Διαχείριση Δεδομένων Μεγάλης Κλίμακας
Αναφορά Εργασίας

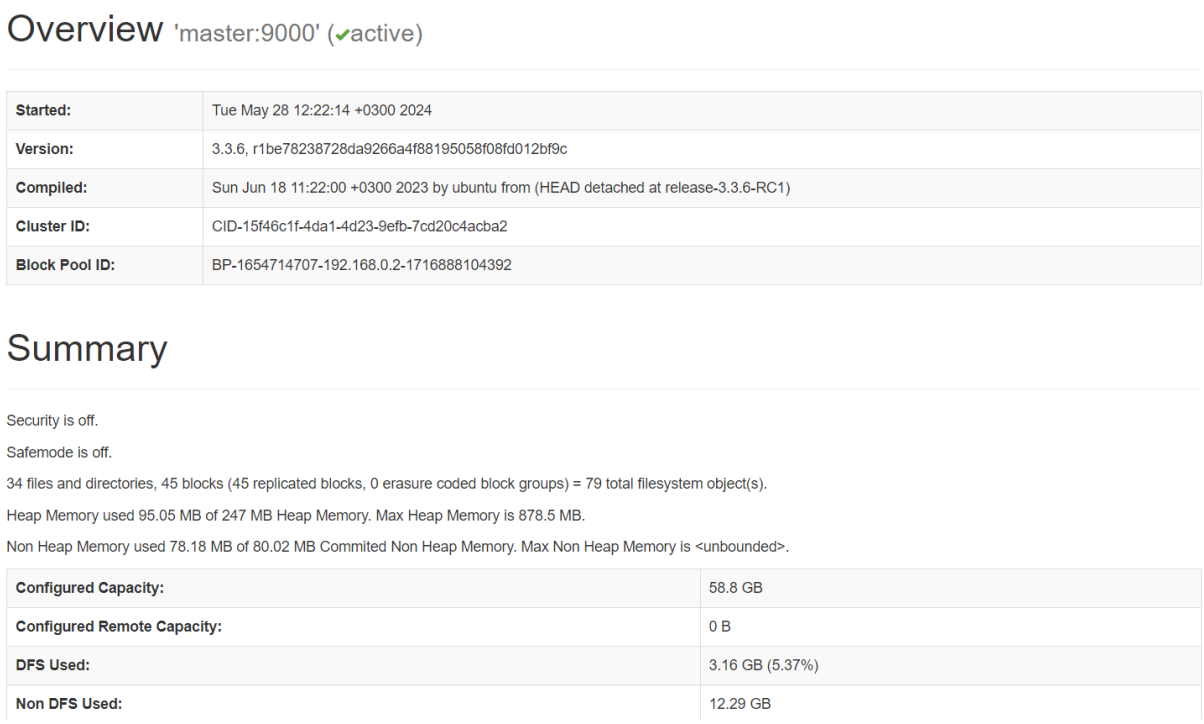
Άγγελος Μαρινάκης
03400225

Λάζαρος Αθανασιάδης
03400201

Ο κώδικας της εργασίας βρίσκεται στο repository https://github.com/lathanasiadis/big-data-project.

Ζητούμενο 1

Έχοντας ακολουθήσει τις οδηγίες της άσκησης για την εγκατάσταση των Apache Spark και HDFS, έχουμε πρόσβαση στην web εφαρμογή του HDFS μέσω της τοποθεσίας `http://<master>:9870/`, όπου <master> η public IP του master node (Screenshot 1).

