Διαχείριση Δεδομένων Μεγάλης Κλίμακας
Εξαμηνιαία Εργασία

ΠΑΠΟΥΛΙΑ ΕΥΓΕΝΙΑ 03400228

ΧΟΝΤΖΑΚΗΣ ΔΙΟΝΥΣΙΟΣ 03400238

ε.δε.μ^2

Περιεχόμενα

Εισαγωγικά Ερωτήματα

1. Εγκατάσταση και Διαμόρφωση HDFS

Για την υλοποίηση των επόμενων ερωτημάτων θα χρησιμοποιηθούν εικονικά μηχανήματα από τον Okeanos-Knossos. Συγκεκριμένα, έχουμε στη διάθεσή μας δύο μηχανήματα, το καθένα με 2 πυρήνες και 4GB μνήμη. Για τον συντονισμό τους, θα χρησιμοποιήσουμε το σύστημα κατανεμημένης επεξεργασίας δεδομένων Apache Spark, καθώς και το κατανεμημένο σύστημα αρχείων Hadoop Distributed File System (HDFS).
Μετά την επιτυχή διαμόρφωση και εγκατάσταση των προαναφερθέντων συστημάτων, διαθέτουμε και τις web εφαρμογές των HDFS και Spark Job History Server, όπως φαίνεται και στα επόμενα screenshots.

Overview 'master:9000' (✔active)

Started:	Sun May 19 16:40:11 +0300 2024
Version:	3.3.6, r1be78238728da9266a4f88195058f08fd012bf9c
Compiled:	Sun Jun 18 11:22:00 +0300 2023 by ubuntu from (HEAD detached at release-3.3.6-RC1)
Cluster ID:	CID-300c3530-8da4-4dcf-8b14-28478b116dfb
Block Pool ID:	BP-780536941-192.168.0.1-1716125988854

Summary

Security is off.

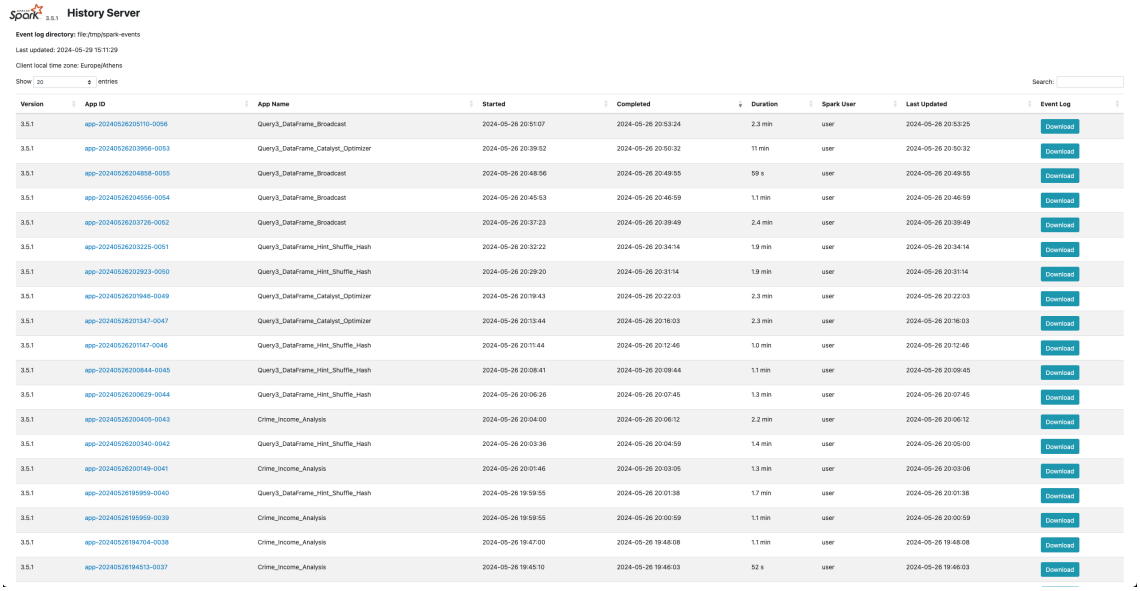
Safemode is off.

