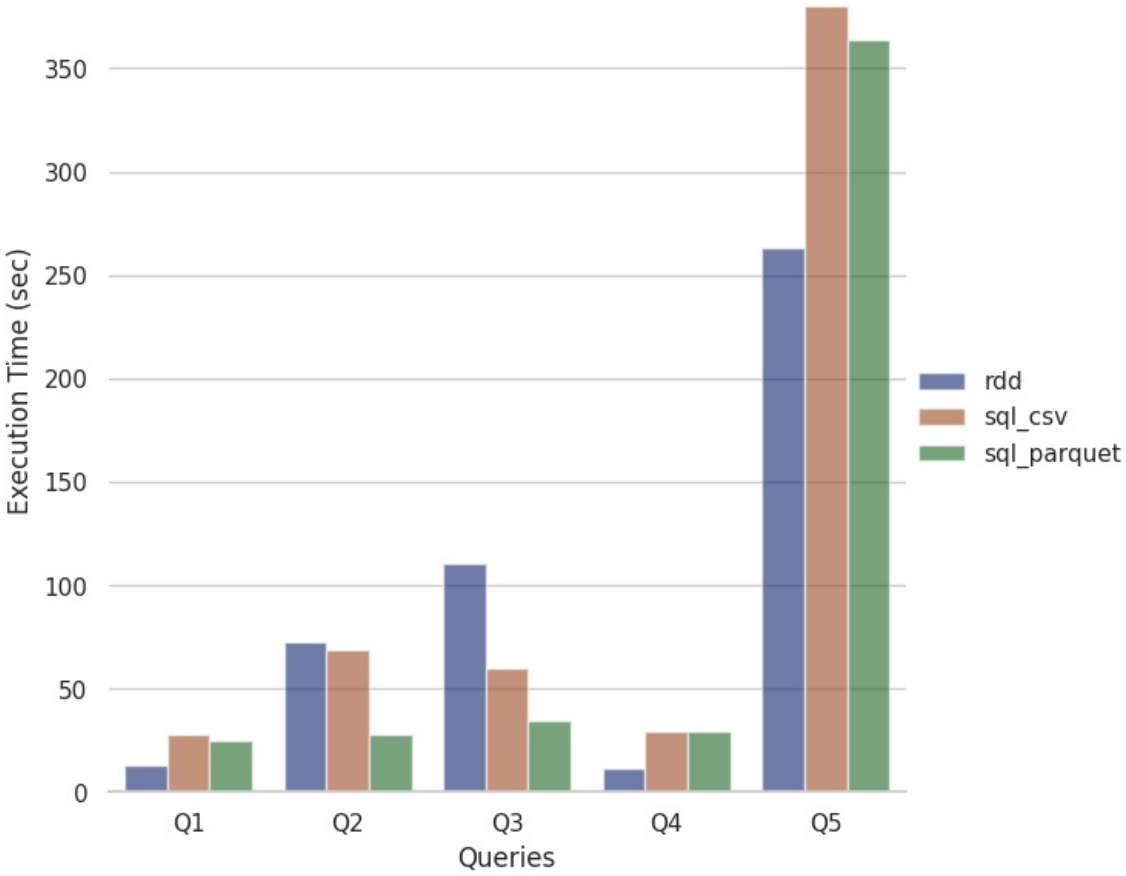
## Ζητούμενο 4

Στο Master VM τρέχουμε το `run_all_queries.sh` script, ώστε να τρέξει όλα τα ερωτήματα στο Spark και να αποθηκεύσει τα logs και τους χρόνους εκτέλεσης:

```
./run_all_queries.sh
```

Τοπικά, τρέχουμε το `plot_queries_exec_times.py` script, ώστε να δημιουργήσουμε το παρακάτω ραβδοδιάγραμμα με τους χρόνους εκτέλεσεις ομαδοποιημένους ανά ερώτημα:

```
./plot_queries_exec_times.py queries_exec_times.txt
```




**Σχολιασμός Χρόνων Εκτέλεσης για κάθε ερώτημα.** Παρατηρούμε ότι, για τα Q2, Q3 το RDD είναι πιο αργό από το αντίστοιχο query σε SQL. Αντίθετα στα Q1, Q4 το RDD είναι πιο γρήγορο από τα αντίστοιχα query σε SQL. Για το Q5, παρατηρούμε ότι η εκτέλεση σε SQL είναι σχεδόν η διπλάσια από ότι με RDD, και αυτό γιατί στην υλοποίηση μας, κάνουμε πολλά Group By, και μετά για να μπορέσουμε να πάρουμε τις τιμές από όλα τα πεδία μιας γραμμής, χρειάζεται να κάνουμε αντίστοιχα πολλά join πινάκων, κάτι που αυξάνει όπως φαίνεται κατά πολύ τον χρόνο εκτέλεσης. Αντίθετα στην περίπτωση του RDD API, δεν χρείαζεται όλη αυτή η επιπλέον δουλειά, καθώς ορίζουμε εμείς από το map ή το reduce ποια πεδία θα κρατήσουμε και πως.

