Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Προχωρημένα Θέματα Βάσεων Δεδομένων 2021-2022

Χρήση του Apache Spark στις Βάσεις Δεδομένων

Όνομα: Γεώργιος

Επίθετο: Κοσμάς

Α.Μ. : 03118071

Μέρος 1ο

Ζητούμενο 1

Δημιουργούμε directory files στο hdfs με την εντολή

```
hadoop fs -mkdir hdfs://master:9000/files
```

Φορτώνουμε τα αρχεία που μας δώθηκαν στο hdfs μέ τις παρακάτω εντολές:

```
hadoop fs -put movies.csv hdfs://master:9000/files/movies.csv

hadoop fs -put movie_genres.csv hdfs://master:9000/files/movie_genres.csv

hadoop fs -put ratings.csv hdfs://master:9000/files/ratings.csv
```

Ζητούμενο 2

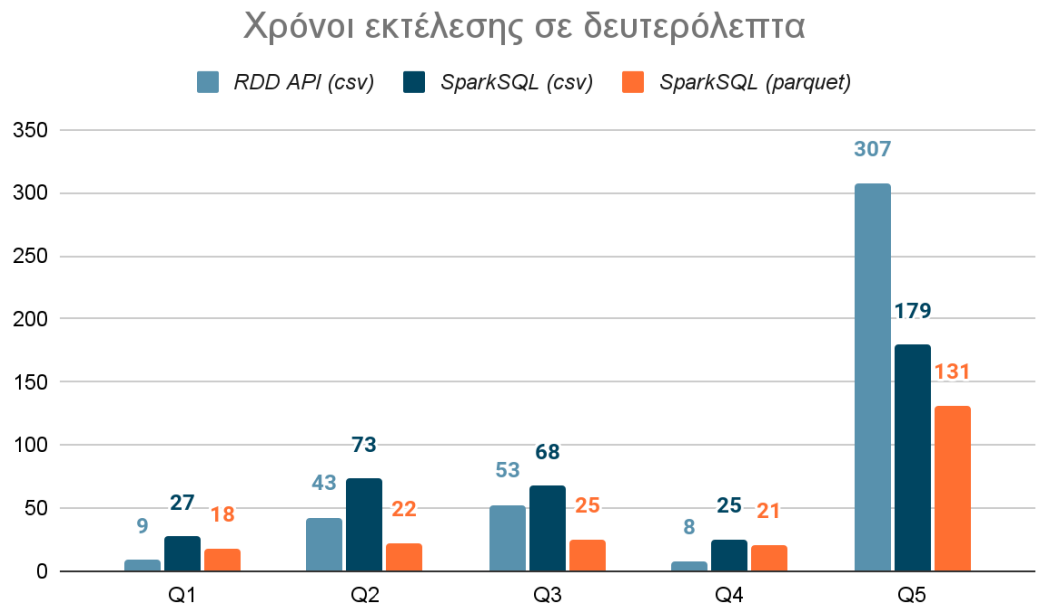
Ο κώδικας για την μετατροπή των αρχείων από csv σε parquet βρίσκεται στο αρχείο *code/exercise_1/save_as_parquet.py*.

