

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

ΕΔΕΜΜ



Διαχείριση Δεδομένων Μεγάλης Κλίμακας

Τίτλος εργασίας	Εξαμηνιαία Εργασία
Μέλη ομάδας - Αριθμοί μητρώου	Νικήτας Χατζής - 03400237
	Κωνσταντίνος Χριστογεώργος - 03400239
Ημερομηνία παράδοσης	12/06/2024

Περιεχόμενα

1. [Ζητούμενο 1](#)
2. [Ζητούμενο 2](#)
3. [Ζητούμενο 3](#)
4. [Ζητούμενο 4](#)
5. [Ζητούμενο 5](#)
6. [Ζητούμενο 6](#)
7. [Ζητούμενο 7](#)

CODE REPOSITORY LINK:

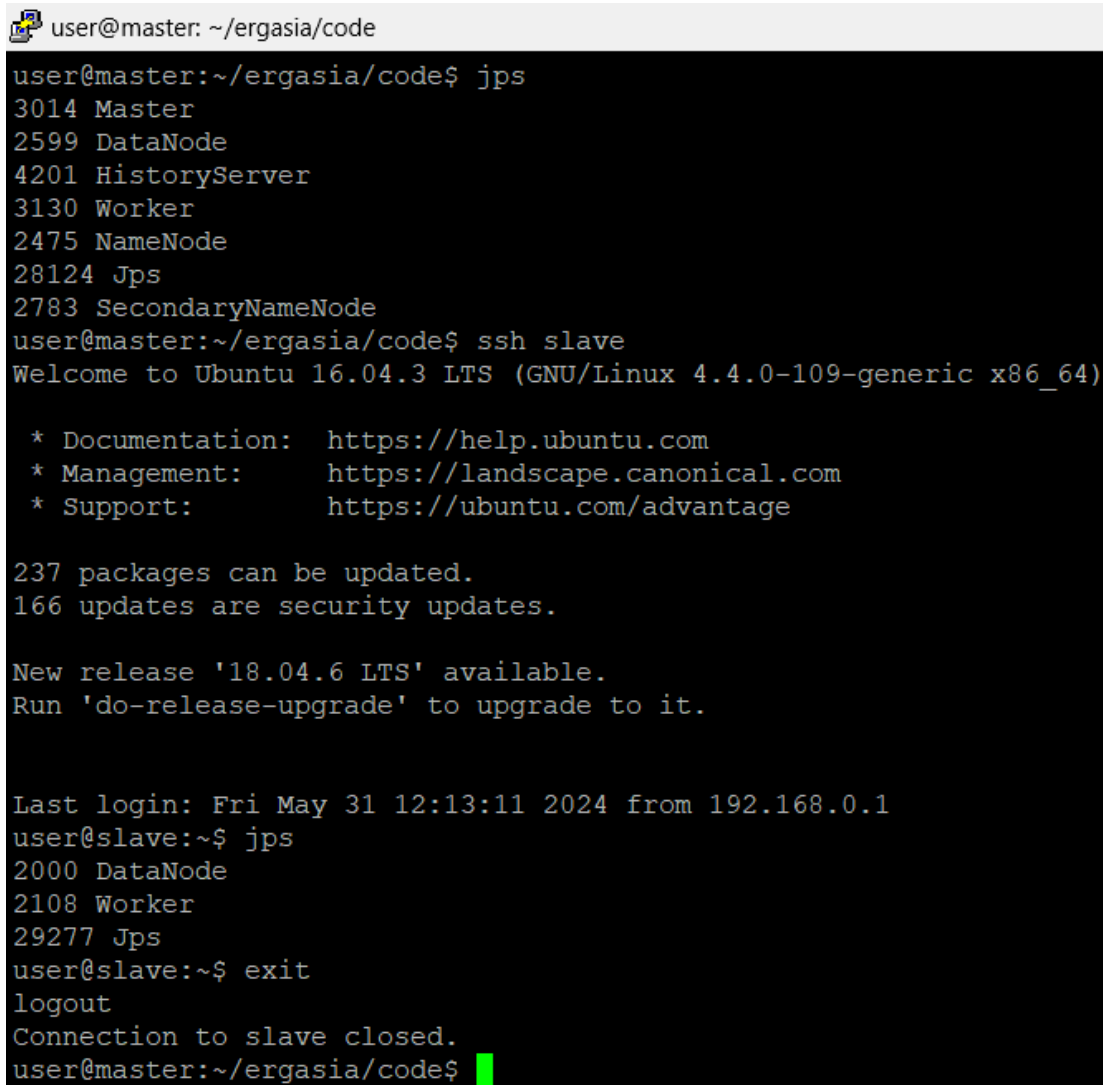
<https://github.com/n-hatz/BD-Assignment/tree/main>

Τα αρχεία κώδικα βρίσκονται στον φάκελο **code/"Ζητούμενο i"** αντίστοιχα για το κάθε ζητούμενο της εργασίας.

Ζητούμενο 1

Για το ζητούμενο αυτό ακολουθήσαμε τις οδηγίες του εργαστηρίου για εγκατάσταση των hdfs και spark. Χρησιμοποιούμε τα Virtual Machines που μας παρέχονται μέσω του okeanos, τα οποία κάναμε set up με βάση τα specs που αναφέρονται στις οδηγίες.

Στο Figure 1 φαίνονται τα αποτελέσματα της εντολής jps για τους master και slave κόμβους αντίστοιχα που δείχνουν ότι το setup έχει γίνει σωστά.

A terminal window showing the execution of the 'jps' command on a master node and an 'ssh slave' command to connect to a slave node. The master node output lists: 3014 Master, 2599 DataNode, 4201 HistoryServer, 3130 Worker, 2475 NameNode, 28124 Jps, and 2783 SecondaryNameNode. The slave node output lists: 2000 DataNode, 2108 Worker, and 29277 Jps. The terminal also shows system updates and login information.

```
user@master: ~/ergasia/code
user@master:~/ergasia/code$ jps
3014 Master
2599 DataNode
4201 HistoryServer
3130 Worker
2475 NameNode
28124 Jps
2783 SecondaryNameNode
user@master:~/ergasia/code$ ssh slave
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.4.0-109-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

237 packages can be updated.
166 updates are security updates.

New release '18.04.6 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Fri May 31 12:13:11 2024 from 192.168.0.1
user@slave:~$ jps
2000 DataNode
2108 Worker
29277 Jps
user@slave:~$ exit
logout
Connection to slave closed.
user@master:~/ergasia/code$
```

Figure 1: Results of jps command on master and slave node

Στο Figure 2 φαίνεται ένα screenshot από το web ui του hdfs που δείχνει πως το setup έχει γίνει σωστά και για τους 2 κόμβους.

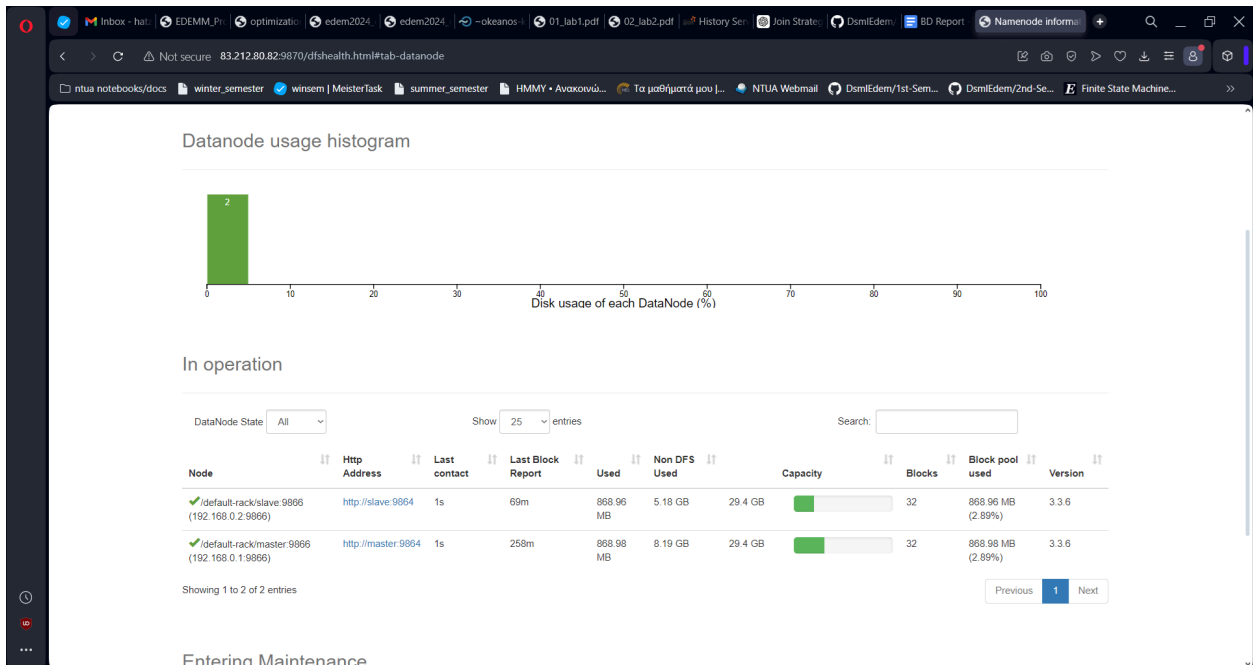


Figure 2: HDFS web UI

Τέλος, στο Figure 3 φαίνεται το web UI του Spark History Server.

