Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Η/Υ

Προχωρημένα Θέματα Βάσεων Δεδομένων - Ροή Λ

Εξαμηνιαία Εργασία - Αναφορά

"Χρήση του Apache Spark στις Βάσεις Δεδομένων"

Ονοματεπώνυμα:
Οικονόμου Ελένη
Σταθά Ευσταθία

Αριθμοί Μητρώου:
el16301
el16190

Πίνακας Περιεχομένων

Μέρος 1

Ζητούμενο 3

Ψευδοκώδικας Map Reduce για τα Queries

Query 1

```
map(line_id, line):
  #line is in csv format
  #line comes from movies.csv file
  if (line.split(‘,’)[3] != ‘’ && int(line.split(‘,’)[3][0:4] >= 2000 && line.split(‘,’)[5] != ‘’ &&
                                                                                    line.split(‘,’)[6] != ‘’)
    date = int(line.split(‘,’)[3][0:4]
    name = line.split(‘,’)[1]
    cost = int(line.split(‘,’)[5])
    income = int(line.split(‘,’)[6])
    profit = 100*((income-cost)/cost))
    emit(date, (name, profit))

reduce(date, tuple_list):
  max = 0
  name = ‘None’
  for tuple in tuple_list:
    if (tuple[1] > max):
      max = tuple[1]
      name = tuple[0]
  emit(date, name, max)
```

Query 2

```
map(line_id, line):
  #line is in csv format
  #line comes from ratings.csv file
  user = line.split(‘,’)[0]
  rating = float(line.split(‘,’))[2]
  emit(user, rating)

reduce(date, rating_list):
  ratings = 0
  sum = 0
  for rat in rating_list:
    ratings += 1
    sum += rat
  avg_rating = sum/ratings
  if (avg_rating > 3.0):
    emit(‘same’, ‘high’)
  else:
  emit(‘same’, ‘all’)
```

```
map(key, value):
  emit(key, value)

reduce(_, value_list):
  all_users = value_list.length()
  users = 0
  for value in value_list:
     if (value == 'high'):
       users += 1
  percentage = 100*users/all_users
  emit('none', percentage)
```

Query 3

```
map(file_type, line):
  #file type is either 'rating' for ratings.csv or 'genres' for movie_genres.csv
  #line is in csv format
  if (file_type == 'genres'):
    genre = line.split(',')[0]
    movie_id = line.split(',')[1]
    emit(movie_id, ('g', genre))
  else:
    movie_id = line.split(',')[1]
    rating = float(line.split(',')[2])
    emit(movie_id, ('r', rating))

reduce(movie_id, tuple_list):
  genres_list = []
  for tuple in tuple_list:
     if (tuple[0] == 'g'):
        genres_list.add(tuple[1]) // we think that l.add(x) adds x in list l
        tuple_list.remove(tuple) // we think that l.remove(x) removes x from list l
   for genre in genres_list:
      for tuple in tuple_list:
        emit((genre, movie_id), tuple[1])

map(key,value):
  emit(key,value)

reduce((genre, movie_id), rating_list):
  sum = 0
  ratings = 0
  for rat in rating_list:
     sum += rat
     ratings += 1
  avg_rating = sum/ratings
  emit(genre, avg_rating)

map(key, value):
  emit(key, value)
```

```
reduce(genre, rating_list):
  sum = 0
  ratings = 0
  for rat in rating_list:
     sum += rat
     ratings += 1
  avg_rating = sum/ratings
  emit(genre, avg_ratings)
```

