Προχωρημένα Θέματα Βάσεων Δεδομένων

Εξαμηνιαία εργασία

Χρήση του Apache Spark στις Βάσεις Δεδομένων

Ακαδημαϊκό έτος 2020 - 2021

Ροή Λ - 9ο Εξάμηνο ΣΗΜΜΥ

Θωμάς Δούκας
03116081

Νικόλαος Ζέρβας
03116035

1 Εισαγωγή

Στα πλαίσια της παρούσας εργασίας καλούμαστε να χρησιμοποιήσουμε το Apache Spark για τον υπολογισμό αναλυτικών ερωτημάτων πάνω σε αρχεία που περιγράφουν σύνολα δεδομένων. Η υλοποίηση των ερωτημάτων αξιοποιεί τα RDD API και Dataframe API / Spark SQL που προσφέρονται από το Apache Spark. Για τις ανάγκες της εργασίας θα χρησιμοποιηθεί ένα σύνολο δεδομένων από ταινίες, το οποίο προέρχεται από το σύνολο Full MovieLens Dataset.

Η συγκεκριμένη αναφορά συντάσσεται με σκοπό να συμπεριλάβει απαντήσεις στις ερωτήσεις που παρατίθενται, τα σχετικά διαγράμματα, ψευδοκώδικα σε MapReduce που περιγράφει τις υλοποιήσεις πρώτου μέρους για κάθε ερώτημα και να περιγράψει συνοπτικά τα υπόλοιπα αρχεία του παραδοτέου.

2 Περιεχόμενα Παραδοτέου

Τα αρχεία που χρησιμοποιήθηκαν για την εκτέλεση των ερωτημάτων που ακολουθούν, εντοπίζονται στον φάκελο ***adb*** του master vm που κατασκευάστηκε μέσω της υπηρεσίας Okeanos. Αντίγραφα των αρχείων εκτέλεσης και των αποτελεσμάτων παρατίθενται στο παραδοτέο που συνοδεύει την παρούσα αναφορά εσωτερικά των φακέλων ***code*** και ***output*** αντίστοιχα. Επιπλέον πληροφορίες για τα vms της υπηρεσίας Okeanos που χρησιμοποιούνται παρέχονται στο αρχείο ***ip***. s

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

Στο πρώτο μέρος της εργασίας θα υπολογιστούν τα αποτελέσματα για 5 βασικά ερωτήματα του συνόλου δεδομένων που παρουσιάζουν ενδιαφέρον. Τα παραπάνω ερωτήματα υλοποιούνται με RDD API λαμβάνοντας τα απαραίτητα δεδομένα από αρχεία .csv αλλά και με Spark SQL διαβάζοντας τα δεδομένα τόσο από αρχεία *.csv* όσο και από αρχεία *.parquet*.

3.1 Ζητούμενο 1

Προκειμένου να ανεβάσουμε τα αρχεία δεδομένων *movies.csv*, *movie_genres.csv* και *ratings.csv* στο hdfs δημιουργήσαμε τον φάκελο ***hdfs://master:9000/movies/***.

3.2 Ζητούμενο 2

Η μετατροπή των αρχείων *.csv* σε *.parquet* πραγματοποιείται με την εκτέλεση του *csv_to_parquet.py* το οποίο βρίσκεται στον φάκελο *code/z1* του παραδοτέου.

3.3 Ζητούμενο 3

Στο σημείο αυτό θα παρουσιάσουμε συνοπτικά την πορεία της σκέψης και τις παραδοχές που κάναμε κατά την υλοποίηση κάθε ερωτήματος. Τόσο στις RDD όσο και στις Spark SQL υλοποιήσεις κάθε ερωτήματος ακολουθούμε την ίδια στρατηγική ως προς την λήψη αποτελεσμάτων, λαμβάνοντας υπ' όψιν μας τις υποδείξεις που παρέχονται από την εκφώνηση.

3.3.1 Παραδοχές

- Στο query 1 απαιτούμε μοναδικότητα καλύτερης ταινίας κάθε έτους, επιλέγοντας την ταινία με το μικρότερο movie_id σε περίπτωση ισοπαλίας.
- Στο query 3 θεωρούμε ότι δεν υπάρχουν εγγραφές του πίνακα ratings χωρίς κριτική. Επομένως, δεν πραγματοποιούμε κάποιον περαιτέρω έλεγχο.
- Στο query 4 θεωρούμε ως delimiter έναν ή περισσότερους κενούς χαρακτήρες. Για τον λόγο αυτό χρησιμοποιούμε το *len(x.split())* κατά την καταμέτρηση των λέξεων της εκάστοτε περίληψης.

- Στο query 5 δεν απαιτούμε μοναδικότητα για τον χρήστη περισσοτέρων κριτικών ανά είδος. Αποτέλεσμα αυτού είναι το γεγονός πως στο είδος History προκύπτουν δύο max users. Βέβαια, όπως προκύπτει από την εκφώνηση, απαιτούμε μοναδικότητα στην περίπτωση της λιγότερο και περισσότερο αγαπημένης ταινίας για κάθε max user, βασίζοντας την επιλογή μας στις κριτικές του. Σε περίπτωση ισοπαλίας επιλέγουμε βάσει της μεγαλύτερης δημοτικότητας. Τέλος, είναι σημαντικό να αναφέρουμε ότι επιλέγονται ως καλύτερες ταινίες, μόνο από εκείνες που ανήκουν στο genre που αντιστοιχεί ο συγκεκριμένος χρήστης.
- Τα queries στην περίπτωση του parquet και του csv είναι ακριβώς ίδια.

3.3.2 Ερώτημα 1 - Ταινία με το μεγαλύτερο κέρδος για κάθε χρονιά

1. **RDD**

   Για τον υπολογισμό του ερωτήματος αρκεί να διατηρήσουμε τις εγγραφές του αρχείου *movies.csv* που αναφέρονται σε ταινίες με ημερομηνία κυκλοφορίας μετά το 2000 και μη μηδενικές τιμές κόστους παραγωγής και εσόδων. Τελικά, διατηρείται η ταινία με το μέγιστο κέρδος και το μικρότερο αναγνωριστικό σε περίπτωση ισοπαλίας. Η στρατηγική της υλοποίησης σε RDD φαίνεται στον ψευδοκώδικα που ακολουθεί.

```
# read from file
map1(key, value):
    # key = null, value = line from movies.csv
    x = value.split(',')
    #  x = (
    #     movie_id, title, abstract, release_date,
    #     movie_length, production_cost, earnings, publicity
    #  )
    movie_info = tuple(x[1:])
    emit(x.movie_id, movie_info)

reduce1(key, values):
    # key = movie_id, values = list of (movie_info)
    # len(values == 1)
    for movie_info in values:
        emit(key, movie_info)

# filter with conditions
map2(key, value)
    # key = movie_id, value = movie_info
    release_year = value.release_date.split('-')[0]
    if(release_year >= 2000 and value.production_cost and value.earnings):
        earnings_percentage = ((earnings - production_cost)*100 / production_cost)
        emit(release_year, (key, value.title, earnings_percentage))

sortByKey2()

#  calculate max percentage
reduce2(key, values):
    # key = release_year, values = list of (movie_id, title, earnings_percentage)
    res = tuple()
    for v in values:
        if(resIsEmptyTuple() or res.earnings_percentage > v.earnings_percentage):
            res = res
        elif(res.earnings_percentage < v.earnings_percentage):
            res = v
        elif(res.earnings_percentage == v.earnings_percentage):
            if(res.movie_id < v.movie_id):
                res = res
            elif(res.movie_id > v.movie_id):
                res = v
    emit(key, res)
# RESULT -> (Year, (Movie_id, Title, Profit))
```

Για λόγους πληρότητας και καλύτερης εποπτείας οι ψευδοκώδικες που περιγράφουν τις υλοποιήσεις του πρώτου μέρους χαρακτηρίζονται από διακριτά στάδια εκτέλεσης. Στη συγκεκριμένη περίπτωση η σειρά εκτέλεσης ακολουθεί την αρίθμηση των σταδίων.