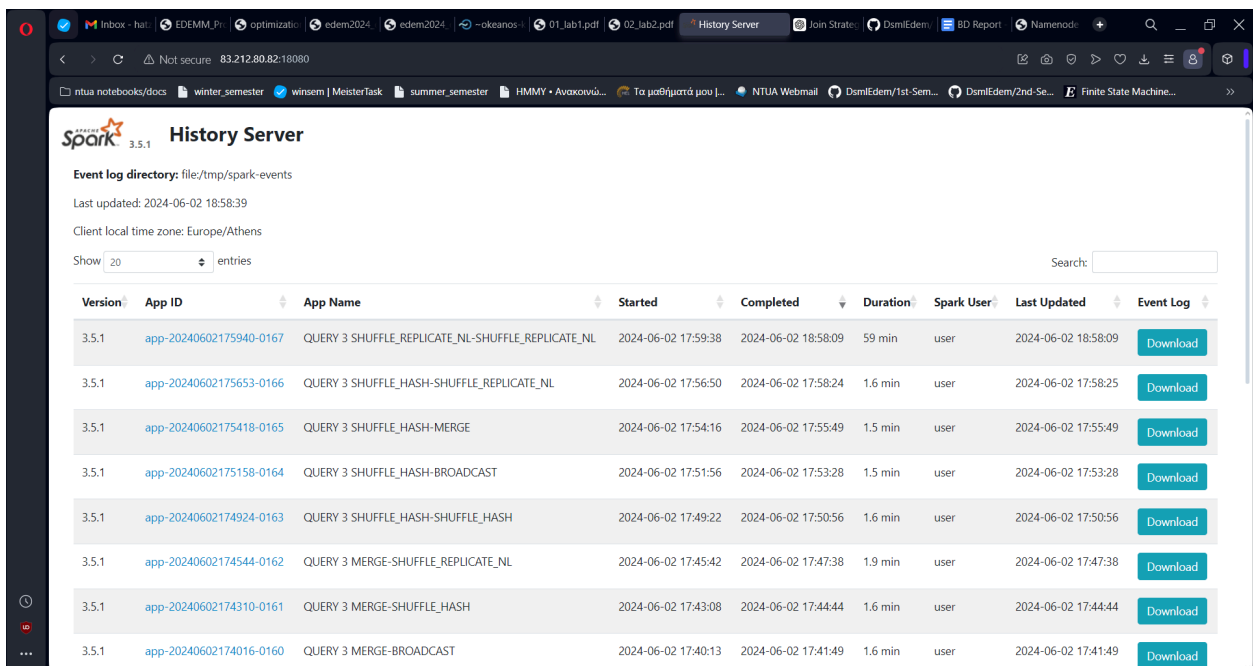


Figure 3: Spark History Server web UI

Όπως φαίνεται και από το Figure 3, το screenshot έχει παρθεί αφότου έχουν εκτελεστεί κάποια queries.

Ζητούμενο 2

Αρχικά δημιουργήσαμε ένα directory στο hdfs με το όνομα **data** στο οποίο μεταφέραμε τα .csv αρχεία της εργασίας. Το directory αυτό το βάλαμε κατευθείαν πάνω στο home, δηλαδή το path είναι **~/data**. Η εντολή φαίνεται στο Code 1.

```
hadoop fs -mkdir ~/data
```

Code 1: creating an hdfs directory

Στη συνέχεια, αφού μεταφέρουμε τα αρχεία στο master VM (μέσω SCP) τα μεταφέρουμε στο hdfs. Οι εντολές φαίνονται στο Figure 4. Για λόγους ευκολίας ονομάσαμε *rows1.csv* το πρώτο αρχείο crime data και *rows2.csv* το δεύτερο. Επίσης για λόγους ευκολίας ονομάσαμε το αρχείο με τα δεδομένα των LA Police Stations ως *LAPD_Police_Stations.csv*. Στο αρχείο *revgecoding.csv* περιέχονται τα νέα δεδομένα που στάλθηκαν στο διορθωτικό email.

```
user@master: ~/ergasia/data
user@master:~/ergasia/data$ ls
LA_income_2015.csv  LA_income_2021.csv      revgecoding.csv
LA_income_2017.csv  LAPD_Police_Stations.csv rows1.csv
LA_income_2019.csv  OLD                      rows2.csv
user@master:~/ergasia/data$ hadoop fs -put *.csv ~/data
user@master:~/ergasia/data$ hadoop fs -ls ~/data
Found 8 items
-rw-r--r--  2 user supergroup      1413 2024-06-01 17:53 /home/user/data/LAPD_
Police_Stations.csv
-rw-r--r--  2 user supergroup    12859 2024-06-01 17:53 /home/user/data/LA_in
come_2015.csv
-rw-r--r--  2 user supergroup    12866 2024-06-01 17:53 /home/user/data/LA_in
come_2017.csv
-rw-r--r--  2 user supergroup    12811 2024-06-01 17:53 /home/user/data/LA_in
come_2019.csv
-rw-r--r--  2 user supergroup    12859 2024-06-01 17:53 /home/user/data/LA_in
come_2021.csv
-rw-r--r--  2 user supergroup    897062 2024-06-01 17:53 /home/user/data/revge
coding.csv
-rw-r--r--  2 user supergroup  537190636 2024-06-01 17:54 /home/user/data/rows1
.csv
-rw-r--r--  2 user supergroup  241051722 2024-06-01 17:54 /home/user/data/rows2
.csv
user@master:~/ergasia/data$ █
```

Figure 4: Αποθήκευση αρχείων στο hdfs

Τέλος, γράψαμε κώδικα που μετατρέπει τα csv αρχεία σε parquet και τα αποθηκεύει στον ίδιο κατάλογο στο hdfs (**~/data**). Ο κώδικας βρίσκεται στο αρχείο **csv2parquet.py** και διαβάζει ένα csv το οποίο το αποθηκεύει μετά ως parquet file. Ο κώδικας φαίνεται στο Code 2.

```

from pyspark.sql import SparkSession
import sys

read_file=sys.argv[1] #path to csv file
read_path = "hdfs://master:9000/home/user/data/"+read_file #where to
read csv file

write_file = sys.argv[2] #name of new file
write_path = "hdfs://master:9000/home/user/data/"+write_file #where
to write parquet file

spark = SparkSession.builder.appName("CSV 2 Parquet").getOrCreate()

df = spark.read.csv(read_path,header=True,inferSchema=True)
df.write.parquet(write_path)

```

Code 2: code for converting csv files to parquet files (csv2parquet.py)

Εν συντομία, ο κώδικας παίρνει 2 command line arguments, το όνομα του csv αρχείου και το όνομα που θα έχει το νέο parquet αρχείο. Ο κώδικας υποθέτει ότι το csv αρχείο βρίσκεται στο ~/data folder του hdfs και πως ο χρήστης θέλει επίσης να αποθηκεύσει το parquet αρχείο στον ίδιο κατάλογο. Ένα παράδειγμα εκτέλεσης του παραπάνω script στην παρακάτω γραμμή:

```
spark-submit csv2parquet.py rows1.csv rows1.parquet
```

