Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Εξάμηνο 9ο :Προηγμένα Θέματα Βάσεων Δεδομένων

Ονοματεπώνυμο :Σκαρμούτσος Σπυρίδων
ΑΜ: 03121644 Ομάδα 9

Αναφορά Εξαμηνιαίας Εργασίας

Abstract

Στο github repository: https://github.com/Sskarm/Advanced-Database-Topics.git βρίσκονται το Python notebook (adb_project.ipynb) το οποίο περιέχει τον ήδη εκτελεσμένο κώδικα. Στον φάκελο Query Results υπάρχουν ως csv τα ζητούμενα αποτελέσματα του κάθε query σε αρχεία csv. Στον φάκελο Query Plans υπάρχουν οι πληροφορίες που μου δόθηκαν από τον Catalyst για τις επιλογές των joins στα τρία τελευταία Queries. Τέλος μια σημείωση για το notebook δεν εχω κανει spark session init καθώς δεν χρειάζεται στο περιβάλλον που μας δώθηκε.

Query 1

Σημείωση στα δεδομένα δεν έχω λάβει υπόψιν ανθρώπους με ηλικία 0 χρονών καθώς δεν βγαζει νόημα.

Τα αποτελέσματα του Query 1 είναι τα εξής:

Adults	121660
Young adults	33758
Children	10904
Elderly	6011

Για το Query 1 δοκιμάσαμε τρεις υλοποιήσεις:
DataFrame χωρίς UDF, DataFrame με UDF και RDD. Οι συμβολικοί χρόνοι ήταν:

T_rdd << T_df_no_udf < T_df_udf

Τα συμπεράσματα είναι τα εξής:

- Η **RDD** υλοποίηση ήταν η πιο γρήγορη, κυρίως επειδή το συγκεκριμένο query είναι πολύ απλό (λίγο mapping και ένα βασικό reduce). Σε τέτοιες περιπτώσεις το "βάρος" του Catalyst στον DataFrame planner είναι μεγαλύτερο από το ίδιο το computation, οπότε το RDD μπορεί να εμφανιστεί ταχύτερο.
- Το **DataFrame χωρίς UDF** ήταν πιο γρήγορο από το **DataFrame με UDF**, όπως αναμένεται. Οι ενσωματωμένες συναρτήσεις του DataFrame API επιτρέπουν στον Catalyst να βελτιστοποιήσει το πλάνο πιο αποτελεσματικά.
- Η χρήση **Python UDF** προσθέτει overhead, λόγω σειριοποίησης μεταξύ JVM και Python. Η διαφορά όμως εδώ ήταν μικρή, επειδή το συνολικό query είναι πολύ απλό, οπότε ο χρόνος κυριαρχείται από το fixed overhead του Spark.
- Παρότι η RDD λύση κέρδισε σε αυτό το πολύ ελαφρύ workload, σε πραγματικές μεγαλύτερες εργασίες το DataFrame API παραμένει η βέλτιστη και πιο αποδοτική επιλογή.

Query 2

Τα αποτελέσματα του Query 2 είναι τα εξής (ενδεικτικά για τα πρώτα έτη):

Year	Victim Descent	#	%
2025	Hispanic/Latin/Mexican	34	40.48
2025	Unknown	24	28.57
2025	White	13	15.48
2024	Hispanic/Latin/Mexican	28576	29.05
2024	White	22958	23.34

Για το Query 2 δοκιμάσαμε δύο υλοποιήσεις:
DataFrame API και SQL API.
Οι συμβολικοί χρόνοι εκτέλεσης ήταν:

T_q2_df < T_q2_sql

Τα συμπεράσματα είναι τα εξής:

• Η υλοποίηση με **DataFrame API** ήταν ταχύτερη. Αυτό είναι αναμενόμενο, καθώς οι DataFrame λειτουργίες εκφράζουν άμεσα το λογικό πλάνο στον Catalyst optimizer, ο

οποίος μπορεί να εφαρμόσει πιο αποδοτικές βελτιστοποιήσεις χωρίς το επιπλέον κόστος του SQL parsing.

• Η **SQL υλοποίηση** ήταν ελαφρώς πιο αργή, κυρίως λόγω του overhead που εισάγει το parsing του SQL query, η δημιουργία ενδιάμεσων σχεδίων και η επιπλέον μετατροπή τους στο optimized physical plan. Παρ' όλα αυτά, παράγει ακριβώς τα ίδια αποτελέσματα.

• Και στις δύο υλοποιήσεις αξιοποιήθηκε σωστά το mapping των κωδικών Vict Descent μέσω του dataset RE_codes.csv, ώστε τα αποτελέσματα να εμφανίζονται με τις πλήρεις περιγραφές φυλετικών ομάδων (π.χ. «Hispanic/Latin/Mexican», «White», «Black»), όπως απαιτεί η εκφώνηση.

