Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Προχωρημένα Θέματα Βάσεων Δεδομένων

Αναφορά για την εξαμηνιαία Εργασία

Καλογιάννης Φοίβος
03114831

Σκουρτσίδης Γιώργος
03114307

Μέρος 1ο

Ζητούμενο 1.1

Σε αυτό το ερώτημα φορτώσαμε τα αρχεία csv στο hdfs χρησιμοποιώντας την εντολή:
`hadoop fs -put <file_name> hdfs://master:9000/<rest_path_to_upload>\\`

Ζητούμενο 1.2

Για να μετρατρέψουμε τα csv αρχεία σε parquet κωδικοποίηση χρησιμοποιήθηκε το αρχείο `|create_parquet.py|` Στη συνέχεια τα αρχεία που δημιουργήθηκαν φορτώθηκαν και στο σύστημα HDFS.

Ζητούμενο 1.3

Για κάθε ερώτημα που μας ζητήθηκε,υλοποιήθηκε μία λύση με το RDD API και μία με Spark SQL. Σε κάθε υλοποίηση που πραγματοποιήθηκε με το Spark SQL API δίνεται μία παράμετρος P ή C, όπου "P" σημαίνει πως το Query θα χρησιμοποιήσει τα αρχεία parquet, ενώ "C" σημαίνει πως θα γίνει χρήση των CSV αρχείων. Παρακάτω παρατίθενται τα Queries σε ψευδοκώδικα με το προγραμματιστικό μοντέλο Map-Reduce.

Query 1

---

map(key,value):
    // key = None
    // value = csv line
    line = value.split(",")
    title = line[1]
    year = line[3][0:4]
    cost = line[5]
    turnover = line[6]
    $profit = 100 * ((turnover - cost)/cost)$
    filter: $year >= 2000$
emit(year, (title,profit))

reduce(key,value):
    // key = year
    // value = (a list of titles,a list of profits)
    max = -∞
    max_title
    for all $prof \in profits$ **do**
      if prof >max **then**
            $max \leftarrow prof$
            max_title = title
      end if
    end for
    emit(key,(max_title, max))

map(year,(list of titles, list of profits):
    sort_by_year(ascending)

Query 2

```
map(key,value):
    // key = None
    // value = csv line
    line = split(",")
    user_id = line[0]
    rating = float(line[2])
    emit(user_id, (rating,1))

reduce(key,value):
    // key = user_id
    // value = (list_of_ratings,list_of_ones)
    count = 0
    total_rating = 0
    for all i ∈ list_of_ones do
      count ← count + i
    end for
    for all i ∈ list_of_ratings do
      total_ratings ← total_ratings + i
    end for
    emit(user_id,(total_ratings,count_ratings))

map(user_id,(total_ratings,count_ratings)):
    emit(user_id, total_ratings / count_ratings)

filter(user_id,mean_rating)):
    mean_rating > 3
    count ←count_num_of_users(user_id)
    emit(count)
```

Στο μέρος 2 παραθέτουμε τον ψευδοκώδικα για τις λειτουργίες "join" χρησιμοποιώντας το Map-Reduce προγραμματιστικό μοντέλο.Συνεπώς, στα ακόλουθα ερωτήματα θα τα χρησιμοποιούμε χωρίς να γράφουμε κάθε φορά εκ νέου την υλοποίηση.

Query 3

```
map1(key,value):
    // key = None
    // value = csv line
    line = split(",")
    movie_id = line[0]
    rating = float(line[2])
    emit(movie_id, (rating,1))

reduce1(key,value):
    // key = user_id
    // value = (list_of_ratings,list_of_ones)
    count = 0
    total_rating = 0
    for all i ∈ list_of_ones do
      count ← count + i
    end for
    for all i ∈ list_of_ratings do
      count ← total_ratings + i
    end for
    emit(movie_id,(total_ratings,count_ratings))

map2(movie_id,(total_ratings,count_ratings)):
    emit(movie_id, total_ratings / count_ratings)

map3(key,value):
    // key = None
    // value = csv line
    line = split(",")
    movie_id = line[0]
    genre = line[1]
    emit(movie_id, genre)

// join results of map2,map3
map4(key,value):
    // key = movie_id
    // value = (genre, mean_rating)
    emit(list_of_genres[i], (list_of_mean_ratings[i], 1)

reduce2(key,value):
    // key = movie_id
    // value = (list_of_mean_ratings, list_of_mean_ratings)
    count1 = 0
    count2 = 0
    for all i ∈ value[0] do
      count1 ← count1 + i
    end for
    for all i ∈ value[1] do
      count2 ← count2 + i
    end for
    emit(key,(count1,count2))

map4(key,value):
    // key = movie_id
    // value = (count1,count2)
    emit(movie_id, (value[0]/value[1], value[1]))
```

