ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΒΑΣΕΩΝ ΓΝΩΣΕΩΝ ΚΑΙ ΔΕΔΟΜΕΝΩΝ

Προχωρημένα Θέματα Βάσεων Δεδομένων

Ακαδημαϊκό έτος 2024-25, 9ο Εξάμηνο

Διδάσκων: Δημήτριος Τσουμάκος

Υπεύθυνος Εργαστηρίου: Νικόλαος Χαλβαντζής

Ομάδα: 12

Μέλη: Ηλιόπουλος Ιωάννης Νεκτάριος 03119919

Καλογεροπούλου Γεωργία 03114174

21 Ιανουαρίου 2025

Διαδικαστικά:

Το link για το Github repo μας που περιέχει τον κώδικα των ερωτημάτων. (Το repo είναι private και θα το κάνουμε public μετά την υποβολή της εργασίας μας)

Τα αρχεία με τα logs καθώς και τα αποτελέσματά τους για κάθε query είναι διαθέσιμα στο S3 bucket της ομάδας μας (Ομάδα 12) στο AWS.

Τα data-sets που χρησιμοποιήθηκαν στην εργασία είναι τα ακόλουθα:

Los Angeles Crime Data:
https://data.lacity.org/Public-Safety/Crime-Data-from-2010-to-2019/63jg-8b9z

https://data.lacity.org/Public-Safety/Crime-Data-from-2020-to-Present/2nrs-mtv8

Los Angeles Census Blocks:

https://data.lacounty.gov/maps/lacounty::2010-census-blocks

Median Household Income by Zip Code:

http://www.laalmanac.com/employment/em12c_2015.php

LA Police Stations:

https://geohub.lacity.org/datasets/lahub::lapd-police-stations/explore

Race And Ethnicity Codes:

Ερώτημα 1:

Στο ερώτημα 1 καλούμαστε να πραγματοποιήσουμε query στο οποίο ταξινομούνται σε φθίνουσα σειρά οι ηλικιακές ομάδες των θυμάτων σε περιστατικά “βαριάς σωματικής βίας”. Για την διεκπαιρέωση του κώδικα χρησιμοποιήσαμε τα δύο LA Crimes Datasets (2010-2019, 2020-2024). Επίσης μας ζητείται να υλοποιήσουμε το πρόβλημα με τη χρήση 4 Spark Executors Και την χρήση Dataframe και RDD APIs.

Παρακάτω παρατίθονται τα αποτελέσματα του κώδικά μας ο οποίος βρίσκεται στο αρχείο ‘Query 1.ipynb’, καθώς και το configuration για την εξασφάλιση 4 spark executors. Παρατηρούμε ότι τα αποτελέσματα είναι τα ίδια και με τις δύο μεθόδους, αλλά ο χρόνος εκτέλεσης έχει μεγάλη απόκλιση.

Configuration:

```
%%configure -f
{
    "conf": {
        "spark.executor.instances": "4"
    }
}
```

```
Current session configs: {'conf': {'spark.sql.catalog.spark_catalog.type': 'hive',
'spark.executor.instances': '4'}, 'kind': 'pyspark'}
```

RDD:

```
[('Adults', 121093), ('Young adults', 38703), ('Children', 10830), ('Elders', 5985)]
Time taken: 57.95 seconds
```

Dataframe:

```
SparkSession available as 'spark'.
+------------+------+
|   Age_group| count|
+------------+------+
|      Adults|121093|
|Young adults| 38703|
|    Children| 10830|
|      Elders|  5985|
+------------+------+

Time taken: 24.80 seconds
```

Η διαφορά μεταξύ των δύο μεθόδων φαίνεται να οφείλεται στη χρήση του Catalyst Optimizer στην προσέγγιση του DataFrame, ο οποίος έχει αναπτυχθεί για να βελτιστοποιεί την τεχνολογία των RDD. Οι βελτιστοποιημένες λειτουργίες του DataFrame API, όπως οι ταχύτεροι αλγόριθμοι για join (π.χ. Broadcast Hash Join), συμβάλλουν σημαντικά στην καλύτερη απόδοση, κάνοντάς το ιδανικό για μεγάλες κλίμακες δεδομένων. Τα DataFrame οργανώνουν τα δεδομένα τους σε στήλες, κάτι που διευκολύνει τόσο τον προγραμματισμό όσο και τις πράξεις σε οργανωμένα

δεδομένα, όπως τα CSV, Parquet ή JSON. Η στενή ενσωμάτωσή του με το Spark SQL επιτρέπει τη σύνταξη υψηλού επιπέδου ερωτημάτων με λιγότερο κώδικα και μεγαλύτερη ευχέρεια. Αντίθετα, το RDD API, αν και πιο γενικευμένο, δεν υποστηρίζει τέτοιες βελτιστοποιήσεις, απαιτώντας σειριακή και λιγότερο αποδοτική επεξεργασία δεδομένων.