Στο Figure 5 φαίνονται τα περιεχόμενα του directory ~/data του hdfs αφού τρέξουμε το *csv2parquet.py* για τα αρχεία *rows1.csv* και *rows2.csv*.

```

user@master: ~/ergasia/code
user@master:~/ergasia/code$ hadoop fs -ls ~/data
Found 10 items
-rw-r--r--  2 user supergroup      1413 2024-06-01 17:53 /home/user/data/LAPD_Police_Stations.csv
-rw-r--r--  2 user supergroup     12859 2024-06-01 17:53 /home/user/data/LA_income_2015.csv
-rw-r--r--  2 user supergroup     12866 2024-06-01 17:53 /home/user/data/LA_income_2017.csv
-rw-r--r--  2 user supergroup     12811 2024-06-01 17:53 /home/user/data/LA_income_2019.csv
-rw-r--r--  2 user supergroup     12859 2024-06-01 17:53 /home/user/data/LA_income_2021.csv
-rw-r--r--  2 user supergroup     897062 2024-06-01 17:53 /home/user/data/revgecoding.csv
-rw-r--r--  2 user supergroup    537190636 2024-06-01 17:54 /home/user/data/rows1.csv
drwxr-xr-x  - user supergroup         0 2024-06-01 18:02 /home/user/data/rows1.parquet
-rw-r--r--  2 user supergroup    241051722 2024-06-01 17:54 /home/user/data/rows2.csv
drwxr-xr-x  - user supergroup         0 2024-06-01 18:04 /home/user/data/rows2.parquet
user@master:~/ergasia/code$

```

Figure 5: HDFS contents after converting csv files to parquet

Ζητούμενο 3

Τα αρχεία που υλοποιούν το query 1 στις διάφορες παραλλαγές του είναι τα εξής:

- *q1_df.py* (dataframe api, csv files)
- *q1_sql.py* (SQL api, csv files)
- *q1_df_parquet.py* (dataframe api, parquet files)
- *q1_sql_parquet.py* (SQL api, parquet files)

Συνοπτικά, ο τρόπος που λειτουργεί ο κώδικας για το Query 1 (ανεξαρτήτως τύπου αρχείου και API) είναι ο εξής: αρχικά διαβάζει το αρχείο ως dataframe και κάνει drop τα columns που δεν χρειαζόμαστε. Παρόλο που δεν είναι απαραίτητα καλή πρακτική να αφαιρούμε εντελώς τα columns που δεν χρησιμοποιούμε, παρατηρήσαμε ότι ο χρόνος εκτέλεσης και η απαιτούμενη μνήμη μειώνονται σημαντικά. Συγκεκριμένα για το Query 1 κρατάμε μόνο το column *“DATE OCC”* καθώς είναι το μόνο που χρειαζόμαστε. Επίσης, επειδή το main dataset βρίσκεται σε 2 ξεχωριστά csv αρχεία, ενώνουμε τα dataframes τους (concatenate από κάτω). Στη συνέχεια ο κώδικας μετατρέπει το πεδίο αυτό σε τύπο timestamp και δημιουργεί από το timestamp αυτό 2 νέα columns *“year”* και *“month”* αντίστοιχα. Έπειτα κάνει group by (year,month) και ένα count aggregation στις εγγραφές για να υπολογίσει το πλήθος crimes ανά έτος και μήνα. Μετά κάνει partition by τη στήλη *“year”* για να διαφοροποιήσει τα έτη και sort ως προς το πλήθος των εγγραφών/crimes ανά έτος. Τέλος, εντός κάθε έτους κατατάσσει τους μήνες σε φθίνουσα σειρά ως προς τον αριθμό crimes δίνοντάς τους έναν αύξων δείκτη σε ένα νέο column *“ranking”* και μετά κάνει filter το dataframe ώστε το *“ranking”* να είναι μικρότερο ή ίσο του 3 (δηλαδή εμφανίζονται μόνο οι 3 μήνες με τα περισσότερα crimes ανά έτος).

Οι χρόνοι για κάθε συνδυασμό API με τύπο αρχείου φαίνονται στον παρακάτω πίνακα.

API	File type	Time (seconds)
dataframe	CSV	42s
SQL	CSV	78s
dataframe	parquet	30s
SQL	parquet	43s

Οι χρόνοι εκτέλεσης φαίνονται επίσης στο Figure 6.

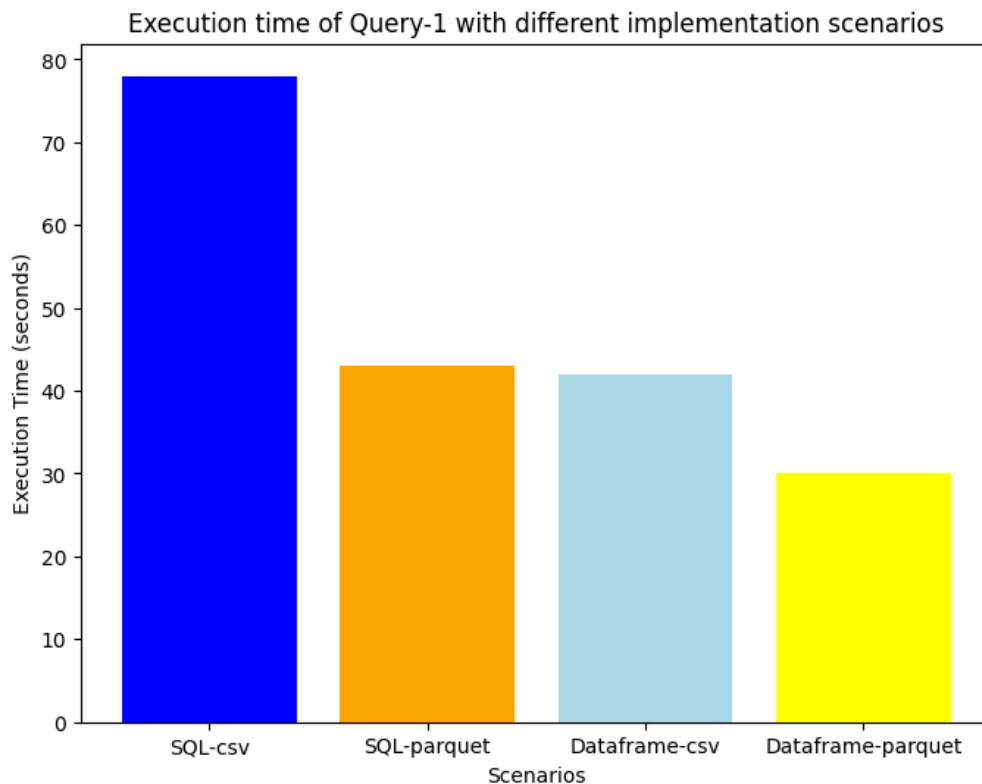


Figure 6: Plot με χρόνους εκτέλεσης του Query 1

Από τους χρόνους παρατηρούμε ότι τα scripts που χρησιμοποιούν τα parquet αρχεία είναι σαφώς πιο γρήγορα. Αυτό είναι αναμενόμενο γιατί το parquet αποθηκεύει τα δεδομένα ανά column (ενώ στα csv είναι ανά row) το οποίο οδηγεί σε καλύτερη συμπίεση και ταχύτερο χρόνο εκτέλεσης στα queries.

Επίσης, η χρήση του SQL API προσθέτει μια επιπλέον καθυστέρηση στην εκτέλεση καθώς δεν είναι βελτιστοποιημένα σε αντίθεση με τα dataframe που γίνονται optimized πιο εύκολα μέσω από το pyspark. Επίσης, τα queries πρέπει να γίνουν και parsed με αποτέλεσμα να δημιουργείται επιπλέον καθυστέρηση.

Τα αποτελέσματα του Query 1 φαίνονται παρακάτω στο Figure 7. Και τα 4 αρχεία προφανώς έχουν το ίδιο αποτέλεσμα.


```

user@master: ~/ergasia/code

```

year	month	crime_total	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

Figure 7: Query 1 results

Ζητούμενο 4

Τα αρχεία που υλοποιούν το Query 2 είναι τα ακόλουθα:

- *q2_df.py* (με χρήση dataframe)
- *q2_rdd.py* (με χρήση rdd)

Συνοπτικά, ο κώδικας λειτουργεί με τον εξής τρόπο: Αρχικά όπως και πριν αφού διαβάσει τα 2 αρχεία του main dataset (crimes) ως dataframe τα κάνει concatenate το ένα κάτω από το άλλο. Στη συνέχεια αφαιρεί τα columns που δεν χρειάζεται. Συγκεκριμένα, τα μόνα columns που δεν αφαιρούμε είναι τα columns “*TIME OCC*” και “*Premis Desc*”. Στη συνέχεια, επειδή το column “*TIME OCC*” διαβάζεται ως αριθμός ενώ βρίσκεται στο format “HHmm”, δημιουργούνται κάποια προβλήματα. Για παράδειγμα, η ώρα 00:30 (δώδεκα και μισή το βράδυ) που είναι αποθηκευμένη ως 0030 και διαβάζεται ως 30 (αφαιρούνται τα μηδενικά στην αρχή). Για το λόγο αυτό ο κώδικας μετατρέπει τη στήλη αυτή σε string, κάνει pad στα αριστερά όσα μηδενικά χρειάζονται ώστε να φτάσει μήκος 4 (αν χρειάζονται) και στη συνέχεια κάνει cast το column σε timestamp της μορφής HHmm χωρίς προβλήματα και το αποθηκεύει ένα νέο column “*time*”. Έπειτα κάνει filter τα δεδομένα ώστε να συμπεριλαμβάνουν μόνο καταγραφές στο δρόμο (“*Premis Desc*”=“STREET”) και μετά δημιουργεί μια νέα στήλη “*time of day*” με βάση τα διαστήματα που δίνονται στην εκφώνηση και τη στήλη “*time*”. Τέλος, κάνει group by (“*time of day*”) με count στις εγγραφές (crime) και sort σε φθίνουσα σειρά ως προς αυτές.