Query 4

```
map1(key,value):
    // key = None
    // value = csv line
    line = split(",")
    movie_id = line[0]
    genre = line[1]
    emit(movie_id, genre)

map2(key,value):
    // key = None
    // value = csv line
    line = split(",")
    movie_id = line[0]
    year = line[3][0:4])
    if year >= 2000 and year <= 2004 then
      year = "2000-2004"
    else if year >= 2005 and year <= 2009 then
      year = "2005-2009"
    else if year >= 2010 and year <= 2014 then
      year = "2010-2014"
    else if year >= 2015 and year <= 2019 then
      year = "2015-2019"
    else
      year = "0"
    end if
    movie_id = line[0]
    words = line[2].split(' ')
    cnt=0
    for _ in words: do
      cnt+=1
    end for
    emit(movie_id, (year, cnt, 1))

// join results of map1,map2
// Filter: keep rows in which: Genre = "Drama" and
//         keep rows in which: year != "0"
map3(key,value):
    // key = movie_id
    // value = (year, count, 1)
    emit(value[0], (value[1], value[2])

reduce1(key,value):
    // key = movie_id
    // value = (list_of_counts, list_of_ones)
    count1 = 0
    count2 = 0
    for all i ∈ value[0] do
      count1 ← count1 + i
    end for
    for all i ∈ value[1] do
      count2 ← count2 + i
    end for
    emit(key,(count1,count2))

map4(key,value):
    // key = movie_id
    // value = (count1,count2)
    emit(movie_id, (value[0]/value[1]))
// Sort Results By Key, Ascending
```

Query 5 Part A

```
// Genres
map1(key,value):
    // key = None
    // value = csv line
    line = split(",")
    movie_id = line[0]
    genre = line[1]
    emit(movie_id, genre)

// Ratings
map2(key,value):
    // key = None
    // value = csv line
    line = split(",")
    movie_id, user_id = line[1], line[2]
    emit(movie_id, (user_id, 1))

// join results of map1,map2
map3(key,value):
    // key = movie_id
    // value = (genre, (user_id, 1)
    emit((value[1][0], value[1][1][0]), value[1][1][1])

reduce1(key,value):
    // key = (genre, user_id)
    // value = (list_of_ones)
    count1 = 0
    for all i ∈ list_of_ones do
      count1 ← count1 + 1
    end for
    emit(genre,(count,user_id))

reduce2(genre,(list_of_count,list_of_user_id)):
    max_count = -Infinity
    for all i ∈ list_of_count do
      if max_count < i then
            max ← i
      end if
    end for
    emit((user_id,genre),max_count)

// Ratings2
map4(key,value):
    // key = None
    // value = CVS line, movie_id, (rating, user_id)
    emit(movie_id, (rating, user_id))

// Movies
map5(key,value):
    // key = None
    // value = CVS line, (movie_id, (title, popularity))
    emit(movie_id, (title, popularity))
```

Query 5 - Part B

```
// Join(movies,genres,ratings2) -> result2
map1(movie_id, (((title,popularity),genre), (rating, user_id))):
    emit(((user_id,genre), ((rating,popularity),(movie_id,title))))


reduce3(((user_id,genre),lists_of_ ((rating,popularity),(movie_id,title))):
    max_rat = -∞
    min_rat = ∞
    for all i ∈ list_of_rating do
      if max_rat < i then
            max_rat ← i
            title ← title_i
      else if max_rat == i then
            if max_pop < pop_i then
                  max_rat ← i
                  best_title ← title_i
            end if
      end if
      if min_rat < i then
            min_rat ← i
            title ← title_i
      else if min_rat == i then
            if min_pop < pop_i then
                  min_rat ← i
                  worst_title ← title_i
            end if
      end if
    end for
    emit((Genre,User Id) , (Favorite Movie ,Best Rating , Least Favorite ,Worst Rating)))

// Join(reduce2, reduce3)
map((Genre,User Id) , (max_count,(Favorite Movie ,Best Rating ,
      Least Favorite ,Worst Rating))):
    emit((Genre,User Id) , (Favorite Movie ,Best Rating , Least Favorite ,Worst Rating))


// Sort Results By Key, Ascending
```