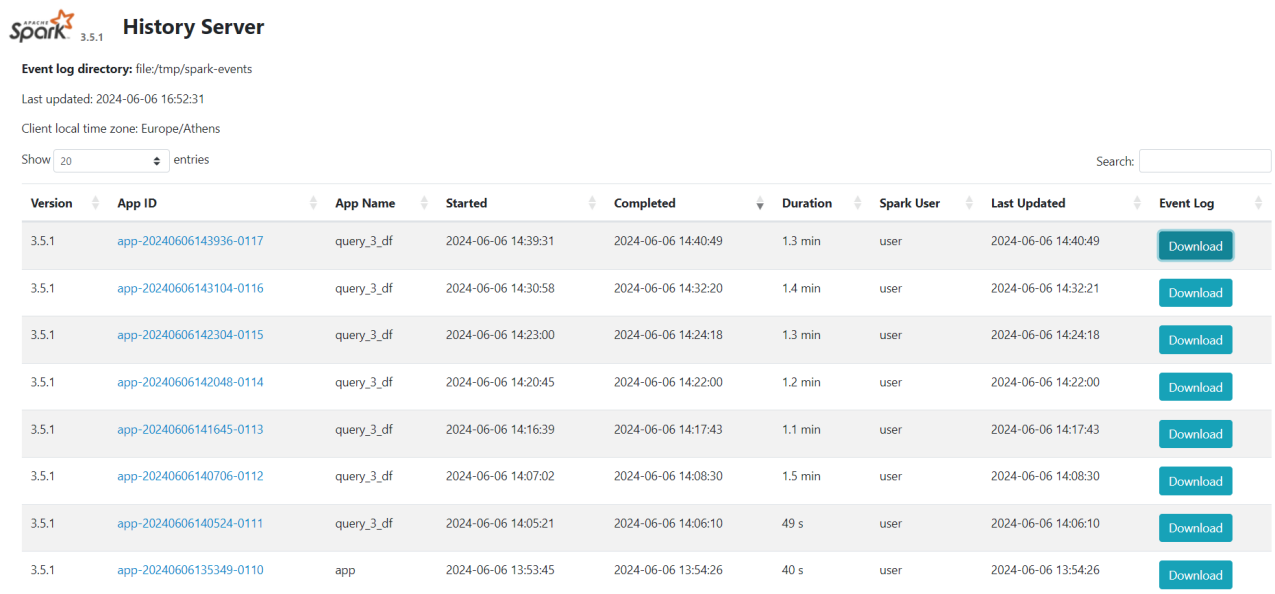


Screenshot 1: Web εφαρμογή του HDFS

Επιπλέον, έχοντας ακολουθήσει τις οδηγίες για την εγκατάσταση του Spark History Server, έχουμε πρόσβαση σε αυτόν μέσω της τοποθεσίας `http://<master>:18080/` (Screenshot 2).

Ζητούμενο 2

Αφού δημιουργήσουμε έναν φάκελο με το όνομα data στον master node και αποθηκεύσουμε σε αυτόν όλα τα δεδομένα που θα χρειαστούμε για την άσκηση, χρησιμοποιούμε τις εντολές στο



Screenshot 2: Spark History Server

```
user@master:~$ hadoop fs -ls ~/project_data
Found 6 items
-rw-r--r--   2 user supergroup  537190637 2024-05-30 15:53 /home/user/project_data/Crime_Data_from_2010_to_2019.csv
-rw-r--r--   2 user supergroup  241051722 2024-05-30 15:53 /home/user/project_data/Crime_Data_from_2020_to_Present.csv
-rw-r--r--   2 user supergroup       1502 2024-06-06 16:27 /home/user/project_data/LAPD_Stations.csv
drwxr-xr-x   - user supergroup          0 2024-05-30 15:53 /home/user/project_data/income
drwxr-xr-x   - user supergroup          0 2024-05-30 17:08 /home/user/project_data/parquet
-rw-r--r--   2 user supergroup     897062 2024-05-30 15:53 /home/user/project_data/revgeocoding.csv
```

Screenshot 3: Περιεχόμενα του φακέλου project_data στο HDFS μετά το ανέβασμα των αρχείων

τμήμα κώδικα 1 για την μεταφορά των αρχείων στο HDFS. Στο Screenshot 3 επιβεβαιώνουμε ότι τα αρχεία έχουν όντως ανέβει στο HDFS στον φάκελο project_data.

```
# create a new directory in HDFS
hadoop fs -mkdir -p ~/project_data
# upload project data to HDFS
hadoop fs -put data/* ~/project_data
# confirm our data has been uploaded by printing the contents of project_data
hadoop fs -ls ~/project_data
```