2. **SQL**

   Για να βρούμε τα max rows ενός πίνακα, εξασφαλίζοντας μοναδικότητα, χρησιμοποιούμε την κλασική μέθοδο της SQL, κατά την οποία πραγματοποιούμε self LEFT OUTER JOIN απαιτώντας το primary key του δεύτερου πίνακα m2._c0 να είναι Null.

   Κάνοντας JOIN με βάση το κλειδί και κάποια conditions (*profit1 < profit2 OR (profit1 = profit2 AND m1._c0 > m2._c0)*), απαιτούμε πρακτικά να επιλεχθούν μόνο οι γραμμές για τις οποίες δεν ισχύει κανένας από τους περιορισμούς. Δηλαδή, επιλέγουμε τις ταινίες με το maximum profit ή το μικρότερο movie_id σε περίπτωση ισοπαλίας.

   Δυστυχώς, η μέθοδος αυτή χαρακτηρίζεται από πολύ μεγάλη πολυπλοκότητα, η οποία όμως είναι απαραίτητη σε περίπτωση που θέλουμε να εξασφαλίσουμε μοναδικότητα αποτελεσμάτων.

   Τέλος, απαιτούμε τα έσοδα και κόστος να είναι διάφορα του μηδενός και το έτος να είναι μεγαλύτερο του 2000. Δεν χρειάζεται να διασφαλίσουμε την ύπαρξη ημερομηνίας, εφόσον αυτό ελέγχεται από το τρίτο condition.

3.3.3 Ερώτημα 2 - Ποσοστό χρηστών με μέση βαθμολογία μεγαλύτερη του 3.0

1. **RDD**

   Στο συγκεκριμένο ερώτημα ξεκινάμε υπολογίζοντας τη μέση βαθμολογία που έχει δώσει ο εκάστοτε χρήστης σε ταινίες, προσδιορίζοντας το άθροισμα των επιμέρους κριτικών κάθε χρήστη δια το πλήθος τον κριτικών του. Στο σημείο αυτό (stage 2) η ροή εκτέλεσης διασπάται σε δύο διαφορετικές, με σκοπό την καταμέτρηση του συνόλου των χρηστών (stage 3.1) αλλά και του πλήθους χρηστών με μέση βαθμολογία μεγαλύτερη του 3.0 (stage 3.2, 4).

```
# read from file
map1(key, value):
    # key = null, value = line from ratings.csv
    x.split(',') # x = user_id, movie_id, rating, rating_timestamp
    emit(x.user_id, (x.rating, rating_counter=1))

# calculate number_of_ratings_per_user, sum_of_ratings_per_user
reduce1(key, values):
    # key = user_id, values = list of (rating, rating_counter=1)
    rating_sum = 0
    count_ratings = 0
    for v in values:
        rating_sum += v.rating
        count_ratings += v.rating_counter
    emit(key, (rating_sum, count_ratings))

# calculate average_rating_per_user
map2(key, value):
    # key = user_id, value = (rating_sum, count_ratings)
    val = value.rating_sum / value.count_ratings
    emit(key, val)

reduce2(key, values):
    # key = user_id, values = list of (average_rating_per_user)
    # len(values) == 1
    for avg_rating in values:
        emit(key, avg_rating)

# use results of stage 2 to calculate total users
# COUNT
map3.1(key, value):
    # key = user_id, value = avg_rating
    constant = "count_users"
    emit(constant, total_users_counter=1)
```

```
35
36 reduce3.1(key, values):
37     # key = constant, values = list of (total_users_counter=1)
38     total_users = 0
39     for v in values:
40         total_users += v.total_users_counter
41     emit(total_users, null)
42
43 # use results from stage2. Filter users with avg_rating> 3.0
44 map3.2(key, value):
45     # key = user_id, value = avg_rating
46     if(value > 3.0):
47         emit(key, value)
48
49 reduce3.2(key, values):
50     # key = user_id, values = list of (avg_rating)
51     # len(values) == 1
52     for v in values:
53         emit(ket, v)
54
55 # use results from stage 3.2 to calculate users with avg_rating> 3.0
56 # COUNT
57 map4(key, value):
58     # key = user_id, value = avg_rating
59     constant = "count_users"
60     emit(constant, users_counter=1)
61
62 reduce4(key, values):
63     # key = constant, values = list of (users_counter=1)
64     users = 0
65     for v in values:
66         users += v.users_counter
67     emit(users, null)
68
69 # Use the results of stages 3.1 and 4 to calculate percentage
70 # RESULT -> Percentage
```

2. **SQL**

   Όπως και στην περίπτωση των RDD, διαιρούμε τον αριθμό (COUNT) των χρηστών με average rating μεγαλύτερο του 3 με το distinct count των χρηστών του πίνακα ratings. Τέλος, πολλαπλασιάζουμε τη διαφορά με 100 για να πάρουμε το ποσοστό.

3.3.4 Ερώτημα 3 - Μέση βαθμολογία και πλήθος ταινιών για κάθε είδος

1. **RDD**

   Ξεκινάμε τη διαδικασία συγκεντρώνοντας τις εγγραφές του αρχείου *ratings.csv*, προσμετρώντας το πλήθος των κριτικών για κάθε ταινία και υπολογίζοντας το άθροισμα τους. Με τον υπολογισμό του πηλίκου αυτών, υπολογίζεται η μέση βαθμολογία κάθε ταινίας. Έτσι, στο στάδιο 2 έχουμε μετασχηματίσει κατάλληλα όλη την απαραίτητη πληροφορία. Εν συνεχεία, συνενώνουμε τα παραπάνω δεδομένα με τα περιεχόμενα του αρχείου *movie_ genres.csv* χρησιμοποιώντας ως join key το αναγνωριστικό ταινίας. Αναδιατάσσοντας τα δεδομένα ως προς την κατηγορία έχουμε τη δυνατότητα να χρησιμοποιήσουμε παρόμοια διαδικασία ώστε, διαιρώντας το άθροισμα επιμέρους βαθμολογιών δια το πλήθος των ταινιών κατηγορίας, να προσδιορίσουμε τη μέση βαθμολογία κάθε είδους. Η υλοποίηση παρουσιάζεται στον ψευδοκώδικα που ακολουθεί.