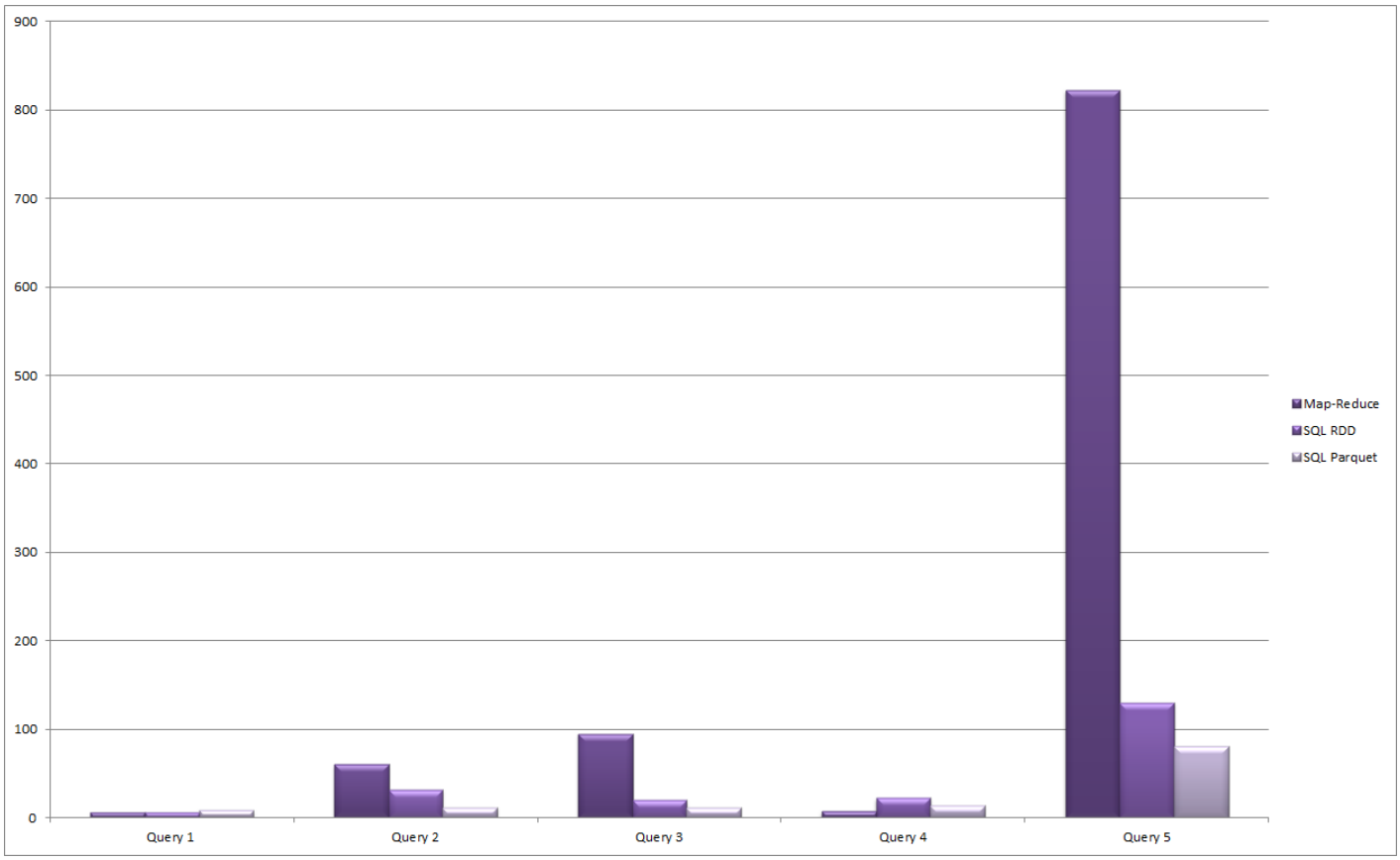
Ζητούμενο 1.4

Παρατηρούμε ότι η πραγματοποίηση queries με τη χρήση των αρχείων parquet είναι αισθητά γρηγορότερη κατά μέσο όρο. Αξιοσημείωτο βέβαια είναι ότι για queries όχι ιδιαίτερα απαιτητικά,που δεν περιέχουν joins και μεγάλα datasets, το overhead που πληρώνουμε για την χρήση του parquet υπερσκελίζει το χρόνο εκτέλεσης με αποτέλεσμα σε απλά ερωτήματα η χρήση του csv να αποδεικνύεται χρονικά αποδοτικότερη. Για τον παραπάνω λόγο, βλέπουμε πως στα Q1 και Q4 το RDD API μας παρέχει τη γρηγορότερη απόδοση.Αντίστοιχα, στα queries που περιέχουν joins βλέπουμε πως την καλύτερη απόδοση μας δίνει το SQL API και συγκεκριμένα υπερτερεί ηκωδικοποίηση parquet.

