Προχωρημένα Θέματα Βάσεων Δεδομένων

Εξαμηνιαία Εργασία

Χρήση του Apache Spark στις Βάσεις Δεδομένων

Φοίβος-Ευστράτιος Καλεμκερής
03116010

Χρήστος Χειλετζάρης
03116645

1 Εισαγωγή

Στην παρούσα εργασία, κληθήκαμε να χρησιμοποιήσουμε το Apache Spark για τον υπολογισμό αναλυτικών ερωτημάτων πάνω σε αρχεία που περιγράφουν σύνολα δεδομένων. Για τις ανάγκες της εργασίας χρησιμοποιήθηκε ένα σύνολο δεδομένων από ταινίες, το οποίο προέρχεται από το σύνολο Full MovieLens Dataset. Ειδικότερα, αξιοποιήθηκε η πληροφορία από 3 CSV αρχεία: movie_genres.csv, movies.csv και ratings.csv

2 Υπολογισμός Αναλυτικών Ερωτημάτων με το Apache Spark

Στο πρώτο μέρος της εργασίας χρησιμοποιήσαμε τα APIs του Apache Spark για τον υπολογισμό 5 ερωτημάτων που παρουσιάζουν ενδιαφέρον από το διαθέσιμο σύνολο δεδομένων.

2.1 Φόρτωση CSV στο HDFS

Προτού προχωρήσουμε στην υλοποίηση των ερωτημάτων, χρειάστηκε να ανεβάσουμε τα CSV αρχεία, τα οποία βρίσκονται αποθηκευμένα τοπικά στο φάκελο movie_data, στο HDFS. Αυτό επιτυγχάνεται με την ακόλουθη εντολή:

```
$ hadoop fs -put movie_data hdfs://master:9000/
```

2.2 Μετατροπή CSV σε Parquet

Στη συνέχεια, ακολουθώντας την υπόδειξη της εκφώνησης, μετατρέψαμε τα CSV αρχεία σε Parquet, προκειμένου να βελτιστοποιήσουμε το I/O και, συνεπώς, να ελαχιστοποιήσουμε το χρόνο εκτέλεσης, εκμεταλλευόμενοι το μειωμένο αποτύπωμα στη μνήμη και τον δίσκο που παρέχει η συγκεκριμένη μορφή αποθήκευσης. Για τη μετατροπή, ξεκινήσαμε διαβάζοντας κάθε CSV σε dataframe και αποθηκεύοντάς το στη συνέχεια σε parquet μορφή πίσω στο hdfs. Αυτό επιτυγχάνεται με το script *toParquet.py* που συμπεριλαμβάνεται στο φάκελο της εργασίας.

2.3 Υλοποίηση Ερωτημάτων

Ερώτημα	**Λεκτική Περιγραφή**
Q1	Από το 2000 και μετά, να βρεθεί για κάθε χρονιά η ταινία με το μεγαλύτερο κέρδος, Αγνοούμε εγγραφές που δεν έχουν τιμή στην ημερομηνία κυκλοφορίας ή έχουν μηδενική τιμή στα έσοδα ή στον προϋπολογισμό.
Q2	Να βρεθεί το ποσοστό των χρηστών (%) που έχουν δώσει σε ταινίες μέση βαθμολογία μεγαλύτερη από 3.
Q3	Για κάθε είδος ταινίας, να βρεθεί η μέση βαθμολογία του είδους και το πλήθος των ταινιών που υπάγονται σε αυτό το είδος. Αν μία ταινία αντιστοιχεί σε περισσότερα του ενός είδη, θεωρούμε ότι μετριέται σε κάθε είδος.
Q4	Για τις ταινίες του είδους "Drama", να βρεθεί το μέσο μήκος περίληψης ταινίας ανά 5ετία από το 2000 και μετά (1η 5ετία: 2000-2004, 2η 2005-2009, 3η 2010 – 2014, 4η 2015 - 2019).
Q5	Για κάθε είδος ταινίας, να βρεθεί ο χρήστης με τις περισσότερες κριτικές, μαζί με την περισσότερο και την λιγότερο αγαπημένη του ταινία σύμφωνα με τις αξιολογήσεις του. Για την περίπτωση που ο χρήστης έχει την ίδια ψηλότερη / χαμηλότερη βαθμολογία σε περισσότερες από 1 ταινίες επιλέξτε την πιο δημοφιλή ταινία από αυτές της κατηγορίας που συμπίπτει η βέλτιστη / χείριστη βαθμολογία του χρήστη. Τα αποτελέσματα να είναι σε αλφαβητική σειρά ως προς την κατηγορία ταινίας και να παρουσιαστούν σε ένα πίνακα με τις ακόλουθες στήλες. • είδος • χρήστης με περισσότερες κριτικές • πλήθος κριτικών, • περισσότερο αγαπημένη ταινία • βαθμολογία περισσότερο αγαπημένης ταινίας • λιγότερο αγαπημένη ταινία • βαθμολογία λιγότερο αγαπημένης ταινίας

Στη συνέχεια, για κάθε ερώτημα του παραπάνω πίνακα υλοποιήσαμε μία λύση με το RDD API και δύο με Spark SQL: η πρώτη διαβάζει από τα CSV αρχεία, χρησιμοποιώντας το option inferSchema, ενώ η δεύτερη διαβάζει από τα αρχεία Parquet.

Q1:
Για την υλοποίηση του ερωτήματος απαιτείται ένα στάδιο Map-Reduce. Ειδικότερα, στη φάση Map κάνουμε split μια εγγραφή του πίνακα movies (χρησιμοποιώντας το complex_split που δίνεται, ώστε να μην υπάρξει πρόβλημα με κόμματα εντός των περιλήψεων) και, στη συνέχεια, αν τα πεδία date, cost και income δεν είναι κενά και αν η χρονιά κυκλοφορίας είναι από το 2000 και μετά, υπολογίζουμε το κέρδος από τα πεδία cost και income και κάνουμε emit ένα tuple με κλειδί τη χρονιά και τιμή τον τίτλο της ταινίας και το κέρδος. Επισημαίνουμε ότι στη φάση map αυτή έχουμε ενσωματώσει τα τρία πρώτα στάδια της RDD υλοποίησης (map-filter-map).
Έπειτα, στη φάση Reduce λαμβάνουμε ως είσοδο ένα tuple με κλειδί τη χρονιά και τιμή μια λίστα όλων των ταινιών που κυκλοφόρησαν εκείνη τη χρονιά, με τα αντίστοιχα κέρδη τους. Ανάμεσα σ' αυτές αναζητούμε την ταινία με τα μεγαλύτερα κέρδη, την οποία και κάνουμε emit στο τέλος. Μετά το πέρας του reduce σταδίου λαμβάνουμε για κάθε χρονιά την πιο προσοδοφόρα ταινία, με τα αντίστοιχα κέρδη.