   Οφείλουμε να τονίσουμε πως για την διαδικασία συνένωσης 2 δομών RDD χρησιμοποιείται η εντολή join όπως παρέχεται από το RDD API. Πρόκειται για την υλοποίηση repartition join η οποία παρουσιάζεται στον ψευδοκώδικα της προς μελέτη αναφοράς του δεύτερου μέρους της παρούσας εργασίας. Για τον λόγο αυτό, τα στάδια join του ψευδοκώδικα αναφέρουν αποκλειστικά τα δεδομένα εισόδου και τη μορφή της εξόδου της συνάρτησης. Παρόμοια πρακτική ακολουθείται για όλα τα επόμενα ερωτήματα που αξιοποιούν την συγκεκριμένη λειτουργία.

```
# read from file
map1(key, value)
    # key = null, value = line from ratings.csv
    x = value.split(',')
    # x = user_id, movie_id, rating, rating_timestamp
    emit(x.movie_id, (x.rating, rating_counter=1))

# calculate sum_of_ratings_per_movie and number_of_ratings_per_movie
reduce1(key, values)
    # key = movie_id, values = (rating, rating_counter=1)
    count_ratings = 0
    rating_sum = 0
    for v in values:
        count_ratings += v.rating_counter
        rating_sum += v.rating
    emit(key, (rating_sum, count_ratings))

# calculate average_rating_per_movie
map2(key, value):
    # key = movie_id, value = (rating_sum, count_ratings)
    average_rating_per_movie = value.rating_sum / value.total_rates
    emit(key, average_rating_per_movie)

reduce2(key, values):
    # key = movie_id, values = list of (average_rating_per_movie)
    # len(values) == 1
    for v in values:
        emit(key, v)
# All essential data from ratings.csv are gathered

# read from file
map3(key, value):
    # key = null, value = line from movie_genres.csv
    x = value.split(',')
    # x = movie_id, genre
    emit(x.movie_id, x.genre)

reduce3(kay, values):
    #key = movie_id, value = list of (genre)
    for v in values:
        emit(key, v)

MR stage 4
# join stage 3 data with stage 2 data (join_key = key)
# -> result in (movie_id, (genre, avg_rating_per_movie))

# rearrange with genre as key and calculate sum_of_avg_ratings and movies_in_genre
map5(key, value):
    # key = movie_id, value = (genre, avg_rating_per_movie)
    new_key = value.genre
    emit(new_key, (value.avg_rating_per_movie, movies_in_genre_counter=1))

reduce5(key, values):
    # key = genre, values = list of (avg_rating_per_movie, movies_in_genre_counter=1)
    sum_of_avg_ratings = 0
    count_movies_in_genre = 0
    for v in values:
        sum_of_avg_ratings += v.avg_rating_per_movie
        count_movies_in_genre += v.movies_in_genre_counter
    emit(key, (sum_of_avg_ratings, count_movies_in_genre))

map6(key, value): # calculate genre_rating
    # key = genre, value = (sum_of_avg_ratings, count_movies_in_genre)
    genre_rating = sum_of_avg_ratings/count_movies_in_genre
    emit(key, (genre_rating, count_movies_in_genre))
```

```
sortByKey6()

reduce6(key, values):
    # key = genre, values = list of (genre_rating, count_movies_in_genre)
    # len(values) == 1
    for v in values:
        emit(key, v)

# RESULT -> (Genre, (Genre rating, Movies in genre))
```

2. **SQL**

   Κάνουμε JOIN τον πίνακα *movie_genres* με τον πίνακα που περιέχει το average rating για κάθε ταινία και κατόπιν παίρνουμε το average των average_ratings ανά genre και το COUNT των ταινιών ανά είδος.

3.3.5 Ερώτημα 4 - Μέσο μήκος περίληψης της κατηγορίας Drama ανά πενταετία

1. **RDD**

   Για τον υπολογισμό του μέσου μήκους περίληψης συγκεντρώνουμε το σύνολο των ταινιών που ανήκουν στην κατηγορία Drama από το αρχείο *movie_genres.csv*. Ταυτόχρονα, μας ενδιαφέρει να διατηρήσουμε τις ταινίες με έτος κυκλοφορίας μεγαλύτερο ή ίσο του 2000 προσδιορίζοντας το μήκος περίληψης για κάθε μία από αυτές. Οι πληροφορίες αυτές λαμβάνονται από το αρχείο *movies.csv*, τα αποτελέσματα του οποίου στη συνέχεια θα συνενώσουμε με εκείνα του *movie_genres.csv*. Στο σημείο αυτό (stage 4) έχουμε συγκεντρώσει το σύνολο των απαραίτητων πληροφοριών. Αρκεί να προσδιορίσουμε την πενταετία στην οποία ανήκει η κάθε ταινία και ύστερα με βάση αυτή να καταμετρήσουμε το σύνολο των δραματικών ταινιών και το άθροισμα των λέξεων περιλήψεων τους. Στο τελικό στάδιο του ερωτήματος προσδιορίζεται το μέσο μήκος των περιλήψεων ανά πενταετία.

```
# read from file and extract only Drama movies
map1(key, value):
    # key = null, value = line from movie_genres.csv
    x = value.split(',')
    # x = movie_id. genre
    if(x.genre == 'Drama'):
        emit(x.movie_id, x.genre)

reduce1(key, values):
    # key = movie_id, values = list of (genre)
    len(values) == 1
    for v in values:
        emit(key, v)
# All essential data from movie_genres.csv are gathered

# read from file
map2(key, value):
    # key = null, value = line from movies.csv
    x = value.split(',')
    #  x = (
    #     movie_id, title, abstract, release_date,
    #     movie_length, production_cost, earnings, publicity
    #  )
    emit(x.movie_id, (x.release_date, x.abstract))

reduce2(key, values):
    # key = movie.id, values = list of (release_date, abstract)
    # len(values) == 1
    for v in values:
        emit(key, v)

# filter with conditions
map3(key, value):
    # key = movie_id, value = (release_date, abstract)
    year = value.release_date.split(',')[0]
```

```
    if(value.abstract and  year >= 2000):
        val = (year, len(value.abstract.split(), movie_counter=1)
        emit(key, val)

reduce3(key, values):
    # key = movie_id, values = list of (year, words_per_abstract, movie_count=1)
    # len(values) == 1
    for v in values:
        emit(key, v)
# All essential data from movies.csv are gathered

MR stage 4
# join stage 3 data with stage 1 data (join_key = key) to keep only Drama movies
# -> result in (movie_id, (release_year, abstract_words, movie_count=1, genre=Drama))

# rearrange with key as quinquennium and calculate total_abstract_length,
    total_drama_movies
map5(key, value):
    # key = movie_id, value = (release_year, abstract_words, movie_count=1, genre=
    Drama)
    temp = (x.realease_year//5)*5
    newKey = str(temp) + '-' + str(temp+4)
    val = (abstract_words, movies_count)
    emit(newKey, val)

reduce5(key, values):
    # key = quinquennium, values = list of (abstract_words, movies_count)
    total_abstract_length = 0
    total_drama_movies = 0
    for v in values:
        total_abstract_length += v.abstract_words
        total_drama_movies += v.movies_count
    emit(key, (total_abstract_length, total_drama_movies))

# calculate average_abstract_length for Drama genre
map6(key, value):
    # key = quinquennium, value = (total_abstract_length, total_drama_movies)
    avg_abstract_length = value.total_abstract_length / value.total_drama_movies
    emit(key, avg_abstract_length)

sortByKey6()

reduce6(key, values):
    # key = quinquennium, values = list of (total_abstract_length, total_drama_movies)
    # len(values) == 1
    for v in values:
        emit(key, v)

# RESULT -> (Five Years, Average words in Abstract)
```

