Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Προχωρημένα Θέματα Βάσεων Δεδομένων

Εξαμηνιαία Εργασία

Ονοματεπώνυμο: Κωνσταντίνος Διβριώτης

Α.Μ.: 03114140

Group: 15

Github Link: https://github.com/kdivriotis/advanced-db-project

Ζητούμενο 1

Υλοποιήσαμε το **Query 1** χρησιμοποιώντας τα **DataFrame** και **RDD** APIs, και εκτελέσαμε με 4 Spark executors. Το πλήθος θυμάτων που βρήκαμε ανά ηλικακή ομάδα φαίνεται παρακάτω, σε φθίνουσα σειρά ως προς το πλήθος περιστατικών:

Ηλικιακή Ομάδα	Περιστατικά Βαριάς Σωματικής Βλάβης
Ενήλικοι	121,093
Νεαροί ενήλικοι	33,605
Παιδιά	15,928
Ηλικιωμένοι	5,985

Η υλοποίηση με χρήση του **DataFrame API** ήταν αρκετά γρηγορότερη από αυτήν με το **RDD API**.

Συγκεκριμένα, η εκτέλεση του ερωτήματος χρειάστηκε **6.78 δευτερόλεπτα** με DataFrames και **30.61 δευτερόλεπτα** με RDD.

Πράγματι, θα αναμέναμε το query να είναι γρηγορότερο με τη χρήση DataFrames, αφού τα RDDs δε μπορούν να βελτιστοποιηθούν από το **Spark** (*Catalyst Optimizer*).

Επίσης, τα RDDs είναι γενικότερα πιο αργά σε non-JVM γλώσσες, όπως η Python στην οποία εκτελέστηκαν τα ερωτήματα.

Ζητούμενο 2

(α)

Υλοποιήσαμε το **Query 2** χρησιμοποιώντας τα **DataFrame** και **SQL** APIs. Τα 3 Αστυνομικά Τμήματα με το υψηλότερο ποσοστό κλεισμένων υποθέσεων ανά έτος φαίνονται παρακάτω:

year	precinct	closed_case_rate	#
2010	RAMPART	32.8471344895	1
2010	OLYMPIC	31.5152898220	2
2010	HARBOR	29.3602833924	3
2011	OLYMPIC	35.0400600901	1
2011	RAMPART	32.4964471814	2
2011	HARBOR	28.5133624632	3
2012	OLYMPIC	34.2970853330	1
2012	RAMPART	32.4600046371	2
2012	HARBOR	29.5095858490	3
2013	OLYMPIC	33.5821794100	1
2013	RAMPART	32.1060382916	2
2013	HARBOR	29.7271648873	3
2014	VAN NUYS	32.0215235282	1
2014	WEST VALLEY	31.4975480951	2
2014	MISSION	31.2249398557	3
2015	VAN NUYS	32.2651406772	1
2015	MISSION	30.4637626737	2
2015	FOOTHILL	30.3530018037	3
2016	VAN NUYS	32.1945184621	1
2016	WEST VALLEY	31.4014643704	2
2016	FOOTHILL	29.9086472281	3
2017	VAN NUYS	32.0554272517	1
2017	MISSION	31.0553871590	2
2017	FOOTHILL	30.4697006571	3
2018	FOOTHILL	30.7313469589	1

year	precinct	closed_case_rate	#
2018	MISSION	30.7270233196	2
2018	VAN NUYS	28.9052069426	3
2019	MISSION	30.7274111123	1
2019	WEST VALLEY	30.5797433547	2
2019	NORTH HOLLYWOOD	29.2380866912	3
2020	WEST VALLEY	30.7711319822	1
2020	MISSION	30.1497464922	2
2020	HARBOR	29.6934865900	3
2021	MISSION	30.3181155901	1
2021	WEST VALLEY	28.9710874400	2
2021	FOOTHILL	27.9937570942	3
2022	WEST VALLEY	26.5363671723	1
2022	HARBOR	26.3375380600	2
2022	TOPANGA	26.2340133178	3
2023	FOOTHILL	26.7607602012	1
2023	TOPANGA	26.5380226165	2
2023	MISSION	25.6627311205	3
2024	NORTH HOLLYWOOD	19.5985289611	1
2024	FOOTHILL	18.6208821887	2
2024	77TH STREET	17.5863181672	3

Οι χρόνοι εκτέλεσης μεταξύ των 2 υλοποιήσεων ήταν πολύ παρόμοιοι - συγκεκριμένα **6.59 δευτερόλεπτα** με DataFrames και **5.57 δευτερόλεπτα** με SQL.