Algorithm 1 Q1

```
getProfit (cost, inc):
    if cost then
        return (inc - cost)*100/cost
    else
        return 0
end

Map (k, v: record of movies):
    movie = v.complex_split(',')
    title = movie[1],   date = movie[3],   cost = movie[5],   income = movie[6]
    Filter movies with null Date, Cost or Income fields
    if date and cost and income then
        year = date[0:4]
        Filter movies from before 2000
        if year >= '2000' then
            emit(year, (title, getProfit(cost, income)))
end

Reduce (year, LIST[v]):
    maxProfit = 0,   mostProfitable = ""
    Find movie with max profit
    foreach movie in LIST[v] do
        profit = movie[1]
        if profit > max then
            maxProfit = profit
            mostProfitable = movie[0]
    end
    emit(year, (mostProfitable, maxProfit))
end
```

Add another Map-Reduce stage to sort by ascending year, to match RDD implementation

Q2:
Ο κορμός της υλοποίησης του ερωτήματος αποτελείται από ένα στάδιο Map-Reduce. Ειδικότερα, στη φάση Map αυτού κάνουμε split μια εγγραφή του πίνακα ratings και, στη συνέχεια, κάνουμε emit ένα tuple με κλειδί το αναγνωριστικό του χρήστη και τιμή ένα tuple που περιλαμβάνει μια βαθμολογία του χρήστη, αλλά και τον αριθμό 1, που θα χρησιμοποιηθούν για τον υπολογισμό του συνολικού αριθμού rating του χρήστη, αλλά και της μέσης βαθμολογίας που έχει δώσει σε ταινίες.
Ακολούθως, στη φάση Reduce λαμβάνουμε ως είσοδο ένα tuple με κλειδί το αναγνωριστικό του χρήστη και με όλα τα ratings που έχει δώσει σε ταινίες. Διατρέχοντας τη λίστα υπολογίζουμε το συνολικό αριθμό ratings του χρήστη (count), αλλά και το άθροισμα των βαθμολογιών του (totalRatings). Από τη διαίρεση totalRatings/count παράγεται το μέσο rating του χρήστη. Τελικά, κάνουμε emit το tuple (user, average).

Βασιζόμενοι στα δεδομένα αυτά, καλούμαστε να κάνουμε δύο μετρήσεις, προκειμένου να παράξουμε το ζητούμενο αποτέλεσμα. Χρειάζεται να υπολογίσουμε, αφενός, τον συνολικό αριθμό χρηστών (total) και, αφετέρου, τον αριθμό χρηστών που έχουν δώσει μέση βαθμολογία μεγαλύτερη του 3 (count), από τα οποία θα προκύψει το ζητούμενο ποσοστό. Καθένα από τα παραπάνω απαιτεί ένα ανεξάρτητο Map-Reduce στάδιο που δέχεται ως είσοδο τις παραχθείσες από τα προηγούμενα (user, average) εγγραφές. Στην πρώτη περίπτωση κάνουμε emit ένα tuple για κάθε εγγραφή, χρησιμοποιώντας το ίδιο κλειδί, ώστε όλες να συγκεντρωθούν στον ίδιο reducer, ο οποίος θα μετρήσει το πλήθος τους. Στη δεύτερη περίπτωση κάνουμε emit ένα αντίστοιχο tuple μόνο για τους χρήστες με μέση βαθμολογία μεγαλύτερη του 3. Μιας και η μέτρηση εγγραφών αποτελεί τετριμμένη διαδικασία, χρησιμοποιήθηκε χάριν συντομίας και στις δύο περιπτώσεις μια συνάρτηση len() για τον υπολογισμό του μήκους της λίστας. Φυσικά αυτό θα μπορούσε να γίνει όπως και στο προηγούμενο Map-Reduce στάδιο, συμπεριλαμβάνοντας το '1' στα tuples του Map και αθροίζοντάς τα στο Reduce.

Algorithm 2 Q2

Map *(k, v: record of ratings)*:
```
ratings = v.split(',')
user = ratings[0],    rating = (float(ratings[2]), 1)
emit(user, rating)
```
end

Reduce *(user, LIST[rating])*:
```
count = 0
totalRatings = 0
foreach rating in LIST[rating] do
    totalRatings += rating[0]
    count += rating[1]
end
average = totalRatings/count
emit(user, average)
```
end

The following 2 Map-Reduce stages use the above generated output, but do not depend on each other.

Count all users

Map *(user, average)*:
```
data = (user, average)
emit('Indlovu', data)
```
end

Reduce *(k, LIST[data])*:
```
total = len(LIST[data])
emit('total', total)
```
end

Count users with an average rating > 3

Map *(user, average)*:
```
if average > 3.0 then
    data = (user, average)
    emit('Wena', data)
```
end

Reduce *(k, LIST[data])*:
```
count = len(LIST[data])
emit('count', count)
```
end

Now we can calculate the percentage

percentage = count*100/total

Q3:
Όπως και στο προηγούμενο ερώτημα, το πρώτο στάδιο Map-Reduce πραγματοποιεί τον υπολογισμό της μέσης βαθμολογίας, αυτή τη φορά ανά ταινία. Προς το σκοπό αυτό, στο Map κάνουμε emit (movie, rating) tuples (στο rating συμπεριλαμβάνουμε πάλι τον αριθμό '1'), ενώ στο Reduce υπολογίζουμε το συνολικό αριθμό ratings για κάθε ταινία, αλλά και το άθροισμα αυτών προκειμένου να παράξουμε τελικά τη μέση βαθμολογία της.
Στη συνέχεια, χρειάζεται να κάνουμε join με τον πίνακα movie_genres, ώστε να υπολογίσουμε τη μέση βαθμολογία, αλλά και τον αριθμό ταινιών ανά είδος. Για το join μπορούμε να χρησιμοποιήσουμε έναν από τους αλγορίθμους που υλοποιήσαμε για το ζητούμενο 2 (broadcast/repartition). Στην έξοδό του λαμβάνουμε εγγραφές της μορφής (movie, (average, genre)).
Ακολουθώντας παρόμοια τακτική με πριν, κάνουμε emit (genre, average) tuples, όπου στο genre συμπεριλαμβάνεται ως συνήθως εκτός από το μέσο rating και ο αριθμός '1'. Στη φάση Reduce, ύστερα, έχοντας λάβει για κάθε genre μια λίστα με τα (avg) ratings των ταινιών του, υπολογίζουμε το πλήθος των ταινιών (numOfMovies) συναθροίζοντας τα '1' και το συνολικό άθροισμα των βαθμολογιών (totalRating). Από τη διαίρεση totalRating/numOfMovies προκύπτει η μέση βαθμολογία του είδους (avgGenreRating), ώστε να επιστρέψουμε, τελικά, εγγραφές της μορφής (genre, (avgGenreRating, numOfMovies))

