Προχωρημένα Θέματα Βάσεων Δεδομένων
9ο εξάμηνο ΗΜΜΥ
Εξαμηνιαία Εργασία

Ιωάννης-Παναγιώτης Κούτρας
Βασίλειος-Παναγιώτης Μηλίγγας

Ζητούμενο 1:

Μετρώντας το χρόνο ολοκλήρωσης των υπολογισμών παρατηρούμε πολύ καλύτερη επίδοση με την χρήση Dataframe APIs σε σχέση με τα RDD. Ένας λόγος είναι οτι τα Dataframes επωφελούνται απο την χρήση του Catalyst Optimizer, που πρόκειται για έναν query optimizer του Spark τον οποίο δεν διαθέτουν τα RDDs καθώς είναι πολύ πιο low-level απο τα dataframes. Ο optimizer δημιουργεί ένα query plan που περιγράφει τις ενέργειες που πρέπει να γίνουν (logical plan), το κάνει optimize (π.χ. αναδιατάσσει την σειρά εντολών με στόχο την μέγιστη αποδοτικότητα), και έπειτα δημιουργεί το physical plan που υποδεικνύει πως θα εκτελεστεί βέλτιστα το query βάσει των διαθέσιμων πόρων. Χρησιμοποιώντας df.explain(mode='formatted') μπορούμε να δούμε τη βέλτιστη στρατηγική που επιλέγει ο Catalyst. Στην δικιά μας περίπτωση:

```
== Physical Plan ==
AdaptiveSparkPlan (8)
+- Sort (7)
   +- Exchange (6)
      +- Union (5)
         :- Filter (2)
         :  +- Scan csv  (1)
         +- Filter (4)
            +- Scan csv  (3)


(1) Scan csv
Output [3]: [DR_NO#3731, Crm Cd Desc#3740, Vict Age#3742]
Batched: false
Location: InMemoryFileIndex [s3://initial-notebook-data-bucket-dblab-905418150721/CrimeData/Crime_Data_from_2010_to_2019_20241101.csv]
PushedFilters: [IsNotNull(Crm Cd Desc), IsNotNull(Vict Age)]
ReadSchema: struct<DR_NO:string,Crm Cd Desc:string,Vict Age:string>

(2) Filter
Input [3]: [DR_NO#3731, Crm Cd Desc#3740, Vict Age#3742]
Condition : (((isnotnull(Crm Cd Desc#3740) AND isnotnull(Vict Age#3742)) AND RLIKE(Crm Cd Desc#3740, AGGRAVATED ASSAULT)) AND (cast(Vict Age#3742 as int) < 18))

(3) Scan csv
Output [3]: [DR_NO#3809, Crm Cd Desc#3818, Vict Age#3820]
Batched: false
Location: InMemoryFileIndex [s3://initial-notebook-data-bucket-dblab-905418150721/CrimeData/Crime_Data_from_2020_to_Present_20241101.csv]
PushedFilters: [IsNotNull(Crm Cd Desc), IsNotNull(Vict Age)]
ReadSchema: struct<DR_NO:string,Crm Cd Desc:string,Vict Age:string>

(4) Filter
Input [3]: [DR_NO#3809, Crm Cd Desc#3818, Vict Age#3820]
Condition : (((isnotnull(Crm Cd Desc#3818) AND isnotnull(Vict Age#3820)) AND RLIKE(Crm Cd Desc#3818, AGGRAVATED ASSAULT)) AND (cast(Vict Age#3820 as int) < 18))

(5) Union

(6) Exchange
Input [3]: [DR_NO#3731, Crm Cd Desc#3740, Vict Age#3742]
Arguments: rangepartitioning(Vict Age#3742 DESC NULLS LAST, 1000), ENSURE_REQUIREMENTS, [plan_id=4869]

(7) Sort
Input [3]: [DR_NO#3731, Crm Cd Desc#3740, Vict Age#3742]
Arguments: [Vict Age#3742 DESC NULLS LAST], true, 0

(8) AdaptiveSparkPlan
Output [3]: [DR_NO#3731, Crm Cd Desc#3740, Vict Age#3742]
Arguments: isFinalPlan=false
```

Παρατηρούμε οτι ο Catalyst επέλεξε να εκτελεστούν τα filters αμέσως μετά την φόρτωση των datasets, για να ελαχιστοποιηθεί ο όγκος των δεδομένων που θα επεξεργαστούν στη συνέχεια. Αντίθετα, τα RDDs δεν έχουν query optimizer οπότε το query εκτελείται με την σειρά που γράφτηκε στον κώδικα.