2. **SQL**

   Κατασκευάζουμε τα udf wordcount και fiveyear τα οποία λειτουργούν ανά στήλη. Αρχικά, κατασκευάζουμε τον πίνακα movie_id, fiveyear, wordcount, κάνοντας INNER JOIN τον πίνακα movies με τον πίνακα movie_genres απαιτώντας το είδος να είναι drama και το έτος μεγαλύτερο ή ίσο του 2000. Τέλος, παίρνουμε το average wordcount ανά genre.

3.3.6 Ερώτημα 5 - Περισσότερο και λιγότερο αγαπημένη ταινία του χρήστη με τις περισσότερες κριτικές ανά κατηγορία

1. **RDD**

   Για την εκτέλεση του τελευταίου ερωτήματος η ροή της εκτέλεσης χωρίζεται σε συγκεκριμένα διακριτά βήματα. Πρωταρχικό στάδιο της διαδικασίας είναι η ανάγνωση των απαραίτητων δεδομένων από τα αντίστοιχα αρχεία. Το βήμα αυτό περιγράφεται αναλυτικά στα MapReduce στάδια 0.1 - 0.4 του ψευδοκώδικα.

Χρησιμοποιώντας τις κατάλληλες δομές επιδιώκουμε να προσδιορίσουμε τους χρήστες με τις περισσότερες κριτικές ανά είδος ταινιών. Για το σκοπό αυτό πραγματοποιούμε join ανάμεσα στα rdd *movie_ genres - ratings* και χρησιμοποιώντας το συνδυασμό (genre, user_id) μετράμε το πλήθος κριτικών του εκάστοτε χρήστη. Θέτοντας ως κύριο κλειδί το genre έχουμε την δυνατότητα να συγκρίνουμε τους επιμέρους χρήστες ως προς το πλήθος κριτικών τους πάνω σε ταινίες συγκεκριμένης κατηγορίας. Στο στάδιο αυτό λαμβάνουμε μέριμνα ώστε σε περίπτωση ισοβαθμίας των κορυφαίων χρηστών να δημιουργούνται ξεχωριστές εγγραφές για κάθε έναν από αυτούς στο τελικό αποτέλεσμα.

Έτσι, γνωρίζουμε τους χρήστες για κάθε είδος ταινιών και καλούμαστε να αναζητήσουμε τις περισσότερο και λιγότερο αγαπημένες ταινίες τους με την προϋπόθεση να ανήκουν στη συγκεκριμένη κατηγορία. Για την επιλογή ταινίας, όπως αναφέρθηκε, βασιζόμαστε στη κριτική του χρήστη ενώ σε περίπτωση ισοβαθμίας ελέγχουμε την τιμή της δημοτικότητας (popularity). Ως εκ τούτου, θα χρειαστούμε πληροφορίες από το σύνολο των αρχείων του dataset (*movies.csv, movie_ genre.csv και ratings.csv*). Χρησιμοποιώντας ως κύριο κλειδί το user_id πραγματοποιούμε διαδοχικά join και δημιουργούμε το σύνολο δεδομένων πάνω στο οποίο θα βασιστούμε για τον προσδιορισμό των ταινιών.

Στο τελευταίο στάδιο της υλοποίησης η ροή εκτέλεσης διαχωρίζεται σε 2 επιμέρους εργασίες. Χρησιμοποιώντας τα αποτελέσματα από το προηγούμενο βήμα, επιδιώκουμε να προσδιορίσουμε περισσότερο και λιγότερο αγαπημένη ταινία αντίστοιχα. Η συνένωση αυτών μας επιστρέφει το επιθυμητό αποτέλεσμα.

Η ροή εκτέλεσης του ερωτήματος παρουσιάζεται αναλυτικότερα στο ψευδοκώδικα που ακολουθεί.

```
# read information from all files
map0.1(key, value) # movie_genres
    # key = null, value = line from movie_genres.csv
    x = value.split(',')
    # x = movie_id, genre
    emit(x.movie_id, x.genre)

reduce0.1(key, values):
    # key = movie_id, values = list of (genre)
    for v in values:
        emit(key, v)

map0.2(key, value) #movies
    # key = null, value = line from movies.csv
    x = value.split(',')
    #  x = (
    #     movie_id, title, abstract, release_date,
    #     movie_length, production_cost, earnings, publicity
    #  )
    val = (x.popularity, x.movie_name)
    emit(x.movie_id, val)

reduce0.2(key, values):
    # key = movie_id, values = list of (popularity, name)
    for v in values:
        emit(key, v)

map0.3(key, value) #ratings
    # key = null, value = line from ratings.csv
    x = value.split(',')
    # x = user_id, movie_id, rating, rating_timestamp
    val = (x.user_id, x.rating, ratings_counter=1)
    emit(x.movie_id, val)

reduce0.3(key, values):
    # key = movie_id, values = list of (user_id, rating, ratings_counter=1)
    for v in values:
        emit(key, v)

map0.4(key, value) #ratings2
    # key = movie_id, value = (user_id, rating, ratings_counter=1)
    val = (key, value.rating)
    emit(value.user_id, val)