Όσον αφορά τον κώδικα σε RDD, τα operations είναι ακριβώς τα ίδια. Η επιλογή των columns που χρειαζόμαστε γίνεται με ένα απλό **map** που κρατάει από κάθε row μόνο τα columns που χρειαζόμαστε (με lambda expression). Το φιλτράρισμα του “*Premis Desc*”=“STREET” γίνεται με τη συνάρτηση **filter**. Η μετατροπή του string σε timestamp και έπειτα σε time of day γίνεται με **map** και δικές μας συναρτήσεις. Επίσης, η συνάρτηση μας που δημιουργεί τις περιόδους της ημέρας (morning, night κλπ. με βάση την εκφώνηση) επιστρέφει τις εγγραφές σε tuples της μορφής (period,1). Άρα για το group by aggregation μας αρκεί ένα **reduceByKey** με το κλασσικό lambda expression που απλώς αθροίζει.

Οι χρόνοι εκτέλεσης φαίνονται στον παρακάτω πίνακα και επίσης στο Figure 8:

File	Time (seconds)
<i>q2_df.py</i> (dataframes)	34s
<i>q2_rdd.py</i> (RDD)	37s



Figure 8: Plot με χρόνους εκτέλεσης του Query 2

Παρατηρούμε ότι το αρχείο που χρησιμοποιεί RDD είναι εν τέλει πιο αργό. Αυτό δεν είναι το αναμενόμενο αποτέλεσμα. Ο κώδικας σε dataframes μεταφράζεται από τον optimizer σε RDDs. Αν ο κώδικας βρισκόταν ήδη σε RDD και ήταν σωστά γραμμένος, θεωρητικά θα έπρεπε να είναι πιο γρήγορος καθώς δεν χρειάζεται η μετάφραση από dataframe. Στην περίπτωση μας, ο κώδικας σε RDD που μεταφράζει ο catalyst optimizer από τα dataframe είναι προφανώς καλύτερα optimized από τον δικό μας κώδικα σε RDDs. Επιπλέον καθυστέρηση πιθανότατα να οφείλεται στη χρήση του IO module και του StringIO. Παρατηρήσαμε πως όταν δεν χρησιμοποιούμε το παραπάνω module και κάνουμε split κάθε γραμμή όταν εμφανίζεται ο χαρακτήρας του κόμματος “,” ,επειδή ορισμένα records του csv περιέχουν το κόμμα, το split για το διαχωρισμό του κάθε πεδίου γίνεται λάθος. Τελικά το διορθώσαμε με τον τρόπο που αναφέραμε παραπάνω ωστόσο οδηγεί σε περαιτέρω καθυστέρηση.

Τα αποτελέσματα του Query 2 φαίνονται στο Figure 9.

```
+-----+-----+
|time of day| count|
+-----+-----+
|      Night|243374|
|   Evening|191792|
| Afternoon|151564|
|   Morning|126611|
+-----+-----+
```

Figure 9: Query 2 results

Ζητούμενο 5

Το αρχείο που υλοποιεί το Query 3 είναι το ακόλουθο:

- *q3_df.py* (με χρήση dataframe)

Συνοπτικά, ο κώδικας λειτουργεί με τον εξής τρόπο:

Αρχικά αφού ενώσουμε τα 2 κομμάτια του crime dataset φιλτράρουμε εκτός δεδομένων τα victimless crimes, δηλαδή αυτά που δεν έχουν θύμα ή δεν υπάρχει καταγραφή της καταγωγής του θύματος (θεωρούμε ότι η καταγωγή "X"=UNKNOWN αντιστοιχεί στις περιπτώσεις που δεν υπάρχει καταγραφή της καταγωγής). Στη συνέχεια ξεφορτωνόμαστε τα columns που δεν χρειαζόμαστε, μετατρέπουμε την στήλη "*TIME OCC*" σε timestamp και δημιουργούμε από αυτό ένα νέο "*year*" column το οποίο κάνουμε filter για να κρατήσουμε μόνο τις εγγραφές με έτος 2015. Στη συνέχεια κάνουμε (inner) join το crime dataset με το reverse geocoding dataset (πάνω στο ζευγάρι από στήλες "*LAT*" και "*LON*").

Επισημαίνουμε πως αντικαταστήσαμε τα δεδομένα στο αρχείο *revgecoding.csv* με αυτά που στάλθηκαν στο διορθωτικό email. Έπειτα φιλτράρουμε εκτός τυχόν εγγραφές με NULL zipcode από το joined dataset. Στη συνέχεια κάνουμε (inner) join τον joined πίνακα με το αρχείο *LA_Income_2015.csv* πάνω στο Zip Code. Αυτό αντιστοιχεί στο column "*ZIPcode*" του πίνακα crimes-revgecoding και στο column "*Zip Code*" του πίνακα *LA_Income_2015*. Μετά μετατρέπουμε τη στήλη "*Estimated Median Income*" σε αριθμητική χρησιμοποιώντας regular expression για την αφαίρεση του συμβόλου \$ και cast σε Double. Επίσης δημιουργήσαμε μια user defined function που χρησιμοποιεί ένα dictionary για να μετατρέψει τους κωδικούς "*Vict Descent*" σε καταγωγές θυμάτων. Κάνουμε broadcast αυτό το dictionary για λόγους ταχύτητας. Στη συνέχεια αποθηκεύουμε σε λίστα τις 3 πρώτες και τις 3 τελευταίες "*Community*", κάνοντας ένα select distinct πάνω στις στήλες "*Community*", "*Estimated Median Income*", sort, limit(3) και τελικά collect για να γίνουν λίστα. Τέλος, δημιουργούμε 2 νέα dataframes, ένα για τα top 3 communities και ένα για τα bottom 3 τα οποία αποτελούν αποτέλεσμα filter πάνω στα joined δεδομένα με βάση τα communities που ανήκουν στις λίστες top3 και bottom3 αντίστοιχα. Τα δυο αυτά dataframes τα κάνουμε group by ("*Vict Descent*") με απλό count εγγραφών ως aggregation και sort για να προκύψουν τα τελικά αποτελέσματα.

Τα αποτελέσματα της εκτέλεσης του Query 3 φαίνονται στο Figure 10.

Vict	Descent	count
White		338
Other Pacific Isl...		106
Hispanic/Latin/Me...		52
Black		16
Other Asian		16

Vict	Descent	count
Hispanic/Latin/Me...		1531
Black		1092
White		703
Other Pacific Isl...		393
Other Asian		103
Korean		9
American Indian/A...		3
Japanese		3
Chinese		2
Filipino		1

Figure 10: Αποτελέσματα εκτέλεσης Query 3

Στη συνέχεια δοκιμάσαμε τα διαφορετικά join strategies του spark.

Δεν φτιάξαμε διαφορετικό αρχείο για κάθε διαφορετικό συνδυασμό από joins, απλώς τρέξαμε πολλές φορές το ίδιο αρχείο αλλάζοντας τις παραμέτρους της συνάρτησης *hint* κάθε φορά.