79 files and directories, 77 blocks (77 replicated blocks, 0 erasure coded block groups) = 156 total filesystem object(s).

Heap Memory used 54.2 MB of 105 MB Heap Memory. Max Heap Memory is 878.5 MB.

Non Heap Memory used 90.22 MB of 92.59 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	58.8 GB
Configured Remote Capacity:	0 B
DFS Used:	3.62 GB (6.16%)
Non DFS Used:	14.41 GB
DFS Remaining:	37.74 GB (64.18%)
Block Pool Used:	3.62 GB (6.16%)
DataNodes usages% (Min/Median/Max/stdDev):	2.49% / 9.83% / 9.83% / 3.67%
Live Nodes	2 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion (including replicas)	0
Block Deletion Start Time	Sun May 19 16:40:11 +0300 2024
Last Checkpoint Time	Wed May 29 14:36:14 +0300 2024
Enabled Erasure Coding Policies	RS-6-3-1024k



Να τονίσουμε πως η δημόσια IP του master node είναι 83.212.80.48 συνεπώς μπορούμε να αναζητήσουμε τις web εφαρμογές ως:

Hadoop : $83.212.80.48 : 9870$ Master Node : $83.212.80.48 : 8080$

History Server : $83.212.80.48 : 18080$ Slave Node : $83.212.80.48 : 8081$

2. Διαχείριση Δεδομένων

Δημιουργήσαμε ένα directory στο HDFS στο οποίο αποθηκεύσαμε τα απαραίτητα αρχεία δεδομένων για την εργασία. Συγκεκριμένα το path που επιλέξαμε για να αποθηκεύσουμε τα αρχεία είναι /home/user/hadoop_data. Για την συλλογή των αρχείων .csv χρησιμοποιήσαμε τις εντολές:

```
$ wget --no-check-certificate https://data.lacity.org/api/views/63jg-8b9z/rows.csv?
    ↪ accessType=DOWNLOAD -O Crime_Data_from_2010_to_2019.csv

$ wget --no-check-certificate -P data https://data.lacity.org/api/views/2nrs-mtv8/
    ↪ rows.csv?accessType=DOWNLOAD -O Crime_Data_from_2020_to_Present.csv

$ wget --no-check-certificate -P data https://data.lacity.org/api/views/2nrs-mtv8/
    ↪ rows.csv?accessType=DOWNLOAD -O revgecoding.csv

$ scp "/Users/user/Downloads/LAPD_Police_Stations_-3946316159051949741.csv" user@83
    ↪ .212.80.48:/home/user/data
```

> Σημείωση
> Το τελευταίο σύνολο δεδομένων, μας δόθηκε μέσω mail και για αυτό ανέβηκε στον master χειροκίνητα και όχι μέσω web.

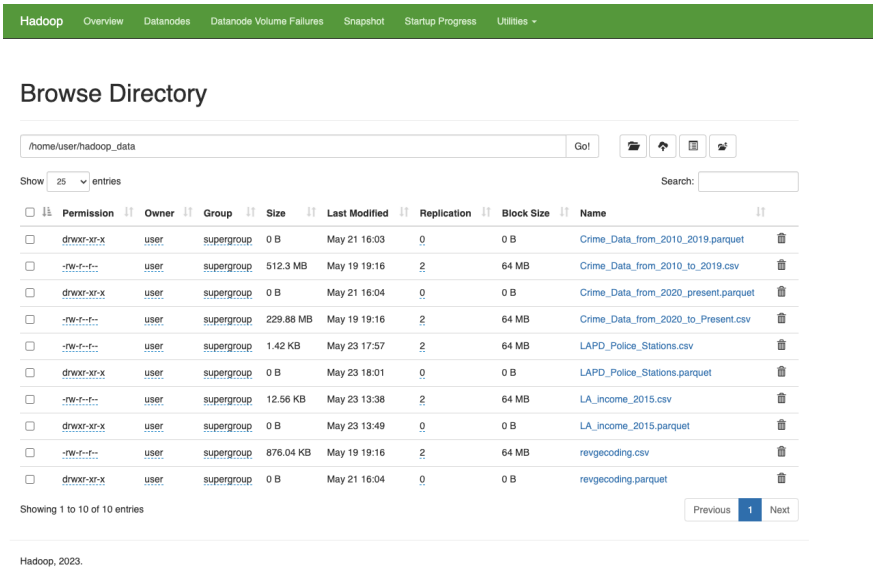
Έπειτα, δημιουργήθηκε ένα directory στο Hadoop για να αποθηκευτούν εκέι τα .csv αρχεία ως:

```
$ hadoop fs -mkdir -p ~/hadoop_data
```