Τμήμα κώδικα 1: Ανέβασμα δεδομένων στο HDFS

Στο τμήμα κώδικα 2 (από το αρχείο question2.py) μετρατρέπουμε τα δύο αρχεία csv που αποτελούν το κυρίως dataset της άσκησης σε μορφή Parquet και τα αποθηκεύουμε στο HDFS στον υποφάκελο parquet του φακέλου project_data.

```
from pyspark.sql import SparkSession

app = SparkSession.builder.appName("app").getOrCreate()

files = ["Crime_Data_from_2010_to_2019",
         "Crime_Data_from_2020_to_Present"]
```

```
path = 'hdfs://master:9000/home/user/project_data/'
output_path = 'hdfs://master:9000/home/user/project_data/parquet/'

for file_name in files:
    csv_path = path + file_name + ".csv"
    df = app.read.csv(csv_path, header=True, inferSchema=True)
    df.write.parquet(output_path + file_name + ".parquet")

app.stop()
```

Τμήμα κώδικα 2: Αποθήκευση αρχείων από μορφή csv σε μορφή Parquet

Ζητούμενο 3

Η υλοποίηση του Query 1 γίνεται στο αρχείο question3dataframe.py χρησιμοποιώντας το DataFrame API και στο αρχείο question3sql.py χρησιμοποιώντας το SQL API. Και οι δύο υλοποιήσεις οδηγούν στα ίδια αποτελέσματα, τα οποία βρίσκονται στον Πίνακα 1.

Σε καθένα από αυτά τα αρχεία χρησιμοποιούμε ένα command line argument για να επιλέξουμε είτε τη μορφή csv είτε τη μορφή Parquet για την υλοποίηση του ερωτήματος. Στους Πίνακες 2 και 3 καταγράφουμε τους χρόνους που μετρήσαμε για κάθε API και μορφή αρχείου. Η στήλη 'Load' αντιστοιχεί στον χρόνο φόρτωσης των δεδομένων, η στήλη 'Query' στον χρόνο εκτέλεσης του ερωτήματος και η στήλη 'Άθροισμα' είναι το άθροισμα των στηλών 'Load' και 'Query'. Οι χρόνοι 'Load' και 'Query' είναι οι διάμεσοι από 5 μετρήσεις.

Έτος	Μήνας	Αριθμός περιστατικών	Κατάταξη μήνα
2010	1	19.520	1
2010	3	18.131	2
2010	7	17.857	3
2011	1	18.141	1
2011	7	17.283	2
2011	10	17.034	3
2012	1	17.954	1
2012	8	17.661	2
2012	5	17.502	3
2013	8	17.441	1
2013	1	16.828	2
2013	7	16.645	3
2014	10	17.331	1
2014	7	17.258	2
2014	12	17.198	3
2015	10	19.221	1
2015	8	19.011	2
2015	7	18.709	3
2016	10	19.660	1
2016	8	19.496	2
2016	7	19.450	3
2017	10	20.437	1
2017	7	20.199	2
2017	1	19.849	3
2018	5	19.976	1
2018	7	19.879	2
2018	8	19.765	3
2019	7	19.126	1
2019	8	18.987	2
2019	3	18.865	3
2020	1	18.542	1
2020	2	17.272	2
2020	5	17.219	3
2021	10	19.326	1
2021	7	18.672	2
2021	8	18.387	3
2022	5	20.450	1
2022	10	20.313	2
2022	6	20.255	3
2023	10	20.029	1
2023	8	20.024	2
2023	1	19.902	3
2024	1	18.762	1
2024	2	17.214	2
2024	3	16.009	3

Πίνακας 1: Αποτελέσματα Query 1

	Load	Query	Άθροισμα
CSV	47,08s	17,92s	65s
Parquet	20,92s	23,76s	44,68s

Πίνακας 2: Χρόνοι εκτέλεσης του Query1 με το DataFrame API

	Load	Query	Άθροισμα
CSV	47,35s	23,23s	70,58s
Parquet	22,28s	21,91s	44,2s

Πίνακας 3: Χρόνοι εκτέλεσης του Query1 με το SQL API

Από τους χρόνους που μετρήθηκαν μπορούμε να βγάλουμε δύο βασικά συμπεράσματα. Το πρώτο είναι ότι τα δύο APIs (DataFrame και SQL) δεν έχουν σημαντικές διαφορές στους αντίστοιχους χρόνους. Το δεύτερο είναι ότι η μορφή Parquet οδηγεί σε αρκετά μικρότερο συνολικό χρόνο σε σχέση με την CSV καθώς έχει αρκετά μικρότερο χρόνο φόρτωσης (Load). Όσον αφορά τους χρόνους εκτέλεσης (Query) οι δύο μορφές Parquet και CSV δεν παρουσιάζουν σημαντικές διαφορές.