Το αποτέλεσμα αυτό είναι αναμενόμενο, καθώς και στις 2 περιπτώσεις το Spark χρησιμοποιεί τον *Catalyst Optimizer* προκειμένου να δημιουργήσει και να βελτιστοποιήσει ένα λογικό και φυσικό πλάνο εκτέλεσης του query.

(β)

Μετατρέπουμε το κυρίως dataset (εγκλήματα 2010-2019 και 2020-σήμερα) σε ένα ενιαίο αρχείο τύπου *parquet*. Επιλέγουμε την υλοποίηση με DataFrame και εκτελούμε ξανά το Query 2, με μοναδική διαφορά ότι διαβάζουμε το αρχείο parquet αντί του CSV.

Παρατηρούμε μία βελτίωση από τα 6.59 στα **3.40 δευτερόλεπτα**. Αυτό οφείλεται αφενός στο ότι το αρχείο parquet πετυχαίνει πολύ καλή συμπίεση, αφού το εξαγόμενο αρχείο έχει μέγεθος 127 MB έναντι των 752 MB των 2 αρχικών CSV.

Η σημαντικότερη διαφορά όμως έχει να κάνει με τον τρόπο που είναι αποθηκευμένα τα δεδομένα, τα οποία είναι σε **Columnar** μορφή αντί για **row-based** όπως το CSV. Αυτό βοηθάει στη γρηγορότερη ανάγνωση και επεξεργασία των δεδομένων.

Τέλος, το αρχείο parquet υποστηρίζει **predicate pushdown**, το οποίο δίνει τη δυνατότητα να φιλτραριστούν τα δεδομένα πριν να διαβαστούν στη μνήμη και συνεπώς πετυχαίνουμε καλύτερη βελτιστοποίηση.

Ζητούμενο 3

Υλοποιήσαμε το **Query 3** χρησιμοποιώντας το **DataFrame** API. Τα πρώτα 20 (από τα 139) αποτελέσματα για το μέσο ετήσιο εισόδημα ανά άτομο και την αναλογία συνολικού αριθμού εγκλημάτων ανά άτομο, ταξινομημένα σε αύξουσα αλφαβητική σειρά με βάση το όνομα της περιοχής (για ευκολότερη σύγκριση) φαίνονται παρακάτω:

COMM	Income per Person	Crimes per Person
Adams-Normandie	8,791.4583014537	0.7148686560
Alsace	11,239.5013642565	0.5416098226
Angeles National Forest	33,079.5882352941	0.4117647059
Angelino Heights	18,427.0598146588	0.5732940185
Arleta	12,110.7794743886	0.4264509064
Atwater Village	28,481.2369671608	0.5288318320
Baldwin Hills	17,303.9064082417	0.9950061114
Bel Air	63,041.3394262196	0.3992252754
Beverly Crest	60,947.4897875482	0.3689607087
Beverlywood	29,267.8211840516	0.5084977849
Boyle Heights	8,494.1082868613	0.6171887393
Brentwood	60,846.8544610554	0.4058638815
Brookside	18,138.6264090177	0.8856682770
Cadillac-Corning	19,572.7846961740	0.5816954239
Canoga Park	19,660.2917380754	0.5506083179
Canthay	49,848.7359264605	0.7628959964
Central	6,973.3056637868	0.6593822350
Century City	45,617.7601345669	0.6329688814
Century Palms/Cove	8,610.3146550310	1.1446474853
Chatsworth	30,694.6064433886	0.5281029088

Για την υλοποίηση απαιτούνται 2 join:

- **blocks_data** (*2010_Census_Blocks.geojson*) ⋈ **income_data** (*LA_income_2015.csv*) με βάση το ZIP Code

- **crime_data** (Crime*_Data_from_*_to_*.csv*) ⋈ **result** (*του προηγούμενου join*) με βάση τη “γεωμετρία” (τόπος εγκλήματος εντός της περιοχής)

Σαν default επιλογή του optimizer, χρησιμοποιεί Broadcast Hash Join για το πρώτο join και Range Join για το δεύτερο, ολοκληρώνοντας την εκτέλεση σε **31.09 δευτερόλεπτα**.