Ερώτημα 2:

a) Στο ερώτημα 2 καλούμαστε να πραγματοποιήσουμε query στο οποίο παραθέτουμε τα 3 Αστυνομικά Τμήματα με το υχηλότερο ποσοστό κλει-σμένων υποθέσεων για κάθε έτος. Επίσης μας ζητείται να υλοποιήσουμε το πρόβλημα με τη χρήση SQL APIs και Dataframe και να συγκρίνουμε τους χρόνους εκτέλεσης. Ο κώδικας και των δύο ερωτημάτων βρίσκεται στο αρχείο "Query 2.ipynb"

Παρακάτω έχουμε τους δύο χρόνους εκτέλεσης των Dataframe και SQL API αντίστοιχα:

Dataframe CSV: `Time taken: 15.34 seconds`

SQL CSV: `Time taken: 29.15 seconds`

Όσον αφορά τον χρόνο εκτέλεσης, περιμένουμε το DataFrame API και το SQL API να έχουν συγκρίσιμες επιδόσεις, καθώς και τα δύο βασίζονται στον ίδιο βελτιστοποιητή, τον Catalyst Optimizer. Ωστόσο, σε πρακτικό επίπεδο, το SQL API βλέπουμε ότι είναι πιο αργό λόγω της πρόσθετης φάσης ανάλυσης του SQL query σε σχέσεις και στη συνέχεια της μετατροπής του σε DataFrame. Το DataFrame API, από την άλλη, προσφέρει καλύτερο χρόνο εκτέλεσης σε λειτουργίες που συνδυάζουν προγραμματιστικά και αναλυτικά στοιχεία, καθώς αποφεύγει αυτό το ενδιάμεσο βήμα. Παρά τις μικρές διαφορές, ο χρόνος εκτέλεσης εξαρτάται συχνά περισσότερο από τη φύση του φορτίου εργασίας και τη δομή των δεδομένων παρά από τη διαφορά μεταξύ των δύο APIs, καθώς χρησιμοποιούν την ίδια υποδομή στο Spark SQL Engine.

b) Στο δεύτερο υπο-ερώτημα μας ζητείται να μετατρέψουμε τα αρχικά δεδομένα από μορφή .csv σε .parquet και στην συνέχεια να εκτελέσουμε τον ίδιο κώδικα με μία απο τις παραπάνω υλοποιήσεις. Το νέο dataset βρίσκεται στο S3 bucket της ομάδας μας (Ομάδα 12). Λόγω των παρατηρήσεων του ερωτήματος α, αποφασίσαμε να δοκιμάσουμε την υλοποίηση με το αρχείο parquet σε Dataframe εφόσον ήταν το πιο γρήγορο.

Από την θεωρία αναμένουμε ότι η ανάγνωση δεδομένων μέσω του DataFrame API θα είναι ταχύτερη στο Parquet απότι με το αρχείο CSV λόγω του σχεδιασμού τους. Το CSV απαιτεί ανάλυση γραμμής προς γραμμή, δημιουργία schema, και επεξεργασία μεγαλύτερων όγκων δεδομένων λόγω έλλειψης συμπίεσης. Το Parquet, αντίθετα, χρησιμοποιεί columnar storage, ενσωματωμένο schema, και συμπίεση, μειώνοντας το I/O και επιταχύνοντας την ανάγνωση. Στον κώδικά μας παρατηρήσαμε ότι η ανάγνωση από Parquet είναι 3 φορές ταχύτερη απ'ότι με CSV, όπως παρατίθεται απο κάτω:

```
Time taken: 4.64 seconds
```

Ερώτημα 3:

Στο ερώτημα 3 καλούμαστε να χρησιμοποιήσουμε δύο νέα dataset, ένα για την απογραφή του πληθυσμού του 2010 και της απογραφής για εισόδημα ανά νοικοκυριό του 2015. Με αυτά καλούμαστε να βρούμε για κάθε περιοχή του Los Angeles το μέσο εισόδειμα ανά νοικοκοιριό και το μέσο όρο εγκλημάτων ανά κάτοικο. Το query αρχικά πραγματοποιείται στο αρχείο “Query 3.ipymb”. Χρησιμοποιήσαμε την υλοποίηση του Apache Sedona όπως δόθηκε στο βοηθητικό υλικό, όπως και το αρχείο .parquet από το προηγουμένο ερώτημα για την πιο γρήγορη υλοποίηση του κώδικα.