Αρχικά εκτελούμε το αρχείο χωρίς το hint όπου ο Catalyst Optimizer επιλέγει BroadcastHashJoin και για τα 2 joins που γίνονται στο αρχείο. Το BroadcastHashJoin είναι ένας τύπος join που συμφέρει όταν ο ένας από τους 2 πίνακες είναι συγκριτικά μικρότερος. Ο πίνακας αυτός γίνεται broadcast σε όλα τα nodes και μετατρέπεται και σε hash map με κλειδί το join column (εξού και η ονομασία). Στην περίπτωση του query 3 και στα 2 joins υπάρχει ένας πολύ μικρός πίνακας (revgecoding και LAPD_Police_Stations αντίστοιχα). Η διάρκεια εκτέλεσης ήταν 102 seconds, και υποθέτουμε πως προστίθεται ένας επιπλέον χρόνος επειδή ο Catalyst Optimizer πρέπει να κάνει υπολογισμούς για να επιλέξει τα κατάλληλα join types. Το physical plan που βρήκαμε στον History Server για την εκτέλεση αυτή φαίνεται στο Figure 11.

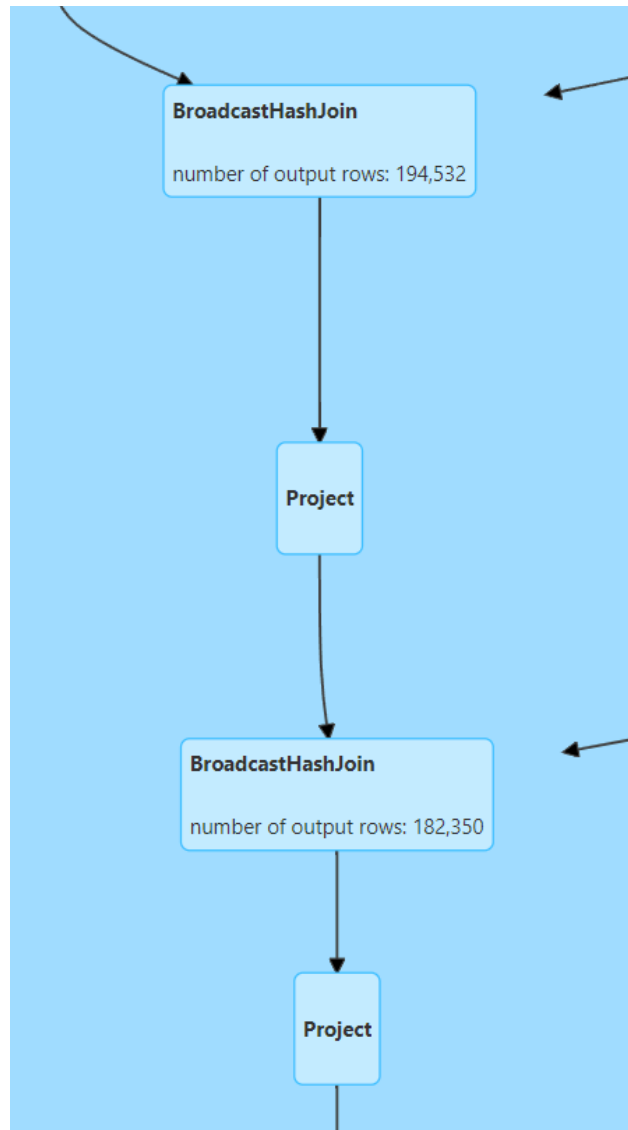


Figure 11: Απόσπασμα physical plan για default joins. Ο optimizer επιλέγει broadcast.

Χρησιμοποιώντας το hint function με παραμέτρους **“broadcast”** οδηγεί στο ίδιο physical plan με αυτό στο Figure 11 καθώς αυτό το join διάλεξε και ο Catalyst Optimizer. Παρουσιάζουμε στο Figure 12 τους χρόνους εκτέλεσης για τις εκτελέσεις όπου με τη μέθοδο hint έχουμε θέσει ως παράμετρο για το join του crime dataset με το revgecoding dataset το **“broadcast”** και για το δεύτερο join με το ενδιαμέσο αποτέλεσμα και το LAPD_Police_Stations χρησιμοποιούμε όλες τις διαθέσιμες παραμέτρους join strategy. Βλέπουμε πως οι ταχύτεροι χρόνοι εκτέλεσης έγιναν με **“broadcast”** και **“shuffle_hash”**.

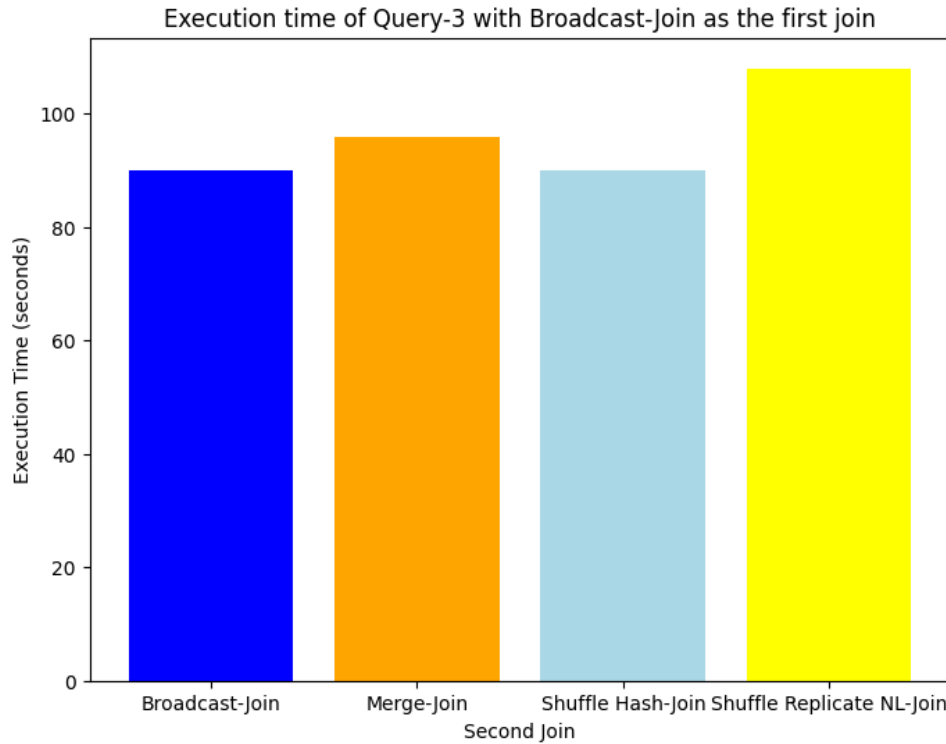


Figure 11: Χρόνοι εκτέλεσης με broadcast join στο πρώτο join και μεταβλητό δεύτερο join

Επαναλαμβάνουμε τα πειράματα για τις διαφορετικές παραμέτρους στη μέθοδο hint. Όταν επιλέγουμε το **"merge"** ως παράμετρο, στο physical plan φαίνεται ένα block που τύπου SortMergeJoin (Figure 12), που είναι ένας διαφορετικός αλγόριθμος join που ταξινομεί τον κάθε πίνακα με βάση το join column και μετά κάνει merge τους 2 πίνακες χρησιμοποιώντας ένα pointer σε κάθε sorted column. Άμα το key στο ένα dataset είναι μικρότερο από αυτό στο δεύτερο τότε ο pointer προχωράει στην επόμενη τιμή.

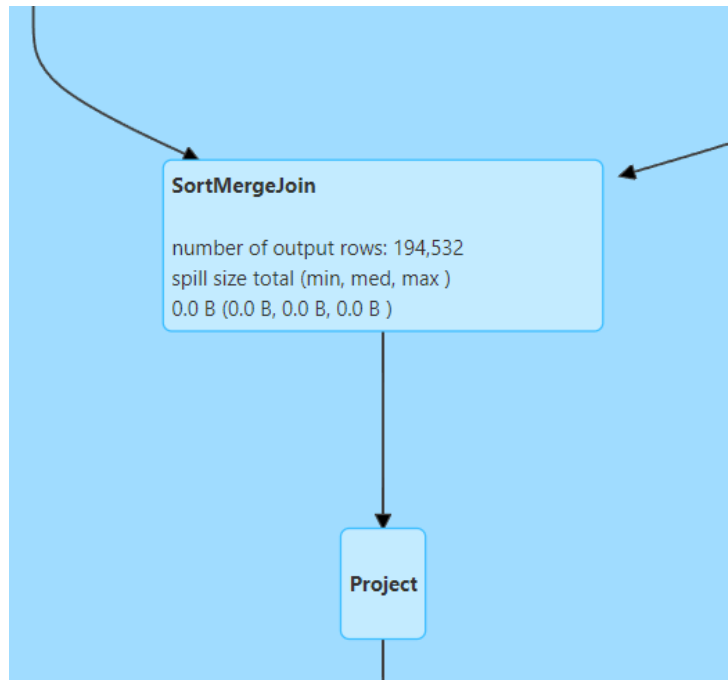


Figure 12: Απόσπασμα physical plan όταν χρησιμοποιούμε merge στη συνάρτηση hint

Παρουσιάζουμε επίσης τους χρόνους εκτέλεσης με **“merge”** στο hint του πρώτου join και μεταβλητό το δεύτερο στο Figure 13.