Ζητούμενο 3

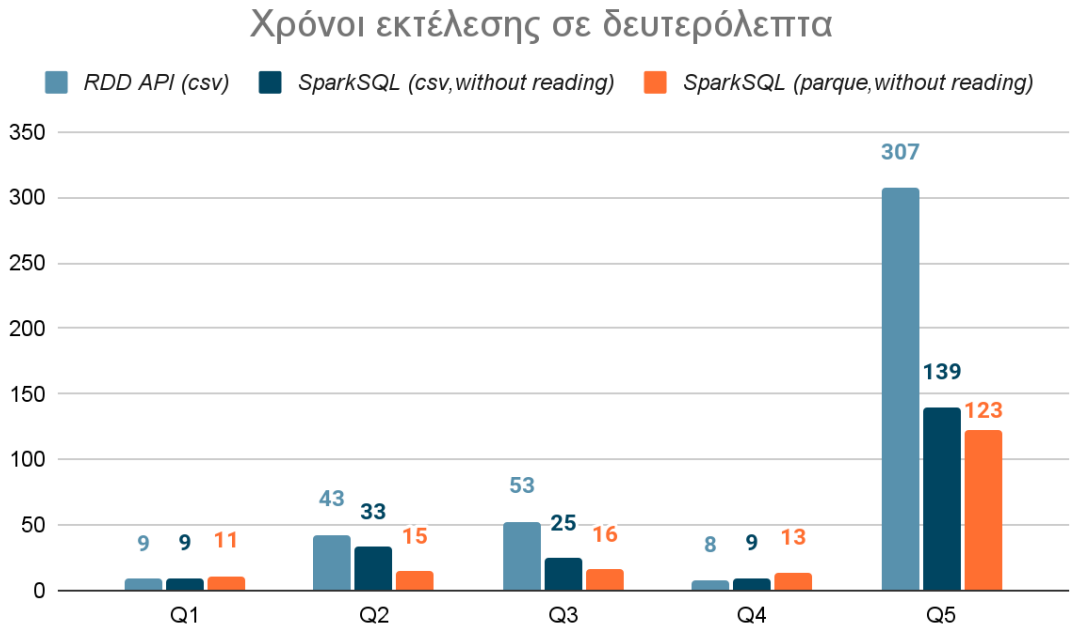
Οι υλοποιήσεις των ζητούμενων ερωτημάτων βρίσκονται εντός του φακέλου *code/exercise_1*.

Ζητούμενο 4

Οι χρόνοι εκτέλεσης των ερωτημάτων για την περίπτωση του SparkSQL στο παρακάτω ραβδόγραμμα περιλαμβάνουν και το “διάβασμα” του αρχείου. Στην περίπτωση που η είσοδος είναι csv αρχείο, χρησιμοποιήσαμε infer schema.



Οι χρόνοι εκτέλεσης των ερωτημάτων για την περίπτωση του SparkSQL στο παρακάτω ραβδόγραμμα δεν περιλαμβάνουν το “διάβασμα” του αρχείου.



Σχολιασμός queries

Query 1

- Παρατηρούμε πως το processing των queries απαιτεί περίπου τον ίδιο χρόνο και στις τρεις περιπτώσεις. Ωστόσο, όταν συνυπολογίσουμε και τον χρόνο για το διάβασμα της εισόδου, η υλοποίηση σε SparkSQL υστερεί σημαντικά όταν η είσοδος δίνεται στη μορφή csv αρχείου. Συγκεκριμένα το διάβασμα της εισόδου διαρκεί διπλάσιο χρόνο σε σχέση με το χρόνο που διαρκεί η επεξεργασία του ερωτήματος. Σημαντικός παράγοντας που οδηγεί σε τόσο μεγάλη καθυστέρηση είναι το infer schema.
- Παρατηρούμε πως ακόμη και σε μια σχετικά μικρή είσοδο (17 MB), το columnar format που χρησιμοποιεί το parquet οδηγεί σε σημαντικά ταχύτερο I/O (7 sec για parquet, 18 sec για csv)

Query 2

- Παρατηρούμε πως για μεγάλα inputs (677 MB) η διαφορά στο I/O performance μεταξύ του csv και του parquet είναι πλέον καθοριστική (7 sec για parquet, 40 sec για csv).
- Οι στατιστικές πληροφορίες που διατηρεί το parquet για κάθε block δεδομένων το καθιστούν σαφώς γρηγορότερο από το csv στο συγκεκριμένο SQL query, το οποίο περιέχει σχετικούς aggregators (COUNT).

Query 3

- Η μεγάλη διαφορά στον χρόνο που απαιτείται για το processing (25 sec για SparkSQL (csv), 53 sec για RDD API) μπορεί να εξηγηθεί αν αναλογιστούμε πως το query που γράψαμε στην υλοποίηση με SparkSQL βελτιστοποιείται από τον Catalyst Optimizer του Spark, σε αντίθεση με την MapReduce ακολουθία που εκτελείται από την υλοποίηση μας σε RDD API, η οποία δεν υφίσταται κάποια βελτιστοποίηση.