Ακόμη, τα Dataframes επωφελούνται απο το γεγονός οτι η αναπαράστασή τους γίνεται με tables που ακολουθούν ένα schema. Κατά την φόρτωση των δεδομένων το Spark δημιουργεί ένα schema για το dataframe διαβάζοντας το header του csv και αναγνωρίζοντας τον τύπο δεδομένων κάθε στήλης. Η ύπαρξη του schema στην περίπτωση μας επιτρέπει να επιλεχθούν απευθείας οι στήλες “DR_NO”, “Crm Dc Desc”, “Vict Age” που θέλαμε με την χρήση του .select(). Αντιθέτως, τα RDDs αντιμετωπίζουν τα δεδομένα απλά ως συλλογές(tuples), και άρα έπρεπε manually να επιλέξουμε τις επιθυμητές στήλες με την χρήση της map. Με αυτόν τον τρόπο γίνεται επεξεργασία ολόκληρου του dataset για την εξαγωγή, γεγονός που καθιστά την διαδικασία πιο αργή.

Τέλος το Dataframe API αξιοποιεί το Tungsten Execution Engine για την εκτέλεση του physical plan, ένα low-level optimization framework που στοχεύει στην βελτίωση της αποδοτικότητας της CPU και της μνήμης. Με το Tungsten τα δεδομένα επεξεργάζονται σε batches και όχι γραμμή προς γραμμή μειώνοντας το overhead στη CPU (ωστόσο στην περίπτωση μας δεν εφαρμόζεται αυτό αφού βλέπουμε πως Batched=False). Επιπλέον παράγει optimized JVM bytecode για το query, και χρησιμοποιεί off-heap μνήμη (μνήμη εκτός του heap της Java) μειώνοντας το overhead απο τον garbage collector.

//Να τσεκαρουμε αν το timer ειναι μετα το show
Σημειώνουμε οτι λόγω του lazy evaluation του spark τα query plans εκτελούνται στο σημείο που κάνουμε show()/take() τις ηλικιακές ομάδες.

Ζητούμενο 2:

α) DataFrame vs SQL API

Υλοποιήσαμε το Query 2 χρησιμοποιώντας τα DataFrame και SQL APIs, μέσω των εξής βημάτων:

Εξηγάγαμε τα year και τα λοιπά ζητούμενα από το dataset.
Υπολογίσαμε το closed case rate με βάση τα case statuses, όπου σύμφωνα με τις οδηγίες στο φόρουμ, οι υποθέσεις που ήταν "IC" (Investigation Continued) ή "UNK" (Unknown) θεωρήθηκαν ως "not closed".
Τέλος, ταξινομήσαμε τα precincts με βάση το closed case rate ετησίως, διαλέγοντας τα 3 κορυφαία.

Οι σχετικοί χρόνοι εκτέλεσης είναι:

DataFrame API: 8.91 seconds
SQL API: 17.55 seconds

Η υλοποίηση με το DataFrame API παρατηρήθηκε αρκετά πιο γρήγορη από εκείνη με το SQL API, έχοντας περίπου 50% μειωμένο χρόνο εκτέλεσης.
Αυτό ενδεχομένως οφείλεται στο γεγονός ότι το DataFrame API αξιοποιεί το optimization του Spark πιο αποτελεσματικά, αποφεύγοντας το επιπλέον κόστος ανάλυσης (parsing overhead) που σχετίζεται με το SQL API.

