Προχωρημένα Θέματα Βάσεων Δεδομένων
Εξαμηνιαία Εργασία 2019-2020

Ονοματεπώνυμο: Κυριάκου Αθηνά
Α.Μ.: 03117405
Εξάμηνο: 9ο

Θέμα 2ο: Υλοποίηση και μελέτη απόδοσης αλγορίθμων συνένωσης για αναλυτική επεξεργασία δεδομένων

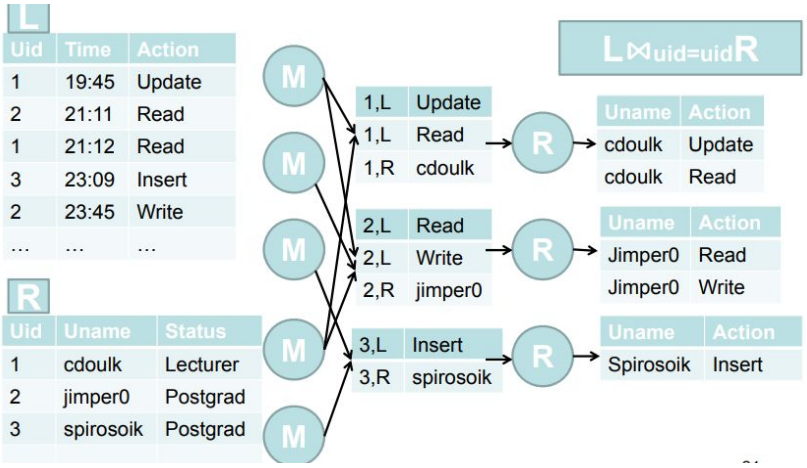
Ζητούμενο του συγκεκριμένου θέματος ήταν η μελέτη διαφορετικών υλοποιήσεων για τη συνένωση δεδομένων στο περιβάλλον Map-Reduce του Spark. Πιο συγκεκριμένα, υλοποιήθηκαν οι αλγόριθμοι Reduce-Side join και Map-Side join, όπως περιγράφονται στη δημοσίευση “A Comparison of Join Algorithms for Log Processing in MapReduce”, Blanas et al, in Sigmod 2010 με το Pyspark API. Στη συνέχεια, συγκρίθηκε η απόδοσή τους με την υλοποίηση join που επιλέγει το SparkSQL API για να βελτιστοποιήσει την εκτέλεση του ερωτήματος.

Ο ψευδοκώδικας των συναρτήσεων map και reduce θα δίνεται στην παρακάτω μορφή:

MAP(key, value):
//επεξεργασία για κάθε <key,value>
emit(key2,value2)

REDUCE(key2,list(value2)):
//επεξεργασία για κάθε <key2,list(value2)>
emit(key3,value3)

Υλοποίηση του Repartition join:



Περιγραφή της υλοποίησης:

Αφού ανακτηθούν τα αρχεία με τις πληροφορίες για τις διαδρομές και τα ταξί από το HDFS σε μορφή RDD, το ερώτημα της συνένωσης πραγματοποιείται με κλήση της συνάρτησης repartition_join() και ορίσματα τα δύο RDDs. Το repartition join αποτελείται από φάσεις map και reduce.

Στη repartition_join() με χρήση της map() κάθε RDD μετασχηματίζεται σε tuples αποτελούμενες από το κοινό key (το ID της διαδρομής) και μία tuple με τις πληροφορίες που παρέχει κάθε αρχείο για το εκάστοτε key. Το πρώτο στοιχείο της tuple των πληροφοριών είναι μία ετικέτα που υποδηλώνει το RDD από το οποίο προέκυψε η πληροφορία˙ για το αρχείο των ταξί η ετικέτα είναι ο χαρακτήρας 'R', ενώ για των διαδρομών ο 'L'.

