ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Προχωρημένα Θέματα Βάσεων Δεδομένων

ΑΝΑΦΟΡΑ

Ονοματεπώνυμο: Παναγιώτης Σταματόπουλος

Α.Μ.: 03120096

Ονοματεπώνυμο: Μαριάνθη Αφροδίτη Χλαπάνη

Α.Μ.: 03120889

Σχόλια

Γενικά

Στο GitHub Repo μας έχουμε 2 Notebooks, ένα για εκτέλεση σε AWS και ένα για τοπική εκτέλεση. Οδηγίες για την εγκατάσταση και εκτέλεση του Notebook βρίσκονται στο README αρχείο στο Repo μας.

Τα σχόλια και τα αποτελέσματά μας βασίζονται σε εκτελέσεις κυρίως σε Local περιβάλλον.

Query 1

Υλοποιήσαμε το Query 1 χρησιμοποιώντας τα DataFrame και RDD APIs και τα δύο με 4 Spark executors.

Η διαφορά στην απόδοση μεταξύ των δύο APIs δεν είναι τόσο εμφανής όσο αναμέναμε. Τα DataFrames αξιοποιούν το optimization και προσφέρουν γενικά μεγαλύτερη ταχύτητα, αν και στη συγκεκριμένη περίπτωση η διαφορά είναι μικρή. Αντιθέτως, τα RDD προσφέρουν μεγαλύτερη ευελιξία, αφού είναι low level, στην επεξεργασία των δεδομένων.

Πιθανόν το μέγεθος των δεδομένων να μην είναι αρκετά μεγάλο ώστε να αρχίσει να φαίνεται μια ουσιαστική διαφορά.

Επίσης, παρατηρήσαμε ότι ο κώδικας ήταν πιο εύκολος και συνοπτικός στην περίπτωση των DataFrames. Για τα RDD, αντιθέτως, αντιμετωπίσαμε και

πρόβλημα με το parsing των δεδομένων μας, γεγονός που αναφέρουμε και στο Jupyter Notebook της εργασίας.

(Συγκεκριμένα αναφέρουμε:

«Δυστυχώς στα δεδομένα μας έχουμε μερικές περιπτώσεις όπου υπάρχει υποδιαστολή εντός quote marks (""), δημιουργώντας έτσι θέμα στο parse. Για αυτό θα χρησιμοποιήσουμε μια βιβλιοθήκη της Python για την ανάγνωση των csv αρχείων και θα την περάσουμε μέσω mapping σε κάθε δεδομένο.»)

Να σημειωθεί ότι η διαφορά στην απόδοση μεταξύ των δύο APIs διαφέρει για τα 2 διαφορετικά περιβάλλοντα εκτέλεσης:

- AWS: Η διαφορά στον χρόνο εκτέλεσης μεταξύ των 2 APIs είναι πολύ μικρή (τάξη του ~ 1 sec) και μάλιστα φαίνεται το RDD API να τερματίζει γρηγορότερα.

- Local: Η διαφορά στο χρόνο εκτέλεσης είναι πραγματικά εμφανής (περίπου 10 φορές γρηγορότερο το DataFrame API)

Μια πιθανή εξήγηση είναι ότι η υπολογιστική ισχύς της AWS υπερνικά το optimization που προσφέρει το DataFrame API και ίσως χρειάζεται πολύ μεγαλύτερο μέγεθος δεδομένων για να φανεί κάποια ουσιαστική διαφορά.

Query 2

α) Υλοποιήθηκε το Query 2 χρησιμοποιώντας τα DataFrame και SQL APIs. Οι τελικοί χρόνοι ήταν οι εξής:

Dataframe: 5.75 seconds

SQL API: 3.83 seconds

Παρατηρούμε ότι οι δύο χρόνοι εκτέλεσης δεν παρουσιάζουν μεγάλες αποκλίσεις. Παραδόξως, οι χρόνοι εκτέλεσης για το SQL API είναι συνήθως σταθεροί για όσες φορές τρέξουμε τον κώδικα, ενώ στο DataFrame API παρατηρούμε μεγάλη απόκλιση μεταξύ των τιμών για πολλαπλές εκτελέσεις του κώδικα.

β) Γράψαμε κώδικα Spark που μετατρέπει το κυρίως data set σε parquet2 file format και αποθηκεύει ένα μοναδικό .parquet αρχείο στο S3 bucket της ομάδας σας. Επιλέξαμε τα DataFrame και εισάγαμε τα δεδομένα σαν .csv και σαν .parquet.