Query 4

- Η χρήση user-defined functions στις υλοποιησεις με SparkSQL επηρεάζει αρνητικά το χρόνο εκτέλεσης, αφού συνεπάγεται μεταφορά δεδομένων μεταξύ της Python και του JVM στο οποίο εκτελείται το Spark (σελίδα 115 στο [2]).

Query 5

- Στο συγκεκριμένο query, η μεγάλη διαφορά μεταξύ της υλοποίησης μέσω RDD API και των υλοποιήσεων μέσω SparkSQL μας επιτρέπει να αντιληφθούμε την καταλυτική επίδραση του Catalyst optimizer της SparkSQL.
- Και σε αυτό το query είναι φανερή η διαφορά μεταξύ cvs και parquet αναφορικά με το I/O performance (8 sec για parquet, 40 sec για csv).

Parquet format

Παρατηρούμε πως η χρήση του parquet οδηγεί σε ταχύτερη εκτέλεση των ερωτημάτων, σε σύγκριση με τις περιπτώσεις που διαβάζουμε την είσοδο κατευθείαν από τα csv αρχεία. Η βελτίωση αυτή οφείλεται:

i. στο βελτιωμένο I/O που παρέχει το columnar format
ii. στο ότι δεν είναι αναγκαίο να συμπεράνουμε το schema μέσω του infer schema το οποίο για μεγάλα αρχεία εισόδου γίνεται υπολογιστικά ακριβό (σελίδα 51 του [2]).
iii. ειδικά για τα queries που απαιτούν υπολογισμό στατιστικών πληροφοριών που αφορούν το dataset (όπως για παράδειγμα max/min κα count), παρατηρούμε πως το performance του parquet υπερισχύει σημαντικά, εξαιτίας των στατιστικών δεδομένων που προ-υπολογίζει για κάθε block των δεδομένων

Στην περίπτωση του parquet δε χρησιμοποιείται το infer schema, διότι το schema των δεδομένων έχει αποθηκευτεί στα metadata του αρχείου parquet κατά τη δημιουργία του, όπως αναφέρεται στη σελίδα 86 του [2]. Στη συγκεκριμένη εφαρμογή θα μπορούσαμε να αποφύγουμε εντελώς τη χρήση του infer schema, δηλώνοντας εκ των προτέρων το schema των csv αρχείων που διαβάζουμε.

Ψευδοκώδικας Map-Reduce

Οι ψευδοκώδικες Map-Reduce για τις υλοποιησεις με RDD API βρίσκονται εντός του φακέλου *map_reduce_pseudocode/*.

Μέρος 2ο

Ζητούμενο 1 - Ζητούμενο 2

Οι ζητούμενες υλοποιήσεις βρίσκονται εντός του αρχείου *code/exercise_2/joins_implementation.py*.

Σημειώσεις:

- Οι συναρτήσεις που υλοποιούν το broadcast join και το repartition join δέχονται ως παραμέτρους RDDs τα οποία είναι της μορφής (key, value) και το join γίνεται με βάση το key. Συνεπώς, πριν την κλήση των συναρτήσεων απαιτείται preprocessing των RDDs μέσω μιας map(), ώστε τα columns πάνω στα οποία θα γίνει το inner join να τοποθετηθούν στο key και τα υπόλοιπα columns να τοποθετηθούν στο value.
- Η συνάρτηση που υλοποιεί το broadcast join δέχεται ως πρώτο όρισμα τον “μικρό” πίνακα και ως δεύτερο όρισμα τον “μεγάλο” πίνακα.
- Στη συνάρτηση που υλοποιεί το repartition join αποφεύγουμε τη χρήση της μεθόδου groupByKey(), διότι δημιουργεί προβλήματα με τη μνήμη (out-of-memory) στην περίπτωση που κάποιο key έχει περισσότερα values από όσα χωράνε στη μνήμη ενος executor. Αντιθέτως, η μέθοδος reduceByKey() δεν προκαλεί αυτό το πρόβλημα. [4]

Ζητούμενο 3

Απομονώνουμε τις 100 πρώτες γραμμές του αρχείου *movie_genres.csv* μέσω της εντολής

```
head -100 movie_genres.csv > movie_genres_100.csv
```