```

```
reduce0.4(key, values):
    # key = movie_id, values = list of (user_id, rating, ratings_counter=1)
    for v in values:
        emit(key, v)

# start by finding the user with the most movie ratings per genre
MR stage 1
# # join stage 0.1 data with stage 0.3 data (genres.join(ratings))
# -> result in (movie_id, (genre, user_id, rating, ratings_counter=1))

# calculate ratings_of_user_in_genre
map2(key, value):
    # key = movie_id, value = (genre, user_id, ratings_counter=1)
    newKey = (value.genre, value.user_id)
    emit(newKey, value.ratings_counter=1)

reduce2(key, values):
    # key = (genre, user_id), values = list of ratings_counter=1
    ratings_of_user_in_genre = 0
    for v in values:
        ratings_of_user_in_genre += ratings_counter
    emit(key, ratings_of_user_in_genre)

# rearrange elements
map3(key, value):
    # key = (genre, user_id), value = ratings_of_user_in_genre
    newKey = key.genre
    val = (key.user_id, value)
    emit(newKey, val)

# keep users with most ratings per genre.
# all users with the same amount of max ratings in genre will move to the next map
    reduce stage
reduce3(key, values)
    # key = genre, values = list of (user_id, ratings_of_user_in_genre)
    res = tuple() #Holds tuple of users with max_number_of_ratings at the moment
    for v in values:
        if(resIsEmptyTuple() or res[0].ratings_of_user_in_genre == v.
    ratings_of_user_in_genre):
            res += v # keep users with same number of_ratings
        elif(res[0].ratings_of_user_in_genre < v.ratings_of_user_in_genre)
            res = tuple(v)
            # all previous values of res have the same number of ratings, which is
    smaller than v's
        elif(res[0].ratings_of_user_in_genre > v.ratings_of_user_in_genre)
            res = res # keep previous users
    # emit all distinct users with maximum number of ratings in genre -> flatmap in
    rdd code
    for r in res:
        emit(key, r)

# rearrange with user_id as key
map4(key, value):
    # key = genre, value = (user_id, ratings_of_user_in_genre)
    newKey = value.user_id
    val = (key, value.ratings_of_user_in_genre)
    emit(newKey, val)

reduce4(key, values):
    # key = user_id, values = genre, ratings_of_user_in_genre
    # len(values) == 1
    for v in values:
        emit(key, v)

# create the dataset to find most and least favorite movies of user in the genre
MR stage 5
# # join stage 4 data with stage 0.4 data (ratings2)
```

```
# -> result (user_id, (genre, num_ratings, movie_id, rating))

map6(kay, value):
    # key = user_id, value = (genre, num_ratings, movie_id, rating)
    newKey = value.movie_id
    var = (value.genre, key, value.rating, value.num_ratings)
    emit(newKey, var)

reduce6(key, value):
    # key = movie_id, value = (category, user_id, rating, num_ratings)
    # len(values) == 1
    for v in values:
        emit(key, v)

MR stage 7
# # join stage 6 data with stage 0.1 data (genres)
# -> result (movie_id, genre, user_id, rating, num_ratings, category)
#  genre = genre in witch user has max ratings

# filter to keep only the genre in witch user has max ratings
map8(kay, value):
    # key = movie_id, value = (movie_id, genre, user_id, rating, num_ratings, category
    )
    if(value.genre == value.category)
        emit(newKey, value)

reduce8(key, value):
    # key = movie_id, value = (movie_id, genre, user_id, rating, num_ratings, category
    )
    # len(values) == 1
    for v in values:
        emit(key, v)

MR stage 9
# join stage 8 data with stage 0.2 data (movies)
# -> result (movie_id, category, user_id, rating, num_ratings, movie_name, popularity)

map10(kay, value):
    # key = movie_id,
    # value = (category, user_id, rating, num_ratings, movie_name, popularity)
    newKey = (category, user_id, num_ratings)
    var = (movie_id, movie_name, rating, popularity)
    emit(newKey, var)

reduce10(key, value):
    # key = (category, user_id, num_ratings)
    # values = list of (movie_id, movie_name, rating, popularity)
    # len(values) == 1
    for v in values:
        emit(key, v)

# at this point all essential data have been selected
# continue by calculating most favorite and least favorite movie
# use popularity as tie breaker

# take data of stage 10
map11.1(key, value): #most_favorite
    # key = (category, user_id, num_ratings)
    # value = (movie_id, movie_name, rating, popularity)
    emit(key, value)

reduce11.1(key, values):
    # key = (category, user_id, num_ratings)
    # values = list of (movie_id, movie_name, rating, popularity)
    res = tuple() # always one value inside res
    for v in values:
        if(resIsEmptyTuple() or res.rating > v.rating):
```

```
            res = res
        elif (res.rating < v.rating):
            res = v
        elif (res.rating == v.rating):
            res = res if(res.popularity >= v.popularity) else v
    emit(key, res)

# take data of stage 10
map11.2(key, value): #least_favorite
    # key = (category, user_id, num_ratings)
    # value = (movie_id, movie_name, rating, popularity)
    emit(key, value)

reduce11.2(key, values):
    # key = (category, user_id, num_ratings)
    # values = list of (movie_id, movie_name, rating, popularity)
    res = tuple() # always one value inside res
    for v in values:
        if(resIsEmptyTuple() or res.rating > v.rating):
            res = v
        elif (res.rating < v.rating):
            res = res
        elif (res.rating == v.rating):
            res = res if(res.popularity >= v.popularity) else v
    emit(key, res)

MR stage 12
# join stage 11.1 data with stage 11.2 data
# -> (category, user_id, $ratings, most_fav_movie_id, most_fav_movie_name, rating,
    least_fav_movie_id, least_fav_movie_name, rating)
```

2. **SQL**

   Προσπαθούμε αρχικά να κατασκευάσουμε τον πίνακα res1 των max users ανά genre. Δημιουργούμε τον πίνακα s1, με όλους τους users και τον αριθμό των κριτικών τους ανά genre. Εφόσον δεν απαιτούμε μοναδικότητα, μπορούμε να μειώσουμε την πολυπλοκότητα κάνοντας INNER JOIN τον πίνακα s1 με τον max πίνακα του s1 (s2), απαιτώντας ο αριθμός των κριτικών ανά genre να είναι ίσος με τον max αριθμό κριτικών ανά genre. Τέλος, αποθηκεύουμε τον πίνακα res1 ως temporary table ώστε να μην απαιτείται εκ νέου ο υπολογισμός του κάθε φορά που χρησιμοποιείται.

   Ακολούθως, προσπαθούμε να κατασκευάσουμε τον πίνακα res3 που περιέχει τους χρήστες, τις ταινίες που έχουν βαθμολογήσει, τα είδη των ταινιών αυτών, το rating αλλά και το popularity τους. Μειώνουμε την πολυπλοκότητα απαιτώντας ο πίνακας αυτός να αφορά μόνο τους distinct users του res1.

   Κατόπιν, πραγματοποιούμε JOIN με τον πίνακα res1 βάσει των genre και user. Επομένως, προκύπτει ο πίνακας res4 που περιέχει για κάθε genre τον/τους max users, τον αριθμό κριτικών για κάθε user και τις ταινίες που έχει βαθμολογήσει με την προϋπόθεση ότι ανήκουν στο αντίστοιχο genre.