Query 4

```
map(file_type, line):
  #file type is either ‘moves’ for movies.csv or ‘genres’ for movie_genres.csv
  #line is in csv format
  #we think that .split() function works like given split_complex()

  if (file_type == ‘genres’):
    genre = line.split(‘,’)[0]
    movie_id = line.split(‘,’)[1]
    if(genre == ‘Drama’):
      emit(movie_id, (‘g’, genre))
  else:
    movie_id = line.split(‘,’)[0]
    length = len(line.split(‘,’)[2])
    if (line.split(‘,’)[3] != ‘’):
      date = int(line.split(‘,’)[3])
      if (date >= 2000 && date <= 2004):
         emit(movie_id, (‘m’, length, ‘2020-2004’))
      else if (date >= 2005 && date <= 2009):
         emit(movie_id, (‘m’, length, ‘2005-2009’))
      else if (date >= 2010 && date <= 2014):
         emit(movie_id, (‘m’, length, ‘2010-2014’))
      else if (date >= 2015 && date <= 2019):
         emit(movie_id, (‘m’, length, ‘2015-2019’))

reduce(movie_id, tuple_list):
  genres_list = []
  flag = false
  for tuple in tuple_list:
     if (tuple[0] == ‘g’):
       tuple_list.remove(tuple)
       flag = true
       brake
  if (flag == true):
      for tuple in tuple_list:
        emit(tuple[2], length)

map(key, value):
  emit(key, value)

reduce(date_stamp, length_list):
  sum = 0
```

```
  lengths = 0
  for len in length_list:
    sum += len
    lengths += 1
  avg_length = sum/lengths
  emit(date_stamp, avg_length)
```