Στη συνέχεια, τα tuples κλειδιών και ετικετοποιημένων πληροφοριών συνενώνονται για τη δημιουργία του RDD full_data. Η φάση του reduce, πραγματοποιείται στο full_data με χρήση της συνάρτησης combineByKey(), η οποία χρησιμοποιείται εσωτερικά και στην υλοποίηση των έτοιμων συναρτήσεων reduce() και reduceByKey() του Spark. Για την combineByKey() ορίστηκαν οι συναρτήσεις[1] create_combiner() και merge_value() που εφαρμόζονται σε κάθε partition και η merge_combiners() που συνενώνει για κάθε key τα αποτελέσματα των δύο προηγούμενων από όλα τα partitions. Στη συγκεκριμένη περίπτωση, η έξοδος της combineByKey() είναι για κάθε κλειδί (ID διαδρομής) μία tuple δύο λιστών που αντιπροσωπεύουν τους δύο buffers BL και BR, όπως περιγράφονται στη δημοσίευση. Ο χωρισμός των πληροφοριών στους buffers γίνεται βάσει των ετικετών τους και η δημιουργούμενη tuple έχει τις πληροφορίες από όλα τα partitions του RDD για κάθε key και βρίσκεται σε όλα τα partitions[2].

Η συνένωση γίνεται καλώντας τη συνάρτηση buff_cross_prod() για κάθε record του RDD buffers που προέκυψε από την combineByKey(). Η συνάρτηση πραγματοποιεί το καρτεσιανό γινόμενο όλων των πληροφοριών του αρχείου του ταξί και αυτές του αρχείου των

[1] https://www.linkedin.com/pulse/spark-pyspark-combinebykey-simplified-swadeep-mishra-1/
[2] https://www.edureka.co/blog/apache-spark-combinebykey-explained

διαδρομών. Για κάθε ζεύγος πληροφορίας, ο εκάστοτε worker στο key δημιουργεί μία tuple μαζί με το ID της διαδρομής και την προσθέτει στον accumulator result_acc.

Τελικά ο driver που μπορεί να διαβάσει τον accumulator, γράφει το αποτέλεσμα στο αρχείο “my_repartition_join_output.txt”.

Ο χρόνος εκτέλεσης για το Repartition Join είναι:

```
Repartition join computational time: 1649.14465213 secs
```

Ψευδοκώδικας Map-Reduce:

Φάση Map:

Για τα ταξί (yellow_tripvendors.csv):

```
MAP (null, record):
//επεξεργασία για κάθε <key,value>
String parts =  record.toString().split(“,”);
int key2 = int(parts[0]);
String value2 = “R, ” + parts[1];
emit(key2,value2)
```

Για τις διαδρομές (yellow_tripdata.csv)

```
MAP (null, record):
//επεξεργασία για κάθε <key,value>
String parts =  record.toString().split(“,”);
int key2 = int(parts[0]);
String value2 = “L, ” + parts[1:];
emit(key2,value2)
```

Φάση Sorting / Shuffling:

Σε αυτή τη φάση θα δημιουργηθεί μία λίστα με τις τιμές που αντιστοιχούν σε κάθε key. Η έξοδος αυτής της φάσης θα είναι της μορφής:
{trip ID – [(R, taxi ID), (L, dpt_time, arr_time, dpt_x, dpt_y, arr_x, arr_y, cost)]}

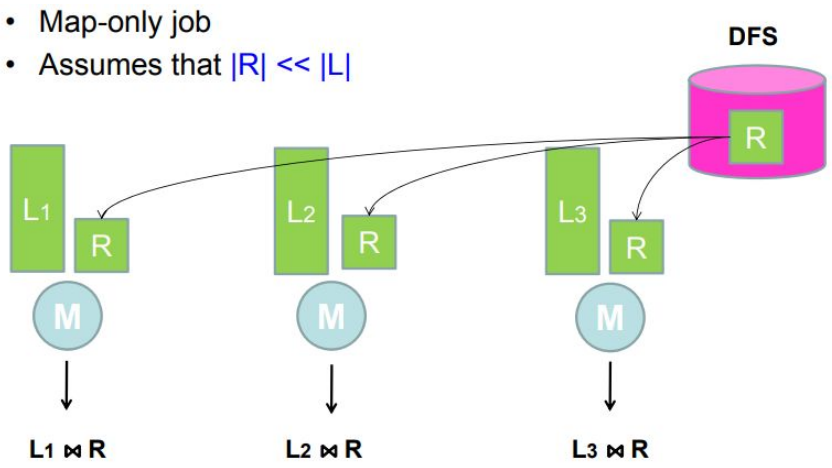
Φάση Reduce:

```
REDUCE (key2, list(value2)):
//επεξεργασία για κάθε <key2,list(value2)>
listValue = list(value2);
List<String> bufferR, bufferL = [];
for (int i = 0; i < listValue.size(); i++){
        String tag = listValue[i][0];
        if (tag.equals(“R”) {
                bufferR.append(listValue[i][1]);
        } else {
                bufferL.append(listValue[i][1:]);
        }
}
```

```
List <String> resultKey2 = [];
for(int i = 0; i < bufferL.size(); i++){
        for(int j=0; j < bufferR.size(); j++){
                resultKey2.append(key2 + bufferL[i] + bufferR[j]);
        }
}
emit(null, resultKey2)
```

Τα output της reducer για κάθε key ενοποιούνται και το τελικό αποτέλεσμα γράφεται σε αρχείο.