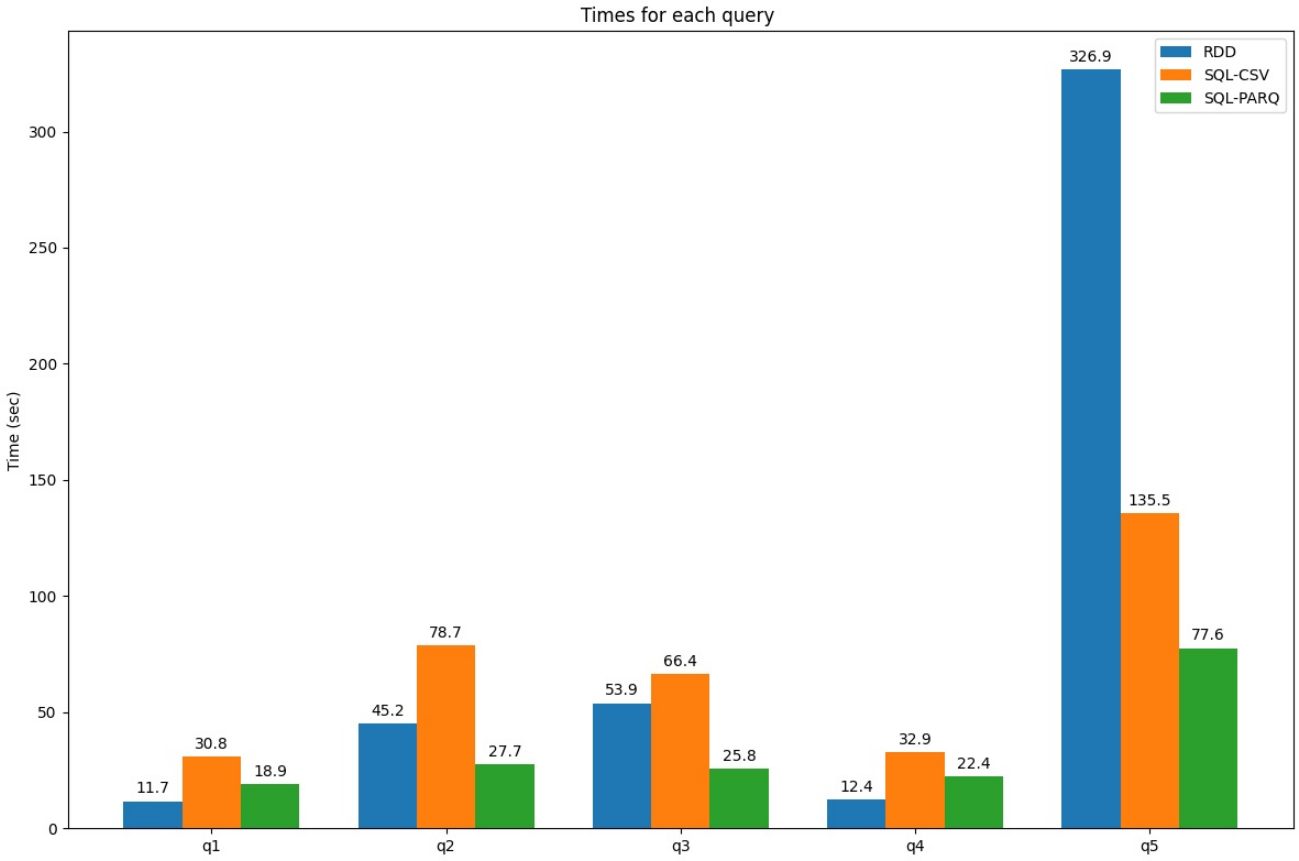
   Χρησιμοποιούμε την τεχνική του query 1 κάνοντας self LEFT OUTER JOIN τον πίνακα res4, μία φορά για τις πιο δημοφιλείς λιγότερο αγαπημένες ταινίες και άλλη μία για τις πιο δημοφιλείς περισσότερο αγαπημένες ταινίες.Τέλος, εκτελούμε INNER JOIN ανάμεσα στους δύο πίνακες που κατασκευάστηκαν ώστε να προκύψει το τελικό αποτέλεσμα.

Αναφέρουμε ότι τόσο οι υλοποιήσεις σε RDD και Spark SQL όσο και τα αρχεία του ψευδοκώδικα που περιγράφει τις πρώτες, εμπεριέχονται στα φάκελο *code/z1* του παραδοτέου.

3.4 Ζητούμενο 4

Μετρήσαμε τους χρόνους εκτέλεσης μέσω της εντολής time spark-submit. Τα αποτελέσματα των εκτελέσεων και τα αρχεία καταγραφών-logs αποθηκεύτηκαν στους φακέλους logs και results εσωτερικά του z1 στο master vm του Okeanos, ενώ εμπεριέχονται και στο φάκελο ***output*** του παραδοτέου. Συγκεντρωτικά οι αντίστοιχοι χρόνοι παρατίθενται στο αρχείου *times.txt* του παραδοτέου. Παρακάτω παρουσιάζονται σε barplot οι χρόνοι (real) για τους οποίους προκύπτουν οι εξής παρατηρήσεις και συμπεράσματα:

3.4.1 Διαγράμματα:



Σχήμα 1: Χρόνοι εκτέλεσης ερωτημάτων

3.4.2 Παρατηρήσεις:

- Το Spark SQL με αρχεία parquet φαίνεται να επιτυγχάνει καλύτερους χρόνους για όλα τα ερωτήματα σε σχέση με το csv. Οι λόγοι γι' αυτό είναι:

  1. Καλύτερο compression του αρχείου λόγω του columnar based format.
  2. Βελτιστοποίηση του I/O.
  3. Λόγω του columnar based format είναι δυνατός ο χωρισμός σε blocks στα οποία αποθηκεύονται οι min, max τιμές. Ως εκ τούτου, κατά το φιλτράρισμα χρειάζεται να ελεγχθούν πολύ λιγότερες γραμμές του πίνακα με αποτέλεσμα να βελτιώνεται ο χρόνος εκτέλεσης. Το φαινόμενο αυτό παρατηρείται περισσότερο όσο τα αρχεία που χρησιμοποιούνται γίνονται μεγαλύτερα σε μέγεθος.
  4. Για το διάβασμα του parquet αρχείου δεν χρειάζεται να συμπεριληφθεί inferSchema εφόσον το schema είναι αποθηκευμένο κατά την κατασκευή του. Το inferSchema απαιτεί ένα επιπλέον πέρασμα από το αρχείο κατά την επεξεργασία του αυξάνοντας τον χρόνο εκτέλεσης.

  Με βάση τα παραπάνω μπορούμε να θεωρήσουμε την χρήση SparkSQLμε είσοδο το parquet αρχείο την πιο αποδοτική από τις 2.

- Βασικό χαρακτηριστικό των queries q1, q4 είναι ότι διαχειρίζονται τα μικρότερα αρχεία *movies*, *movie_ genres* και όχι το αρχείο *ratings*. Στην περίπτωση αυτή, φαίνεται ότι το RDD API παρουσιάζει μακράν καλύτερη επίδοση από τις δύο περιπτώσεις του Spark SQL. Ταυτόχρονα, λόγω του μικρού μεγέθους αρχείων οι parquet και csv υλοποιήσεις δεν έχουν τόση μεγάλη χρονική διαφοροποίηση. Ωστόσο, είναι εμφανές ότι η είσοδος parquet εξακολουθεί να εμφανίζει μικρότερους χρόνους εκτέλεσης.

- Στα ερωτήματα q2,q3 όπου χρησιμοποιείται το μεγαλύτερο σε μέγεθος αρχείο *ratings*, παρατηρούμε ότι το rdd api εμφανίζει καλύτερη επίδοση από το Spark SQL csv αλλά χειρότερη από το Spark SQL

parquet. Επιπλέον, η διαφορά μεταξύ της csv και parquet υλοποίησης φαίνεται αρκετά μεγαλύτερη και ιδιαίτερα στην περίπτωση του q2 όπου χρησιμοποιούμε δύο φορές τον πίνακα ratings.

