Προχωρημένα Θέματα Βάσεων Δεδομένων: Εξαμηνιαία Εργασία

Δήμητρα Νεστορή
03121840

Παρασκευή Πυρπιρή
03121154

Query 1

Για το συγκεκριμένο query χρησιμοποιήσαμε 4 executors (1 core, 2GB memory) και το υλοποιήσαμε με τρεις διαφορετικούς τρόπους. Γνωρίζουμε ότι είναι αποδοτικότερο με τη χρήση Dataframe API αφού επωφελείται από τον Catalyst Optimizer ώστε να βελτιστοποιηθούν αυτόματα οι εκτελούμενες εργασίες. Ωστόσο, χάνει επίδοση με την προσθήκη UDF λόγω της απαίτησης για σειριοποίηση των δεδομένων για τη μεταφορά από το JVM (όπου τρέχει η spark) στο python process και έπειτα της αποσειριοποίησης για την αντίθετη διαδικασία. Στην πράξη δεν υπήρχε μεγάλη διαφορά στους δυο χρόνους γιατί το μειονέκτημα του UDF δεν ήταν αρκετά μεγάλο, ενώ το όφελος του Catalyst ήταν ιδιαίτερα σημαντικό. Η υλοποίηση του με RDD API ήταν η πιο αργή, καθώς είναι χαμηλότερου επιπέδου από το Dataframe και δεν χρησιμοποιεί τον Catalyst Optimizer για βελτιστοποίηση.

Query 2

Το 2ο query υλοποιήθηκε με Dataframe και SQL APIs με σχεδόν ίδιο χρόνο εκτέλεσης και στις 2 περιπτώσεις. Αυτό συμβαίνει επειδή και τα δύο API χρησιμοποιούν τον Catalyst Optimizer ο οποίος και στις δύο περιπτώσεις δημιουργεί ένα λογικό πλάνο και στη συνέχεια ένα βελτιστοποιημένο φυσικό πλάνο που έχει τον ίδιο αριθμό εργασιών για τους 4 executors.

Query 3

Στο Query 3,για την ταξινόμηση με φθίνουσα σειρά συχνότητας των μεθόδων διάπραξης εγκλημάτων και των αντίστοιχων περιγραφών,.ο Catalyst Optimizer αναλύει το query, εφαρμόζει λογικούς και φυσικούς μετασχηματισμούς και επιλέγει τη φθηνότερη στρατηγική εκτέλεσης με βάση τα στατιστικά μεγέθους των σχέσεων. Το dataset των MO Codes είναι μικρό, οπότε η καταλληλότερη στρατηγική που επέλεξε ο catalyst ήταν το Broadcast Hash Join, όπου η μικρή σχέση αναπαράγεται σε όλους τους executors και το join εκτελείται τοπικά, αποφεύγοντας shuffle και επαναδιαμερισμό. **,** χρόνος εκτέλεσης**:BROADCAST join time: 14.85282301902771**

Γενικά παρατηρώ ότι ο catalyst επιλέγει να χρησιμοποιήσει τις μεθοδους που οριζω εγω καθε φορα με hint.

Με **+- BroadcastHashJoin [MOCODE#81], [MOCODE#94], Inner, BuildRight, false**

BROADCAST join time: 12.5029 sec

Ισως βγαίνει πιο γρήγορο επειδή εγω ορίζω ήδη ποια σχέση θα γίνει broadcast και δεν χρειάζεται επιπλέον χρόνος για τον Catalyst να το ελέγχει τα στατιστικά των σχέσεων Αποδείχθηκε η καλύτερη στρατητική!

+- SortMergeJoin [MOCODE#81], [MOCODE#94], Inner

MERGE join time: 18.3418 sec

Το **Sort-Merge Join** απαιτεί:

shuffle και repartitioning και για τις δύο σχέσεις,ταξινόμηση με βάση το join key πριν τη συγχώνευση.
Αποτελεί μια σταθερή μέθοδο join (δηλαδή διατηρεί τη σχετική σειρά των πλειάδων με ίδιο join key), ωστόσο για μεγάλα datasets μπορεί να είναι ιδιαίτερα κοστοβόρα λόγω των επιπλέον σταδίων shuffle και sorting.

+- ShuffledHashJoin [MOCODE#68], [MOCODE#81], Inner, BuildRight

SHUFFLE_HASH join time: 15.5652 sec

Το **Shuffled Hash Join** πραγματοποιεί shuffle και για τις δύο σχέσεις και κατασκευάζει hash table σε κάθε partition. Αποδίδει ικανοποιητικά όταν τουλάχιστον ένα από τα datasets χωράει στη μνήμη μετά το shuffling, ωστόσο εξακολουθεί να έχει κόστος λόγω της μετακίνσης των δεδομένων **(**shuffle) μεταξύ των executors μέσω δικτύου.