Τέλος, μεταφέρθηκαν τα αρχεία στο Hadoop μέσω των:

```
$ hadoop fs -put Crime_Data_from_2010_to_2019.csv Crime_Data_from_2020_to_Present.csv
    ↪  LAPD_Police_Stations_-3946316159051949741.csv revgecoding.csv ~/hadoop_data
```

Στο επόμενο screenshot παρουσιάζεται η διαθεσιμότητα των εν λόγω αρχείων στο HDFS.



Να σημειώσουμε πως μαζί με τα .csv αρχεία, υπάρχουν και τα .parquet αρχεία που μας ζητήθηκε να δημιουργήσουμε.

Query 1

3. Υλοποίηση με DataFrame και SQL

Για το πρώτο query μας ζητείται να βρούμε, για κάθε έτος, τους 3 μήνες με τον υψηλότερο αριθμό καταγεγραμμένων εγκλημάτων και να τους παρουσιάσουμε με συγκεκριμένη διάταξη. Συγκεκριμένα, ζητείται να τυπωθούν οι μήνες αυτοί ανά έτος, ο συνολικός αριθμός περιστατικών και η θέση του συγκεκριμένου μήνα στην κατάταξη μέσα στο αντίστοιχο έτος. Τα αποτελέσματα πρέπει να δοθούν σε σειρά αύξουσα ως προς το έτος και φθίνουσα ως προς τον αριθμό καταγραφών.

Αυτό υλοποιήθηκε με 4 διαφορετικές μεθόδους. Συγκεκριμένα, για κάθε υλοποίηση επιλέξαμε μεταξύ DataFrame ή SQL APIs, και διαλέξαμε μεταξύ csv ή parquet format αρχείων. Για κάθε υλοποίηση συλλέξαμε χρόνους εκτέλεσης από τον History Server, έχοντας προσέξει να καθαρίσουμε την cache.
Οι χρόνοι εκτέλεσης για κάθε μία από αυτές τις μεθόδους είναι οι εξής:

API	Format	Χρόνος (min)
DataFrame	csv	2.8
DataFrame	parquet	1.2
SQL	csv	2.0
SQL	parquet	1.4

Σχολιασμός:

Αναλύοντας τα αποτελέσματα, σκοπός μας είναι να επιλέξουμε τη βέλτιστη μέθοδο για την υλοποίηση του ερωτήματος. Οι δύο επιλογές που αξιολογούμε είναι οι APIs των DataFrame και της SQL. Και οι δύο μέθοδοι χαρακτηρίζονται από την πινακική δομή τους, δηλαδή την οργάνωση των δεδομένων σε γραμμές και στήλες. Ωστόσο, καθεμία εστιάζει σε διαφορετικά κομμάτια της επεξεργασίας δεδομένων.

Τα DataFrame είναι εύκολα κατανοητά και σχεδιασμένα για φιλική προς τον χρήστη χρήση. Αντίθετα, η SQL επικεντρώνεται στη διαχείριση δεδομένων μεγάλου όγκου και στην εκτέλεση σύνθετων ερωτημάτων. Επομένως, οι μηχανές SQL συνήθως περιλαμβάνουν optimizers που βελτιστοποιούν τα ερωτήματα για ταχύτερη εκτέλεση, γεγονός που επιβεβαιώνεται από τους χρόνους εκτέλεσης. Παρατηρούμε ότι, για csv τύπο αρχείου, η εκτέλεση μέσω του SQL API είναι ταχύτερη, με χρόνους εκτέλεσης 2.0 min έναντι 2.8 min. Για τα parquet αρχεία βέβαια, έχουμε αντίθετα αποτελέσματα, με το DataFrame API να παρέχει γρηγορότερη υλοποίηση με 1.2 έναντι 1.4 για SQL.