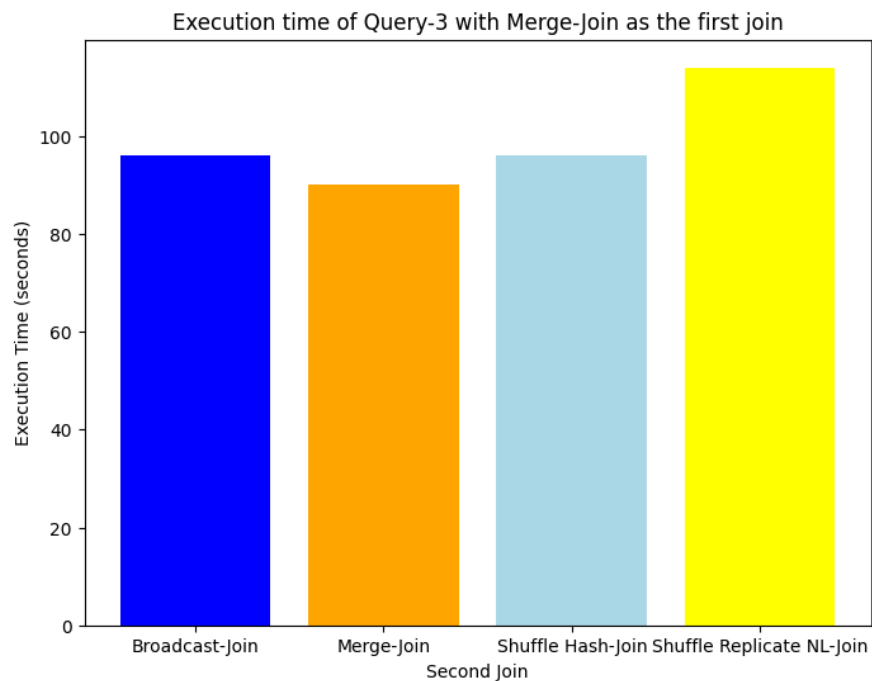


Figure 13: Χρόνοι εκτέλεσης με “merge” στο πρώτο join και μεταβλητό δεύτερο join

Όπως και πριν οι χρόνοι εκτέλεσης είναι αρκετά κοντινοί μεταξύ τους με μόνο το “shuffle_replicate_nl” να καθυστερεί περισσότερο όταν εφαρμόζεται στο δεύτερο join.

Όταν επιλέγουμε “**shuffle_hash**”, στο physical plan υπάρχουν block τύπου ShuffledHashJoin (Figure 14). Το ShuffledHashJoin είναι ένας τύπος join που είναι ιδιαίτερα χρήσιμος σε περιπτώσεις που έχουμε μεγάλα dataset. Οι 2 πίνακες γίνονται shuffled και partitioned με βάση το join column και δημιουργείται ένα hash map από το μικρό table για να γίνει το merge των tables. Παρουσιάζουμε τα αποτελέσματα των συνδυασμών εκτέλεσης φαίνονται στο Figure 15.

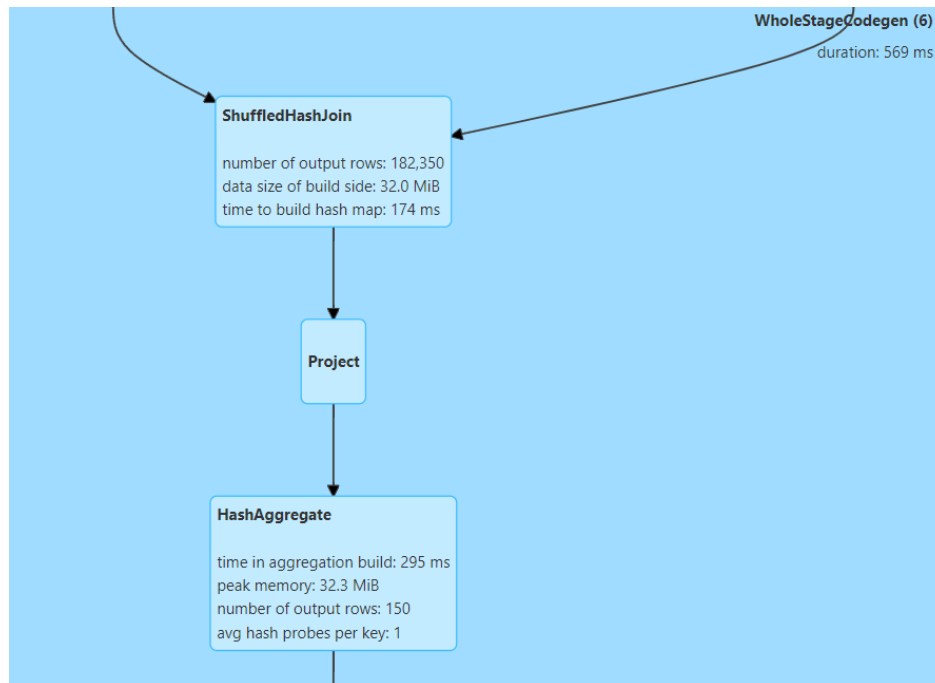


Figure 14: Απόσπασμα physical plan με χρήση “shuffle_hash” στο hint

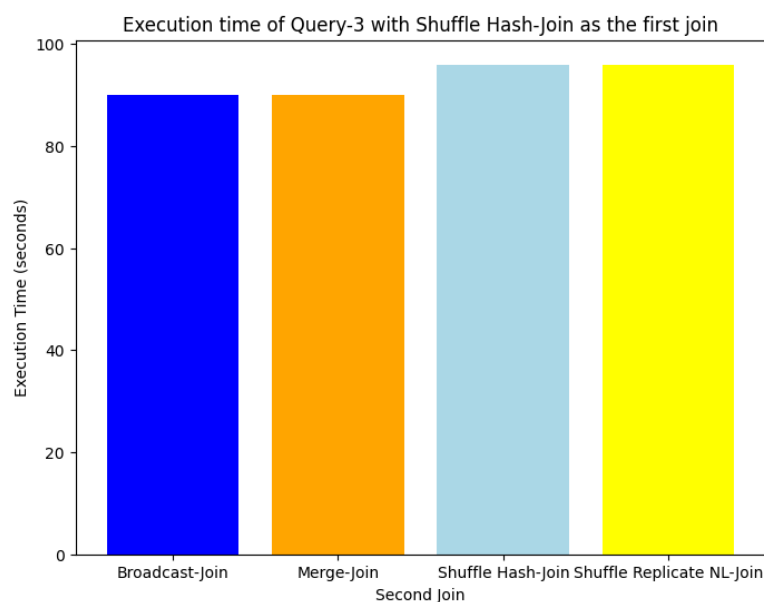


Figure 15: Χρόνοι εκτέλεσης για συνδυασμούς “shuffle_hash”

Τέλος, όταν επιλέγουμε “**shuffle_replicate_nl**” στο physical plan φαίνεται ένα block τύπου Cartesian Product που ακολουθείται από HashAggregate. Ο χρόνος εκτέλεσης με την παράμετρο “shuffle_replicate_nl” και για τα 2 joins ήταν 59 λεπτά. Ο λόγος που συμβαίνει αυτό είναι επειδή ο αλγόριθμος “shuffle_replicate_nl” περιέχει nested loop (cartesian product) που οδηγεί σε μεγάλη καθυστέρηση λόγω του μεγέθους του crime dataset. Δεν δοκιμάσαμε και τους υπόλοιπους συνδυασμούς με shuffle_replicate_nl ως το πρώτο join λόγω των **μεγάλων** χρόνων εκτέλεσης.