- Το τελευταίο αποτελεί το πολυπλοκότερο ερώτημα από όσα μελετήσαμε. Έχει ενδιαφέρον να μελετήσουμε την περίπτωση του προσπαθώντας να λάβουμε συμπεράσματα για την επίδοση των διάφορων τεχνολογιών. Το γεγονός πως, το q5 αποτελεί ερώτημα που όχι μόνο χρησιμοποιεί μεγάλα σε μέγεθος αρχεία αλλά απαιτεί και σημαντικό πλήθος υπολογισμών, μας στερεί την εγγύηση οι υλοποιήσεις μας είναι οι βέλτιστες δυνατές. Ωστόσο, με τα δεδομένα αποτελέσματα παρατηρούμε εκτόξευση του χρόνου επεξεργασίας στο RDD API σε σχέση με το Spark SQL. Επιπρόσθετα, τα πλεονεκτήματα του parquet επιτρέπουν σημαντική μείωση του απαιτούμενου χρόνου τόσο σε σχέση με το RDD όσο και με το αντίστοιχο csv.

3.4.3 Συμπεράσματα:

- Για σχετικά απλά ερωτήματα σε μικρότερα αρχεία (της τάξης των δεκάδων mb) φαίνεται ότι το RDD API εμφανίζει πολύ πιο ικανοποιητικές επιδόσεις από το SparkSql. Ταυτόχρονα, η βελτιστοποίηση του parquet αρχείου αν και παρουσιάζει μικρότερους χρόνους εκτέλεσης, δεν φαίνεται να επηρεάζει τόσο πολύ την ταχύτητα της επεξεργασίας σε σχέση με το csv.

- Για εξίσου απλά ερωτήματα σε μεγαλύτερα αρχεία (της τάξης των εκατοντάδων mb) φαίνεται ότι το Spark SQL parquet αποτελεί καλύτερη πρακτική, παρουσιάζοντας έντονη η διαφορά χρόνου επεξεργασίας σε σχέση με το csv.

- Για συνθετότερα queries με μεγάλα αρχεία φαίνεται ότι η υλοποίηση με RDD γίνεται ανεπαρκής. Η Spark SQL αποτελεί βέλτιστη επιλογή για τέτοιου είδους προβλήματα με την επιλογή μορφοποίησης αρχείου να παίζει σημαντικό ρόλο.

Καταληκτικά, βλέπουμε ότι η επιλογή ανάμεσα σε RDD, Spark SQL με αρχεία csv και Spark SQL με αρχεία parquet εξαρτάται από πληθώρα παραγόντων. Είναι απαραίτητο να λαμβάνουμε υπόψη μας τόσο το μέγεθος του dataset όσο και την πολυπλοκότητα του προβλήματος που καλούμαστε να επιλύσουμε.

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

4.1 Ζητούμενο 1 - Υλοποίηση broadcast join

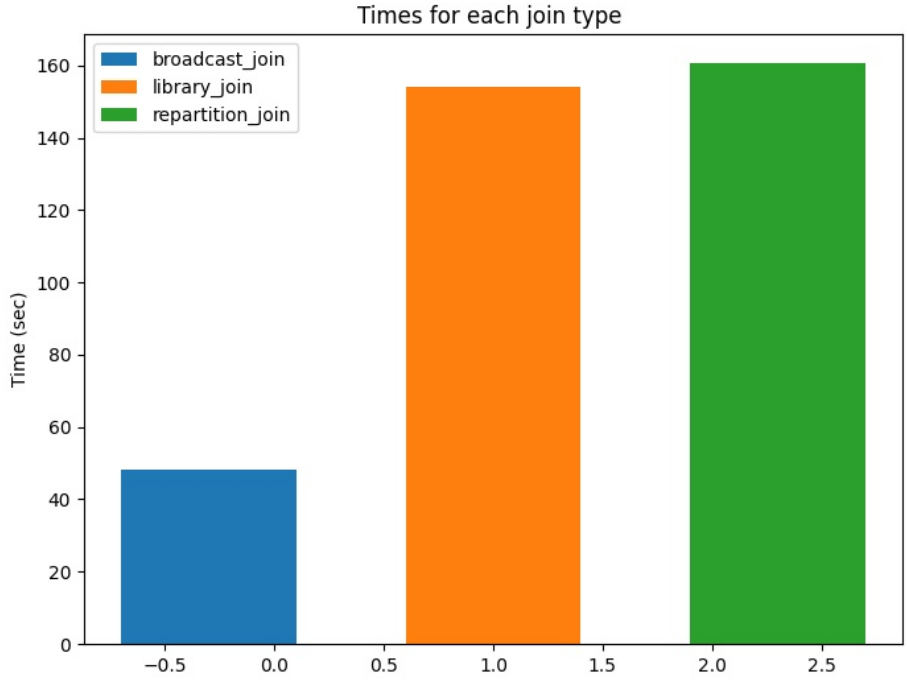
Υλοποιήθηκε το broadcast join ως μέθοδος της κλάσης RDD και βρίσκεται στον φάκελο *code/z2/broadcast/* του παραδοτέου.

4.2 Ζητούμενο 2 - Υλοποίηση repartition join

Υλοποιήθηκε το repartition join ως μέθοδος της κλάσης RDD και βρίσκεται στον φάκελο *code/z2/repartition/* του παραδοτέου.

4.3 Ζητούμενο 3

Προκειμένου να απομονώσουμε τις πρώτες 100 γραμμές του αρχείου *movie_ genres.csv* αρκεί να εκτελεστεί το πρόγραμμα *reduce_ genres.py*. Χρησιμοποιώντας το παραγόμενο αρχείο εκτελούμε, για τις πρώτες 100 σειρές του πίνακα movie genres και τον πίνακα ratings, το broadcast join, την υλοποίηση join της βιβλιοθήκης του Spark (repartition join) και την δική μας έκδοση του repartition join. Αναλυτικότερα τα αποτελέσματα δίνονται στον φάκελο *output/z2/* του παραδοτέου. Οι χρόνοι εκτέλεσης μετρήθηκαν μέσω της εντολής time spark-submit και τα αποτελέσματα παρουσιάζονται στο διάγραμμα που ακολουθεί.