Algorithm 3 Q3

Calculate AvgMovieRating

Map *(k, v: record of ratings)*:
```
ratings = v.split(',')
movie = ratings[1],    rating = (float(ratings[2]), 1)
emit(movie, rating)
```
end

Reduce *(movie, LIST[rating])*:
```
count = 0
totalRating = 0
foreach rating in LIST[rating] do
    totalRating += rating[0]
    count += rating[1]
end
average = totalRating/count emit(movie, average)
```
end

Join movie_genres using one of the already implemented join algorithms

Calculate AvgGenreRating and NumOfMovies per Genre

Map *(movie, v: (average, genre))*:
```
average = v[0],    genre = v[1]
rating = (average, 1)
emit(genre, rating)
```
end

Reduce *(genre, LIST[rating])*:
```
numOfMovies = 0
totalRating = 0
foreach rating in LIST[rating] do
    totalRating += rating[0]
    numOfMovies += rating[1]
end
avgGenreRating = totalRating/numOfMovies
emit(genre, (avgGenreRating, numOfMovies))
```
end

Q4:
Και σ' αυτό το ερώτημα χρειάστηκε να υπολογίσουμε έναν μέσο όρο, αυτή τη φορά του μήκους περίληψης για κάθε 5ετία. Ο υπολογισμός του μέσου όρου έγινε με τις ίδιες τεχνικές που ακολουθήθηκαν και στα προηγούμενα. Προτού μπορέσουμε, ωστόσο, να επεξεργαστούμε τα δεδομένα, χρειάστηκε μια προεπεξεργασία, για την οποία ορίσαμε 2 συναρτήσεις: την summaryLength για τον υπολογισμό του μήκους περίληψης, η οποία όπως σχολιάζουμε θα μπορούσε να αναπαρασταθεί με ένα Map-Reduce στάδιο, και η quinquennium για τον υπολογισμό της 5ετίας από τη χρονιά. Πριν τον υπολογισμό του μέσου μήκους ανά 5ετία, εξάγουμε τις τιμές που μας ενδιαφέρουν από τα movies και movie_genres, οι οποίες και συνενώνονται στη συνέχεια βάσει του αναγνωριστικού ταινίας (movie), με κάποιον από τους αλγορίθμους που υλοποιήσαμε στο 2ο μέρος της άσκησης (κατά προτίμηση repartition, ως reduce-side join).

Algorithm 4 Q4

summaryLength *(summary)*:
    Calculating summary length could be a separate Map-Reduce stage as in previous queries
    return len(summary.split())
end

quinquennium *(year)*:
    quint = 0
    if *year > '1999'* **then**
        quint = (int(year)-2000)//5 + 1
    return (str(quint) + "η 5ετία")
end

Split movies: Filter out null summaries and dates
Map *(k, v: record of movies)*:
    movies = v.split$_{c}omplex$(',')
    $movie = movies[0], \quad summary = movies[2], \quad date = movies[3]$
    if *summary and date* **then**
        emit(movie, (**summaryLength**(summary), **quinquennium**(year)))
end

Split movie_genres: Only emit 'Drama' movies
Map *(k, v: record of movie_genres)*:
    genres = v.split(',')
    movie = genres[0], genre = genres[1]
    if *genre == 'Drama'* **then**
        emit(movie, genre)
end

Join the above using a reduce side join like repartition (impl. in next section)

Calculate average summary length the usual way
Map *(movie, v: ((summaryLength, quinquennium), genre))*:
    summaryLength = (v[0][0], 1), quinquennium = v[0][1]
    emit(quinquennium, summaryLength)
end

Reduce *(quinquennium, LIST[summaryLength])*:
    count = 0
    totalLength = 0
    foreach *rating* ***in*** *LIST[summaryLength]* **do**
        totalLength += summaryLength[0]
        count += summaryLength[1]
    end
    averageSummaryLength = totalLength/count
    emit(quinquennium, averageSummaryLength)
end

Q5:
Το τελευταίο ερώτημα αποτελείται απο πολλά μικρότερα υποερωτήματα και, συνεπώς, περισσότερα Map-Reduce στάδια από τα προηγούμενα. Για το λόγο αυτό θα χωρίσουμε τον ψευδοκώδικα σε τρία μέρη, προκειμένου να διευκολύνουμε την εποπτεία.
Στο πρώτο μέρος που ακολουθεί πραγματοποιείται η προεπεξεργασία των δεδομένων, δηλαδή το διάβασμα κι η αρχική μορφοποίηση των εγγραφών από τα 3 αρχεία. Ειδικότερα, διαβάζουμε αρχικά από τα αρχεία movies και movie_genres, σπλιτάροντας τις αντίστοιχες εγγραφές, όπως και στα προηγούμενα ερωτήματα. Για τον πρώτο πίνακα κατασκευάζουμε εγγραφές της μορφής (movie_id, (title, popularity)), στις οποίες θα αναφερόμαστε στο εξής για συντομία ως "movies", ενώ για τον δεύτερο (movie_id, genre), στις οποίες θα αναφερόμαστε ως "movie_genres". Προετοιμάζουμε και μια δεύτερη εκδοχή των movie_genres, την οποία ονομάζουμε "movie_genres_complex" και θα μας χρειαστεί στη συνέχεια.
Επιπλέον, θα χρειαστούμε δύο εκδοχές του πίνακα ratings: η πρώτη αποτελείται από εγγραφές (movie_id, (user_id, rating)), όπως προκύπτουν από την ανάγνωση του πίνακα ratings και στις οποίες θα αναφερόμαστε στο εξής ως "ratings1". Η δεύτερη προκύπτει με τον ίδιο τρόπο, άλλα έχει ως κλειδί το αναγνωριστικό του χρήστη: (user_id, (movie_id, rating)). Στις εγγραφές αυτές θα αναφερόμαστε ως "ratings2".
Σε όλες τις προηγούμενες περιπτώσεις περιγράφονται λειτουργίες που επιτελούνται σε ένα μόνο στάδιο Map, χωρίς να ακολουθεί στάδιο Reduce. Αυτό δε μας πειράζει στην προκειμένη περίπτωση, αφού τα παραπάνω θα χρησιμοποιηθούν σε reduce-side joins μεταξύ τους ή με άλλους πίνακες.