Ζητούμενο 4

Έχουμε υλοποιήσει το Query 2 με το RDD API στο αρχείο question4rdd.py και με το DataFrame API στο αρχείο question4dataframe.py. Τα αποτελέσματα είναι τα ίδια και για τις δύο υλοποιήσεις και βρίσκονται στον Πίνακα 4.

Τμήμα Ημέρας	Πλήθος Εγκλημάτων (στους δρόμους)
Νύχτα	243.374
Βράδυ	191.792
Απόγευμα	151.564
Πρωί	126.611

Πίνακας 4: Αποτελέσματα Query 2

	Load	Query	Άθροισμα
RDD API	2,42s	26,5s	28,92s
DataFrame API	45,45s	14,64s	60,09s

Πίνακας 5: Χρόνοι εκτέλεσης του Query 2.

Victim Descent	Total Vicitms
Hispanic/Latin/Mexican	1.550
Black	1.093
White	705
Other	400
Other Asian	104
Unknown	65
Korean	9
American Indian/Alaskan Native	3
Japanese	3
Chinese	2
Filipino	2

(α΄) Αριθμός θυμάτων ανά καταγωγή για τις τρεις περιοχές με το χαμηλότερο εισόδημα.

Victim Descent	Total Vicitms
White	347
Other	110
Hispanic/Latin/Mexican	55
Unknown	32
Black	18
Other Asian	16

(β΄) Αριθμός θυμάτων ανά καταγωγή για τις τρεις περιοχές με το υψηλότερο εισόδημα.

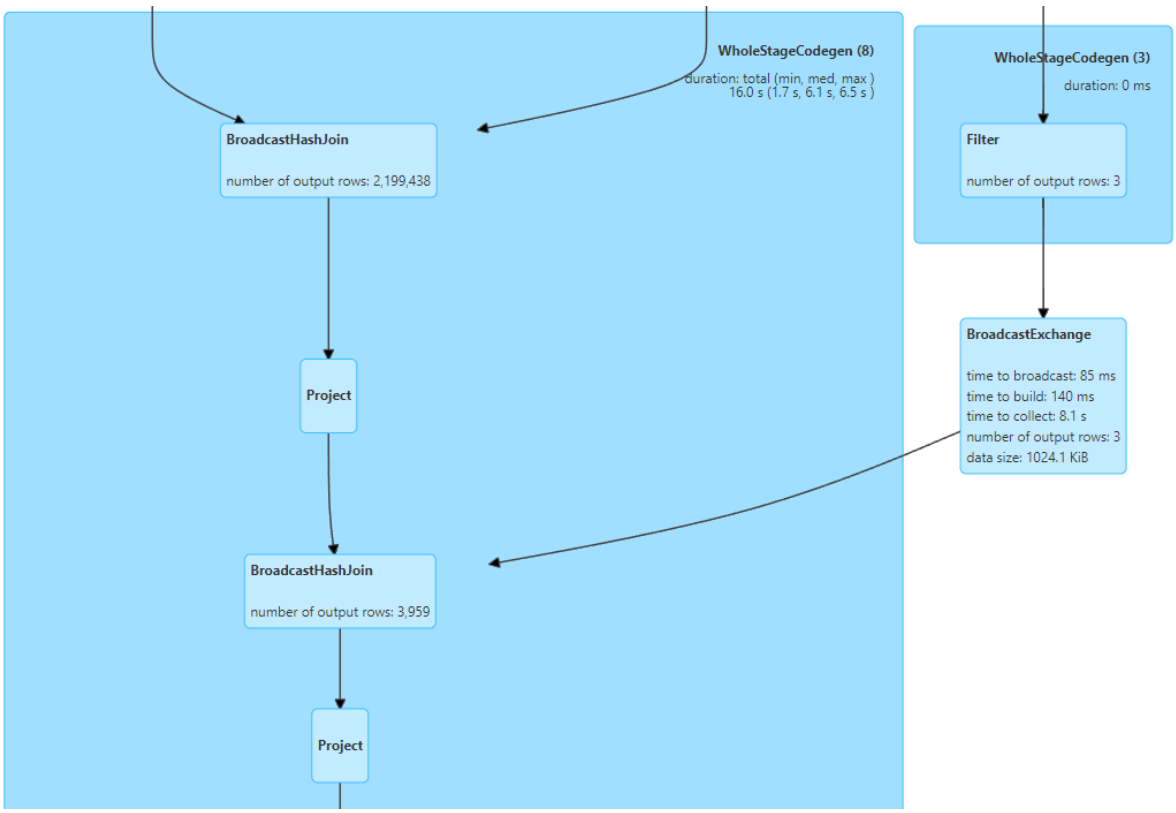
Πίνακας 6: Αποτελέσματα Query 3.