Υλοποίηση του Broadcast join:



Περιγραφή της υλοποίησης:

Για την υλοποίηση του Broadcast join όπως περιγράφεται στη δημοσίευση, ανακτώνται αρχικά τα αρχεία από το HDFS, συμβολίζοντας R το αρχείο των ταξί και L των διαδρομών, και με map συναρτήσεις δημιουργούνται όπως και πριν τα tuples.

Σημειώνεται ότι εδώ χρησιμοποιείται επιπλέον η συνάρτηση partitionBy(). Χρησιμοποιείται ο ίδιος partitioner και για τα δύο RDDs, ώστε κάθε key και από τα δύο αρχεία να βρίσκεται στο ίδιο partition[3]. Αυτό θα χρειαστεί στην περίπτωση που το R είναι μεγαλύτερο κάποιου Li, οπότε το join γίνεται στην close(), όπως περιγράφεται στη δημοσίευση έχοντας διαμερίσει τα R και L με τον ίδιο τρόπο και εφόσον δεν απαιτείται sorting ως προς το ID των διαδρομών[4]. Για τα συγκεκριμένα datasets αυτό δεν ισχύει αφού το R είναι μικρότερο κάθε partition του L, αλλά υλοποιήθηκε για λόγους πληρότητας ώστε να υπάρχει αντιστοίχιση με τον αλγόριθμο της δημοσίευσης. Επιλέχθηκε διαμέριση σε 27 partitions εφόσον το Spark διαμερίζει by default το L σε 27 partitions και το R αποτελείται από μόνο 25 records.

Φάση Init και Map:
Στη συνέχεια γίνεται από τον driver broadcast το R και το μέγεθός του (σε bytes). Η διαδικασία της συνένωσης ξεκινά με την εφαρμογή της συνάρτησης init_n_map σε κάθε partition του L, Li. Στην περίπτωση που το μέγεθος του R είναι μικρότερο του Li, ανακτάται στη μνήμη του worker το R το οποίο έχει γίνει broadcast σε μορφή dictionary (hash table) όπου key είναι το ID της διαδρομής. Κάθε γραμμή του Li συνενώνεται με την αντίστοιχη του R με hash στο κοινό key και το αποτέλεσμα εισάγεται στη λίστα του accumulator result_acc.

---

3 https://stackoverflow.com/questions/34368202/when-create-two-different-spark-pair-rdd-with-same-key-set-will-spark-distribut

4 “On the other hand, if the split of L is smaller than R, the join is not done in the map function. Instead, the map function partitions L in the same way as it partitioned R.” pg.978 “A Comparison of Join Algorithms for Log Processing in MapReduce”, Blanas et al 2 , in Sigmod 2010

Για το συγκεκριμένο dataset όπου το R είναι μικρότερο κάθε partition Li, η συνένωση ολοκληρώνεται εδώ. Όπως προαναφέρθηκε περιλαμβάνονται στον κώδικα η init() αν το Li είναι μικρότερο του R (else σκέλος της init_n_map()) και η close() που εκτελείται από τον driver.

Η μορφή αυτή της συνένωσης είναι map side εφόσον η συνένωση γίνεται στη map συνάρτηση αν το R είναι μικρότερο από το partition Li του L ή στην close() αν είναι μεγαλύτερο. Σε κάθε περίπτωση δε χρησιμοποιούνται reduce συναρτήσεις.

Τελικά ο driver που μπορεί να διαβάσει τον accumulator, γράφει το αποτέλεσμα στο αρχείο “my_broadcast_join_output.txt”.