• Συνολικά, η DataFrame λύση αποδείχθηκε πιο αποδοτική, ενώ η SQL λύση παραμένει απόλυτα έγκυρη αλλά με λίγο μεγαλύτερο χρόνο εκτέλεσης.

Query 3

Τα αποτελέσματα του Query 3 είναι τα εξής (ενδεικτικά για τα πρώτα frequencies):

MO Code	Description	Frequency
0344	Removes vict property	1002900
1822	Stranger	548422
0416	Hit-Hit w/ weapon	404773

Για το Query 3 συγκρίνω πρώτα το DF με τα RDDs και μετά τις διάφορες υλοποιήσεις των join μεταξύ τους στο RDD.

Μέρος Α

Για το Query 3 δοκιμάσαμε δύο υλοποιήσεις: DataFrame API και RDD API. Οι συμβολικοί χρόνοι ήταν:

T_q3_rdd << T_q3_df

Τα συμπεράσματα είναι τα εξής:

- Η RDD υλοποίηση ήταν σημαντικά πιο γρήγορη από την DataFrame υλοποίηση. Όπως και στο Query 1, το συγκεκριμένο query είναι σχετικά απλό (explode σε κωδικούς, count και ένα join με μικρό πίνακα), οπότε το σταθερό κόστος του Catalyst (logical/physical planning, code generation) γίνεται συγκρίσιμο ή και μεγαλύτερο από το ίδιο το computation.

- Η DataFrame εκδοχή παραμένει πιο “δηλωτική” και ευανάγνωστη, αλλά πληρώνει επιπλέον overhead σε σχέση με την πιο “χαμηλού επιπέδου” RDD υλοποίηση, η οποία εδώ κάνει ουσιαστικά ένα flatMap → map → reduceByKey → sort.
- Παρότι η RDD λύση κερδίζει σε αυτό το συγκεκριμένο, σχετικά ελαφρύ aggregation, σε μεγαλύτερα και πιο σύνθετα workloads (joins πολλών πινάκων, φίλτρα, σύνθετες εκφράσεις) το DataFrame API εξακολουθεί να είναι η πιο κατάλληλη επιλογή λόγω βελτιστοποιήσεων του Catalyst.

Μέρος Β

Για το Query 3 εξετάστηκαν οι στρατηγικές default, broadcast, merge, shuffle_hash και shuffle_replicate_nl. Οι συμβολικοί χρόνοι εκτέλεσης ήταν:

$T_{shuffle\_hash} > T_{merge} \approx T_{shuffle\_replicate\_nl} > T_{broadcast} \approx T_{default}$

Τα βασικά συμπεράσματα είναι τα εξής:

• Το **shuffle_hash** join ήταν consistently το πιο αργό. Αυτό είναι αναμενόμενο, καθώς απαιτεί πλήρες shuffle των δεδομένων και έχει σημαντικό κόστος σε network I/O και disk spill. Σε μικρά dimension tables η στρατηγική αυτή είναι η λιγότερο κατάλληλη.

• Οι στρατηγικές **default** και **broadcast** είχαν τους καλύτερους χρόνους. Ο μικρός πίνακας mo_df επιτρέπει στο Spark να χρησιμοποιήσει αποτελεσματικό BroadcastHashJoin, είτε μέσω explicit hint είτε ως επιλογή του Catalyst από το default join strategy.

• Το **merge** (SortMergeJoin) και το **shuffle_replicate_nl** είχαν ενδιάμεσες επιδόσεις. Το merge χρειάζεται ταξινόμηση και των δύο πλευρών, ενώ το shuffle_replicate_nl έχει μικρό κόστος replication λόγω του μικρού μεγέθους του πίνακα MO codes. Αυτό εξηγεί γιατί οι δύο στρατηγικές είχαν παρόμοια απόδοση.

• Γενικά, όταν το “δεξί” input του join είναι μικρό (όπως εδώ), οι broadcast-style και replicate-style joins είναι οι πιο κατάλληλες επιλογές, ενώ τα shuffle-based joins είναι τα λιγότερο αποδοτικά.

Query 4

Τα αποτελέσματα του Query 4 είναι τα εξής (ενδεικτικά για τα πρώτα division):

division	average_distance	crime_count
HOLLYWOOD	2.266	212904
VAN NUYS	3.167	209295
WILSHIRE	2.921	198499

Στο μέρος Α θα σχολιάσω τον τρόπο Join και μετά στο μέρος β τους πόρους.