β) CSV vs Parquet Data Formats

Στο β) μέρος, μετατρέψαμε το dataset σε ένα ενιαίο αρχειό Parquet το οποίο αποθηκεύσαμε στο S3 bucket της ομάδας μας. Στη συνέχεια,

εκτελέσαμε το query 2 τόσο στο CSV όσο και στο Parquet version του dataset, ώστε να συγκρίνουμε τους χρόνους εκτέλεσης. Επιλέξαμε το Dataframe API.

Execution Times:

CSV Format: 12.84 seconds
Parquet Format: 7.27 seconds

Υπολογίσαμε ότι το Parquet format είναι περίπου 43% γρηγορότερο από το CSV format.
Η βελτίωση στην ταχύτητα οφείλεται στο ότι το Parquet format είναι "column-oriented", το οποίο βοηθά στην πιο αποδοτική εξαγωγή μόνο των σχετικών στηλών και υποστηρίζει πιο προχωρημένες βελτιστοποιήσεις όπως το predicate pushdown.

Ζητούμενο 3:

Στην υλοποίησή μας πραγματοποιήθηκαν 3 joins: Των dataframes 2010 census blocks, Median Household Income με βάση το Zip Code, των Crime Data και 2010 census blocks με βάση το geometry, και τέλος των αποτελεσμάτων μετά απο αυτά τα 2 joins. Τα join strategies που εξετάζουμε παρακάτω είναι τα εξής:

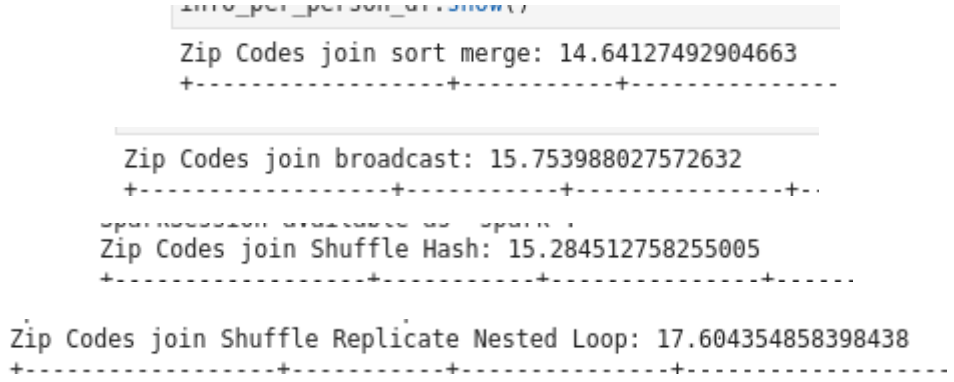
- Broadcast join: Αποδοτικό join strategy σε περιπτώσεις όπου το ένα dataset είναι αρκετά μικρό ώστε να χωράει εξ ολοκλήρου στην μνήμη των worker nodes. Το μικρό dataset αντιγράφεται σε κάθε worker node του cluster, και έπειτα κάθε worker node, τοπικά, εκτελεί το join (χρησιμοποιώντας μια hash function ή ένα nested loop) με τα partitions του μεγάλου συνόλου δεδομένων που καταφθάνουν στον worker. Έτσι αποφεύγεται το shuffling μεγάλου όγκου δεδομένων στο cluster, το οποίο μπορεί να αποτελέσει bottleneck στα κατανεμημένα συστήματα.

- Sort-Merge Join: Στο Sort-Merge Join, αρχικά γίνεται shuffle στα δύο datasets, ώστε τα δεδομένα με το ίδιο join key να καταλήξουν στο ίδιο partition. Κατά τη φάση του shuffle, τα partitions ταξινομούνται με βάση το join key. Στη συνέχεια, τα ταξινομημένα partitions με το ίδιο join key αποστέλλονται στον ίδιο worker node, όπου οι γραμμές τους ενώνονται όταν ικανοποιείται ισότητα του join key. Αυτό το join strategy αποτελεί το default που χρησιμοποιείται απο το Spark όταν κανένα απο τα 2 datasets δεν χωράει ολόκληρο στη μνήμη ενός worker.