Algorithm 5 Q5: Part 1 - Data Preprocessing

Split movies
Map *(k, v: record of movies)*:
    movies = v.split_complex(',')
    movie = movies[0], title = movies[1], popularity = movies[7]
    emit(movie, (title, popularity)) // *movies*
end

Split movie_genres
Map *(k, v: record of movie_genres)*:
    genres = v.split(',')
    movie = genres[0], genre = genres[1]
    emit(movie, genre) // *movie_genres*
end

Split movie_genres_complex
Map *(k, v: record of movie_genres)*:
    genres = v.split(',')
    emit(genres, ()) // *movie_genres_complex*
end

Split ratings 1
Map *(k, v: record of ratings)*:
    ratings = v.split(',')
    user = ratings[0], movie = ratings[1], rating = float(ratings[2])
    emit(movie, (user, rating)) // *ratings1*
end

Split ratings 2
Map *(k, v: record of ratings)*:
    ratings = v.split(',')
    user = ratings[0], movie = ratings[1], rating = float(ratings[2])
    emit(user, (movie, rating)) // *ratings2*
end

Στη συνέχεια, χρησιμοποιώντας τα παραπάνω, επιθυμούμε να υπολογίσουμε για κάθε είδος το χρήστη με τις περισσότερες κριτικές, καθώς και να βρούμε τη μεγαλύτερη και τη μικρότερη βαθμολογία του. Θα υπολογίσουμε το παραπάνω σε δύο διαδοχικά Map-Reduce στάδια.

Algorithm 6 Q5: Part 2 - User with the most reviews

Join ratings1 with movie_genres using a reduce side join like repartition

Calculate reviews, max and min rating per user per genre

Map *(movie, ((user, rating), genre))*:
  ratings = (1, rating, rating) emit((user, genre), ratings)
end
Reduce *((user, genre), LIST[ratings])*:
  reviews = 0, max_rating = 0, min_rating = 5.0
  foreach *ratings* ***in*** *LIST[ratings]* **do**
    reviews += ratings[0]
    if *max_rating < ratings[1]* **then**
      max_rating = ratings[1]
    if *min_rating > ratings[2]* **then**
      min_rating = ratings[2]
  end
  emit((user, genre), (reviews, max_rating, min_rating))
end

Calculate user with the most reviews per genre

Map *((user, genre), (reviews, max_rating, min_rating))*:
  ratings = (user, reviews, max_rating, min_rating)
  emit(genre, ratings)
end
Reduce *(genre, LIST[ratings])*:
  max_reviews = 0, user_info = ()
  foreach *ratings* ***in*** *LIST[ratings]* **do**
    if *max_reviews < ratings[1]* **then**
      max_reviews = ratings[1]
      user_info = ratings
  end
  In the RDD implementation we would just emit(genre, user_info), followed by an extra map step to expose the user as key. Here, however, we can do that directly, without the extra step.
  emit(user_info[0], (genre, max_reviews, max_rating, min_rating)) *// users*
end

Έπειτα, έχοντας βρει για κάθε είδος το χρήστη με τις περισσότερες κριτικές, αλλά και τις πληροφορίες που ζητούνται γι' αυτόν, προχωράμε στον υπολογισμό της περισσότερο και της λιγότερο αγαπημένης του ταινίας αυτού σε κάθε περίπτωση. Θα χρειαστεί να συνδυάσουμε την πληροφορία από όλους τους πίνακες που έχουμε φτιάξει, οπότε συνενώνουμε τους πίνακες ratings2, users, movies και movie_genres_complex. Επισημαίνουμε ότι εισάγουμε για πρώτη φορά τον πίνακα movies. Μας ενδιαφέρει να κρατήσουμε μόνο τις ταινίες από κάθε είδος που έχουν βαθμολογηθεί από το χρήστη είτε με τη μέγιστη είτε με την ελάχιστη βαθμολογία του. Μετά τη συνένωση και με τον πίνακα movie_genres_complex, θα χρησιμοποιήσουμε την πρώτη θέση του value για την εύρεση και αποθήκευση της αγαπημένης ταινίας του χρήστη και τη δεύτερη για τα αντίστοιχα στοιχεία της λιγότερο αγαπημένης του ταινίας. Σύμφωνα, δε, με την υπόδειξη της εκφώνησης, αν δύο ταινίες έχουν την ίδια βαθμολογία, κρατάμε εκείνη με τη μεγαλύτερη δημοφιλία.
Το τελευταίο Map-Reduce στάδιο δεν προσδίδει κάτι ουσιαστικό στην υλοποίηση, παρά μόνο προετοιμάζει τα δεδομένα εξόδου, ώστε να έχουν τη δομή που ζητείται, ενώ εκμεταλλεύεται τη διαδικασία ταξινόμησης που παρέχει το framework πριν από το στάδιο reduce, με σκοπό την τελική εμφάνιση των αποτελεσμάτων σε αλφαβητική σειρά ως προς το είδος.