Query 5

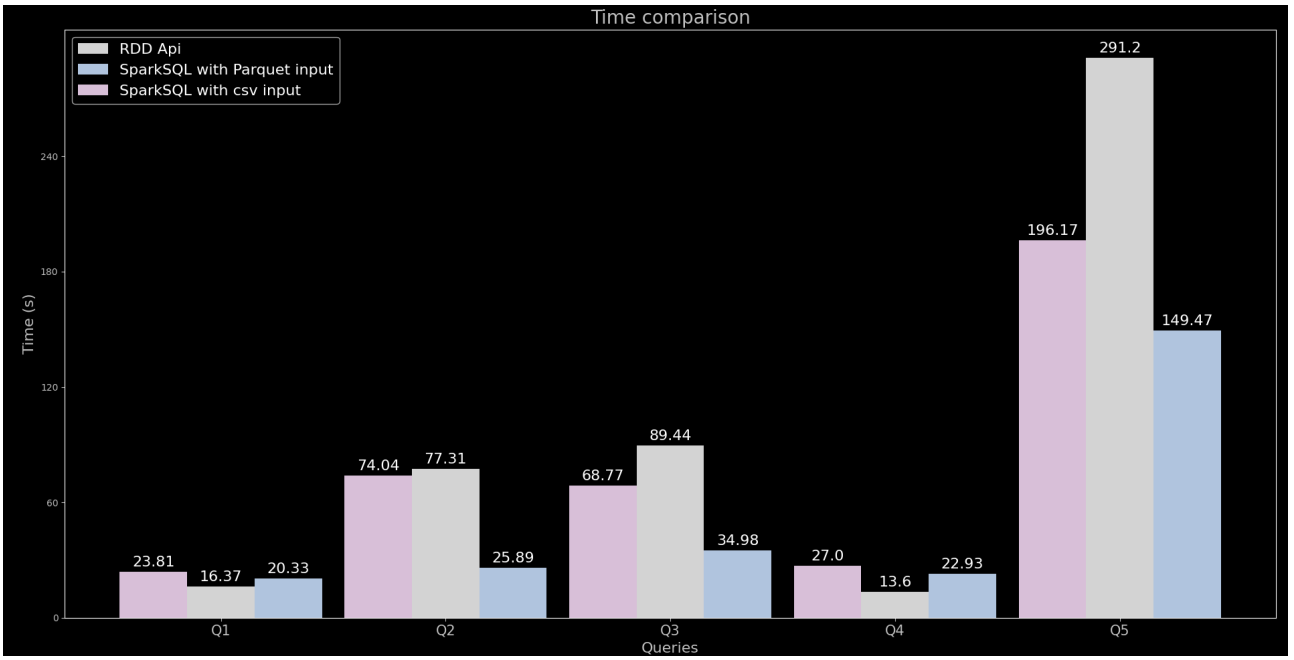
```
map(file_type, line):
  #file type is either ‘rating’ for ratings.csv or ‘genres’ for movie_genres.csv
  #line is in csv format
  if (file_type == ‘genres’):
    genre = line.split(‘,’)[0]
    movie_id = line.split(‘,’)[1]
    emit(movie_id, (‘g’, genre))
  else:
    movie_id = line.split(‘,’)[1]
    user_id = line.split(‘,’)[0]
    emit(movie_id, (‘r’, user_id))

reduce(movie_id, tuple_list):
  genres_list = []
  for tuple in tuple_list:
     if (tuple[0] == ‘g’):
        genres_list.add(tuple[1]) // we think that l.add(x) adds x in list l
        tuple_list.remove(tuple) // we think that l.remove(x) removes x from list l
   for genre in genres_list:
      for tuple in tuple_list:
         emit((genre, tuple[1]), 1)

map(key,value):
  emit(key,value)

reduce((genre, user_id), user_id_list):
  ratings = 0
  for user in user_id_list:
     ratings += 1
  emit(genre, (user_id, ratings))

map(key, value):
  emit(key, value)

reduce(genre, tuple_list):
  max = 0
  user = ‘none’
  for tuple in tuple_list:
     if (tuple[1] > max):
        max = tuple[1]
        user = tuple[0]
  write_to_file(‘category_ratings.csv’, (genre, user, max) # we use write to file to keep the result
```

```
map(file_type, line):
  #file type is ‘movies’ for movies.csv, ‘genres’  for movie_genres.csv, or ‘ratings’ for ratings.csv
   #line is in csv format
   if (file_type == ‘genres’):
      genre = line.split(‘,’)[0]
      movie_id = line.split(‘,’)[1]
      emit(movie_id, (‘g’, genre))
   else if(file_type == ‘ratings’):
      movie_id = line.split(‘,’)[1]
      rating = float(line.split(‘,’)[2])
      user_id = line.split(‘,’)[0]
      emit(movie_id, (‘r’, user_id, rating))
else:
      movie_id = line.split(‘,’)[1]
      user_id = line.split(‘,’)[0]
      emit(movie_id, (‘r’, user_id))

reduce(movie_id, tuple_list):
   genres_list = []
   for tuple in tuple_list:
       if (tuple[0] == ‘g’):
          genres_list.add(tuple[1]) // we think that l.add(x) adds x in list l
          tuple_list.remove(tuple) // we think that l.remove(x) removes x from list l
       if (tuple[0] == ‘m’):
          movie_name = tuple[1]
          movie_popularity = tuple[2]
          tuple_list.remove(tuple) // we think that l.remove(x) removes x from list l
    for genre in genres_list:
        for tuple in tuple_list:
           emit((genre, tuple[1]), (movie_name, tuple[2], movie_popularity))

map(key, value):
   emit(key, value)

reduce((genre, user_id), tuple_list):
   max_rat = 0
   min_rat = 0
   max_pop = 0
   min_pop = 0
   max_name = ‘none’
   min_name = ‘none’
   for tuple in tuple_list:
      if (max_rat < tuple[1] || (max_rat == tuple[1] && max_pop < tuple[2]):
         max_name = tuple[0]
         max_pop = tuple[2]
         max_rat = tuple[1]
      if (min_rat > tuple[1] || (min_rat == tuple[1] && min_pop < tuple[2]):
         min_name = tuple[0]
         min_pop = tuple[2]
         min_rat = tuple[1]
   write_to_file(‘fav_worst_movie.csv’, (genre, user_id, max_name, max_rat, min_name, min_rat)
```

```
map(file_type, line):
  #file type is either ‘f_w_movie’ for fav_worst_movie.csv or ‘cat_rating’ for category_rating.csv
  #line is in csv format
  if (file_type == ‘f_w_movie’):
    genre = line.split(‘,’)[0]
    user_id = line.split(‘,’)[1]
    fav_name = line.split(‘,’)[2]
    fav_rat = line.split(‘,’)[3]
    worst_name = line.split(‘,’)[4]
    worst_rat = line.split(‘,’)[5]
    emit((genre, user_id), (‘f_w’, fav_name, fav_rat, worst_name, worst_rat))
  else:
    genre = line.split(‘,’)[0]
    user_id = line.split(‘,’)[1]
    ratings = line.split(‘,’)[2]
    emit((genre, user_id), (‘c_r’, ratings))

reduce((genre, user_id), tuple_list):
  c_r_list = []
  for tuple in tuple_list:
     if (tuple[0] == ‘c_r’):
        c_r_list.add(tuple[1]) // we think that l.add(x) adds x in list l
        tuple_list.remove(tuple) // we think that l.remove(x) removes x from list l
   for ratings in c_r_list:
      for tuple in tuple_list:
         emit(genre, (user_id, ratings, tuple[1], tuple[2], tuple[3], tuple[4]))
```