Οι χρόνοι εκτέλεσης είναι:

- Time of Q1 using Map-Reduce with csv is: 6.2992103099823 seconds
- Time of Q2 using Map-Reduce with csv is: 59.95288586616516 seconds
- Time of Q3 using Map-Reduce with csv is: 94.70336699485779 seconds
- Time of Q4 using Map-Reduce with csv is: 7.6572792530059814 seconds
- Time of Q5 using Map-Reduce with csv is: 821.560733795166 seconds
- Time elapsed for q1sql.py using csv is 6.1324 sec.
- Time elapsed for q1sql.py using parquet is 8.4735 sec.
- Time elapsed for q2sql.py using csv is 31.7143 sec.
- Time elapsed for q2sql.py using parquet is 11.6561 sec.
- Time elapsed for q3sql.py using csv is 19.8163 sec.
- Time elapsed for q3sql.py using parquet is 11.6330 sec.
- Time elapsed for q4sql.py using csv is 22.4371 sec.
- Time elapsed for q4sql.py using parquet is 13.8103 sec.
- Time elapsed for q5sql.py using csv is 129.8353 sec.
- Time elapsed for q5sql.py using parquet is 81.0783 sec.

Τα αποτελέσματα παρουσιάζονται συγκεντρωμένα στο bar plot που ακολουθεί παρακάτω.



Μέρος 2ο

Ζητούμενο 2.1

Υλοποιήσαμε ένα Broadcast join στον πίνακα ratings, καθώς και σε ένα υποσύνολο του *movie_genres*,αποτελούμενο απο 100 σειρές. Αρχικά διαλέγουμε 2 πεδία για τον κάθε πίνακα. Για τον πίνακα ratings επιλέγεται ως κλειδι το movies_id και ως value η βαθμολογία μίας ταινίας(rating),ενώ για τον πίνακα *movie_genres* επιλέγονται το *movie_id* και το είδος(genre) ταινίας αντίστοιχα.
Στη συνέχεια, ο μικρότερος πίνακας movie_genres γίνεται groupByKey προκειμένου να συγκεντρωθούν όλα τα όμοια κλειδιά μαζί. Έτσι, ο πίνακας παίρνει τη μορφή (key,(value[0],value[1]). Μετατρέπουμε επίσης τον πίνακα σε Dictionary,χρησιμοποιώντας την εντολή collectAsMap και μετά κάνουμε broadcast το Dictionary.
Με τη μέθοδο mapPartitions εφαρμόζουμε μια συνάρτηση σε ένα partition του μεγαλύτερου πίνακα ratings.Για κάθε σειρά του partition ελέγχουμε εάν υπάρχει το κλειδί της σειράς(tuple) μέσα στο Broadcasted Dictionary movie_genres. Εάν δεν υπάρχει,τότε τίποτα δε συμβαίνει και η επανάληψη συνεχίζεται.Εάν όμως υπάρχει, τότε φτιάχνω μία νέα τουπλα *(movie_id,(rating,genre))*.
Το broadcast join που υλοποιήθηκε αποτελεί μια πολύ συγκεκριμένη περίπτωση Join για παράδειγμα. Δεν είναι δύσκολη όμως η γενίκευση του παραδείγματος.Αντί να το τελικό tuple να έιναι (movie_id,(rating,genre)), θα μπορούσε να είναι *(common_id,((values_of_smallRDD),(values_of_bigRDD))*.