Έπειτα, χρησιμοποιούμε το αρχείο *movie_genres_100.csv* και το αρχείο *ratings.csv* για να συγκρίνουμε τις επιδόσεις των δύο υλοποιήσεων μας. Λαμβάνουμε τις παρακάτω μετρήσεις:

join	**time**
Broadcast Join	48,807013273239136 seconds
Repartition Join	610,1717803478241 seconds

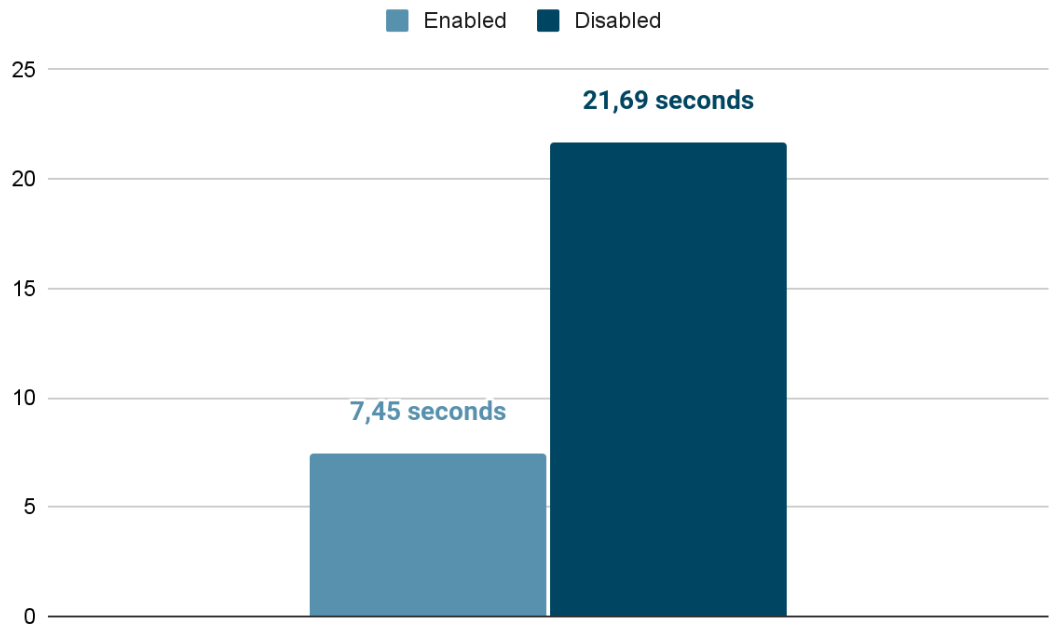
Συμπεραίνουμε πως το broadcast join πραγματοποιεί τη συγκεκριμένη συνένωση 12,7 φορές ταχύτερα από το repartition join. Το αποτέλεσμα αυτό είναι αναμενόμενο με βάση τα όσα αναφέρονται στο [3].

Αναλυτικότερα, το broadcast join μεταφέρει μόνο έναν πίνακα σχετικά μικρού μεγέθους μεταξύ των μηχανημάτων. Συνεπώς, μειώνει σε πολύ σημαντικό βαθμό τη μεταφορά δεδομένων μέσω δικτύου, η οποία συχνά αποτελεί bottleneck σε ό,τι αφορά την ολοκλήρωση ενος distributed join. Για την υλοποίηση του broadcast join αξιοποιούμε μόνο τη μέθοδο map. Ως εκ τούτου, το broadcast join περιέχει μόνο narrow transformations, κάτι που σημαίνει πως κάθε executor δε χρειάζεται πληροφορίες από άλλα partitions για να ολοκληρώσει tasks που αφορούν κάποιο partition που είναι αποθηκευμένο τοπικά. Αντιθέτως, το repartition join χρησιμοποιεί τη μέθοδο reduceByKey(), η οποία συνιστά wide transformation. Αυτό σημαίνει πως στη repartition join έχουμε έντονη ανταλλαγή δεδομένων μεταξύ των executors, και κατ' επέκταση μεγάλο χρόνο ολοκλήρωσης λόγω καθυστερήσεων του δικτύου.