Algorithm 7 Q5: Part 3 - Most and least favourite movie

Join ratings2 with users

Calculate reviews, max and min rating per user per genre

Map *(user, ((movie, rating), (genre, max_reviews, max_rating, min_rating)))*:
    Combined Filter and Map stage: emit only records with a rating of interest
    if *(rating == max_rating **or** rating == min_rating)* **then**
        user_tuple = (user, rating, genre, max_reviews)
        emit(movie, user_tuple)
end

Join the above with movies using a reduce-side join like repartition

Map *(movie, (user_tuple, (title, popularity)))*:
    user, rating, genre, max_reviews = user_tuple *// extract info from user_tuple*
    emit((movie, genre), (user, rating, max_reviews, title, popularity))
end

Join the above with movie_genres_complex using a reduce-side join like repartition

Map *((movie, genre), ((user, rating, max_reviews, title, popularity), ()))*:
    genre_user_info = (genre, user, max_reviews)
    movies = ((title, rating, popularity), (title, rating, popularity))
    emit(genre_user_info, movies)
end

Reduce *(genre_user_info, LIST[movies])*:
    most_favourite = (), least_favourite = ()
    max_rating = 0, min_rating = 5.0
    popularity_max = 0, popularity_min = 0
    foreach *movie **in** LIST[movies]* **do**
        max = movie[0][1], popularity1 = movie[0][2]
        min = movie[1][1], popularity2 = movie[1][2]
        if *(max_rating < max **or** (max_rating == max **and** popularity_max < popularity1))* **then**
            max_rating = max
            most_favourite = movie[0]
        if *(min_rating > min **or** (min_rating == min **and** popularity_max < popularity2))* **then**
            min_rating = min
            least_favourite = movie[1]
    end
    emit(genre_user_info, (most_favourite, least_favourite))
end

Prepare output

Map *(genre_user_info, (most_favourite, least_favourite))*:
    genre, user, max_reviews = genre_user_info *// extract info from genre_user_info*
    most_favourite = most_favourite[0], max_rating = most_favourite[1]
    least_favourite = least_favourite[0], min_rating = least_favourite[1]

    info = (user, max_reviews, most_favourite, max_rating, least_favourite, min_rating)
    emit(genre, info)
end

Reduce stage to sort by genre

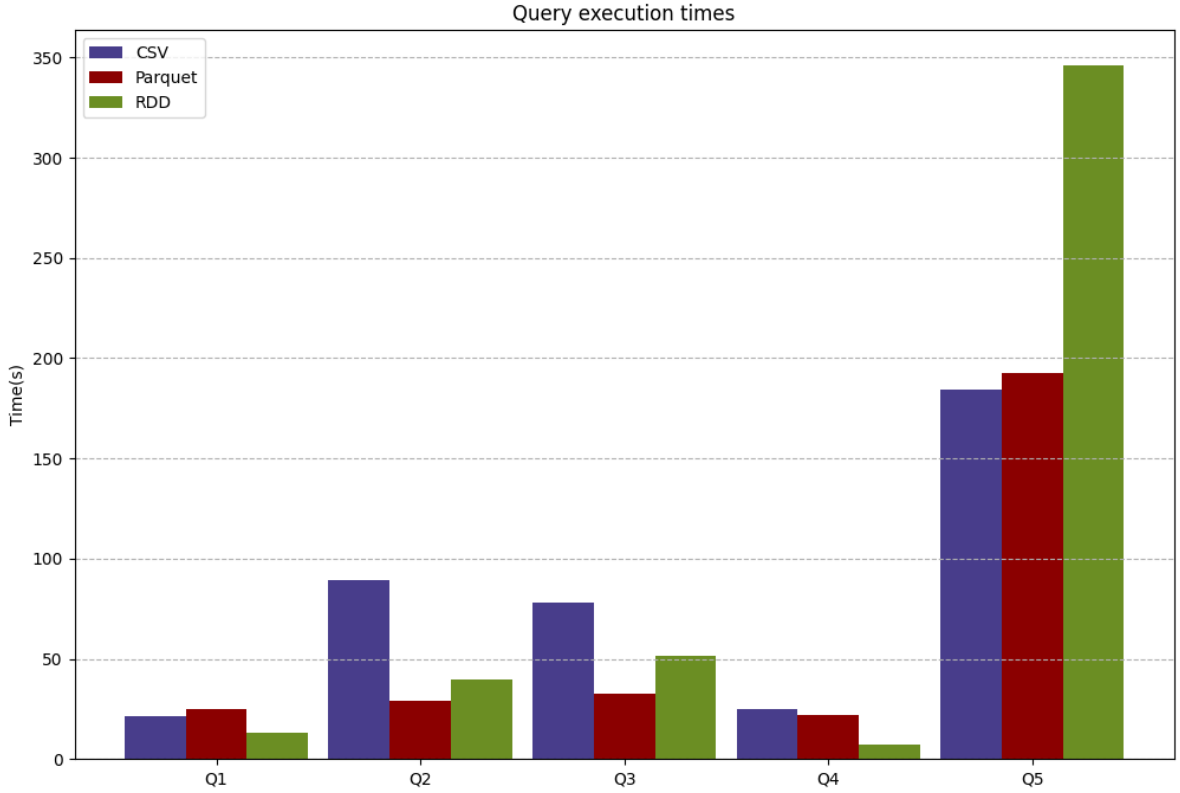
Reduce *(genre, LIST[info])*:
    Hopefully by now there's just one tuple per genre, but still, we need to go through a list
    foreach *tuple **in** LIST[info]* **do**
        emit(genre, info)
    end
end

2.4 Αξιολόγηση Υλοποιήσεων

Τέλος, κληθήκαμε να εκτελέσουμε τις προηγούμενες υλοποιήσεις μας και να μετρήσουμε την επίδοσή τους. Ειδικότερα, πραγματοποιήσαμε για κάθε ερώτημα 3 μετρήσεις, μία για κάθε υλοποίηση:

1. Spark SQL με είσοδο το CSV αρχείο (infer schema)
2. Spark SQL με είσοδο το Parquet αρχείο
3. Map Reduce Queries – RDD API

Τα αποτελέσματα συγκεντρώθηκαν και συμπεριλαμβάνονται στο φάκελο της εργασίας εντός του *output/*. Για κάθε εκτέλεση, αντλήσαμε από το αντίστοιχο αρχείο εξόδου τη μέτρηση του χρόνου- η οποία πραγματοποιήθηκε με 1η λήψη του χρόνου μετά την κατασκευή του Spark Context και 2η στο τέλος του κάθε προγράμματος- και την τοποθετήσαμε στο ακόλουθο συγκεντρωτικό ραβδόγραμμα (*plot_ queries.py*).