Μετρήσαμε τον χρόνο εκτέλεσης κάθε υλοποίησης, ακολουθώντας την ίδια μεθοδολογία με το προηγούμενο ζητπύμενο. Τα αποτελέσματα βρίσκονται στον Πίνακα 5. Παρατηρούμε πως η δημιουργία του dataframe από το σύνολο δεδομένων είναι πολύ χρονοβόρα, αλλά οι πράξεις πάνω σε αυτό είναι βελτιστοποιημένες, με αποτέλεσμα να εκτελούνται γρηγορότερα από τις αντίστοιχες των RDD. Επιπλέον, το διάβασμα των δεδομένων είναι πιο αργό στο DataFrame API γιατί, μεταξύ άλλων, χειρίζεται αυτόματα την csv μορφή των αρχείων, φορτώνοντάς τα σε κατάλληλες στήλες. Αντιθέτως, το RDD API τα διαβάζει σε "raw" μορφή, και πρέπει να τα προ-επεξεργαστούμε μόνοι μας. Για το dataset της εργασίας, δεν αρκεί ένα απλό split με βάση το κόμμα, καθώς υπάρχουν στοιχεία σε κάθε γραμμή που περιέχουν κόμμα χωρίς να αντιστοιχεί σε αλλαγή στήλης (επειδή το στοιχείο βρίσκεται μέσα σε εισαγωγικά). Για αυτόν τον λόγο, στην υλοποίηση με το RDD API χρησιμοποιήσαμε την μέθοδο `csv.reader` για το διάβασμα των αρχείων.

Ζητούμενο 5

Ο κώδικας για το ζητούμενο 5 βρίσκεται στο question5.py. Χρησιμοποιήσαμε το σύνολο δεδομένων στην μορφή parquet, καθώς όπως είδαμε στα προηγούμενα ερωτήματα οδηγεί σε μικρότερους χρόνους εκτέλεσης. Ως σύνολο δεδομένων για το εισόδημα των νοικοκυριών, χρησιμοποιήσαμε την εκδοχή που αντιστοιχεί στο 2015, ακολουθώντας την εκφώνηση. Τα αποτελέσματα βρίσκονται στον Πίνακα 6.

Στην υλοποίησή μας για το Query 3, χρησιμοποιούμε την πράξη join δύο φορές. Μπορούμε να παρατηρήσουμε την στρατηγική που επιλέγει ο Catalyst Optimizer για κάθε join είτε χρησιμοποιώντας τη μέθοδο explain του Spark που εκτυπώνει το physical plan για ένα query, είτε μέσω του



Screenshot 4: Τμήμα από την απεικόνιση του physical plan για το Query 3 από τον History Server

History Server. Όπως μπορούμε να δούμε στο Screenshot 4, ο Catalyst Optimizer επιλέγει Broadcast Hash Join και για τα δύο joins.

Η επιλογή αυτή βγάζει νόημα, καθώς τα σύνολα δεδομένων με τα οποία κάνουμε join (Reverse Geocoding και Income) έχουν μικρό μέγεθος (1.5KB και 13K, αντίστοιχα). Έτσι, τόσο η αποστολή τους σε κάθε mapper, όσο και η φόρτωσή τους σε buffers σε κάθε έναν από αυτούς, είναι υπολογιστικά φτηνή.