- Shuffle Hash Join: Γίνεται shuffling ακριβώς όπως στο sort-merge ώστε γραμμές με το ίδιο κλειδί να σταλούν στο ίδιο partition. Σε κάθε worker node φορτώνονται τα partitions και δημιουργείται ένα hash table για ένα απο τα datasets. Έπειτα οι γραμμές του άλλου dataset ταιριάζονται μέσω του hash table με βάση το join key.

- Shuffle Replicate Nested Loop Join: Ένα απο τα 2 datasets αντιγράφεται σε κάθε partition και στη συνέχεια γίνεται γίνεται μια nested loop σύγκριση των 2 datasets στο join key ώστε να βρεθούν οι γραμμές που ταιριάζουν. Αυτό το μοντέλο είναι ακριβό υπολογιστικά.

Δοκιμάσαμε τις διαφορετικές στρατηγικές join στο πρώτο join που αφορά την ένωση των census blocks με το median household income by zip code και οι χρόνοι που λάβαμε είναι οι εξής:



Παρατηρούμε πως το sort-merge join αναδείχθηκε το πιο αποδοτικό με 2ο να είναι το shuffle hash και να ακολουθούν τα broadcast και shuffle replicated. Γενικά τα datasets δεν είναι ιδιαίτερα μεγάλα και οι αποκλίσεις στα 3 πρώτα join είναι μικρές. Κάνοντας explain() στο joined dataset χωρίς να δώσουμε hint για συγκεκριμένο join, παρατηρούμε οτι η default επιλογή από το Spark ήταν το broadcast join. Αυτή η επιλογή είναι λογική αφού το Median Household Income by Zip Code dataset είναι εξαιρετικά μικρό (μόλις μερικά Kbs) οπότε συμφέρει να αντιγραφεί στη μνήμη κάθε executor. Τέλος παρατηρούμε πως το Shuffle Replicate Nested Loop join απέχει απο τα άλλα σε επίδοση λόγω του μεγάλου υπολογιστικού του κόστους.

Δοκιμάσαμε τις διαφορετικές join στρατηγικές στο τελικό join που έγινε μεταξύ των 2 dataframes που υπολογίστηκαν για το ερώτημα (μέσο εισόδημα ανά άτομο και εγκλήματα ανά άτομο). Τα αποτελέσματα φαίνονται παρακάτω:

Παρατηρούμε πολύ καλύτερη επίδοση για το broadcast join. Πράγματι και τα 2 dataframes είναι αρκετά μικρά (μόλις 139 rows το καθένα αφού κάθε row αφορά ένα community) οπότε μπορεί οποιοδήποτε απο τα 2 να αντιγραφεί στη μνήμη των executors. Ωστόσο κάνοντας explain χωρίς κάποιο hint για το join, παρατηρούμε πως το Spark επέλεξε την default join επιλογή που είναι το sort-merge join. Πάλι εδώ παρατηρούμε πως το shuffle replicate nested loop είναι το αργότερο.

Για το άλλο join που γίνεται με βάση το geometry, κάνοντας explain() παρατηρούμε οτι το Spark επιλέγει τη στρατηγική range join. Το γεγονός αυτό δικαιολογείται αφού στη συγκεκριμένη

περίπτωση εξετάζεται αν ένα σημείο (συντεταγμένες εγκλήματος) βρίσκεται σε ένα γεωγραφικό εύρος (census block). Τα dataset που ενώνονται είναι μεγάλα, και κάνοντας collect() για να μετρήσουμε την επίδοση του join είναι πιθανό να τελειώνει η μνήμη του driver. Για αυτό το λόγο δεν δοκιμάσαμε άλλα join strategies για αυτό το join.

Ζητούμενο 4:

Παρακάτω φαίνονται τα αποτελέσματα της κλιμάκωσης των υπολογιστικών πόρων:

```
Executor Instances: 2
Executor Memory: 2g
Executor Cores: 1
Elapsed time: 14.232461214065552
+                  +   +

 Executor Instances: 2
 Executor Memory: 4g
 Executor Cores: 2
 Elapsed time: 13.613950490951538
 '                  '   '

Executor Instances: 2
Executor Memory: 8g
Executor Cores: 4
Elapsed time: 12.111953735351562
```

Παρατηρούμε οτι όσο αυξάνεται ο αριθμός των cores και η μνήμη σε κάθε executor, το query εκτελείται πιο γρήγορα. Το γεγονός αυτό ήταν αναμενόμενο αφού με την αύξηση των cores στους 2 executors αυξάνεται ο παραλληλισμός στην επεξεργασία των partitions. Έτσι περισσότερα tasks μπορούν να εκτελεστούν ταυτόχρονα οδηγώντας σε ταχύτερα αποτελέσματα.