Σχήμα 1: Χρόνος εκτέλεσης των Queries για τις 3 υλοποιήσεις

Από τα παραπάνω καθίσταται, αρχικά, σαφής η διαφορά επίδοσης μεταξύ των δύο υλοποιήσεων της SparkSQL. Ειδικότερα, ενώ στα Q1, Q4 οι δύο υλοποιήσεις εμφανίζουν παραπλήσια επίδοση, το parquet φαίνεται να υπερτερεί στις περιπτώσεις που οι πίνακες εισόδου είναι πολύ μεγάλοι, όπως συμβαίνει στα Q2 και Q3, όπου χρησιμοποποιείται ολόκληρος ο πίνακας ratings ($\backsim$ 26.000.000 εγγραφές, σχεδόν 300 φορές μεγαλύτερος από τον movie_genres), επιβεβαιώνοντας τα όσα αναφέραμε στην αρχή περί των λόγων της επιλογής του. Αποδεικνύεται στις περιπτώσεις αυτές ότι πράγματι η ανάγνωση από parquet αρχεία είναι μακράν ταχύτερη αυτής από csv.
Εκτός από το διαφορετικό τρόπο αποθήκευσης των δύο τύπων αρχείων, σημαντικό ρόλο στην επίδοση διαδραματίζει και η διαδικασία εξαγωγής (συμπερασμού) του σχήματος των δεδομένων (schema inference). Στην περίπτωση των csv αρχείων, χρειάζεται να ενεργοποιήσουμε την επιλογή inferSchema, προκειμένου να αναγνωριστεί το σχήμα, ο τύπος, δηλαδή, της κάθε στήλης των δεδομένων. Για να γίνει αυτό, ωστόσο, το σύστημα χρειάζεται να διατρέξει τα δεδομένα και μια δεύτερη φορά, προκαλώντας σημαντική αύξηση του χρόνου εκτέλεσης, ειδικά για πολύ μεγάλα αρχεία όπως το ratings.csv. Από την άλλη, για την ανάγνωση από αρχεία parquet δεν απαιτείται η ενεργοποίηση κάποιας παρόμοιας επιλογής, αφού η διαδικασία εξαγωγής του σχήματος πραγματοποιείται αυτόματα by default κατά τη δημιουργία του αρχείου και συμπεριλαμβάνεται στα μεταδεδομένα του.

Από την άλλη, το RDD μοιάζει να υπερισχύει και των δύο στις απλές περιπτώσεις όπου διαχειριζόμαστε μικρότερους πίνακες (Q1, Q4) παρουσιάζοντας διπλάσια ταχύτητα εκτέλεσης από τη χειρότερη υλοποίηση. Όσο μεγαλώνουν οι πίνακες εισόδου, αλλά και ο όγκος των δεδομένων, ωστόσο, παρατηρείται μια σταδιακή μείωση της επίδοσής του. Αυτό φαίνεται έντονα με τη μετάβαση από το Q1- που εκτελείται πάνω στον μικρότερο από όλους τους πίνακες (movies)- στο Q2- που εκτελείται πάνω στον μεγαλύτερο πίνακα (ratings)- και, τελικά, στο Q3, όπου διαχειριζόμαστε τη συνένωση των δύο μεγαλύτερων πινάκων, movie_genres και ratings. Αξίζει να σημειώσουμε ότι το parquet παρουσιάζει πολύ πιο ήπια μεταβολή για την ίδια μετάβαση.

Μπορούμε, συνεπώς, να αντιληφθούμε ότι, αν και το RDD αποτελεί αξιόπιστη και αποδοτική λύση- παρέχοντας ταυτόχρονα ευκολία υλοποίησης- για τις απλές περιπτώσεις, όπου το σύνολο δεδομένων είναι σχετικά μικρό, είναι υποδεέστερο της SparkSQL υλοποίησης με ανάγνωση από parquet αρχεία, η οποία εμφανίζει μεγάλη συνέπεια στην επίδοση και διαχειρίζεται αποδοτικότερα μεγάλες εισόδους και σύνθετα προβλήματα με διαδοχικές συνενώσεις.
Τα παραπάνω αποδεικνύονται περίτρανα στο ερώτημα Q5, το οποίο αποτελεί έτσι κι αλλιώς μια ιδιάζουσα περίπτωση. Ειδικότερα, τόσο στην RDD, όσο και στη SparkSQL υλοποίηση, απαιτούνται πολλαπλές συνενώσεις μεταξύ όλων των πινάκων, γεγονός που αυξάνει κατά πολύ των όγκο των δεδομένων, παρόλο που προσπαθήσαμε κάθε φορά να πραγματοποιούμε συνενώσεις σε υποσύνολα των πινάκων. Πάντως, σε αυτό το ερώτημα αναδεικνύεται χαρακτηριστικά η αδυναμία του RDD να διαχειριστεί τον αυξημένο όγκο δεδομένων, σε τέτοιο βαθμό ώστε ακόμα και SparkSQL υλοποίηση με ανάγνωση από csv να είναι κατά πολύ ταχύτερη.

Οφείλουμε να σχολιάσουμε, καταληκτικά, ότι τα παραπάνω συμπεράσματα δεν μπορούν να θεωρηθούν απόλυτα, καθώς πολλοί παράγοντες επηρεάζουν την επίδοση μιας εκτέλεσης. Ταυτόχρονα, δεν ισχυριζόμαστε, ότι όλες οι υλοποιήσεις είναι βέλτιστες, αφού βασικό μας μέλημα κατά το σχεδιασμό υπήρξε η κατά το δυνατόν εφαρμογή κοινής στρατηγικής μεταξύ των δύο υλοποιήσεων (RDD - SparkSQL), με σκοπό αφενός να διευκολύνεται η συγκριτική ανάλυση των επιδόσεων και, αφετέρου, να λαμβάνουμε τα ίδια αποτελέσματα. Αξίζει να σημειωθεί ότι μερικές από τις υλοποιήσεις-προσεγγίσεις που δοκιμάσαμε έδιναν ελαφρώς διαφορετική έξοδο. Αυτό οφείλεται στη διαφορετική σειρά/ προτεραιότητα των εφαρμοζόμενων πράξεων, αφού τα δεδομένα περιέχουν διπλότυπα, τα οποία ευθύνονται και για "περίεργα" αποτελέσματα, όπως η διπλή εγγραφή για το είδος "History" στο Q5.

3 Υλοποίηση και μελέτη συνένωσης σε ερωτήματα - Μελέτη του βελτιστοποιητή του Spark

Στο δεύτερο μέρος της εργασίας κληθήκαμε να μελετήσουμε και να αξιολογήσουμε τις διαφορετικές υλοποιήσεις που υπάρχουν στο περιβάλλον Map-Reduce του Spark για τη συνένωση (join) δεδομένων και συγκεκριμένα το repartition join και το broadcast join, όπως έχουν περιγραφεί στην δημοσίευση "A Comparison of Join Algorithms for Log Processing in MapReduce", Blanas et al , in Sigmod 2010".