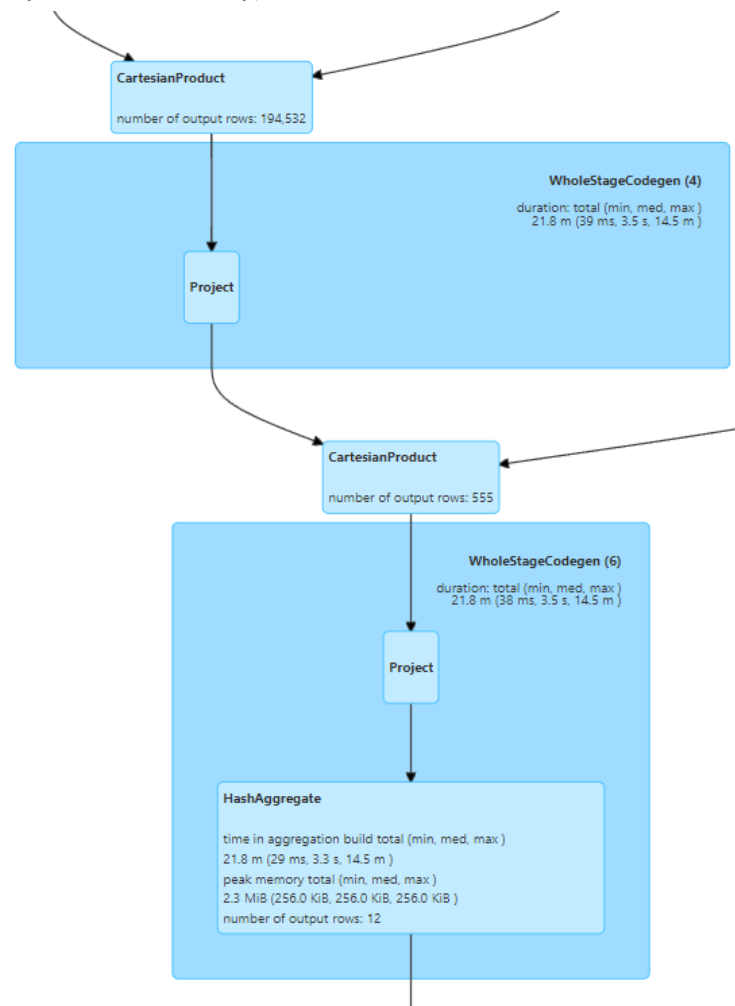


Figure 16: Αποσπασμα physical plan με shuffle replicate nested loop join.

Με βάση τον χρόνο εκτέλεσης και τον τρόπο που λειτουργούν οι αλγόριθμοι, παρατηρούμε πως οποιοσδήποτε από τους “**broadcast**”, “**merge**” και “**shuffle_hash**” είναι κατάλληλοι λόγω των παρόμοιων χρόνων εκτέλεσης και της δυνατότητάς τους να διαχειρίζονται μεγάλα dataset και να λειτουργούν σε distributed περιβάλλοντα. Αντιθέτως, ο shuffle_replicate_nl δεν είναι κατάλληλος (για τους λόγους που εξηγήσαμε παραπάνω), οδηγώντας σε μεγάλο χρόνο εκτέλεσης.

Ζητούμενο 6

Ο κώδικας για το Query 4 (με custom joins και RDD) βρίσκεται στα παρακάτω αρχεία:

- *q4_broadcast.py* (υλοποίηση του broadcast join)
- *q4_repartition.py* (υλοποίηση του repartition join)

Αρχικά θα αναφέρουμε περιγραφικά τα operations που χρειάζονται για το Query 4 (παρακάτω θα εξηγήσουμε αναλυτικότερα τις υλοποιήσεις με RDD για το Ζητούμενο 6 και DF για το Ζητούμενο 7). Συγκεκριμένα, χρειάζεται ένα filter στο αρχικό dataset με τα crimes ώστε να βρούμε τα περιστατικά με πυροβόλα όπλα και για να αφαιρέσουμε τυχόν εγγραφές στο Null Island. Έπειτα χρειάζεται ένα join μεταξύ του αρχικού συνόλου δεδομένων (crimes) με το σύνολο δεδομένων των LA Police stations. Τέλος πρέπει να υπολογίσουμε τη μέση απόσταση των crimes από το αντίστοιχο αστυνομικό τμήμα.

Operations εκτός join

Όσον αφορά τα operations σε **RDD**, τα υλοποιήσαμε ως εξής. Για να ενώσουμε (concatenate από κάτω) τα 2 αρχεία που αποτελούν το crimes dataset κάναμε **union**. Κρατάμε τα columns που χρειαζόμαστε με ένα απλό map και μετά κάνουμε **filter** ώστε να βρούμε μόνο τα crimes με πυροβόλα όπλα όπως επισημαίνεται στην εκφώνηση. Μέσα στο lambda expression βάζουμε **.startswith("1") and len==3** ώστε να διατηρούνται μόνο εγγραφές της μορφής "1xx". Για να κάνουμε αφαιρέσουμε τις Null Island εγγραφές κάνουμε ένα ακόμα **filter** ώστε τα LAT και LON να είναι ταυτόχρονα μη μηδενικά. Αφού διαβάσουμε και το LAPD Police Stations αρχείο κάνουμε ένα **map** για να κρατήσουμε τα columns που χρειαζόμαστε. Στη συνέχεια γίνεται ένα join όπου το column "AREA" του αρχικού dataset πρέπει να ισούται με το "PREC" του LAPD Police Station. Επισημαίνουμε πως κάνουμε cast και τα 2 αυτά columns σε integer για να λάβουμε τα σωστά αποτελέσματα. Πληροφορίες για την υλοποίηση του join (broadcast και repartition) υπάρχουν παρακάτω. Μετά από το join υπολογίζουμε την απόσταση μέσω της geopy (κώδικας που δίνεται στην εκφώνηση) και των columns LAT, LON, X, Y, αποθηκεύοντας την σε ένα νέο column "distance". Τέλος, κάνουμε group by ("DIVISION") και 2 aggregations, 1 count στα rows (αριθμός εγκλημάτων) και ένα average στο "distance" και sort ως προς τη μέση απόσταση σε φθίνουσα σειρά.

Broadcast join

Υλοποιήσαμε το broadcast join με RDD βασισμένοι στον ψευδοκώδικα και τις πληροφορίες που παρέχονται στο paper που αναφέρει η εκφώνηση. Όπως αναφέρει και το paper, το υλοποιήσαμε ως **map only**. Επισημαίνουμε ότι και τα 2 tables χώρεσαν ολόκληρα στη μνήμη οπότε δεν χρειάστηκε να κάνουμε split σε chunks κανένα από τα 2. Αρχικά, αφού εκτελεστούν τα operations που αναφέρθηκαν [παραπάνω](#) (μέχρι πριν το

join) κάνουμε **broadcast** το μικρό table (LAPD Police Stations) και το μετατρέπουμε σε Hash Map με κλειδί το join column ("*PREC*") και values όλα τα άλλα columns (μαζί με το "*PREC*"). Αυτό γίνεται με τις συναρτήσεις **broadcast** και **keyBy** αντίστοιχα. Στη συνέχεια για να γίνει το join φτιάχνουμε μια δικιά μας συνάρτηση, την **merge_tables**, η οποία παίρνει ως όρισμα ένα row από το μεγάλο table. Χρησιμοποιώντας το "*AREA*" του row αυτού ως κλειδί (το "*AREA*" αντιστοιχεί στο "*PREC*") αναζητεί στο Hash Map του μικρού table την αντίστοιχη εγγραφή και την ενώνει με το row που πήρε ως είσοδο η συνάρτηση. Στο τέλος επιστρέφεται το νέο row, υλοποιώντας έτσι το join μεταξύ των 2 tables. Τη συνάρτηση αυτή την δίνουμε ως όρισμα σε ένα **map** πάνω στο μεγάλο table (crimes). Στη συνέχεια αφού πραγματοποιήθηκε το join, γίνονται τα υπόλοιπα operations που αναφέρθηκαν παραπάνω (μετά το join). Για να υλοποιήσουμε το group by και τα aggregations κατάλληλα, αρχικά υπολογίζουμε την απόσταση μέσω της συνάρτησης που δίνεται στην εκφώνηση ως όρισμα σε ένα **map** που κάνει append το αποτέλεσμά της συνάρτησης αυτής στο τέλος κάθε row (ουσιαστικά δημιουργείται ένα νέο column με την απόσταση). Έπειτα μετατρέπουμε κάθε row σε tuple (key-value pair) της μορφής ("*DIVISION*", ("distance",1)) και κάνουμε ένα **reduceByKey** προσθέτοντας τα distances μεταξύ τους και τους άσσους μεταξύ τους. Το 1 σε κάθε key value pair αντιστοιχεί σε 1 crime, οπότε αν τα προσθέσουμε βρίσκουμε το συνολικό αριθμό crime ανά "*DIVISION*". Προσθέτοντας τα distances μεταξύ τους έχουμε ένα άθροισμα των αποστάσεων όλων των crimes αυτού του "*DIVISION*". Τέλος, κάνουμε ένα ακόμα **map** στο οποίο διαιρούμε το άθροισμα των distances με τον αριθμό των crimes για να βρούμε τη μέση απόσταση από το τμήμα για κάθε "*DIVISION*". Το τελικό αποτέλεσμα γίνεται sorted ως προς τη μέση απόσταση κατά φθίνουσα σειρά.