Ζητούμενο 4

Διαγράμματα - Σύγκριση χρόνων



Σχολιασμός

Αρχικά, αξίζει να αναφερθεί πως στην σύγκριση που αφορά τη χρήση του RDD API, έναντι της SparkSQL θα ασχοληθούμε με τα συμπεράσματα που προκύπτουν από τα queries που τρέχουν για περισσότερο από μισό λεπτό. Ο λόγος για την επιλογή αυτή είναι πως σε τόσο "μικρά" (ως προς το χρόνο επεξεργασίας) ερωτήματα, μια κακή υλοποίηση μπορεί να επηρεάσει σημαντικά το αποτέλεσμα, όσων αφορά τον χρόνο εκτέλεσης της υλοποίησης τους με την μια ή την άλλη επιλογή. Αναγνωρίζουμε, λοιπόν, πως για το Query 1 και για το Query 4 ενδεχομένως η ταχύτερη εκτέλεση της υλοποίησης με RDD API να οφείλεται, απλώς, σε μια όχι βέλτιστη υλοποίηση με το SparkSQL. Στα υπόλοιπα queries είναι εμφανές πως η SparkSQL τα πηγαίνει εμφανώς καλύτερα από το RDD API, με τις εκτελέσεις που αφορούν είσοδο με csv μορφή να είναι αρκετά κοντά, όμως η διαφορά ανάμεσα στους διαφορετικούς τύπους εισόδου θα σχολιαστεί ακολούθως. Η καλύτερη απόδοση της SparkSQL ήταν σε μεγάλο βαθμό αναμενόμενη, αφού η ίδια ενσωματώνει έναν βελτιστοποιητή (Catalyst Optimizer), ο οποίος φροντίζει για την βελτιστοποίηση της απόδοσης των queries που γράφονται από τον εκάστοτε προγραμματιστή. Συγκεκριμένα, ο ίδιος είναι υπεύθυνος για την παραλλαγή των queries προκειμένου αυτά να τρέχουν με τον καλύτερο δυνατό τρόπο χρησιμοποιώντας τεχνικές όπως το φιλτράρισμα και, ακόμα, την εύρεση του καλύτερου δυνατού τρόπου εκτέλεσης των Joins. Στο Query 5 για παράδειγμα, όπου το join 3 διαφορετικών πινάκων είναι απαραίτητο, ο βελτιστοποιητής φροντίζει για την πραγματοποίηση του με την καλύτερη δυνατή επιλογή ανάμεσα σε αυτές που προσφέρει η SparkSQL για τα joins. Επιπρόσθετα, στη διαφορά ανάμεσα στη χρήση SparkSQL και RDD API συνδράμει το γεγονός πως, διαδικασίες όπως το group ή το aggregate επί των δεδομένων μας είναι πολύ πιο βαριές όταν δουλεύουμε με RDD API, καθώς σε αυτή την περίπτωση υπάρχει το στάδιο της μεταφοράς δεδομένων μέσω του δικτύου που αναπόφευκτα οδηγεί σε επιπλέον χρονική καθυστέρηση.