Στη συνέχεια προσπαθούμε να αναγκάσουμε το Spark να χρησιμοποιήσει διαφορετικές στρατηγικές, και συγκεκριμένα τις: ***BROADCAST***, ***MERGE***, ***SHUFFLE_HASH***, ***SHUFFLE_REPLICATE_NL***.

Για το δεύτερο join που αφορά την ένωση με βάση τα γεωχωρικά δεδομένα, το Sedona επιτρέπει 2 στρατηγικές join: **Range Join** και **Broadcast Index Join**.

Οι χρόνοι εκτέλεσης για καθένα απ’τα προαναφερθέντα join strategies όσον αφορά το πρώτο join φαίνονται στον πίνακα:

Join Strategy	Time (seconds)
Broadcast Hash Join	33.36
Sort Merge Join	32.95
Shuffle Hash Join	28.10
Shuffle Replicate Nested Loop (Cartesian Product)	27.93

Για τις 2 καλύτερες στρατηγικές, SHUFFLE_HASH και SHUFFLE_REPLICATE_NL, δοκιμάζουμε στο δεύτερο join να χρησιμοποιήσουμε Broadcast Index Join αντί για Range Join:

First Join Strategy	Second Join Strategy	Time (seconds)
Shuffle Hash Join	Range Join	28.10
Shuffle Hash Join	Broadcast Index Join	28.80
Shuffle Replicate Nested Loop	Range Join	27.93
Shuffle Replicate Nested Loop	Broadcast Index Join	30.92

Οι χρόνοι που παρατηρούμε έχουν γενικά μικρές διαφορές μεταξύ τους, ωστόσο φαίνεται ότι για το γεωχωρικό join έχει καλύτερα αποτελέσματα το optimized **Range Join** του Sedona σε σχέση με το broadcast, πιθανόν επειδή το dataframe που κάνουμε broadcast δεν είναι αρκετά μικρό ώστε να επωφεληθεί ο χρόνος εκτέλεσης από αυτό.

Όσον αφορά το πρώτο join, και πάλι τα αποτελέσματα δεν είναι ξεκάθαρα, καθώς το join αυτό έχει σχετικά μικρό κόστος, συγκριτικά με το κόστος του δεύτερου.

Ζητούμενο 4

Υλοποιήσαμε το **Query 4** χρησιμοποιώντας το **DataFrame** API. Εκτελούμε με 2 executors και τα εξής configurations:

- 1 core / 2 GB μνήμη
- 2 cores / 4 GB μνήμη
- 4 cores / 8 GB μνήμη

Το πλήθος εγκλημάτων ανά φυλετικό προφίλ των καταγεγραμμένων θυμάτων στις 3 περιοχές με το υψηλότερο και το χαμηλότερο κατά κεφαλήν εισόδημα φαίνεται παρακάτω:

Υψηλότερο Εισόδημα		Χαμηλότερο Εισόδημα	
Victim Descent	**#**	**Victim Descent**	**#**
White	649	Hispanic/Latin/Mexican	2,815
Other	72	Black	761
Hispanic/Latin/Mexican	66	White	330
Unknown	38	Other	187
Black	37	Other Asian	113
Other Asian	21	Unknown	22
Chinese	1	American Indian/Alaskan Native	21
American Indian/Alaskan Native	1	Korean	5
		Chinese	3
		AsianIndian	1
		Filipino	1

Οι χρόνοι εκτέλεσης για καθένα απ’τα προαναφερθέντα configurations φαίνονται στον πίνακα:

Cores	**Memory (GB)**	**Time (seconds)**
1	2	79.66
2	4	62.67
4	8	60.42

Παρατηρούμε ότι η βελτίωση είναι σημαντική κατά την αύξηση από **1 core/2 GB** σε **2 cores/4 GB**, από τα **79.66 δευτερόλεπτα** στα **62.67 δευτερόλεπτα**. Αυτό σημαίνει ότι τα περισσότερα cores εκμεταλλεύονται την παράλληλη εκτέλεση εργασιών, επιτρέποντας σε

μέρη της υλοποίησης όπως τα JOIN και τα geospatial queries να εκτελεστούν γρηγορότερα, και ότι η αύξηση της μνήμης επιτρέπει την αποθήκευση περισσότερων δεδομένων στη RAM.