(Standard) Repartition join

Υλοποιήσαμε το standard repartition join με RDD βασισμένοι στον ψευδοκώδικα και τις πληροφορίες που παρέχονται στο paper που αναφέρει η εκφώνηση. Επισημαίνουμε ότι και τα 2 tables χώρεσαν ολόκληρα στη μνήμη οπότε δεν χρειάστηκε να κάνουμε split σε chunks κανένα από τα 2. Αρχικά, αφού γίνουν τα operations που αναφέραμε [παραπάνω](#), κάνουμε ένα **map** σε κάθε table μετατρέποντας κάθε row σε key value pairs. Ως key βάζουμε το join column, δηλαδή το "*AREA*" στο μεγάλο table (crimes) και το "*PREC*" στο LAPD Police Stations. Ως values έχουμε μια λίστα με πρώτη τιμή ένα tag "**R**" ή "**L**" και υπόλοιπες τιμές τα columns του αντίστοιχου table. Κάνουμε **union** (concatenate) το αποτέλεσμα των παραπάνω map operations σε ένα νέο RDD με όνομα **LIST_V** (όπως στο paper). Στη συνέχεια κάνουμε **groupByKey** στο **LIST_V** και **flatMap** με όρισμα μια δικιά μας συνάρτηση την *buffers*. Η συνάρτηση αυτή παίρνει ως είσοδο ένα σύνολο εγγραφών που έγιναν grouped με βάση το key ("*AREA*" ή "*PREC*") προηγουμένως στο **groupByKey**, δηλαδή είναι εγγραφές από τα 2 tables με ίδιο join key. Στη συνέχεια εξετάζει αυτές τις εγγραφές και μία μία και ανάλογα το tag που τους δώσαμε προηγουμένως (**R** ή **L**) τις

προσθέτει σε μια λίστα (buffer) **buffer_L** ή **buffer_R** αντίστοιχα. Τέλος, με ένα double for loop (καρτεσιανό γινόμενο) προσθέτει το joined rows (μια λίστα με τα columns των εγγραφών L, το κοινό join key και columns των εγγραφών R) σε μια λίστα από νέα rows την οποία επιστρέφει στο τέλος. Επειδή η συνάρτηση buffer που μόλις περιγράψαμε επιστρέφει μια λίστα από rows για κάθε πιθανή τιμή του κοινού join key (list of lists) κάνουμε **flatMap** για να γίνουν όλες flattened σε μια κοινή λίστα από τα νέα rows του joined table.

Μετά από το join υλοποιούνται με τον ίδιο τρόπο τα post join operations που αναφέραμε πριν για να βρεθεί η μέση απόσταση κλπ.

Τα αποτελέσματα του Query 4 (και για τα 2 custom joins) φαίνονται στο Figure 17. Παρουσιάζουμε ταυτόχρονα τα αποτελέσματα του Query 4 με dataframe ([Ζητούμενο 7](#)) για να φανεί η ορθότητα της υλοποίησης των joins μας.

division	average_distance	incidents	total
77TH STREET	2.6894953276176987	17019	
SOUTHEAST	2.105776256800535	12942	
NEWTON	2.0171208341434532	9846	
SOUTHWEST	2.7021348727134993	8912	
HOLLENBECK	2.6493492384728237	6202	
HARBOR	4.075035449343677	5621	
RAMPART	1.5746749254915302	5115	
MISSION	4.7083878975924485	4504	
OLYMPIC	1.8206840904852264	4424	
NORTHEAST	3.907108617915897	3920	
FOOTHILL	3.8029708851056117	3774	
HOLLYWOOD	1.461112865377521	3641	
CENTRAL	1.1383382396350337	3614	
WILSHIRE	2.3129170711246045	3525	
NORTH HOLLYWOOD	2.716400414919105	3466	
WEST VALLEY	3.5323691679613356	2902	
VAN NUYS	2.219864019721527	2733	
PACIFIC	3.728939103459969	2708	
DEVONSHIRE	4.011825365635368	2471	
TOPANGA	3.481381058322757	2283	
WEST LOS ANGELES	4.248587515715767	1541	

Figure 17: Αποτελέσματα Query 4 με RDD και dataframes αντίστοιχα

Ζητούμενο 7

Ο κώδικας για το Ζητούμενο 7 βρίσκεται στο αρχείο

- *q4_df.py* (υλοποίηση Query 4 με dataframes).

Η υλοποίηση με dataframes είναι straightforward καθώς χρησιμοποιούμε απλώς τις συναρτήσεις που αντιστοιχούν στα operations που αναφέραμε στο [Ζητούμενο 6](#). Αρχικά, αφού ενωθούν τα 2 τμήματα του crimes dataset και γίνουν τα ίδια filters που αναφέραμε στο Ζητούμενο 6 που επιλέγουν μόνο τα crimes με πυροβόλα όπλα και αφαιρούν τις εγγραφές με τοποθεσία το Null Island, κάνουμε cast τα join columns σε integer type (column "AREA" για το crimes dataset και column "PREC" για το LAPD_Police_Stations dataset). Μετά κάνουμε (inner) join τα 2 tables με βάση αυτά τα columns. Στη συνέχεια δημιουργούμε μια user defined function δίνοντας ως όρισμα στην αντίστοιχη συνάρτηση του dataframe API το signature της συνάρτησης **get_distance** που δίνεται στην εκφώνηση και χρησιμοποιούμε το udf αυτό για να δημιουργήσουμε ένα νέο column "*distance*" με τις αποστάσεις από τα αστυνομικά τμήματα. Τέλος, κάνουμε ένα group by ("*DIVISION*") με aggregations ένα count στον αριθμό των εγγραφών και ένα average στο "*distance*" και ταξινομούμε με βάση αυτό σε φθίνουσα σειρά. Τα αποτελέσματα τα δείξαμε και νωρίτερα μαζί με τα RDD οπότε τώρα θα τα δείξουμε ξεχωριστά.

```
division average_distance incidents total
77TH STREET 2.6894953276176987 17019
SOUTHEAST 2.105776256800535 12942
NEWTON 2.0171208341434532 9846
SOUTHWEST 2.7021348727134993 8912
HOLLENBECK 2.6493492384728237 6202
HARBOR 4.075035449343677 5621
RAMPART 1.5746749254915302 5115
MISSION 4.7083878975924485 4504
OLYMPIC 1.8206840904852264 4424
NORTHEAST 3.907108617915897 3920
FOOTHILL 3.8029708851056117 3774
HOLLYWOOD 1.461112865377521 3641
CENTRAL 1.1383382396350337 3614
WILSHIRE 2.3129170711246045 3525
NORTH HOLLYWOOD 2.716400414919105 3466
WEST VALLEY 3.5323691679613356 2902
VAN NUYS 2.219864019721527 2733
PACIFIC 3.728939103459969 2708
DEVONSHIRE 4.011825365635368 2471
TOPANGA 3.481381058322757 2283
WEST LOS ANGELES 4.248587515715767 1541
```

Figure 18: Αποτελέσματα Query 4 με RDD και custom joins

DIVISION	average_distance	incidents_total
77TH STREET	2.689495327617719	17019
SOUTHEAST	2.1057762568005245	12942
NEWTON	2.017120834143444	9846
SOUTHWEST	2.70213487271351	8912
HOLLENBECK	2.649349238472807	6202
HARBOR	4.075035449343705	5621
RAMPART	1.5746749254915222	5115
MISSION	4.708387897592439	4504
OLYMPIC	1.8206840904852262	4424
NORTHEAST	3.907108617915903	3920
FOOTHILL	3.8029708851056223	3774
HOLLYWOOD	1.461112865377517	3641
CENTRAL	1.1383382396350257	3614
WILSHIRE	2.312917071124613	3525
NORTH HOLLYWOOD	2.7164004149191237	3466
WEST VALLEY	3.5323691679613343	2902
VAN NUYS	2.2198640197215234	2733
PACIFIC	3.7289391034599784	2708
DEVONSHIRE	4.011825365635365	2471
TOPANGA	3.4813810583227505	2283
WEST LOS ANGELES	4.248587515715773	1541

Figure 19: Αποτελέσματα Query 4 με dataframe API