Αναφορικά με τη διαφορά στην απόδοση των ιδίων SparkSQL ερωτημάτων, όταν αυτά γίνουν πάνω σε αρχεία σε csv ή parquet μορφή, αναμέναμε καλύτερη απόδοση των ερωτημάτων μας στη δεύτερη περίπτωση. Αυτό συμβαίνει λόγω, τόσο του μικρότερου αποτυπώματος αυτών των αρχείων στη μνήμη και στο δίσκο, με αποτέλεσμα την πολύ ταχύτερη ανάγνωση και εγγραφή τους, αλλά και λόγω των ιδιοτήτων αυτών των αρχείων. Συγκεκριμένα, λόγω της μεταπληροφορίας που καταγράφουν παρέχουν πολλαπλές δυνατότητες βελτιστοποίησης της επεξεργασίας των queries, καθώς σε αρκετές περιπτώσεις το πλήθος των δεδομένων που χρειάζεται να σκαναριστούν μειώνεται σε πολύ μεγάλο βαθμό κι, έτσι, η ταχύτητα της εκτέλεσης των ερωτημάτων αυξάνεται. Επομένως, σε όλα μας τα queries υπήρχαν δύο διαφορετικά σημεία όπου το parquet αναμέναμε να επιταχύνει τη διαδικασία και αυτό πράγματι αποδείχτηκε στην πράξη, όπως φαίνεται ξεκάθαρα και στο παραπάνω διάγραμμα.

Όσων αφορά τη χρήση του option inferSchema κατά το διάβασμα αρχείων σε csv μορφή, αυτή επιβαρύνει ακόμα περισσότερο το χρόνο εκτέλεσης, καθώς απαιτεί ένα επιπλέον σκανάρισμα του αρχείου εισόδου, προκειμένου να αναγνωρίσει τον τύπο δεδομένων κάθε κολώνας και, έτσι, επιβραδύνει την ανάγνωση. Ασφαλώς, στον αντίποδα μας προσφέρει ένα dataframe που έχει ορθά ορισμένους τους τύπους στις κολώνες, ωστόσο, στην περίπτωση δεδομένων, όπως αυτά που διαχειριζόμαστε, για τα οποία είμαστε σε θέση να εξάγουμε με ευκολία αυτή την πληροφορία χωρίς να χρειαστεί η διαδικασία να αυτοματοποιηθεί για εμάς είναι προτιμότερο να γλιτώσουμε την χρονική αυτή καθυστέρηση.

Μέρος 2

Ζητούμενο 3

Ψευδοκώδικας Map Reduce για τα Joins

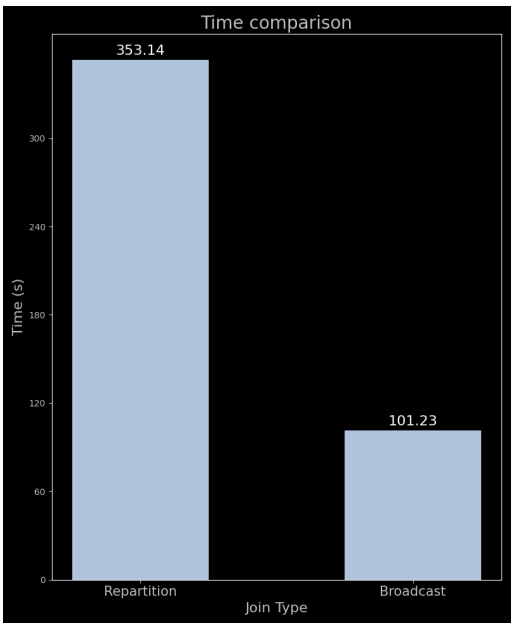
Broadcast Join

```
Preprocess(small_table):
  broadcast small table to all machines

map(key, value):
  # key is the join key and value is a tuple with all the other data
  # small_table is cached
  for tuple in small_table:
    # tuple.join_key returns the join_key of the tuple
    # tuple.values returns the tuple without join_key
    if tuple.join_key == key:
    emit(key, value, tuple.values)
```

Repartition Join

```
map(file_type, tuple):
  # file_type is either t1 or t2
  # tuple has the key and all the other data and supports the two "methods" explained in broadcast join
  if file_type == 't1':
    emit(tuple.join_key, ('t1', tuple.values))
  else:
    emit(tuple.join_key, ('t2', tuple.values))

reduce(join_key, tuple_list):
  emit(join_key, tuple)

map(join_key, tuple_list):
  t1 = []
  for tuple in tuple_list:
    if (tuple[1][0] == 't1'):
      t1.append(tuple[1][1])
    else:
      t2.appent(tuple[1][1])
  for tuple1 in t1:
     for tuple2 it t2:
       emit(join_key, tuple1, tuple2)
```