Αντίθετα, βλέπουμε ότι η αύξηση σε **4 cores/8 GB** δεν έφερε μεγάλη αύξηση στην επίδοση του συστήματος, αφού ο χρόνος παρέμεινε σχεδόν ο ίδιος στα **60.42 δευτερόλεπτα**. Αυτό πιθανόν να υποδηλώνει ότι το query αρχίζει να περιορίζεται από άλλους παράγοντες όπως το I/O και δεν επωφελείται περαιτέρω από την αύξηση των πόρων.

Ζητούμενο 5

Υλοποιήσαμε το **Query 5** χρησιμοποιώντας το **DataFrame** API. Εκτελούμε με συνολικά 8 cores και 16 GB μνήμης με τα εξής configurations:

- 2 executors x 4 cores / 8 GB μνήμη
- 4 executors x 2 cores / 4 GB μνήμη
- 8 executors x 1 core / 2 GB μνήμη

Ο αριθμός εγκλημάτων που έλαβαν χώρα πλησιέστερα σε κάθε αστυνομικό τμήμα καθώς και η μέση απόσταση των εγκλημάτων από αυτό, φαίνονται στον παρακάτω πίνακα, ταξινομημένα σε φθίνουσα σειρά ως προς το πλήθος των περιστατικών:

division	avg_distance	#
HOLLYWOOD	2.0762639602	224,340
VAN NUYS	2.9533697428	210,134
SOUTHWEST	2.1913988058	188,901
WILSHIRE	2.5926655330	185,996
77TH STREET	1.7165449720	171,827
OLYMPIC	1.7236036972	170,897
NORTH HOLLYWOOD	2.6430060941	167,854
PACIFIC	3.8500706553	161,359
CENTRAL	0.9924764375	153,871
RAMPART	1.5345341879	152,736
SOUTHEAST	2.4218662159	152,176
WEST VALLEY	3.0356712163	138,643
TOPANGA	3.2969548418	138,217
FOOTHILL	4.2509217084	134,896
HARBOR	3.7025615994	126,747
HOLLENBECK	2.6801812377	115,837
WEST LOS ANGELES	2.7924572890	115,781
NEWTON	1.6346357397	111,110
NORTHEAST	3.6236655246	108,109
MISSION	3.6909426143	103,355
DEVONSHIRE	2.8247654128	77,094

Οι χρόνοι εκτέλεσης για καθένα απ’τα προαναφερθέντα configurations φαίνονται στον πίνακα:

Executors	Cores	Memory (GB)	Time (seconds)
2	4	8	41.76
4	2	4	19.32
8	1	2	11.18

Το configuration με την καλύτερη απόδοση είναι η χρήση 8 executors, με 1 core και 2 GB μνήμης ανά executor, πετυχαίνοντας χρόνο ολοκλήρωσης του query σε μόλις **11.18 δευτερόλεπτα**.

Η εκτέλεση του query φαίνεται να επωφελείται σημαντικά από την παράλληλη εκτέλεση καθώς αυξάνεται το πλήθος των εκτελεστών. Συγκεκριμένα, παρατηρούμε ότι ο χρόνος εκτέλεσης μειώνεται σχεδόν γραμμικά με την αύξηση του πλήθος των executors, καθώς για **2,4 και 8 executors** πετυχαίνουμε χρόνους εκτέλεσης **41.76, 19.32 και 11.18 δευτερόλεπτα** αντίστοιχα.

Αντιθέτως, η μείωση των πυρήνων και της μνήμης ανά executor που επιφέρει η αύξησή του πλήθους τους (καθώς κρατάμε σταθερά σύνολικά 8 cores και 16 GB μνήμης), καθώς και η αύξηση του κόστους επικοινωνίας μεταξύ τους, φαίνεται να επηρεάζει ελάχιστα το συνολικό χρόνο εκτέλεσης.