Παρατηρούμε αρχικά ότι το Q5 είναι το πιο χρονοβόρο από όλα, καθώς γίνονται πολλά joins και group by, και με τα τρία αρχεία.

Επίσης τα Q2, Q3 είναι πιο χρονοβόρα από τα Q1, Q4 ανεξάρτητα τις υλοποίησης και αυτό γιατί και τα δύο χρησιμοποιούν το `ratings` αρχείο, το οποίο είναι πολύ μεγαλύτερο των άλλων αρχείων:

- `movies.csv` : 17 MB
- `movie_genres.csv` : 1.3 MB
- `ratings.csv` : 677 MB

Επίσης το Q2 χρησιμοποιεί μόνο το αρχείο `ratings`, ενώ το Q3, εφαρμόζει join πάνω στα αρχεία `ratings` και `movie_genres`, οπότε για αυτό είναι ακόμα πιο αργό.

**Τι παρατηρούμε με την χρήση του parquet;** Για τις SQL υλοποιήσεις παρατηρούμε ότι, όταν τρέχουμε τα ερωτήματα με τα `.parquet` αρχεία έχουμε πάντα μικρότερο (έως και μισό) χρόνο εκτέλεσης σε σχέση με τα αντίστοιχα `.csv` αρχεία. Το parquet είναι ένα columnar storage format, το οποίο είναι συμπιεσμένο άρα έχει μικρότερο αποτύπωμα στη μνήμη και στον δίσκο και βελτιστοποιεί το I/O, μειώνοντας τον χρόνο εκτέλεσης. Επίσης διατηρεί επιπλέον πληροφορίες (metadata) για το dataset και μπορεί να διατηρήσει και το schema που μπορεί να έχουν τα δεδομένα. Οπότε είναι λογικό να έχει καλύτερες επιδόσεις από το να χρησιμοποιούμε απλά text formats όπως το CSV.

**Γιατί δεν χρησιμοποιούμε `infer schema` με την χρήση του parquet;** Όπως αναφέραμε το parquet έχει την δυνατότητα να διατηρεί το schema των δεδομένων, οπότε όταν μετατρέψαμε στην αρχή τα `.csv` δεδομένα μας σε `.parquet` αρχεία, ορίσαμε και περάσαμε ως παράμετρο τότε το schema των δεδομένων. Αυτό το schema παρέμεινε και όταν τρέχαμε τα ερωτήμα, οπότε γλιτώσαμε και χρόνο, καθώς για το τρέξιμο στα `.csv` αρχεία, έπρεπε κάθε φορά να ορίζουμε το schema για το κάθε αρχείο-πίνακας.

# Μέρος 2ο

## Ζητούμενο 1

Η υλοποίηση του `broadcast join` στο RDD API βρίσκεται στο `code/join_broadcast.py`.

## Ζητούμενο 2

Η υλοποίηση του `repartition join` στο RDD API βρίσκεται στο `code/join_repartition.py`.

## Ζητούμενο 3

Στο Master VM τρέχουμε το `create_movie_genres_100.py` με παράμετρο το `movie_genres.csv`:

```
./create_movie_genres_100_local.py movie_genres.csv
```

Αυτό απομονώνει τις 100 πρώτες γραμμές του αρχείου `movie_genres.csv` και τις αποθηκεύει στο `movie_genres_100.csv` τοπικά καθώς και στο hdfs.