Διαγράμματα - Σύγκριση χρόνων



Σχολιασμός

Όπως φαίνεται και στο ανωτέρω διάγραμμα οι δύο μέθοδοι παρουσιάζουν σημαντικά μεγάλη απόκλιση ως προς το χρόνο εκτέλεσης τους. Συγκεκριμένα, η εκτέλεση του Repartition Join απαιτεί υπερτριπλάσιο χρόνο σε σχέση με την εκτέλεση του Broadcast Join. Το αποτέλεσμα αυτό είναι καθ' όλα αναμενόμενο, καθώς στη μια περίπτωση έχουμε ένα Map Side Join, ενώ στην άλλη ένα Reduce Side Join. Στο Repartition Join αφού ολοκληρώσουμε τη διαδικασία από την πλευρά του mapper, οδηγούμαστε στην εκτέλεση ενός groupByKey (το οποίο λειτουργεί σαν reduce) για να προσθέσουμε έπειτα ένα στάδιο flatMap (λειτουργεί σαν map), που θα μας βγάλει το τελικό αποτέλεσμα. Η αλυσίδα του RDD είναι σημαντικά πολυπλοκότερη και μακρύτερη και, ως εκ τούτου, η διαδικασία απαιτεί πολύ περισσότερο χρόνο, ενώ, ακόμη, υπάρχει το στάδιο της μεταφοράς των δεδομένων πάνω από το δίκτυο (η οποία μεσολαβεί ανάμεσα στο αρχικό map και reduce) που μας οδηγεί αναπόφευκτα σε περαιτέρω χρονική επιβάρυνση. Ενδεχομένως η απόκλιση να μπορεί να είναι μικρότερη, στην περίπτωση διαφορετικής υλοποίησης, καθώς και η ίδια η επιλογή του groupByKey στο Repartition Join κάνει την καθυστέρηση μεγαλύτερη, αφού πρόκειται για μια αρκετά βαριά επιλογή σε σύγκριση με το ελαφρότερο reduceByKey, με το δεύτερο να κάνει ένα combine των δεδομένων που βρίσκονται στον ίδιο maper πριν το shuffle, κάτι που το groupByKey δεν κάνει. Έτσι, μια πιθανή υλοποίηση με reduceByKey ίσως να προξενούσε μικρότερη χρονική διαφορά ανάμεσα στις δυο μεθόδους join που υλοποιήθηκαν.

Ζητούμενο 4

Enabled - Πλάνο εκτέλεσης

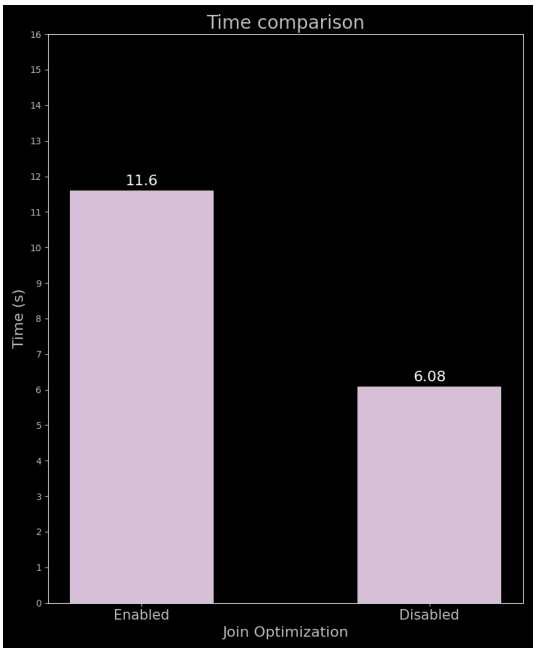
```
== Physical Plan ==
*(3) BroadcastHashJoin [_c0#8], [_c1#1], Inner, BuildLeft
:- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as bigint)))
:  +- *(2) Filter isnotnull(_c0#8)
:     +- *(2) GlobalLimit 100
:        +- Exchange SinglePartition
:           +- *(1) LocalLimit 100
:              +- *(1) FileScan parquet [_c0#8,_c1#9] Batched: true, Format: Parquet, Location:
InMemoryFileIndex[hdfs://master:9000/movies/movie_genres.parquet], PartitionFilters: [], Pushe
```