3.1 Broadcast Join

Για την υλοποίηση του broadcast join βασιστήκαμε στον σχετικό ψευδοκώδικα της παραπάνω δημοσίευσης με μικρές τροποποιήσεις. Ειδικότερα, δεν υλοποιήσαμε τη συνάρτηση Close(), αφού θεωρούμε ότι, αφενός, η διαχείριση των partitions πραγματοποιείται από το framework του Spark και, αφετέρου, γνωρίζουμε τα μεγέθη των R και L πινάκων εκ των προτέρων, οπότε φροντίζουμε να καλέσουμε τη συνάρτηση έτσι ώστε R « L.
Αναφορικά με την υλοποίηση, στη συνάρτηση Init() γίνεται η προετοιμασία για το map-side join. Πιο συγκεκριμένα, για κάθε κλειδί του πίνακα R, δημιουργούμε μια λίστα από τις εγγραφές στις οποίες εμφανίζεται και, ύστερα, δημιουργούμε ένα Hash Map (hmap) για τα tuples (κλειδί-R, λίστα εγγραφών), το οποίο και κάνουμε broadcast. Στη συνέχεια, στο κύριο μέρος της συνάρτησης, λαμβάνεται ως είσοδος μια εγγραφή του πίνακα L. Με βάση το κλειδί της, αναζητούμε στο hmap- το οποίο βρίσκεται αποθηκευμένο τοπικά μετά το broadcast- και επιστρέφουμε μια λίστα από tuples της μορφής (κλειδί-L, (εγγραφή-R, εγγραφή-L)) για κάθε εγγραφή-R αποθηκευμένη στη θέση κλειδί-L του hmap.

Algorithm 8 Broadcast Join

```
Init ():
    foreach rec of R do
        Extract join key
        Map (k: null, v : a record from R):
            emit(v[0], v)
        end
        Reduce stage to group by key
        Reduce (join_key, LIST[v]):
            emit(join_key, LIST[v])
        end
    end
    Create HashMap hmap with join keys as keys, LIST[v]'s as values.
    Broadcast hmap
end

Map (k, v: a record from an L split):
    foreach r in hmap[k] do
        emit(k, (r[0], v))
    end
end
```

3.2 Repartition Join

Έπειτα, υλοποιήσαμε το repartition join στο RDD API, ακολουθώντας πιστά τον ψευδοκώδικα της δημοσίευσης. Όπως φαίνεται και παρακάτω, στο στάδιο map προσθέτουμε απλώς ένα tag στο value της εγγραφής, ανάλογα με το αν προέρχεται από τον πίνακα L ή R. Η συνένωση των εγγραφών πραγματοποιείται στο στάδιο reduce (reduce-side join). Ειδικότερα, κάθε εγγραφή στη LIST[tagged_record] που φτάνει στο στάδιο reduce, τοποθετείται σε έναν buffer, ανάλογα με το tag της. Στο τέλος, κάνουμε emit όλους τους συνδυασμούς εγγραφών από τους δύο buffers (καρτεσιανό γινόμενο), με κλειδί το join_key.

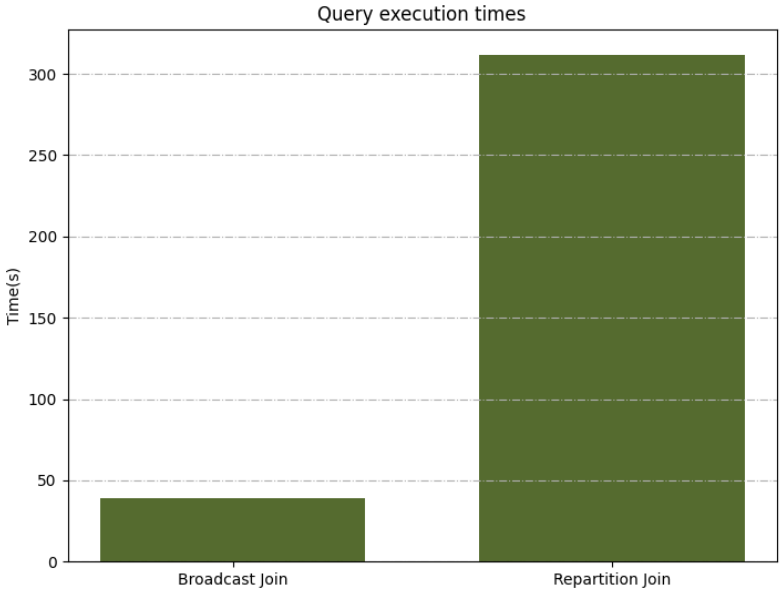
Algorithm 9 Repartition Join

```
Map (k, v : a record from a split of either R or L):
    join_key = k
    tag = a tag of either 'R' or 'L'
    tagged_record = (tag, v)
    emit(join_key, tagged_record)
end

Reduce (join_key, LIST[tagged_record]):
    lbuf, rbuf = [], [] Append to buffers accordingly
    foreach (tag, v) in LIST[tagged_record] do
        if tag == 'L' then
            lbuf.append(v)
        else
            rbuf.append(v)
    end
    Emit the cross product of the buffers
    foreach r in rbuf do
        foreach l in lbuf do
            emit(k, (l, r))
        end
    end
end
```

3.3 Σύγκριση Υλοποιήσεων

Έχοντας υλοποιήσει τους δύο αλγορίθμους συνένωσης, καλούμαστε να απομονώσουμε 100 γραμμές του πίνακα movie genres σε ένα άλλο CSV (genres100.csv) και να συγκρίνουμε τους χρόνους εκτέλεσης των δύο υλοποιήσεων, για τη συνένωση των 100 γραμμών με τον πίνακα ratings. Τοποθετήσαμε τους χρόνους εκτέλεσης για το broadcast και το repartition join σε ένα ραβδόγραμμα, για καλύτερη εποπτεία.