Όσον αφορά τη μορφή των αρχείων, το CSV format είναι σχεδιασμένο με γενικότερη προοπτική, παραμένοντας επεξεργάσιμο από διάφορα λογισμικά και γλώσσες προγραμματισμού. Από την άλλη πλευρά, το Parquet format είναι σχεδιασμένο για αποδοτική αποθήκευση και ανάγνωση μεγάλων όγκων δεδομένων, προσφέροντας ταχύτερες αναγνώσεις και εγγραφές για συγκεκριμένες στήλες και συγκρατώντας μεταδεδομένα για ευκολότερη επεξεργασία. Συνοψίζοντας όλα τα παραπάνω, είναι εμφανές γιατί το αρχείο μορφής Parquet είναι πιο κατάλληλο από το αρχείο CSV. Αυτό το συμπέρασμα ενισχύεται από τους χρόνους εκτέλεσης των υλοποιήσεων, όπου το Parquet αρχείο χρειάστηκε 1.2 λεπτά σε σύγκριση με τα 2.8 λεπτά του αρχείου CSV με την υλοποίηση των DataFrame APIs και 1.4 λεπτά έναντι 2.00 λεπτών του CSV αρχείου με την υλοποίηση των SQL APIs.

Εν κατακλείδι ,η επιλογή της SQL μεθόδου API και της μορφής αρχείου DataFrame φαίνεται ότι είναι η βέλτιστη επιλογή για το συγκεκριμένο query.

Ο πίνακας αποτελεσμάτων του πρώτου Query παρατίθενται στον παρακάτω πίνακα:

Year	Month	CrimeTotal	Ranking
2010	1	19520	1
2010	3	18131	2
2010	7	17857	3
2011	1	18141	1
2011	7	17283	2
2011	10	17034	3
2012	1	17954	1
2012	8	17661	2
2012	5	17502	3
2013	8	17441	1
2013	1	16828	2
2013	7	16645	3
2014	10	17331	1
2014	7	17258	2
2014	12	17198	3
2015	10	19221	1
2015	8	19011	2
2015	7	18709	3
2016	10	19660	1
2016	8	19496	2
2016	7	19450	3
2017	10	20437	1
2017	7	20199	2
2017	1	19849	3
2018	5	19976	1
2018	7	19879	2
2018	8	19765	3
2019	7	19126	1
2019	8	18987	2
2019	3	18865	3
2020	1	18542	1
2020	2	17272	2
2020	5	17219	3
2021	10	19326	1
2021	7	18672	2
2021	8	18387	3
2022	5	20450	1
2022	10	20313	2
2022	6	20255	3
2023	10	20029	1
2023	8	20024	2
2023	1	19902	3
2024	1	18762	1
2024	2	17214	2
2024	3	16009	3

Query 2

4. Υλοποίηση με DataFrame και RDD

Έχοντας αναλύσει το DataFrame API παραπάνω, είναι κρίσιμο να επικεντρωθούμε αρχικά στο RDD API και ακολούθως να εξετάσουμε τις διαφορές μεταξύ τους. Το RDD API παρέχει ένα σύνολο εργαλείων υψηλού επιπέδου για την επεξεργασία και διαχείριση κατανεμημένων δεδομένων, το οποίο προσφέρεται από το Apache Spark. Ειδικότερα, το RDD API είναι ένα ισχυρό εργαλείο για την επεξεργασία δεδομένων μεγάλης κλίμακας, προσφέροντας απλότητα και ευελιξία καθώς επιτρέπει στους χρήστες να μπορούν να εκτελούν λειτουργίες όπως map, filter, reduce και πολλές άλλες σε μεγάλα σύνολα δεδομένων. Ακόμη,τα RDDs παρέχουν ανθεκτικότητα στα σφάλματα. Αν κάποιο μέρος των δεδομένων χαθεί, μπορεί να αναπαραχθεί από τα αρχικά δεδομένα χάρη στον κατανεμημένο χαρακτήρα τους.