dFilters: [], ReadSchema: struct<_c0:int,_c1:string>
+- *(3) Project [_c0#0, _c1#1, _c2#2, _c3#3]
  +- *(3) Filter isnotnull(_c1#1)
    +- *(3) FileScan parquet [_c0#0,_c1#1,_c2#2,_c3#3] Batched: true, Format: Parquet, Location:
InMemoryFileIndex[hdfs://master:9000/movies/ratings.parquet], PartitionFilters: [], PushedF
ilters: [IsNotNull(_c1)], ReadSchema: struct<_c0:int,_c1:int,_c2:double,_c3:int>

Disabled - Πλάνο Εκτέλεσης

```
== 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/movies/movie_genres.parquet], PartitionFilters: [], Pu
shedFilters: [], ReadSchema: struct<_c0:int,_c1:string>
+- *(5) Sort [_c1#1 ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(_c1#1, 200)
      +- *(4) Project [_c0#0, _c1#1, _c2#2, _c3#3]
         +- *(4) Filter isnotnull(_c1#1)
            +- *(4) FileScan parquet [_c0#0,_c1#1,_c2#2,_c3#3] Batched: true, Format: Parquet, Location:
InMemoryFileIndex[hdfs://master:9000/movies/ratings.parquet], PartitionFilters: [], P
ushedFilters: [IsNotNull(_c1)], ReadSchema: struct<_c0:int,_c1:int,_c2:double,_c3:int>
```

Διάγραμμα - Σύγκριση χρόνων



Σχολιασμός

Επεμβαίνοντας στον βελτιστοποιητή και κλείνοντας την δυνατότητα του να κάνει optimization τα join χρησιμοποιώντας το BroadcastHash Join, ο ίδιος οδηγείται σε ένα πλάνο εκτέλεσης με SortMerge Join, το οποίο συνιστά την αμέσως καλύτερη επιλογή. Όπως είναι αναμενόμενο, το πλάνο εκτέλεσης αυτό έχει σημαντικά χειρότερη απόδοση, οδηγώντας σε, σχεδόν, διπλάσιο χρόνο εκτέλεσης. Άλλωστε στη μια περίπτωση έχουμε ένα Map-side Join που αποφεύγει τελείως τη μεταφορά δεδομένων πάνω από το δίκτυο (shuffle), ενώ στην άλλη περίπτωση αν και δουλεύουμε και πάλι με Map-side Join η μεταφορά των δεδομένων είναι αναπόφευκτη, καθώς για να ολοκληρωθεί το join πρέπει τα ίδια κλειδιά από κάθε dataset να φτάσουν στους ίδιους mapper. Επιπρόσθετα, χρειάζεται τα κλειδιά αυτά να ταξινομηθούν (sort) με τρόπο τέτοιο ώστε να μπορούν να γίνουν parse παράλληλα και να γίνει το join ανάμεσα στις τούπλες που έχουν τα ίδια κλειδιά. Δηλαδή, το SortMerge Join προσθέτει κάποια περαιτέρω στάδια επεξεργασίας για να υπάρχει εγγύηση ότι τα δεδομένα που αντιστοιχούν στα ίδια keys θα οδηγηθούν στα ίδια partitions και θα είναι ταξινομημένα με τον ίδιο τρόπο. Και μόνο η μεταφορά των δεδομένων είναι ένα εξαιρετικά χρονοβόρο και βαρύ στάδιο, το οποίο σε μεγάλο βαθμό εξηγεί τη χρονική διαφορά ανάμεσα στις δύο μεθόδους, όμως η προσθήκη της ταξινόμησης επιφέρει κι αυτή μια περαιτέρω χρονική καθυστέρηση που επιβαρύνει την κατάσταση.