Σχήμα 2: Χρόνοι εκτέλεσης υλοποιήσεων join

Φαίνεται ότι η υλοποίηση μας για το repartition join έχει παρόμοια αποτελέσματα με το join του RDD API. Αντιθέτως το broadcast join εμφανίζει πολύ καλύτερα αποτελέσματα. Οι λόγοι που οδηγούν σε αυτό είναι:

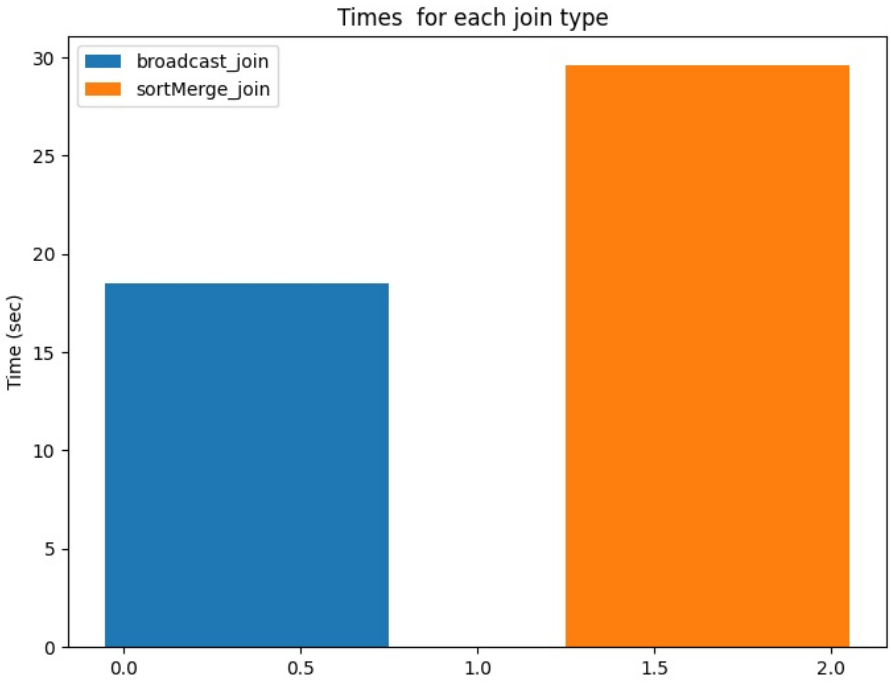
- Στην περίπτωση του repartition join ξεκινάμε αποδίδοντας τα κατάλληλα tags σε κάθε εγγραφή, προκειμένου να είναι γνωστός ο πίνακας προέλευσης καθεμιάς από αυτές. Κατόπιν το σύνολο των εγγραφών (*union*) ομαδοποιείται με βάση το κλειδί (*groupByKey*) γεγονός που προκαλεί αναδιάταξη των δεδομένων (partition, sort and merge) από το framework. Στη συνέχεια, τα δεδομένα που αντιστοιχούν σε κάθε join key, εισάγονται στον reducer ο οποίος διαχωρίζει τις εγγραφές ανάλογα με το tag και αποθηκεύει τα παραγόμενα σύνολα σε 2 buffers. Τέλος, για την λήψη των αποτελεσμάτων συνένωσης αρκεί να επιστραφεί το cross product των παραπάνω συνόλων. Η διαδικασία αυτή εισάγει σημαντικό overhead στον υπολογισμό του αποτελέσματος. Η καθυστέρηση που δημιουργείται οφείλεται κυρίως στην αναδιάταξη και το sorting ενός μεγάλου πίνακα, με την απαίτηση μεταφοράς μεγάλου όγκου δεδομένων στο δίκτυο.
- Αντίθετα, στο broadcast join δεν χρειάζεται sorting ή μετακίνηση του μεγαλύτερου πίνακα L. Αρχικά, προσδιορίζουμε το hashtable του μικρότερου πίνακα R (*groupByKey, collectAsMap*) και πραγματοποιούμε broadcast του παραγόμενου dictionary. Ακολούθως, ενώνουμε κάθε γραμμή του πίνακα L με κάθε στοιχείο της λίστας του H_r που αντιστοιχεί στο ίδιο κλειδί. Χρησιμοποιώντας την παραπάνω στρατηγική επιτυγχάνεται σημαντική μείωση του απαιτούμενο φόρτου, καθώς αρκεί η μεταφορά ενός μικρού σε μέγεθος hashtable προς τα μηχανήματα του συστήματος. Κάθε μηχάνημα διατηρεί cached το hashtable ώστε να μην απαιτείται εκ νέου η μεταφορά δεδομένων κάθε φορά που χρησιμοποιούνται από ενα task.

Αναφέρουμε πως οι αντίστοιχες υλοποιήσεις και τα αποτελέσματα εκτέλεσης βρίσκονται στο φάκελο *adb/code/z2* του master vm στον Okeanos.

4.4 Ζητούμενο 4

Για την εκτέλεση του ερωτήματος μετασχηματίζουμε κατάλληλα το script που παρέχεται από την εκφώνηση. Η τελική έκδοση περιέχεται στο αρχείο *code/z2/sql/sql_join.py* του παραδοτέου. Στο σημείο αυτό τονίζουμε ότι SparkSQL με απενεργοποιημένο τον βελτιστοποιητή χρησιμοποιεί την μέθοδο sort merge join. Όπως

αναφέρεται και στο paper στο οποίο βασιζόμαστε, η μέθοδος repartition join για τα RDDs είναι ανάλογη της μεθόδου sort merge join για τα παράλληλα RDBMS. Αντιθέτως, η ενεργοποίηση του βελτιστοποιητή οδηγεί στην εφαρμογή του broadcast join από το Spark SQL. Τα αποτελέσματα της εκτέλεσης παρουσιάζονται στο διάγραμμα που ακολουθεί.



Σχήμα 3: Χρόνοι εκτέλεσης υλοποιήσεων join σε Spark SQL

Παρατηρούμε, όπως και στο προηγούμενο ερώτημα, ότι το broadcast join έχει καλύτερα αποτελέσματα, 24s σε σχέση με 29s. Η χρονική διαφοροποίηση, βέβαια, φαίνεται ακριβέστερα με την χρήση της βιβλιοθήκης time (broadcast : 5s , sortMerge: 15s). Για την ερμηνεία των αποτελεσμάτων αρκεί να παρατηρήσουμε το πλάνο εκτέλεσης που παράγει ο βελτιστοποιητής σε κάθε μια από τις προηγούμενες περιπτώσεις. Όπως βλέπουμε από το *spark.sql(query).explain()* η μέθοδος SortMerge join προβαίνει πρώτα σε ταξινόμηση και των δύο πινάκων και κατόπιν εκτελεί merge. Αντίθετα, όπως σε προηγούμενα ερωτήματα, η Broadcast Join κατασκευάζει τον hashtable του πίνακα movie_genres, πραγματοποιεί broadcast και στο τέλος υλοποιεί το hash join. Αυτό το επιπλέον sorting του μεγαλύτερου πίνακα προσθέτει επιπλέον πολυπλοκότητα.

```
*(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[hdfs://master:9000/movies/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: InMemoryFileIndex[hdfs://master:9000/movies/ratings.parquet],
    PartitionFilters: [], PushedFilters: [IsNotNull(_c1)], ReadSchema: struct<_c0:string,_c1
    :string,_c2:string,_c3:string>
```

```
Broadcast Join :
== 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/movies/movie_genres.parquet],
    PartitionFilters: [], PushedFilters: [], ReadSchema: struct<_c0:string,_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/movies/ratings.parquet], PartitionFilters
    : [], PushedFilters: [IsNotNull(_c1)], ReadSchema: struct<_c0:string,_c1:string,_c2:
    string,_c3:string>
```