Σχήμα 2: Χρόνος εκτέλεσης των Broadcast-Repartition Join

Όπως καθίσταται σαφές από το παραπάνω γράφημα, το broadcast join είναι μακράν ταχύτερο του repartition στη συγκεκριμένη εφαρμογή. Αυτό φυσικά ήταν απολύτως αναμενόμενο, αφού το broadcast join είναι πολύ πιο αποδοτικό στην περίπτωση συνένωσης ενός μικρού πίνακα R με έναν πολύ μεγαλύτερο πίνακα L. Αυτό συμβαίνει, διότι, αντί να μεταφέρουμε και τους δύο πίνακες σε ολόκληρο το δίκτυο- όπως συμβαίνει στην περίπτωση του repartition join- κάνουμε broadcast τον μικρό πίνακα R, ώστε αυτός να βρίσκεται τοπικά αποθηκευμένος σε όλα τα μηχανήματα, αποφεύγοντας το σορτάρισμα και των δύο πινάκων, καθώς και τη μεταφορά του μεγάλου L στο δίκτυο, η οποία θα προκαλούσε σημαντική αύξηση του χρόνου εκτέλεσης. Οφείλουμε να σημειώσουμε πάντως, πως και στις δύο περιπτώσεις τα αποτελέσματα της συνένωσης είναι τα ίδια (απλά επιστρέφονται με διαφορετική σειρά).

3.4 Query Optimizer

Στο τελευταίο μέρος της εργασίας, κληθήκαμε να πειραματιστούμε με τον βελτιστοποιητή ερωτημάτων που παρέχεται από το DataFrame API του SparkSQL. Όπως πληροφορούμαστε από την εκφώνηση, ο συγκεκριμένος βελτιστοποιητής έχει τη δυνατότητα αυτόματης επιλογής της υλοποίησης που θα χρησιμοποιήσει για ένα ερώτημα join, λαμβάνοντας υπόψη το μέγεθος των δεδομένων, αλλά και αλλαγής της σειράς ορισμένων τελεστών, προσπαθώντας να μειώσει τον συνολικό χρόνο εκτέλεσης του ερωτήματος. Σύμφωνα με τα παραπάνω, αν ο ένας πίνακας είναι αρκετά μικρός (με βάση ένα όριο που ρυθμίζει ο χρήστης) θα χρησιμοποιήσει broadcast join, διαφορετικά θα χρησιμοποιήσει repartition join.
Προκειμένου να απενεργοποιήσουμε την προαναφερθείσα δυνατότητα επιλογής, μπορούμε να θέσουμε τη μεταβλητή "spark.sql.autoBroadcastJoinThreshold" σε -1. Βασιζόμενοι στο script της εκφώνησης πραγματοποιήσαμε πειράματα με τη βελτιστοποίηση του join ενεργοποιημένη και απενεργοποιημένη. Τα αποτελέσματα φαίνονται παρακάτω.

Σχήμα 3: Χρόνος εκτέλεσης χωρίς και με βελτιστοποίηση του join

Όπως γίνεται αντιληπτό, η εκτέλεση του ερωτήματος με βελτιστοποιητή είναι σαφώς ταχύτερη από αυτή χωρίς. Αυτό παρατηρείται, διότι, στην πρώτη περίπτωση ο βελτιστοποιητής εντοπίζει ότι ο πίνακας genres είναι πολύ μικρός (επιστρέφονται μόνο 100 γραμμές) και αποφασίζει να χρησιμοποιήσει broadcast join, το οποίο- όπως διαπιστώσαμε και στο προηγούμενο ερώτημα- αποδίδει πολύ καλύτερα για τα συγκεκριμένα δεδομένα εισόδου (R « L). Από την άλλη, φαίνεται ότι στην unoptimized εκτέλεση έχει χρησιμοποιηθεί το default join (sort-merge join), το οποίο οδηγεί σε τετραπλάσιο χρόνο εκτέλεσης.
Παρακάτω φαίνονται και τα πλάνα εκτέλεσης που παράγει ο βελτιστοποιητής για τις δύο περιπτώσεις.

Optimized

```
$== Physical Plan ==
*(3) BroadcastHashJoin [_c0#8], [_c1#1], Inner, BuildLeft
:- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, false]))
:  +- *(2) Filter isnotnull(_c0#8)
:     +- *(2) GlobalLimit 100
:        +- Exchange SinglePartition
:           +- *(1) LocalLimit 100
:              +- *(1) FileScan parquet [_c0#8,_c1#9] Batched: true, Format: Parquet,
Location: InMemoryFileIndex[hdfs://master:9000/movie_data/movie_genres.parquet],
PartitionFilters: [], PushedFilters: [], ReadSchema: struct<_c0:string,_c1:string>
+- *(3) Project [_c0#0, _c1#1, _c2#2, _c3#3]
   +- *(3) Filter isnotnull(_c1#1)
      +- *(3) FileScan parquet [_c0#0,_c1#1,_c2#2,_c3#3] Batched: true, Format: Parquet,
      Location: InMemoryFileIndex[hdfs://master:9000/movie_data/ratings.parquet],
      PartitionFilters: [], PushedFilters: [IsNotNull(_c1)],
      ReadSchema: struct<_c0:string,_c1:string,_c2:string,_c3:string>$
```

Unoptimized

```
$== Physical Plan ==
*(6) SortMergeJoin [_c0#8], [_c1#1], Inner
:- *(3) Sort [_c0#8 ASC NULLS FIRST], false, 0
:  +- Exchange hashpartitioning(_c0#8, 200)
:     +- *(2) Filter isnotnull(_c0#8)
:        +- *(2) GlobalLimit 100
:           +- Exchange SinglePartition
:              +- *(1) LocalLimit 100
:                 +- *(1) FileScan parquet [_c0#8,_c1#9] Batched: true, Format: Parquet,
Location: InMemoryFileIndex[hdfs://master:9000/movie_data/movie_genres.parquet],
PartitionFilters: [], PushedFilters: [], ReadSchema: struct<_c0:string,_c1:string>
+- *(5) Sort [_c1#1 ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(_c1#1, 200)
      +- *(4) Project [_c0#0, _c1#1, _c2#2, _c3#3]
         +- *(4) Filter isnotnull(_c1#1)
            +- *(4) FileScan parquet [_c0#0,_c1#1,_c2#2,_c3#3] Batched: true,
            Format: Parquet,
            Location: InMemoryFileIndex[hdfs://master:9000/movie_data/ratings.parquet],
            PartitionFilters: [], PushedFilters: [IsNotNull(_c1)],
            ReadSchema: struct<_c0:string,_c1:string,_c2:string,_c3:string>$
```