+- CartesianProduct (MOCODE#68 = MOCODE#81)SHUFFLE_REPLICATE_NL join time: 19.0149 sec. Η συγκεκριμένη στρατηγική παρουσίασε τον **μεγαλύτερο χρόνο εκτέλεσης**, καθώς απαιτεί τη δημιουργία καρτεσιανού γινομένου των δύο σχέσεων.

Το **Shuffle Replicate Nested Loop Join**:replicate τη μικρή σχέση,shuffle την μεγάλη σχέση και εκτελεί nested loop join σε κάθε κόμβο.(Γενικά,είναι κατάλληλο μόνο όταν έχουμε πολύ μικρό dataset ή για **non-equi joins**.)

Η υλοποίηση με RDDs παρουσιάζει μεγαλύτερο χρόνο εκτέλεσης (30.16 sec) σε σύγκριση με τα DataFrames, κυρίως επειδή στα RDDs οι μετασχηματισμοί και οι πράξεις εκτελούνται σε χαμηλότερο επίπεδο αφαίρεσης,(δουλεύω με map και filter σε tuples),ενώ με τα DataFrames με δομημένα δεδομένα-schema, εντολές τύπου join,groupby κλπ) χωρίς την υποστήριξη αυτόματης βελτιστοποίησης από τον Catalyst Optimizer. Επιπλέον, τα δεδομένα στα RDDs αναπαρίστανται ως Java/Scala objects, τα οποία σειριοποιούνται και μεταφέρονται μεταξύ των κόμβων, αυξάνοντας το υπολογιστικό κόστος.

Αντίθετα, τα DataFrames αξιοποιούν τον Catalyst Optimizer και τη μηχανή Tungsten, η οποία επιτρέπει αποδοτικότερη διαχείριση μνήμης μέσω off-heap αποθήκευσης, εκτέλεση πράξεων απευθείας στη μνήμη και μείωση του κόστους δημιουργίας και διαχείρισης αντικειμένων, καθώς και της μεταφοράς δεδομένων.

Η διαφορά στον χρόνο εκτέλεσης δεν είναι ιδιαίτερα μεγάλη, γεγονός που αποδίδεται στο σχετικά μικρό μέγεθος των δεδομένων και στο απλό workload, χωρίς ιδιαίτερα απαιτητικούς μετασχηματισμούς που θα φαινόταν πιο έντονα η βελτιστοποίηση στην απόδοση με τον Catalyst.

Σημείωση για την τρόπο μέτρησης των χρόνων εκτέλεσης:

Ξεκίνησαμε το χρονόμετρο αμέσως πριν διαβάζουμε τα αρχεία και θέσαμε το τέλος αμέσως μετά την count (), ως action function. Επειδή το spark είναι lazy ξεκινάει να μετράει χρόνο και να εκτελεί τους μετασχηματισμούς από τότε που βλέπει μια action function που τους χρειάζεται, όπως count,show,corr.Επέλεξα την count ώς την πιο αντιπροσωπευτική για όλο το πλάνο, καθώς δεν θα ήταν αντιπροσωπευτικό του να τυπώσουμε μόνο τα 20 πρώτα(το οποιοι και αυτό παίρνει χρόνο αλλά δεν το θέτω στην σύγκριση.)

Επίσης, κάθε φορά εκτελούσαμε μια μέθοδο την φορά, έχοντας απόι πριν κανει stop περιμενονταας καποια ωρα και μετα restart το notebook,ειδάλως έμεναν δεδομένα αποθηκευμένα στην cache και οι χρόνοι δεν ήταν καθόλου αντιπροσωπευτικόι(παροτι προσπαθησαμε με clearcache εντολη και τρέχοντας πολλές φορές το κελι για να παρουμε καποιο μεσο ορο, θεωρησαμε τον τελικο τροπο πιο πρακτικο).