Στα πλαίσια πειραματισμού με τις στρατηγικές join, δοκιμάσαμε τις διαφορετικές υλοποιήσεις που έχει το Spark. Αρχικά, η χρήση του Broadcast join αλλά με μετάδοση του κυρίου συνόλου δεδομένων οδηγεί σε out of memory προβλήματα κατά την διάρκεια της αποστολής του. Με αυτόν τον τρόπο επιβεβαιώνουμε και πειραματικά πως αυτή η στρατηγική join έχει νόημα μόνο αν μεταδόσουμε τα μικρά σύνολα δεδομένων.

Το Shuffle Hash join είναι παρόμοιο με το Broadcast join, μόνο που αντί να μεταδίδουμε ολόκληρο το μικρό dataset, κάνουμε shuffle όλα τα δεδομένα (αναθέτοντάς τα σε partitions με βάση το κλειδί) και μετά δημιουργείται ένα hash table από το μικρότερο σύνολο δεδομένων σε κάθε κόμβο. Αυτή η στρατηγική join οδηγεί σε ελαφρά χειρότερους χρόνους εκτέλεσης από το Broadcast join (70s από 64s), το οποίο εξηγείται από τα παραπάνω operations που πρέπει να γίνουν (shuffle αντί για μετάδοση του μικρού συνόλου δεδομένων).

Στο Merge join, το πρώτο βήμα είναι το ανακάτεμα των εγγραφών των συνόλων δεδομένων με βάση το key, όπως και στο Shuffle Hash join. Έπειτα, ακολουθεί ταξινόμηση των εγγραφών και ταυτόχρονη διαπέρασή τους για να ενωθούν. Σημειώνουμε πως είναι υπολογιστικά πιο ακριβό από το Shuffle Hash join, καθώς απαιτούνται ακόμη περισσότερες πράξεις (ταξινόμηση και διαπέραση των συνόλων δεδομένων αντί για δημιουργία και probing ενός hash table). Αυτό επιβεβαιώνεται και από τον μεγαλύτερο χρόνο εκτέλεσης (77s αντί για 70s).

Χρησιμοποιώντας το hint `SHUFFLE_REPLICATE_NL`, αναγκάζουμε το spark να υλοποιήσει το join υπολογίζοντας το καρτεσιανό γινόμενο των δύο συνόλων δεδομένων, χρησιμοποιώντας εμφωλευμένες επαναλήψεις. Αυτό είναι μια εξαιρετικά ακριβή διαδικασία, καθώς πρέπει να υπολογιστούν

όλοι οι πιθανοί συνδυασμοί. Το γεγονός ότι πρέπει να εκτελεστεί δύο φορές (μία για κάθε join) αυξάνει ακόμη περισσότερο τον αναμενόμενο χρόνο εκτέλεσης. Δεν μας εκπλήσσει, λοιπόν, που ο συνολικός χρόνος εκτέλεσης του query ήταν κοντά στις 2 ώρες.

Συμπεραίνουμε πως η επιλογή του Catalyst Optimizer (Broadcast join) είναι η καταλληλότερη, όπως και σε κάθε περίπτωση που κάνουμε join με ένα πολύ μικρό σύνολο δεδομένων.

Ζητούμενο 6

Οι υλοποιήσεις των repartition join και broadcast join βρίσκονται στα αρχεία question6repartition.py και question6broadcast.py. Για να μπορούμε να ελέγξουμε εύκολα την έξοδο των αλγορίθμων, έχουμε κάνει την προεπεξεργασία στα RDD που θα χρειάζοταν αν υλοποιούσαμε και το Query 4, δηλαδή την επιλογή των σχετικών στηλών και το φιλτράρισμα για τα εγκλήματα στα οποία έγινε χρήση πυροβόλου όπλου.

Repartition Join

Αφού διαβάσουμε τα δύο σύνολα δεδομένων σε RDD μορφή, χρησιμοποιούμε μια map εντολή σε κάθε ένα από αυτά που τα μετατρέπει σε ζεύγη κλειδιού-τιμής, με κλειδί την μεταβλητή πάνω στην οποία θέλουμε να κάνουμε join (κωδικός αστυνομικού τμήματος). Επιπλέον, με την ίδια map εντολή σημαδεύουμε (tag) κάθε ζεύγος με μια τιμή που δείχνει το σύνολο δεδομένων από το οποίο προήλθε, ακριβώς όπως περιγράφεται στο [1].