Αρχικά μας ζητείται να χρησιμοποιήσουμε την εντολή explain για να δούμε το φυσικό πλάνο που χρησιμοποίησε ο Catalyst Optizer:

```
== Physical Plan ==
AdaptiveSparkPlan (29)
+- Sort (28)
   +- Exchange (27)
      +- HashAggregate (26)
         +- Exchange (25)
            +- HashAggregate (24)
               +- Project (23)
                  +- RangeJoin (22)
                     :- Filter (18)
                     :  +- ObjectHashAggregate (17)
                     :     +- Exchange (16)
                     :        +- ObjectHashAggregate (15)
                     :           +- Project (14)
                     :              +- BroadcastHashJoin Inner BuildRight (13)
                     :                 :- ObjectHashAggregate (8)
                     :                 :  +- Exchange (7)
                     :                 :     +- ObjectHashAggregate (6)
                     :                 :        +- Project (5)
                     :                 :           +- Filter (4)
                     :                 :              +- Generate (3)
                     :                 :                 +- Filter (2)
                     :                 :                    +- Scan geojson  (1)
                     :                 +- BroadcastExchange (12)
                     :                    +- Project (11)
                     :                       +- Filter (10)
                     :                          +- Scan csv  (9)
                     +- Project (21)
                        +- Filter (20)
                           +- Scan parquet  (19)
```

Παρατηρούμε ότι στα δύο join που πραγματοποιήσαμε ο Catalyst Optimizer χρησιμοποίησε αρχικά ένα Broadcast Hash Join με Build Right. Αυτό συμβαίνει επειδή το res_income είναι πολύ πιο μικρό σε σύγκριση με το zip_comm_population. Έτσι, συμφέρει να σταλθεί ολόκληρο το dataset στους workers για να αποφεφχθεί το shuflling κόστος. Έπειτα χρησιμοποίησε ένα Range Join λόγω της συνθήκης σύγκρισης που περιλαμβάνει το zip_comm_population["ZCTA10"] == res_income["Zip Code"]. Το Spark, όταν εκτελεί την ένωση και επεξεργάζεται τις στήλες με αριθμητικούς υπολογισμούς, μπορεί να εφαρμόσει Range Join για να προσαρμόσει τις εκτελέσεις στον βέλτιστο σχεδιασμό, ειδικά εάν οι υπολογισμοί εμπλέκουν μεγάλο εύρος τιμών.

Αυτό συχνά βελτιώνει την απόδοση σε δεδομένα όπου εφαρμόζεται το broadcasting , μειώνοντας τη μεταφορά δεδομένων μεταξύ των κόμβων.

Στη συνέχεια αναγκάσαμε το Spark να χρησιμοποιήσει τις μεθόδους BROADCAST, MERGE, SHUFFLE_HASH, SHUFFLE_REPLICATE_NL στα δύο join του κώδικα μας με τη χρήση του hint. Οι κώδικες βρίσκονται στα αρχεία με το αντίστοιχο όνομα (πχ Query 3 Broadcast.ipynb) και τα αποτελέσματα κάθε μεθόδου αποθηκεύτηκαν σε ένα Json στο S3 bucket μας, με την προβολή του Json να γινεται στο βασικό αρχείο κώδικα (Query 3.ipymb). Τα αποτελέσματα τους είναι τα ακόλουθα:

```
+--------------------+--------------+
|Join_Strategy       |Execution_Time|
+--------------------+--------------+
|SHUFFLE_REPLICATE_NL|3.28          |
|SHUFFLE_HASH        |6.64          |
|BROADCAST           |6.67          |
|MERGE               |7.87          |
+--------------------+--------------+
```