Broadcast Join

```
        Init ()
        if R in local storage then
        remotely retrieve R partition R into p chunks R_1...R_p
                        save R_1 ...R_p to local storage
      end if
      if R < a split of L then
        H_R ←build a hash table from R_1 ...R_p
      else
        H_L1...H_Lp ←initialize p hash tables for L
      end if

  Map (K: null, V : a record from an L split)
  if H_R exist then
    probe H_R with the join column extracted from V
    for each match r from H_R do
      emit (null, new_record(r, V ))
    end for
  else
    add V to an H_Li
    hashing its join column
  end if

  Close ()
  if H_R not exist then
    for each non-empty H_Li do
        load R_i in memoryfor each record r in R_i do  probe H_Li with r's join column
        for each match l from H_Li do
          emit (null, new record(r, l))
        end for
      end for
    end for
  end if
```

Ζητούμενο 2.2

Standard Repartition Join

Map(K: null, V : a record from a split of either R or L)
join_key ←extract the join column from V
tagged_record ←add a tag of either R or L to V
emit (join key, tagged record)
Reduce (K': a join key,LIST_V': records from R and L with join key K')
create buffers B_R and B_L for R and L, respectively
for each record t ∈LIST_V⊠ **do**
append t to one of the buffers according to its tag
for each pair of records (r, l) $\in B_R \times B_L$ **do**
end for
end for
emit (null, new record(r, l))

Ζητούμενο 2.3

Υλοποίησαμε στα προηγούμενα ερωτήματα τα 2 είδη joins (repartition και broadcast) πάνω στους πίνακες genres και ratings. Καθώς ο σκοπός ήταν καθαρά για λόγους σύγκρισης χρησιμοποίσαμε μόνο 2 πεδία από τον κάθε πίνακα.

- `Time Using Broadcast Join is 82.61 sec.`
- `Time Using Repartition Join is 574.88 sec.`

Τα αποτελέσματα μας δείχνουν συγκριτικό πλεονέκτημα του broadcast join. Το αποτέλεσμα ήταν αναμενόμενο λόγω του μικρού μεγέθους του movie_genres πίνακα. Κατά τη συννένωση ενός μεγάλου με ενός μικρού πινακα αυτό το είδος Join θεωρείται ιδανικού καθώς αποστέλλεται (broadcasting) ένα αντιγραφο του μικρού πίνακα σε όλους τους workers και ο κάθε worker πραγματοποιεί συνένωση ενός partition του μεγάλου πίνακα με τον μικρότερο.
Όταν χρησιμοποιούμε repartition join κάθε πραγματοποιείται μια map-reduce διαδικασία. Κάθε εγγραφή λαμβάνει ένα χαρακτηριστικό που προσδιορίζει από ποιόν πίνακα προέρχεται. Στη συνέχεια οι εγγραφές γίνονται merged και περνάμε στη φάση reduce. Σε αυτό το στάδιο οι εγγραφές τοποθετούνται σε buffers ανάλογα με το κλειδί προέλευσης από πινακα. Στη συνέχεια συνενώνονται οι πιθανοί συνδυασμοί.
Είναι προφανές πως χρησιμοποιώντας το broadcast join κερδίζουμε χρόνο καθώς αποφεύγουμε το περιττό (για το συγκεκριμένο παράδειγμα) overhead λόγω κίνησης στο δίκτυο.

Ζητούμενο 2.4

Στο σημείο αυτό τρέχουμε το ίδιο script με δύο διαφορετικές ρυθμίσεις κάθε φορά για ένα σχετικά μικρό αρχείο εισόδου. Οι έξοδοι που λαμβάνουμε είναι οι εξής:

- `Time with choosing join type disabled is 3.6126 sec.`
- `Time with choosing join type enabled is 5.9754 sec.`

Στην περίπτωση του `disable` χαμηλώνουμε το κατώφλι πάνω από το οποίο πραγματοποιείται το *broadcast join*. Με τον τρόπο αυτό, το spark αναγκάζεται να πραγματοποιήσει broadcast join κατά το οποίο ο μικρότερος πίνακας αποστέλλεται σε κάθε reducer. Αυτό προφανώς δεν είναι δυνατό αν ο ένας από τους δύο πίνακες στους οποίους πρέπει να γίνει *join* δεν είναι αρκετά μικρός.

Στην περίπτωση αυτή χρειάζεται να πραγματοποιήσουμε *shuffle join*. Όμως, στην προκειμένη περίπτωση ο εν λόγω αλγόριθμος απαιτεί περισσότερο χρόνο καθώς πρέπει τα δεδομένα να υποστούν shuffling και να διανεμηθούν σε κάθε worker όσα έχουν ίδιο κλειδί. Ενώ είναι σίγουρο ότι αυτό δουλεύει πάντα, εν προκειμένω, το overhead για τη διαδικασία αυτή υπερσκελίζει τον χρόνο εκτέλεσης του *broadcast join* το οποίο δεν χρειάζεται την διαδικασία του shuffling.