Query 4

Στο παρόν query, αναλύσαμε τις στρατηγικές που επιλέγει ο catalyst optimizer του spark με τη βοήθεια της μεθόδου explain. Πιο συγκεκριμένα, παρατηρήσαμε ότι χρησιμοποιεί Broadcast Nested Loop Join (BuildRight, Cross), δηλαδή έγινε broadcast το dataframe των αστυνομικών τμημάτων. Αυτή η επιλογή είναι η ιδανική καθώς ο πίνακας των τμημάτων ήταν αισθητά μικρότερος από αυτόν των εγκλημάτων και για την αποφυγή του shuffling ο κάθε executor λαμβάνει το δικό του μισό από τον πίνακα των εγκλημάτων και ολόκληρο αντιγραφο των stations για να εκτελέσει τον υπολογισμό του καρτεσιανού γινομένου. Με αυτόν τον τρόπο η διαδικασία βελτιστοποιείται καθώς δεν μετακινούνται δεδομένα μεταξύ των executors.
Επιπλέον, καθώς κλιμακώνονται οι υπολογιστικοί πόροι, δηλαδή αυξάνονται οι πυρήνες και η μνήμη βελτιώνεται η επίδοση. Επομένως, η τρίτη επιλογή είναι η καλύτερη (4 cores, 8GB memory). Αυτό συμβαίνει επειδή το query είναι εξαιρετικά υπολογιστικά εντατικό λόγω των

εκατομμύρια υπολογισμών αποστάσεων. Με την αύξηση των πόρων κατανέμονται οι υπολογισμοί και εκτελούνται παράλληλα εξοικονομώντας χρόνο.

Query5

Στο configuration με **2 executors × 4 cores × 8 GB**, κάθε executor διαθέτει περισσότερους cores και αρκετή μνήμη, κάτι που επιτρέπει την αποδοτική εκτέλεση των tasks και τις απαιτητικές πράξεις όπως τα joins,χωρίς ιδιαίτερο overhead. ωστόσο έχει μικρό parallelism και χρόνο εκτέλεσης, **79.59 sec seconds.**

Στο configuration με **4 executors × 2 cores × 4 GB**, ο parallelism κατανέμεται σε περισσότερους executors, όμως οι λιγότεροι cores και η μικρότερη μνήμη ανά executor αυξάνουν το overhead για την διαχείριση. Έτσι, παρότι υπάρχει θεωρητικά καλύτερη κατανομή, ο συνολικός χρόνος εκτέλεσης είναι μεγαλύτερος απο πριν, **77.83 seconds**.

Στο configuration με **8 executors × 1 core × 2 GB,** το parallelism είναι το υψηλότερο, καθώς μπορούν να εκτελούνται πολλά tasks ταυτόχρονα, αλλά η πολύ μικρή μνήμη ανά executor και ο μεγάλος αριθμός JVM αυξάνουν το overhead, και έτσι έχουμε ο χρόνος εκτέλεσης είναι ο μεγαλύτερος. **84.02 seconds**

Τελικά, το configuration **4 executors × 2 cores × 4 GB** έχει τον καλύτερο διαθέσιμο parallelism με επαρκή μνήμη και χαμηλό overhead, αποτελώντας την πιο αποτελεσματική επιλογή για το task.

Οι μέθοδοι που χρησιμοποιεί ο catalyst είναι(παίρνω για ένα απο τα configurations, τα ίδια είναι κάθε φορά)

1.Για τον υπολογισμό των εγκλημάτων ανά κοινότητα(crime_per_comm_year)ο catalyst με συναρτήσεις της Sedonna χρησιμοποιείται spatial join

RangeJoin crime_point#382: geometry, geometry#220: geometry, WITHIN

Είναι (non-equi) join, και ελέγχει αν κάθε σημείο (έγκλημα ως (lat, lon)) ανήκει σε ένα πολύγωνο (περιοχή), Είναι η κατάλληλη μέθοδος για χωρικά δεδομενα αν και σχετικά αργή σε σχέση με πχ Hash Join

2.Για την εύρεση των εισοδημάτων ανά ZipCode (κάθε περιοχή μπορεί να περιέχει πολλά ZipCodes/ZCTA20) Income with comm

BroadcastHashJoin [ZipCode#265], [ZCTA20#577], Inner, BuildLeft, false

Το αριστερό data frame ,income_df ειναι μικρού μεγεθους και αντιγράφεται σε όλους τους κόμβους του cluster και γινεται τοπικη εκτέλεση με hash join με το zip_codes_comm..Έτσι αποφεύγεται το shuffle και μειώνεται το κόστος για την μεταφορά των δεδομένων.

3.Για το τελικο join στο final_df που αφορά την ετήσια μέση αναλογία εγκλημάτων ανά άτομο για κάθε περιοχή του Λος Άντζελες και το εισοδημα

,SortMergeJoin [COMM#50], [COMM#557], Inner

Και οι δύο πίνακες είναι μεγάλοι για broadcast, οπότε ταξινομούνται ως προς το key `COMM` και συγχωνεύονται τμηματικά. Αυτή η στρατηγική είναι ασφαλής για μεγάλες σχέσεις, αν και είναι πιο αργή.

Link του repository στο Github: https://github.com/dimitranestori/Advanced_DB_2026