Παρατηρούμε ότι σύμφωνα με τα αποτελέσματα που λάβαμε, ο χρόνος εκτέλεσης για το SHUFFLE_REPLICATE_NL βγαίνει μικρότερος από τις υπόλοιπες στρατηγικές. Αυτό μας εκπλήσει γιατί όπως είδαμε πιο πρίν, θα περιμέναμε να είναι πρώτο το Broadcast που επιλέγει ο Catalyst Optimizer στο πρώτο Join. Παρ'ολα αυτά, το Broacast φαίνεται να μην είναι τόσο efficient για το δεύτερο Join που είναι και συγκριτικά μεγαλύτερο σε όγκο, με αποτέλεσμα η μέθοδος Broadcast να είναι η 3η γρηγορότερη. Η στρατηγική SHUFFLE_REPLICATE_NL εμφανίζεται ταχύτερη με μεγάλη διαφορά σε σύγκριση με τις υπόλοιπες (μισός χρόνος) το οποίο μάλλον οφείλεται στη χρήσης χωρικού partitioning και indexing (π.χ. R-Tree), που μειώνουν το πλήθος γεωμετριών προς σύγκριση μέσω pruning. Σε γεωχωρικά δεδομένα, οι πιο σύνθετες συνθήκες όπως το ST_Within απαιτούν περισσότερους υπολογισμούς από απλές ισότητες, και το nested loop, σε συνδυασμό με τοπικές βελτιστοποιήσεις, αποφεύγει overhead όπως shuffle, sort, ή broadcast preparation, που συναντάται σε άλλες στρατηγικές (Broadcast, Hash, Merge). Έτσι, όταν το μέγεθος ή η κατανομή δεδομένων δεν ευνοούν άλλες προσεγγίσεις, η SHUFFLE_REPLICATE_NL προσφέρει καλύτερη απόδοση.

Με βάση τα παραπάνω, εικάζουμε ότι η ταχύτερη απόδοση θα ήταν στο πρώτο join να πραγματοποιηθεί Broadcast, ενώ στο δεύτερο SHUFFLE_REPLICATE_NL.

Ερώτημα 4:

Στο ερώτημα 4 καλούμαστε να πραγματοποιήσουμε ένα query το οποίο θα επιστρέφει το φυλετικό προφίλ των θυμάτων στις 3 περιοχές με ψηλότερο και χαμηλότερο κατα κεφαλήν εισόδημα το έτος 2015. Η υλοποίηση μας έγινε με τη χρήση dataframe, που όπως είδαμε πιο πάνω είναι και η ταχύτερη. Επίσης εκτελέστηκε τρεις φορές συμφωνα με τις οδηγίες της άσκησης, με σταθερό αριθμό 2 executors αλλά διαφορετικές ρυθμίσεις σε πυρήνες (cores) και μνήμη (GB). Στην πρώτη περίπτωση

χρησιμοποιήσαμε 1 core και 2GB μνήμης για κάθε executor, στη δεύτερη 2 cores και 4GB μνήμης, και στην τρίτη 4 cores και 8GB μνήμης. Τα αποτελέσματα παρατίθονται παρακάτω:

Top 3:

Victim Descent	Count
White	649
Other	72
Hispanic/Latin/Me...	66
Unknown	38
Black	37
Other Asian	21
American Indian/A...	1
Chinese	1

Bottom 3:

Victim Descent	Count
Hispanic/Latin/Me...	2815
Black	761
White	330
Other	187
Other Asian	113
Unknown	22
American Indian/A...	21
Korean	5
Chinese	3
AsianIndian	1
Filipino	1

spark_executor_cores	spark_executor_memory	execution_time
4	8g	58.45950675010681
2	4g	64.49426293373108
1	2g	82.67877650260925

Όπως διαπιστώνεται, η αύξηση του αριθμού των πυρήνων και της μνήμης οδηγεί σε μείωση του χρόνου εκτέλεσης. Αυτό οφείλεται κυρίως στην καλύτερη εκμετάλλευση του παράλληλου υπολογισμού (parallelism) από τον Spark, καθώς περισσότερα tasks μπορούν να εκτελούνται ταυτόχρονα. Επίσης, η μεγαλύτερη μνήμη επιτρέπει στο σύστημα να διαχειρίζεται καλύτερα τους ενδιάμεσους υπολογισμούς και να ελαχιστοποιεί τις καθυστερήσεις από το garbage collection ή τη μεταφορά δεδομένων στον δίσκο (spill to disk).