Τόσο τα RDD όσο και τα DataFrame επωφελούνται από τις βελτιστοποιήσεις της SQL engine του Apache Spark, που περιλαμβάνει τον Catalyst Optimizer. Αυτός ο βελτιστοποιητής αναλαμβάνει την εκτέλεση των ερωτημάτων με τον πιο αποδοτικό τρόπο.

Είναι επίσης σημαντικό να αναφέρουμε τις διαφορές μεταξύ RDD και DataFrame. Τα RDD, από τη μια πλευρά, δεν έχουν schema, γεγονός που δεν παρέχει εγγυήσεις για τη δομή των δεδομένων. Αντίθετα, τα DataFrame διαθέτουν schema, το οποίο καθορίζει τη δομή των δεδομένων σε μορφή στήλης κάτι που εξασφαλίζει τη συνοχή των δεδομένων. Αυτή η αυστηρή δομή επιτρέπει στον Catalyst Optimizer να εκτελεί περισσότερες και καλύτερες βελτιστοποιήσεις σε σχέση με τα RDD, τα οποία δεν έχουν schema.

Η καθορισμένη μορφή στήλης στα DataFrame επιτρέπει επίσης άλλες βελτιστοποιήσεις, όπως η αποθήκευση σε στήλες και η εκτέλεση λειτουργιών σε στήλες πιο αποδοτικά από ό,τι σε γραμμές. Αυτό μπορεί να βελτιώσει την απόδοση σε πολλαπλά σενάρια χρήσης, ειδικά σε αναλύσεις μεγάλων δεδομένων και επεξεργασία δεδομένων.

Επομένως, η αυστηρότερη δομή των DataFrame τα καθιστά πιο αποδοτικά σε σύγκριση με τα RDD, εκμεταλλευόμενα πλήρως τις βελτιστοποιήσεις της SQL engine του Apache Spark.

Ο πίνακας με τους αντίστοιχους χρόνους εκτέλεσης είναι ο εξής:

API	Χρόνος (min)
DataFrame	2.0
RDD	2.2

Παρατηρούμε ότι οι παραπάνω σχολιασμοί συμφωνούν με τα αριθμητικά αποτελέσματα των υλοποιήσεων, επιβεβαιώνοντας την πιο αποδοτική φύση του DataFrame API.

Ο πίνακας των τμημάτων της ημέρας ανάλογα με τις καταγραφές εγκλημάτων που έλαβαν χώρα στο δρόμο ("STREET"), με φθίνουσα σειρά υλοποιώντας το Query 2 παρατίθενται παρακάτω :

TimeOfDay	CrimeCount
Νύχτα	283152
Βράδυ	202863
Απόγευμα	163451
Πρωί	63875

Query 3

5. Υλοποίηση με Catalyst Optimizer, Broadcast, Merge, Shuffle Hash, Shuffle Replicate NL

Σ' αυτό το ερώτημα μας ζητείται να υλοποιηθεί το τρίτο query με πέντε διαφορετικές μεθόδους join. Η πρώτη υλοποίηση πραγματοποιείται με τη στρατηγική που επιλέγει ο Catalyst Optimizer,βελτιστοποιητής που παρέχει το Spark SQL. Ο Catalyst Optimizer είναι υπεύθυνος για τη διαχείριση και βελτιστοποίηση τόσο του λογικού (logical) όσο και του φυσικού (physical) πλάνου εκτέλεσης για κάθε query. Η διαδικασία αυτή περιλαμβάνει την ανάλυση, τον μετασχηματισμό και τη βελτιστοποίηση του λογικού πλάνου εκτέλεσης, ακολουθούμενη από τη μετάφρασή του σε ένα αποδοτικό φυσικό πλάνο εκτέλεσης. Μέσω αυτών των βημάτων, ο Catalyst Optimizer διασφαλίζει ότι τα queries εκτελούνται με τον πιο αποδοτικό και αποτελεσματικό τρόπο, αξιοποιώντας πλήρως τις δυνατότητες της υποκείμενης υποδομής.