Στο Master VM τρέχουμε το `run_all_joins.sh` script, ώστε να τρέξει τα δύο joins ( `broadcast` , `repartition` ) του πίνακα `movie_genres_100` και του `ratings` στο Spark και να αποθηκεύσει τα logs και τους χρόνους εκτέλεσης:

```
./run_all_joins.sh
```

**Χρόνοι Εκτέλεσης των 2 joins:**

```
join_broadcast.py: 74.14392161369324 seconds
join_repartition.py: 1089.001785993576 seconds
```

**Παρατηρήσεις** Παρατηρούμε ότι προφανώς το repartition join, είναι πιο αργό από το broadcast join, καθώς απαιτείται το στάδιο του reduce και άρα και τα αντίστοιχα στάδια shuffle and short, όπου είναι αρκετά χρονοβόρα και υπολογιστικά ακριβά. Αντίθετα, το broadcast join απαιτεί μόνο Map στάδια, και άρα είναι λιγότερο υπολογιστικά ακριβό, και όπως φάνηκε από τις μετρήσεις πολύ καλύτερη υλοποίηση. Το μόνο μειονέκτημα του broadcast όμως είναι ότι απαιτείται να έχουμε αρκετή μνήμη για να γίνει broadcast ο μικρότερος πίνακας, κάτι όμως που στην δικιά μας περίπτωση ισχύει, όποτε δεν αποτελεί πρόβλημα για εμάς.

## Ζητούμενο 4

### Απενεργοποίηση optimizer και εκτέλεση

Χρησιμοποιώντας το script της εκφώνησης, συμπληρώσαμε τα path των parquet πινάκων στο hdfs, καθώς και την επιλογή για να μπορέσουμε να απενεργοποιήσετε την επιλογή του join από το βελτιστοποιητή. Αυτή ήταν η παρακάτω:

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)
```

η οποία στην ουσία απενεργοποιεί το broadcasting, οπότε τα μόνα join που γίνονται είναι από την μεριά του reduce.

Στο Master VM τρέχουμε το `run_all_optimizer.sh` script, ώστε να τρέξει τα 2 join μεταξύ των πινάκων `movie_genres` και `ratings` στο Spark, με ή χωρίς τον optimizer και να αποθηκεύσει τα logs και τους χρόνους εκτέλεσης:

```
./run_all_optimizer.sh
```

Τρέχει το αρχείο `optimizer_joins.py` με τα arguments:

- `./optimizer_joins.py N`
- `./optimizer_joins.py Y`

#### Πλάνα Εκτέλεσης

Το πλάνο εκτέλεσης για το **join χωρίς τον optimizer**:

```
== Physical Plan ==
*(6) SortMergeJoin [movieId#8], [movieId#1], Inner
:- *(3) Sort [movieId#8 ASC NULLS FIRST], false, 0
:  +- Exchange hashpartitioning(movieId#8, 200)
:     +- *(2) Filter isnotnull(movieId#8)
:        +- *(2) GlobalLimit 100
:           +- Exchange SinglePartition
:              +- *(1) LocalLimit 100
:                 +- *(1) FileScan parquet [movieId#8,genre#9] Batched: true, Format: Parquet, Location:
InMemoryFileIndex[hdfs://master:9000/movie_data/movie_genres.parquet], PartitionFilters: [], PushedFilters: [],
ReadSchema: struct<movieId:int,genre:string>
+- *(5) Sort [movieId#1 ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(movieId#1, 200)
      +- *(4) Project [userId#0, movieId#1, rating#2, timestamp#3]
         +- *(4) Filter isnotnull(movieId#1)
            +- *(4) FileScan parquet [userId#0,movieId#1,rating#2,timestamp#3] Batched: true, Format: Parquet,
Location: InMemoryFileIndex[hdfs://master:9000/movie_data/ratings.parquet], PartitionFilters: [], PushedFilters:
[IsNotNull(movieId)], ReadSchema: struct<userId:int,movieId:int,rating:double,timestamp:string>
```

Βλέπουμε ότι χρησιμοποιείται `SortMergeJoin` (repartition join).

Το πλάνο εκτέλεσης για το **join με τον optimizer**:

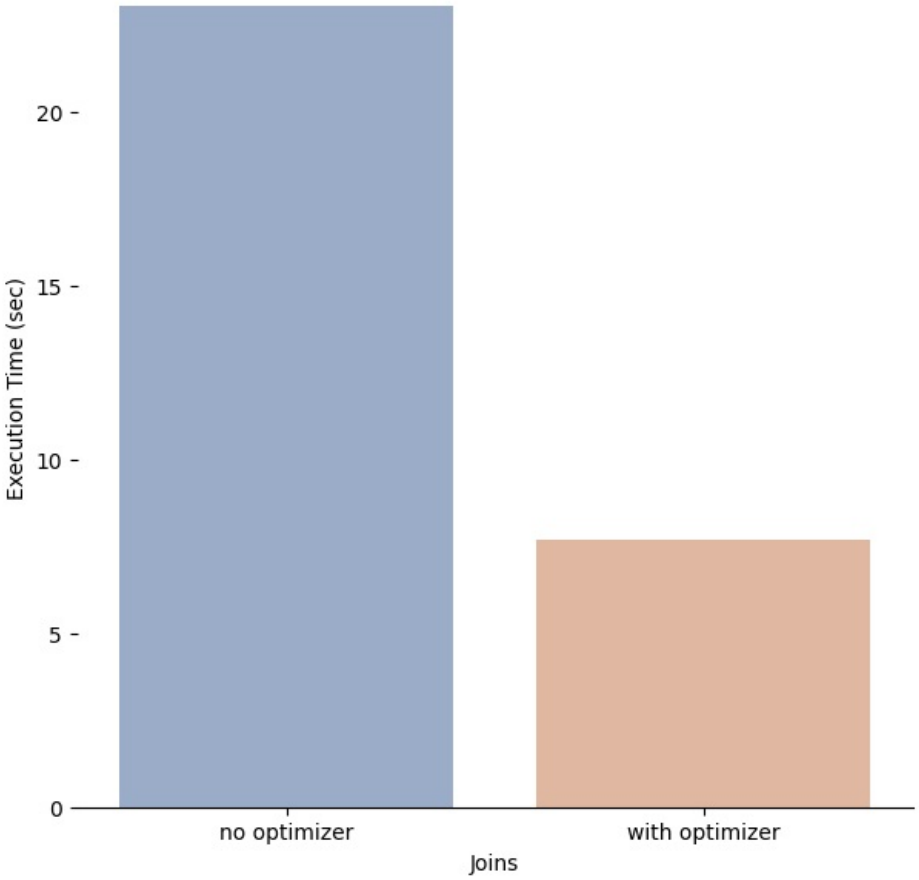
```
== Physical Plan ==
*(3) BroadcastHashJoin [movieId#8], [movieId#1], Inner, BuildLeft
:- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as bigint)))
:  +- *(2) Filter isnotnull(movieId#8)
:     +- *(2) GlobalLimit 100
:        +- Exchange SinglePartition
:           +- *(1) LocalLimit 100
:              +- *(1) FileScan parquet [movieId#8,genre#9] Batched: true, Format: Parquet, Location:
InMemoryFileIndex[hdfs://master:9000/movie_data/movie_genres.parquet], PartitionFilters: [], PushedFilters: [],
ReadSchema: struct<movieId:int,genre:string>
+- *(3) Project [userId#0, movieId#1, rating#2, timestamp#3]
   +- *(3) Filter isnotnull(movieId#1)
      +- *(3) FileScan parquet [userId#0,movieId#1,rating#2,timestamp#3] Batched: true, Format: Parquet,
Location: InMemoryFileIndex[hdfs://master:9000/movie_data/ratings.parquet], PartitionFilters: [], PushedFilters:
[IsNotNull(movieId)], ReadSchema: struct<userId:int,movieId:int,rating:double,timestamp:string>
```

Βλέπουμε ότι χρησιμοποιείται `BroadcastHashJoin` (broadcast join).

#### Ραβδοδιάγραμμα

Τοπικά, τρέχουμε το `plot_optimizer_exec_times.py` script, ώστε να δημιουργήσουμε το παρακάτω ραβδοδιάγραμμα με τους χρόνους εκτέλεσεις για τα δύο joins με ή χωρίς τον optimizer:

```
./plot_optimizer_exec_times.py optimizer_exec_times.txt
```




**Σχολιασμός Χρόνων Εκτέλεσης για κάθε join**

Παρατηρούμε, ότι απενεργοποιώντας τον optimizer για τα joins, χρησιμοποιείται το default join που είναι το repartition, ενώ με ενεργοποιημένο τον optimizer για τα joins, επιλέγει το broadcast, αφού ο ένας πίνακας είναι μικρότερος από το default όριο που θέτει το Spark για να τον κάνει broadcast. Σύμφωνα με το Spark, και το property `spark.sql.autoBroadcastJoinThreshold`, το default όριο είναι τα 10MB (Πήγη).

Η επιλογή του broadcast join, βλέπουμε ότι εκτελείται στο 1/3 του χρόνου που εκτελείται το repartition join, για τους λόγους που έχουμε αναφέρει και στο ζητούμενο 3.