Χρησιμοποιώντας τα parquet δεδομένα, παρατηρούμε σταθερά μια αρκετά μεγάλη βελτίωση στην ταχύτητα. Συγκεκριμένα: 4.21 seconds για τα parquet δεδομένα.

Query 3

Υλοποιήσαμε το Query 3 χρησιμοποιώντας DataFrame.

Χρησιμοποιώντας τις μεθόδους hint & explain βρήκαμε ποιες στρατηγικές join χρησιμοποιεί ο catalyst optimizer. Συγκεκριμένα, χρησιμοποιεί:

- Broadcast Hash Join,
- Range Join και
- Hash Partitioning.

Έπειτα, πειραματιστήκαμε αναγκάζοντας το Spark να χρησιμοποιήσει διαφορετικές στρατηγικές (BROADCAST, MERGE, SHUFFLE_HASH, SHUFFLE_REPLICATE_NL) και παρατηρήσαμε ότι:

- **Προεπιλογή (45,40 δευτερόλεπτα)**:

  Η προεπιλεγμένη επιλογή του Catalyst Optimizer είχε ως αποτέλεσμα τον πιο αργό χρόνο εκτέλεσης.

  Αυτό υποδηλώνει ότι ενώ η απόφαση του βελτιστοποιητή μπορεί να ήταν λογική δεδομένων των χαρακτηριστικών του συνόλου δεδομένων, δεν ήταν βέλτιστη σε σύγκριση με τις άλλες στρατηγικές που δοκιμάστηκαν.

- **Broadcast (25.24 δευτερόλεπτα)**:

  Η ταχύτερη στρατηγική, ξεπερνώντας σημαντικά όλες τις άλλες. Οι συνδέσεις εκπομπής λειτουργούν καλύτερα όταν ένα table είναι αρκετά μικρό ώστε να χωράει στη μνήμη και προφανώς τα δεδομένα μας διαθέτουν αυτά τα χαρακτηριστικά.

- **Merge (37.58 δευτερόλεπτα)**: Η δεύτερη ταχύτερη στρατηγική.

Η Merge συνδέει excel με μεγάλα, ταξινομημένα σύνολα δεδομένων, αλλά μπορεί να είναι πιο αργή εάν απαιτείται ταξινόμηση ή προετοιμασία δεδομένων κατά την εκτέλεση.

Αυτό το αποτέλεσμα υποδηλώνει κάποια επιβάρυνση ταξινόμησης, αλλά ήταν ακόμα πολύ πιο γρήγορη από τις στρατηγικές Base, Shuffle Hash και Shuffle Replicate.

- **Shuffle Hash (44.58 δευτερόλεπτα)**:

  Ελαφρώς ταχύτερη από την προεπιλεγμένη στρατηγική, αλλά και πάλι αργή. Η Shuffle hash κάνει επαναδιαμερισμό δεδομένων (repartitioning data), με σημαντικό κόστος τυχαίας αναπαραγωγής (shuffle costs). Αυτή η στρατηγική δεν ήταν η κατάλληλη.

- **Shuffle Replicate (40.72 δευτερόλεπτα)**:

  Το Shuffle Replicate περιλαμβάνει την αναπαραγωγή ενός συνόλου δεδομένων σε όλους τους κόμβους, το οποίο είναι υπολογιστικά ακριβό και λιγότερο αποτελεσματικό για μεγάλα σύνολα δεδομένων.

Από τις διαθέσιμες στρατηγικές join του Spark, λοιπόν, η καταλληλότερη είναι η Broadcast Join με σημαντικά ταχύτερο χρόνο εκτέλεσης (25,24 δευτερόλεπτα), γιατί αποφεύγει αποτελεσματικά τις λειτουργίες τυχαίας αναπαραγωγής και αξιοποιεί τις δυνατότητες μνήμης του Spark, καθιστώντας το ιδανικό όταν ένα τραπέζι είναι αρκετά μικρό για να μεταδοθεί.

Query 4

Υλοποιήσαμε το Query 4 χρησιμοποιώντας DataFrame.

Εφαρμόσαμε κλιμάκωση στο σύνολο των υπολογιστικών πόρων ως εξής (σε 2 executors):

- **1 core/2 GB memory (124,71 δευτερόλεπτα):**

  Ο χρόνος εκτέλεσης είναι μέτριος, αναμενόμενο λόγω των περιορισμένων υπολογιστικών πόρων (μονό πυρήνα και ελάχιστη μνήμη).