Οι επόμενες υλοποιήσεις πραγματοποιήθηκαν με τη χρήση της μεθόδου hint του DataFrame API, ώστε τα joins να εκτελεστούν παρακάμπτοντας τον Catalyst Optimizer. Έπειτα έχουμε τη μέθοδο hint("broadcast"). Γενικά το broadcast join είναι μια τεχνική που χρησιμοποιείται όταν ένα από τα δύο σύνολα δεδομένων που συμμετέχουν στο join είναι μικρό σε μέγεθος και μπορεί να μεταδοθεί (broadcast) σε όλους τους εκτελεστικούς κόμβους του cluster. Αυτό επιτρέπει την αποτελεσματική εκτέλεση του join χωρίς την ανάγκη για ανακατανομή δεδομένων (shuffle), κάτι που είναι δαπανηρό σε όρους χρόνου και πόρων. Η χρήση του hint(broadcast) είναι ένας τρόπος να δώσεις μια συμβουλή (hint) στον Spark να εκτελέσει ένα συγκεκριμένο join ως broadcast join. Αυτός ο τρόπος είναι πιο δηλωτικός και χρησιμοποιείται μέσα στο DataFrame API. Στη συνέχεια υλοποιήσαμε το Query 3 με τη μέθοδο hint("merge"). Για να είναι αποδοτική αυτή η μέθοδος, οι πίνακες πρέπει να είναι μεγάλοι και ταξινομημένοι. Η μέθοδος merge εκμεταλλεύεται την ταξινομημένη μορφή των πινάκων και ξεκινά τη διαδικασία του join ταξινομώντας τα δύο dataset με βάση ένα κλειδί. Στη συνέχεια, εκτελείται μια γραμμική σάρωση των δύο ταξινομημένων datasets για την εκτέλεση του join. Προχωράμε στη υλοποίηση με μέθοδο hint("shuffle_hash"). Αυτή η μέθοδος υποδεικνύει στον Spark να χρησιμοποιήσει το shuffle hash join για την εκτέλεση του join. Το shuffle hash join έχει υψηλή κλιμακωσιμότητα, καθώς είναι μια τεχνική join που χρησιμοποιείται όταν τα δεδομένα που συμμετέχουν στο join είναι αρκετά μεγάλα και δεν μπορούν να μεταδοθούν (broadcast) σε όλους τους κόμβους. Σε αυτή τη μέθοδο, τα δεδομένα ανακατανέμονται (shuffle) σε κόμβους με βάση το κλειδί του join, και στη συνέχεια εφαρμόζεται η λειτουργία hash για την εκτέλεση του join. Τα δεδομένα κατανέμονται σε διαφορετικά partitions στους κόμβους του cluster με βάση τα κλειδιά join. Σε κάθε partition, χρησιμοποιείται ένας πίνακας κατακερματισμού για το join. Το shuffle hash join είναι αποδοτικό για μεγάλα datasets που δεν είναι ταξινομημένα. Η τελευταία υλοποίηση πραγματοποιήθηκε με τη μέθοδο hint("shuffle_replicate_nl"). Αυτή η μέθοδος χρησιμοποιείται όταν οι άλλες μέθοδοι join δεν μπορούν να βελτιστοποιηθούν εύκολα. Ουσιαστικά, ένα από τα datasets αναπαράγεται και κατανέμεται σε όλους τους κόμβους του cluster. Στη συνέχεια, οι κόμβοι εκτελούν έναν nested loop για να εντοπίσουν τις αντιστοιχίες, κάτι που την καθιστά ιδιαίτερα αργή για μεγάλα datasets.