Ζητούμενο 4

Θέτουμε τη ρύθμιση *spark.sql.autoBroadcastJoinThreshold* ίση με -1, ώστε να απενεργοποιήσουμε την μετατροπή sort-merge join σε broadcast join.

Οι χρόνοι εκτέλεσης των ερωτημάτων για την περίπτωση του SparkSQL στο παρακάτω ραβδόγραμμα:



Επεξήγηση

- Αν ένας από τους δύο πίνακες που συμμετέχουν στο join είναι αρκετά μικρός ώστε να χωράει στη μνήμη των executors και η χρήση broadcast joins από τον optimizer επιτρέπεται, τότε το join εκτελείται με Broadcast Join.
- Αν, όμως, που η χρήση broadcast joins δεν επιτρέπεται, ο optimizer αναγκάζεται να αναζητήσει τον αμέσως καλύτερο τρόπο για να εκτελεστεί το join, ο οποίος είναι το Sort-Merge Join, που υστερεί από άποψη performance συγκριτικά με το Broadcast Join. Στην περίπτωση που τα κλειδιά δεν ήταν sortable, ο optimizer θα κατέληγε στο Shuffle Hash Join.

Παρακάτω παρατίθενται τα πλάνα εκτέλεσης για τις δύο περιπτώσεις.

Physical plan όταν ο query optimizer είναι ενεργοποιημένος

```
*(3) BroadcastHashJoin [_c0#4], [cast(_c1#1 as int)], Inner, BuildLeft

:- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as bigint)))

:  +- *(2) Filter isnotnull(_c0#4)

:     +- *(2) GlobalLimit 100

:     +- Exchange SinglePartition

:           +- *(1) LocalLimit 100

:           +- *(1) FileScan parquet [_c0#4,_c1#5,_c2#6,_c3#7] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://master:9000/files/ratings.parquet], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<_c0:int,_c1:int,_c2:double,_c3:int>

+- *(3) Project [_c0#0, _c1#1]

   +- *(3) Filter isnotnull(_c1#1)

      +- *(3) FileScan parquet [_c0#0,_c1#1] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://master:9000/files/movie_genres.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(_c1)], ReadSchema: struct<_c0:int,_c1:string>
```

Physical plan όταν ο query optimizer είναι απενεργοποιημένος

```
*(6) SortMergeJoin [_c0#4], [cast(_c1#1 as int)], Inner

:- *(3) Sort [_c0#4 ASC NULLS FIRST], false, 0

:  +- Exchange hashpartitioning(_c0#4, 200)

:     +- *(2) Filter isnotnull(_c0#4)

:     +- *(2) GlobalLimit 100

:           +- Exchange SinglePartition

:           +- *(1) LocalLimit 100

:                 +-   *(1)   FileScan   parquet   [_c0#4,_c1#5,_c2#6,_c3#7]
Batched:          true,          Format:          Parquet,          Location:
InMemoryFileIndex[hdfs://master:9000/files/ratings.parquet],
PartitionFilters:       [],       PushedFilters:       [],       ReadSchema:
struct<_c0:int,_c1:int,_c2:double,_c3:int>

+- *(5) Sort [cast(_c1#1 as int) ASC NULLS FIRST], false, 0

   +- Exchange hashpartitioning(cast(_c1#1 as int), 200)

      +- *(4) Project [_c0#0, _c1#1]

      +- *(4) Filter isnotnull(_c1#1)

            +- *(4) FileScan parquet [_c0#0,_c1#1] Batched: true, Format:
Parquet,                                                          Location:
InMemoryFileIndex[hdfs://master:9000/files/movie_genres.parquet],
PartitionFilters:    [],    PushedFilters:    [IsNotNull(_c1)],    ReadSchema:
struct<_c0:int,_c1:string>
```

References

[1] A Comparison of Join Algorithms for Log Processing in MapReduce", Blanas et al , in Sigmod 2010

[2] Learning Spark: Lightning-Fast Big Data Analysis

[3] High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark

[4]https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html