Έπειτα, ενώνουμε τις πλειάδες από κάθε dataset (`union` μέθοδος των RDD αντικειμένων), τις ομαδοποιούμε με βάση το κλειδί τους (`reduceByKey`) και χρησιμοποιούμε μια flatMap εντολή, η οποία εκτελεί την λειτουργικότητα του Reducer που περιγράφεται στο [1]. Συγκεκριμένα, ξεχωρίζει και αποθηκεύει σε buffers τις τιμές που αντιστοιχούν σε κάθε κλειδί με βάση το σημάδι (tag) που τους έχουμε βάλει. Μετά, αρκεί να επιστρέψει ως τιμές που αντιστοιχούν στο κάθε κλειδί το καρτεσιανό γινόμενο των δύο buffers.

Broadcast Join

Γνωρίζουμε πως το μέγεθος του συνόλου δεδομένων με τα αστυνομικά τμήματα (1.5KB) είναι κατά πολύ μικρότερο του default split του Spark (128MB). Για αυτόν τον λόγο, δεν ελέγχουμε προγραμματιστικά τα μεγέθη των συνόλων δεδομένων, αλλά κάνουμε broadcast το σύνολο δεδομένων με τα αστυνομικά τμήματα.

Στην υλοποίηση που περιγράφεται στο [1], το μικρό σύνολο δεδομένων διανέμεται (broadcast) σε κάθε κόμβο, οι οποίοι φτιάχνουν ένα hash table με κλειδί την μεταβλητή πάνω στην οποία θέλουμε να κάνουμε join. Στο Spark έχουμε την δυνατότητα να στείλουμε κατευθείαν το hash table, χρησιμοποιώντας την μέθοδο `broadcast` του `sparkContext`.

Αφού διανείμουμε το hash table σε κάθε κόμβο, αρκεί να χρησιμοποιήσουμε μία map εντολή για να ελέγξουμε την τιμή του hash table για κάθε πλειάδα, χρησιμοποιώντας το στοιχείο της που αντιστοιχεί στην μεταβλητή που θέλουμε να κάνουμε join ως το κλειδί.

Ζητούμενο 7

Η υλοποίηση του Query 4 με χρήση του Dataframe API γίνεται στο αρχείο question7.py. Για τον υπολογισμό της γεωδαισιακής απόστασης με χρήση της βιβλιοθήκης geopy, χρειάστηκε να μετατρέψουμε τις στήλες "x" και "y" στο dataset με τα αστυνομικά τμήματα από το σύστημα συντεταγμένων στο οποίο δίνονται (προβολικό σύστημα συντεταγμένων "EPSG:2229" για την Καλιφόρνια) σε γεωγραφικό μήκος και πλάτος (σύστημα συντεταγμένων "EPSG:4326"). Η μετατροπή

Αστ. Τμήμα	Πλήθος Περιστατικών	Μέση απόσταση από το Αστ. τμήμα
77th Street	17.019	2,69 km
Southeast	12.942	2,1 km
Newton	9.846	2,01 km
Southwest	8.912	2,7 km
Hollenbeck	6.202	2,64 km
Harbor	5.621	4,07 km
Rampart	5.115	1,57 km
Mission	4.504	4,70 km
Olympic	4.424	1,82 km
Northeast	3.920	3,90 km
Foothill	3.774	3,8 km
Hollywood	3.641	1,46 km
Central	3.614	1,13 km
Wilshire	3.525	2,31 km
North Hollywood	3.466	2,71 km
West Valley	2.902	3,53 km
Van Nuys	2.733	2,22 km
Pacific	2.708	3,72 km
Devonshire	2.471	4,01 km
Topanga	2.283	3,48 km

Πίνακας 7: Αποτελέσματα Query 4

γίνεται στον κώδικα με χρήση της βιβλιοθήκης pyproj. Τα αποτελέσματα φαίνονται στον πίνακα 7.

Αναφορές

[1] Spyros Blanas κ.ά. ``A comparison of join algorithms for log processing in MapReduce''. Στο: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, σσ. 975–986. DOI: 10.1145/1807167.1807273. URL: https://doi.org/10.1145/1807167.1807273.