Ο πίνακας με τους αντίστοιχους χρόνους εκτέλεσης είναι ο εξής:

Μέθοδος	**Χρόνος (min)**
Catalyst Optimizer	2.9
Broadcast	3.5
Merge	2.9
Shuffle Hash	2.7
Shuffle Replicate NL	56

Σχολιασμός:

Είναι εμφανές ότι οι μέθοδοι Catalyst Optimizer, Broadcast, Merge και Shuffle Hash είναι σημαντικά ταχύτερες από τη Shuffle Replicate NL. Όπως αναλύσαμε προηγουμένως, η Shuffle Replicate NL χρησιμοποιείται ως έσχατη λύση όταν καμία άλλη μέθοδος δεν είναι αποδοτική, καθώς δεν διαθέτει κάποια "έξυπνη" στρατηγική και βασίζεται στη δύναμη των διαθέσιμων μηχανημάτων για μια brute force συνένωση. Συμπερασματικά, για την υλοποίηση αυτού του query, η επιλογή της κατάλληλης μεθόδου join θα πρέπει να γίνει μεταξύ των Catalyst Optimizer, Broadcast, Merge και Shuffle Hash.

Οι πίνακες αποτελεσμάτων για την καταγωγή των καταγεγραμμένων θυμάτων εγκλημάτων με φθίνουσα σειρά στο Los Angeles για το έτος 2015 στις 3 περιοχές με το υψηλότερο και χαμηλότερο εισόδημα ανά νοικοκυριό παρατίθενται παρακάτω αντίστοιχα :

victim descent	total victims
White	320
Other	104
Hispanic/Latin/Mexican	53
Unknown	26
Black	16
Other Asian	16

Table 1: Υψηλότερο εισόδημα ανά νοικοκυριό

victim descent	total victims
Hispanic/Latin/Mexican	1526
Black	1098
White	700
Other	390
Other Asian	101
Unknown	65
Korean	8
Japanese	3
American Indian/Alaskan Native	3
Chinese	2
Filipino	1

Table 2: Χαμηλότερο εισόδημα ανά νοικοκυριό

Query 4

6. Broadcast και Repartition Join

Broadcast

Ο Broadcast Join λειτουργεί αποκλειστικά σαν map-only job. Ειδικότερα, ανάλογα με το μέγεθος του μικρού πίνακα και το μέγεθος του split του μεγάλου πίνακα, υλοποιούνται δύο συναρτήσεις, η init ή η close. Έστω R, L οι δύο πίνακες με $|R| << |L|$. Αν το μέγεθος του R, είναι μικρότερο από το split του L, τότε η init συνάρτηση αναλαμβάνει να συλλέξει τα δεδομένα (R και split του L) και να φτιάξει το hash table με βάση τον R. Αντίθετα, αν το split του L είναι μικρότερο από τον R, στο map κομμάτι γίνεται partition του L όπως ο R και όχι join. Το join γίνεται στη συνάρτηση close, όπου εκεί έχουν ήδη φορτωθεί τα partitions του L και τα partitions του R, που αντιστοιχούν σε μη μηδενικά κομμάτια του L. Αν και δεν υπάρχει τρόπος να ελέγξουμε ακριβώς τη φυσική τοποθέτηση των αντιγράφων στο DFS, με την αύξηση του replication factor για τον R μπορούμε να διασφαλίσουμε ότι οι περισσότεροι κόμβοι στο cluster έχουν ένα τοπικό αντίγραφο του R. Αυτό μπορεί να επιτρέψει στην ένωση broadcast να αποφύγει την ανάκτηση του R από το DFS στη συνάρτηση init().

Στη δικιά μας περίπτωση, έχουμε τα σύνολα δεδομένων Los Angeles Crime Data και LA Police Stations, όπου το Los Angeles Crime Data έχει τον ρόλο του L πίνακα και το LA Police Stations ως το R.

Improved Repartition Join

Ο Repartition Join υλοποιείται σε ένα μόλις MapReduce job. Στη map διαδικασία, κάθε map task λειτουργεί πάνω σε μια είσοδο από τα δύο αυτά σύνολα. Για να γνωρίζουμε από ποιο σύνολο προέρχεται κάθε είσοδος, το κάθε task σημειώνει ένα tag, ορίζοντας την προέλευση της εισόδου και εξάγοντας ένα tag-αρισμένο ζεύγος (key, value). Στη συνέχεια, σε αυτά τα ζεύγη εφαρμόζεται partition, sort και merge, ώστε στο τέλος να τροφοδοτήσουν τον reducer ανάλογα με το key. Εκεί ο reducer διαχωρίζει τις εισόδους ανάλογα με το tag που τους έχει δοθεί στο map στάδιο, και εν τέλει πραγματοποιεί ένα cross-product.