Όταν εφαρμόζουμε scaling στη μνήμη ο χρόνος εκτέλεσης βελτιώνεται καθώς περισσότερα δεδομένα αποθηκεύονται στην μνήμη και άρα μειώνεται το Disk I/O.

Ωστόσο, το μέγεθος των datasets που επεξεργαζόμαστε δεν είναι ιδιαίτερα μεγάλο (~675 MB) και απο ένα σημείο και μετά η αύξηση της μνήμης των executors μπορεί να είναι περιττή. Για αυτό το λόγο μάλλον δεν παρατηρούμε ιδιαίτερα μεγάλη επίδραση των ταυτόχρονων κλιμακώσεων μνήμης και cores.

Ζητούμενο 5:

Στο 5ο ζητούμενο, καλούμαστε να υπολογίσουμε τα εξής:

Τον συνολικό αριθμό από reported crimes για κάθε police division.
Τη μέση απόσταση για τα παραπάνω crimes σε σχέση με τα αντίστοιχα police stations και τέλος
να ταξινομήσουμε τα αποτελέσματα με βάση τον αριθμό τους σε φθίνουσα σειρά.

Σε σχέση με τα datasets και το preprocessing που πραγματοποιήθηκε, κάναμε filter σύμφωνα με τη σχετική οδηγία τα Invalid coordinates: (LAT, LON = 0,0).

Τα crime datasets έγιναν joined με το police station dataset με βάση τα area codes (AREA and PREC), ενώ για κάθε έγκλημα η απόσταση από το αντίστοιχο police station υπολογίστηκε μέσω των geospatial tools.

Κάναμε group τα αποτελέσματα με βάση το police division για να υπολογίσουμε:
Τον συνολικό αριθμό total number of crimes.
Τη μέση απόσταση τους από τα police stations.

Στη συνέχεια ταξινομήσαμε με βάση το total number of crimes σε φθίνουσα σειρά, σύμφωνα με το υπόδειγμα του Πίνακα 4 της εκφώνησης.

Το query 5 εκτελέστηκε με 3 διαφορετικά Spark configurations:

```
Executor Instances: 2
Executor Memory: 8g
Executor Cores: 4
Elapsed time: 9.37587022781372
+---------------+---------------

Executor Instances: 4
Executor Memory: 4g
Executor Cores: 2
Elapsed time: 8.37051272392273
+---------------+---------------+

Executor Instances: 8
Executor Memory: 2g
Executor Cores: 1
Elapsed time: 8.604315996170044
```

Ερμηνεύουμε τα αποτελέσματα ως εξής:

Το configuration με 4 Executors × 2 Cores/4 GB ήταν εκείνο με το γρηγορότερο execution time (8.37 seconds), λόγω της ισορροπίας μεταξύ της παράλληλης επεξεργασίας και της κατανομής μνήμης. Κατά αυτόν τον τρόπο είχαμε αποδοτικό distribution των tasks χωρίς σημαντικό overhead. Αντίθετα, το configuration με 2 Executors × 4 Cores/8 GB (9.38 seconds) είχε λιγότερους executors και άρα περιορισμένη δυνατότητα για παράλληλη επεξεργασία παρά την επαρκή μνήμη, το οποιο είχε ως αποτέλεσμα χειρότερη επίδοση. Τέλος, μια ερμηνεία για το αποτέλεσμα για το configuration με 8 Executors × 1 Core/2 GB (8.60 seconds) είναι πως εισήγαγε υψηλότερο communication overhead ανάμεσα στους executors και περιορισμένο task parallelism μέσα σε κάθε executor. Ως αποτέλεσμα, είχαμε ελαφρώς χειρότερη επίδοση.