Βλέπουμε επίσης ότι η διαφορά μεταξύ 2 cores / 4GB και 4 cores / 8GB είναι υπαρκτή αλλά όχι τόσο δραματική όσο μεταξύ 1 core / 2GB και των άλλων δύο ρυθμίσεων. Παρ' όλα αυτά, η αύξηση των πόρων (cores και μνήμης) συνεχίζει να βελτιώνει τις επιδόσεις, καθώς επιτρέπει μεγαλύτερο βαθμό παράλληλης εκτέλεσης και αποφυγή bottlenecks στη διαχείριση της μνήμης.

Ερώτημα 5:

Στο ερώτημα 5 καλούμαστε να πραγματοποιήσουμε ένα query το οποίο θα επιστρέφει τον αριθμό εγκλημάτων που έλαβαν χώρα πλησιέστερα σε κάθε αστυνομικό τμήμα, καθώς και τη μέση απόστασή του από τις τοποθεσίες όπου σημειώθηκαν τα συγκεκριμένα περιστατικά. Η υλοποίηση μας έγινε με τη χρήση dataframe. Επίσης εκτελέστηκε τρεις φορές συμφωνα με τις οδηγίες της άσκησης, με σταθερό αριθμό πυρήνων (8 cores) και μνήμης (16GB), αλλά διαφορετικό αριθμό executors(2,4,8). Τα αποτελέσματα παρατίθονται παρακάτω:

Division	average_distance_km	crime_count
HOLLYWOOD	2.0763	224340
VAN NUYS	2.9534	210134
SOUTHWEST	2.1914	188901
WILSHIRE	2.5927	185996
77TH STREET	1.7165	171827
OLYMPIC	1.7236	170897
NORTH HOLLYWOOD	2.6430	167854
PACIFIC	3.8501	161359
CENTRAL	0.9925	153871
RAMPART	1.5345	152736
SOUTHEAST	2.4219	152176
WEST VALLEY	3.0357	138643
TOPANGA	3.2970	138217
FOOTHILL	4.2509	134896
HARBOR	3.7026	126747
HOLLENBECK	2.6802	115837
WEST LOS ANGELES	2.7925	115781
NEWTON	1.6346	111110
NORTHEAST	3.6237	108109
MISSION	3.6909	103355

Executor instances	cores	memory	time
4	2	4g	32.254109621047974
2	4	8g	41.54272270202637
8	1	2g	53.72337985038757

Παρατηρούμε ότι το configuration με τα 4 executors οδήγησε στον μικρότερο χρόνο εκτέλεσης, περίπου 32 δευτερόλεπτα. Αντίθετα, τα 8 executors είχε τον μεγαλύτερο χρόνο, περίπου 53 δευτερόλεπτα, ενώ τα 2 executors διήρκησαν κάπου ενδιάμεσα με ~41.5 δευτερόλεπτα. Τα αποτελέσματα αυτά μας εκπλήσσουν διότι διαισθητικά θα περιμέναμε ο χρόνος να είναι αντιστρόφως ανάλογος με το πλήθος των executors, και να είναι ταχύτερη η εκτέλεση με τους 8 έναντι των 4.

Μία εικασία που μπορούμε να κάνουμε για τους χρόνους είναι ότι στην εκτέλεση με τους 4 executors το σύστημα αξιοποιεί πιο αποτελεσματικά τους διαθέσιμους πόρους του ανάμεσά τους, αφού έχει ο καθένας 2 πυρήνες και 4GB μνήμη. Αυτή η ισορροπημένη κατανομή επιτρέπει καλή παράλληλη επεξεργασία χωρίς να επιβαρύνει το σύστημα με υπερβολικό overhead ή ανεπάρκεια μνήμης. Αντίθετα, οι 8 executors

με 1 πυρήνα και μόλις 2GB μνήμη ο καθένας δημιουργούν σημαντικό overhead στη διαχείριση και επικοινωνία, ενώ μπορεί να οδηγήσουν σε αυξημένο shuffle και συχνό garbage collection. Τέλος οι 2 executors με 4 πυρήνες και 8GB μνήμη περιορίζουν τον βαθμό παράλληλης εκτέλεσης, ειδικά σε σενάρια με μεγάλα γεωχωρικά datasets.

Ετσι, η εκτέλεση με τους 4 executors προσφέρει τη βέλτιστη ισορροπία μεταξύ παράλληλης επεξεργασίας και επάρκειας μνήμης, ενώ μειώνει το overhead και τις καθυστερήσεις που συνδέονται με το shuffle και τη διαχείριση μνήμης.