Ένα πιθανό πρόβλημα με το Repartition Join είναι ότι όλα τα αρχεία για ένα συγκεκριμένο κλειδί σύνδεσης, τόσο από το L όσο και από το R, πρέπει να αποθηκευτούν προσωρινά στη μνήμη. Για αυτό, παρουσιάζεται μια βελτιωμένη μορφή του Repartition Join, η Improved Repartition Join.

Στη βελτιωμένη εκδοχή, στη συνάρτηση map, το κλειδί εξόδου αλλάζει σε ένα σύνθετο κλειδί που αποτελείται από το join key και το table tag. Οι ετικέτες των πινάκων δημιουργούνται με τέτοιο τρόπο ώστε να εξασφαλίζεται ότι τα αρχεία από το R θα ταξινομηθούν πριν από εκείνα από το L για ένα δεδομένο κλειδί σύνδεσης. Ύστερα, η συνάρτηση διαμερισμού (partitioning) προσαρμόζεται έτσι ώστε το hashcode να υπολογίζεται μόνο από το join key του σύνθετου κλειδιού. Με αυτόν τον τρόπο, τα αρχεία με το ίδιο join key εξακολουθούν να ανατίθενται στο ίδιο reduce task. Το grouping στον reducer προσαρμόζεται επίσης έτσι ώστε τα αρχεία να ομαδοποιούνται μόνο βάσει του join key. Τέλος, καθώς τα αρχεία από τον μικρότερο πίνακα R είναι εγγυημένο ότι θα είναι μπροστά από αυτά από το L για ένα δεδομένο join key, μόνο τα αρχεία του R αποθηκεύονται προσωρινά στη μνήμη και τα αρχεία του L χρησιμοποιούνται για να παράγουν την έξοδο της σύνδεσης.

Σημείωση:
Οι αντίστοιχοι χρόνοι εκτέλεσης του Query4 μέσω των Broadcast και Improved Repartition Join είναι 2.2min και 1.7min αντίστοιχα.

7. Υλοποίηση με DataFrame

Παρακάτω παρατίθενται ο αριθμός των εγκλημάτων με χρήση πυροβόλων όπλων και η μέση απόσταση των περιστατικών αυτών από τα αντίστοιχα αστυνομικά τμήματα. Οι πληροφορίες παρουσιάζονται ανά αστυνομικό τμήμα και είναι ταξινομημένες με φθίνουσα σειρά βάσει του αριθμού των περιστατικών. Αυτό μας επιτρέπει να κατανοήσουμε ποια τμήματα έχουν μεγαλύτερη συχνότητα τέτοιων εγκλημάτων και πόσο μακριά από το τμήμα λαμβάνουν χώρα, παρέχοντας σημαντικά δεδομένα για την κατανομή των εγκλημάτων και την απόκριση των τμημάτων.

DIVISION	average_distance	incidents total
77TH STREET	2.69	17019
SOUTHEAST	2.11	12942
NEWTON	2.02	9846
SOUTHWEST	2.7	8912
HOLLENBECK	2.65	6202
HARBOR	4.08	5621
RAMPART	1.58	5115
MISSION	4.72	4504
OLYMPIC	1.82	4424
NORTHEAST	3.9	3920
FOOTHILL	3.8	3774
HOLLYWOOD	1.46	3641
CENTRAL	1.14	3614
WILSHIRE	2.31	3525
NORTH HOLLYWOOD	2.72	3466
WEST VALLEY	3.53	2902
VAN NUYS	2.22	2733
PACIFIC	3.73	2708
DEVONSHIRE	4.01	2471
TOPANGA	3.49	2283
WEST LOS ANGELES	4.24	1541

GitHub Link

GitHub Repository: https://github.com/Chontzakis/Big_Data