Ο χρόνος εκτέλεσης για το Broadcast Join είναι:

```
Broadcast join computational time: 347.020092964 secs
```

Ψευδοκώδικας Map-Reduce:

Φάση Map:

Για τα ταξί (yellow_tripvendors.csv):

```
MAP (null, record):
//επεξεργασία για κάθε <key,value>
String parts =  record.toString().split(“,”);
int key2 = int(parts[0]);
String value2 = parts[1];
emit(key2,value2)
```

Για τις διαδρομές (yellow_tripdata.csv)

```
MAP (null, record):
//επεξεργασία για κάθε <key,value>
String parts =  record.toString().split(“,”);
int key2 = int(parts[0]);
String value2 = parts[1:];
emit(key2,value2)
```

Για την περίπτωση όπου ||R|| << ||L|| που ισχυεί στα συγκεκριμένα datasets:

Broadcast R σε dictionary -> broadR

```
MAP (key2, value2):
//επεξεργασία για κάθε <key,value> του L
int key3 = key2;
String valL = value2;
String valR = broadR.get(key2);
if (valR not equals None){
        String value3 = valL + valR;
```

```
        emit(key3,value3)
}
```

Σχολιασμός των χρόνων εκτέλεσης για τις δύο υλοποιήσεις:

Παρατηρούμε ότι ο χρόνος εκτέλεσης του Broadcast Join είναι σχεδόν 5 φορές μικρότερος του Repartition Join. Όπως επισημαίνεται και στη δημοσίευση, αυτό συμβαίνει διότι μέσω broadcast, αποθηκεύεται στη μνήμη κάθε node ένα read-only αντίγραφο του R[5]. Έτσι η συνένωση πραγματοποιείται τοπικά σε κάθε node για τα partition του L και αποφεύγεται η φάση του reduce όπου υπάρχει αναδιοργάνωση των τιμών στα partitions βάσει κλειδιού και άρα μεταφορά τιμών του R και του L στο δίκτυο, όπως γίνεται στο Repartition Join.

Αξίζει να σημειωθεί ότι στο συγκεκριμένο dataset δεν υπάρχουν καθυστερήσεις sorting βάσει κλειδιού γιατί τα ID των διαδρομών είναι ήδη διατεταγμένα σε αύξουσα σειρά.

[5] https://spark.apache.org/docs/1.6.1/api/java/org/apache/spark/broadcast/Broadcast.html

Εκτέλεση της συνένωσης μέσω του Spark SQL API:

Αφού τα αρχεία από το HDFS μετατράπηκαν σε Apache Parquet Format και πραγματοποιήθηκε join με το SparkSQL API.

Η υλοποίηση join που επιλέχθηκε είναι η **Broadcast Hash Join (map side join)**. Η επιλογή αυτή είναι η αναμενόμενη καθώς το reference αρχείο με τις πληροφορίες για τους παρόχους ταξί είναι πολύ μικρότερο του αρχείου των διαδρομών. Το μέγιστο μέγεθος του αρχείου reference για να επιλεχθεί αυτός ο αλγόριθμος είναι 10MB ώστε να χωράει στη μνήμη του worker, κάτι που ισχύει.

```
== Physical Plan ==
*(2) Project [trip_id#24, dpt_time#25, arr_time#26, dpt_x#27, dpt_y#28, arr_x#29, arr_y#30, cost#31, taxi_id#5]
+- *(2) BroadcastHashJoin [trip_id#24], [trip_id#4], Inner, BuildRight
   :- *(2) Project [_c0#8 AS trip_id#24, _c1#9 AS dpt_time#25, _c2#10 AS arr_time#26, _c3#11 AS dpt_x#27, _c4#12 A
S dpt_y#28, _c5#13 AS arr_x#29, _c6#14 AS arr_y#30, _c7#15 AS cost#31]
   :  +- *(2) Filter isnotnull(_c0#8)
   :     +- *(2) FileScan parquet [_c0#8,_c1#9,_c2#10,_c3#11,_c4#12,_c5#13,_c6#14,_c7#15] Batched: true, Format: P
arquet, Location: InMemoryFileIndex[hdfs://master:9000/yellow_tripdata_1m.parquet], PartitionFilters: [], PushedFi
lters: [IsNotNull(_c0)], ReadSchema: struct<_c0:string,_c1:string,_c2:string,_c3:string,_c4:string,_c5:string,_c6:
string,_c7:string>
   +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true]))
      +- *(1) Project [_c0#0 AS trip_id#4, _c1#1 AS taxi_id#5]
         +- *(1) Filter isnotnull(_c0#0)
            +- *(1) FileScan parquet [_c0#0,_c1#1] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdf
s://master:9000/yellow_tripvendors_25.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(_c0)], ReadSchema:
struct<_c0:string,_c1:string>

Computational time: 0.665962934494 secs
```