Μέρος Α

Στο Query 4 ο Spark επέλεξε **BroadcastNestedLoopJoin (Cross BuildRight)**.
Τα συμπεράσματα συνοπτικά:

- Ο πίνακας των αστυνομικών τμημάτων είναι **πολύ μικρός**, άρα ο Catalyst τον κάνει broadcast σε κάθε executor μέσω BroadcastExchange.
- Το join είναι **spatial**: δεν υπάρχει equi-join key, αλλά υπολογισμός απόστασης με ST_Distance. Επομένως ο Spark δεν μπορεί να χρησιμοποιήσει ούτε Hash Join ούτε Sort-Merge Join.
- Ο συνδυασμός *μικρό dimension table* + *spatial condition* οδηγεί στο **Broadcast Nested Loop** ως πιο αποδοτική επιλογή.
- Μετά το join, εφαρμόζεται row_number() για να κρατηθεί ο **πλησιέστερος σταθμός ανά crime**, και στη συνέχεια group by για count και average distance.

Συμπέρασμα: Λόγω μεγάλου crime dataset, μικρού station dataset και spatial λογικής, το **BroadcastNestedLoopJoin** είναι η σωστή και αναμενόμενη επιλογή.

Μέρος Β

Οι συμβολικοί χρόνοι εκτέλεσης ήταν:

T_q4_ 2×4_8g < T_q4_ 2×2_4g < T_q4_ 2×1_2g

Συμπεράσματα:

- Αυξάνοντας cores/μνήμη, οι χρόνοι **μειώνονται σημαντικά**, ειδικά στο πέρασμα από 1 → 2 cores. Το Query 4 είναι CPU-heavy (ST_Distance, sort, window), άρα κερδίζει από περισσότερα task slots.
- Η αύξηση σε 4 cores/8GB βελτιώνει κι άλλο τον χρόνο αλλά με **φθίνουσες αποδόσεις**, καθώς ορισμένα στάδια (window, shuffle, sort) δεν παραλληλοποιούνται τέλεια και το Spark προσθέτει scheduling overhead.

Έτσι, το Query 4 δείχνει ξεκάθαρη επιτάχυνση με περισσότερους πόρους, αλλά όχι γραμμική.

Query 5

Τα αποτελέσματα του Query 5 είναι τα εξής (ενδεικτικά για τα πρώτα division):

corellation	%
All	0.0978
Top 10	-0.0413
Bottom 10	-0.1172

Στο μέρος Α θα σχολιάσω τον τρόπο Join και μετά στο μέρος β τους πόρους.

Μέρος Α

- Στο spatial κομμάτι, ο Catalyst μετατρέπει το ST_Contains σε Sedona **RangeJoin (WITHIN)**, δηλαδή χωρικό join πάνω σε γεωμετρίες point–polygon.

- Για το join των blocks με το income χρησιμοποιείται **BroadcastHashJoin**, επειδή ο πίνακας εισοδημάτων είναι μικρός και μπορεί να γίνει broadcast.

- Στο τελικό join των δύο aggregated πλευρών (crimes per COMM και income per COMM), ο Catalyst επιλέγει **SortMergeJoin**, επειδή και οι δύο πίνακες είναι μεσαίου μεγέθους και ήδη έχουν προηγηθεί shuffles και aggregates, οπότε η ταξινόμηση είναι η πιο σταθερή και αποδοτική λύση.

Συμπέρασμα: Χωρικό κομμάτι = Sedona RangeJoin,
dimension join = BroadcastHashJoin,

τελικό join = **SortMergeJoin** (λογικό λόγω μεσαίου μεγέθους και ανάγκης για ordered/shuffle consistency).

Μέρος Β

Οι συμβολικοί χρόνοι εκτέλεσης για το Query 5 ήταν:

T_conf1 (1 core / 2GB) < T_conf2 (2 cores / 4GB) < T_conf3 (4 cores / 8GB).

Συμπεράσματα:

- Σε αντίθεση με το Query 4, εδώ η αύξηση cores/μνήμης **δεν βελτιώνει τον χρόνο**, αλλά τον επιβαρύνει.Το Query 5 είναι πιο "μεσαίου" μεγέθους workload, με μεγάλο μέρος του κόστους να βρίσκεται στο Sedona **RangeJoin**, το οποίο δεν παραλληλοποιείται τέλεια και έχει σταθερό spatial overhead.
- Η αύξηση των πόρων εισάγει **περισσότερα shuffle partitions, επιπλέον scheduling overhead**, και περισσότερη επικοινωνία μεταξύ executors, τα οποία τελικά αυξάνουν τον συνολικό χρόνο.
- Το αποτέλεσμα είναι ότι το query **τρέχει γρηγορότερα στη μικρότερη διαμόρφωση**, επειδή το workload δεν είναι αρκετά μεγάλο ώστε να αξιοποιήσει αποτελεσματικά περισσότερους πυρήνες.

Έτσι, στο Query 5, η μεγαλύτερη παραλληλοποίηση **δεν μεταφράζεται σε επιτάχυνση**, αλλά αντίθετα οδηγεί σε ελαφρώς χειρότερη απόδοση.