Με έναν μόνο πυρήνα, το Spark επεξεργάζεται τις εργασίες διαδοχικά και η περιορισμένη μνήμη (2 GB) μπορεί να έχει οδηγήσει σε συχνότερη garbage collection ή disk spilling, εάν το σύνολο δεδομένων ήταν μεγάλο.

- **2 cores/4GB memory (118,83 δευτερόλεπτα)**:

  Ο χρόνος εκτέλεσης βελτιώθηκε ελαφρώς σε σύγκριση με τη διαμόρφωση 1 πυρήνα / 2 GB.

  Ο πρόσθετος πυρήνας επιτρέπει την παράλληλη επεξεργασία εργασιών και η αυξημένη μνήμη (4 GB) μειώνει την πιθανότητα επιβάρυνσης που σχετίζεται με τη μνήμη (π.χ. disk spilling ή garbage collection).

  Η βελτίωση είναι μέτρια, γεγονός που μπορεί να υποδεικνύει ότι ο φόρτος εργασίας δεν είναι ιδιαίτερα παραλληλιζόμενος ή ότι άλλα σημεία συμφόρησης (π.χ. λειτουργίες εισόδου/εξόδου ή τυχαίας αναπαραγωγής) περιορίζουν τα κέρδη απόδοσης.

- **4 cores/8GB memory (156,95 δευτερόλεπτα)**:

  Παραδόξως, ο χρόνος εκτέλεσης αυξήθηκε σημαντικά σε σύγκριση με τις άλλες διαμορφώσεις.

  Ίσως με περισσότερους πυρήνες, το Spark χρειάζεται χρόνο για το συντονισμό εργασιών και τη διαχείριση πόρων. Εάν τα δεδομένα δεν είναι ομοιόμορφα κατανεμημένα, η προσθήκη περισσότερων πυρήνων μπορεί να οδηγήσει σε αναποτελεσματικότητα όπου ορισμένες εργασίες τελειώνουν γρήγορα ενώ άλλες χρειάζονται περισσότερο χρόνο.

Η διαμόρφωση με 2 πυρήνες και μνήμη 4 GB παρείχε την καλύτερη απόδοση.

Query 5

Υλοποιήσαμε το Query 5 χρησιμοποιώντας το DataFrame. Χρησιμοποιώντας συνολικούς πόρους 8 cores και 16GB μνήμης, είχαμε τα εξής configurations:

- **2 executors × 4 cores/8GB memory (26,78 δευτερόλεπτα)**:

  Αυτή η διαμόρφωση είχε τους λιγότερους executors αλλά τους περισσότερους cores και μνήμη ανά executor.

Παρά το γεγονός ότι υπήρχαν περισσότεροι πόροι ανά executor, ο συνολικός χρόνος εκτέλεσης ήταν ο πιο αργός. Αυτό θα μπορούσε να οφείλεται σε χαμηλότερο παραλληλισμό και πιθανή αναποτελεσματικότητα στην κατανομή εργασιών.

- **4 executors × 2 cores/4GB memory (23,51 δευτερόλεπτα)**:

  Αυτή η ρύθμιση πέτυχε τον ταχύτερο χρόνο εκτέλεσης.
  Ο αυξημένος αριθμός executors πιθανώς βελτίωσε την κατανομή εργασιών και μείωσε τα γενικά έξοδα, οδηγώντας σε καλύτερη παράλληλη επεξεργασία.
  Η εκχώρηση μνήμης (4 GB ανά executor) ήταν πιθανότατα επαρκής για κάθε εργασία, αποτρέποντας το υπερβολικό garbage collection ή memory constraints.

- **8 executors × 1 core/2 GB memory (23,93 δευτερόλεπτα)**:

  Αυτή η διαμόρφωση είχε το υψηλότερο επίπεδο παραλληλισμού (πολλοί executors) αλλά τη λιγότερη μνήμη ανά executor.
  Ενώ είχε καλύτερη απόδοση από την πρώτη ρύθμιση (με 2 executors), ήταν ελαφρώς πιο αργή από τη ρύθμιση 4 executors.
  Ο υψηλότερος αριθμός executors μπορεί να έχει οδηγήσει σε αυξημένη προσπάθεια συντονισμού και επικοινωνίας, επηρεάζοντας το συνολικό performance.

Github link

https://github.com/aphrochl/Advanced-DB-2025

Installation

https://github.com/aphrochl/Advanced-DB-2025/blob/main/README.md