Στην περίπτωση που απενεργοποιηθεί η δυνατότητα χρήσης του Broadcast Hash Join, θέτοντας το autoBroadcastJoinThreshold=-1, η υλοποίηση που επιλέγεται είναι η **Sort Merge Join (reduce side join)**. Η υλοποίηση αυτή επιλέγεται όταν το key στο οποίο γίνεται το join μπορεί να ταξινομηθεί και δε μπορούν να εφαρμοστούν οι Broadcast Hash Join και Shuffle Hash Join.

```
== Physical Plan ==
*(5) Project [trip_id#24, dpt_time#25, arr_time#26, dpt_x#27, dpt_y#28, arr_x#29, arr_y#30, cost#31, taxi_id#5]
+- *(5) SortMergeJoin [trip_id#24], [trip_id#4], Inner
   :- *(2) Sort [trip_id#24 ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(trip_id#24, 200)
   :     +- *(1) Project [_c0#8 AS trip_id#24, _c1#9 AS dpt_time#25, _c2#10 AS arr_time#26, _c3#11 AS dpt_x#27, _c
4#12 AS dpt_y#28, _c5#13 AS arr_x#29, _c6#14 AS arr_y#30, _c7#15 AS cost#31]
   :        +- *(1) Filter isnotnull(_c0#8)
   :           +- *(1) FileScan parquet [_c0#8,_c1#9,_c2#10,_c3#11,_c4#12,_c5#13,_c6#14,_c7#15] Batched: true, For
mat: Parquet, Location: InMemoryFileIndex[hdfs://master:9000/yellow_tripdata_1m.parquet], PartitionFilters: [], Pu
shedFilters: [IsNotNull(_c0)], ReadSchema: struct<_c0:string,_c1:string,_c2:string,_c3:string,_c4:string,_c5:strin
g,_c6:string,_c7:string>
   +- *(4) Sort [trip_id#4 ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(trip_id#4, 200)
         +- *(3) Project [_c0#0 AS trip_id#4, _c1#1 AS taxi_id#5]
            +- *(3) Filter isnotnull(_c0#0)
               +- *(3) FileScan parquet [_c0#0,_c1#1] Batched: true, Format: Parquet, Location: InMemoryFileIndex
[hdfs://master:9000/yellow_tripvendors_25.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(_c0)], ReadSch
ema: struct<_c0:string,_c1:string>

Computational time: 0.180924892426 secs
```

Παρατηρούμε ότι μεταξύ των 2 αλγορίθμων υλοποίησης, μικρότερο χρόνο εκτέλεσης έχει ο Sort Merge Join. Αυτό μπορεί να αιτιολογηθεί δεδομένου ότι τα αρχεία είναι ήδη ταξινομημένα βάσει του ID των διαδρομών και άρα ο Sort Merge Join δεν πραγματοποιεί sorting αλλά μόνο shuffling και merging (Sort false από το αρχείο εκτέλεσης).

Το συμπέρασμα αυτό δεν προκύπτει από τις υλοποιήσεις του προηγούμενου μέρους καθώς η reduce side join είναι πιο χρονοβόρα από τη map side. Ενδεχομένως το shuffling δεν είναι τόσο optimized με την combineByKey() που χρησιμοποιήθηκε.

Το **link στο HDFS site** με τα datasets είναι: http://83.212.76.46:50070/explorer.html#/

Βιβλιογραφία:

https://community.cloudera.com/t5/Support-Questions/spark-spark-socketexception-connection-reset-by-peer/td-p/196930
https://medium.com/parrot-prediction/partitioning-in-apache-spark-8134ad840b0
http://mycourses.ntua.gr/courses/ECE1060/document/%C4%E9%E1%EB%DD%EE%E5%E9%F2/%C4%E9%DC%EB%E5%EE%E7_11_MapReduce_Joins/joins.pdf
https://www.oreilly.com/library/view/learning-spark/9781449359034/ch04.html
https://spark.apache.org/docs/latest/configuration.html
https://spark.apache.org/docs/2.3.0/sql-programming-guide.html
https://spark.apache.org/docs/latest/sql-data-sources-parquet.html
https://stackoverflow.com/questions/31464727/increase-memory-available-to-pyspark-at-runtime
https://stackoverflow.com/questions/31424396/how-does-hashpartitioner-work
https://stackoverflow.com/questions/34368202/when-create-two-different-spark-pair-rdd-with-same-key-set-will-spark-distribut