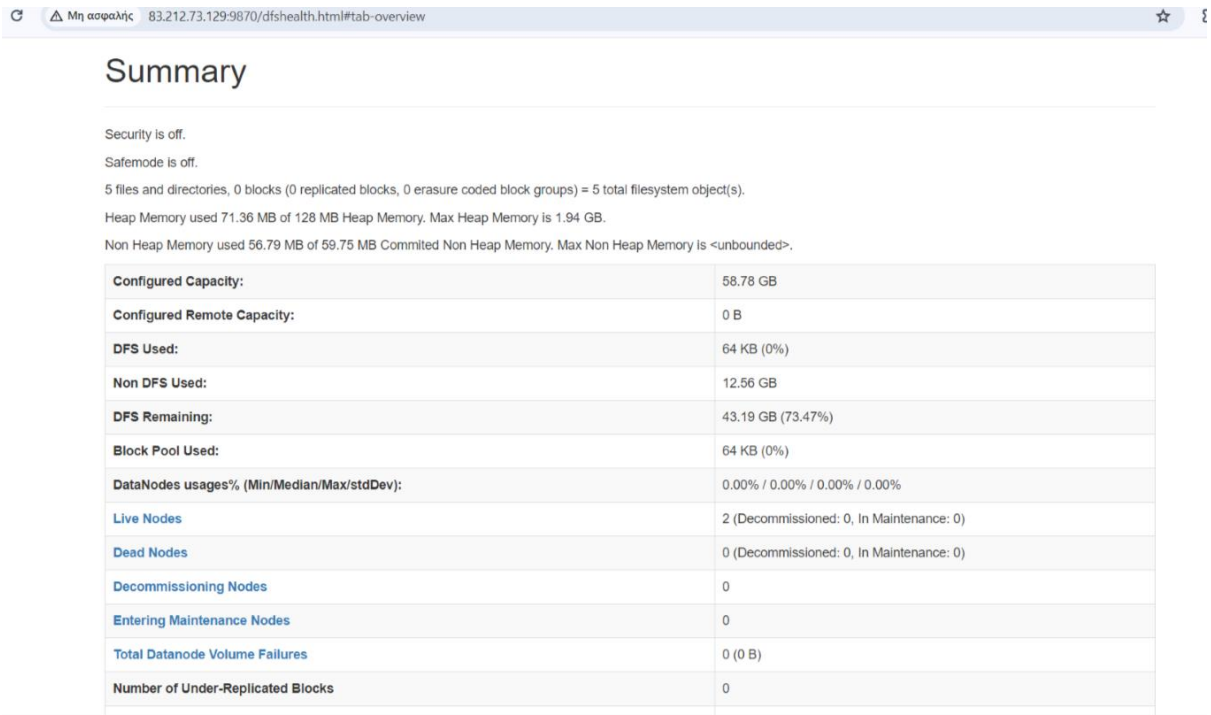
ΠΡΟΧΩΡΗΜΕΝΑ ΘΕΜΑΤΑ ΒΑΣΕΩΝ ΔΕΔΟΜΕΝΩΝ

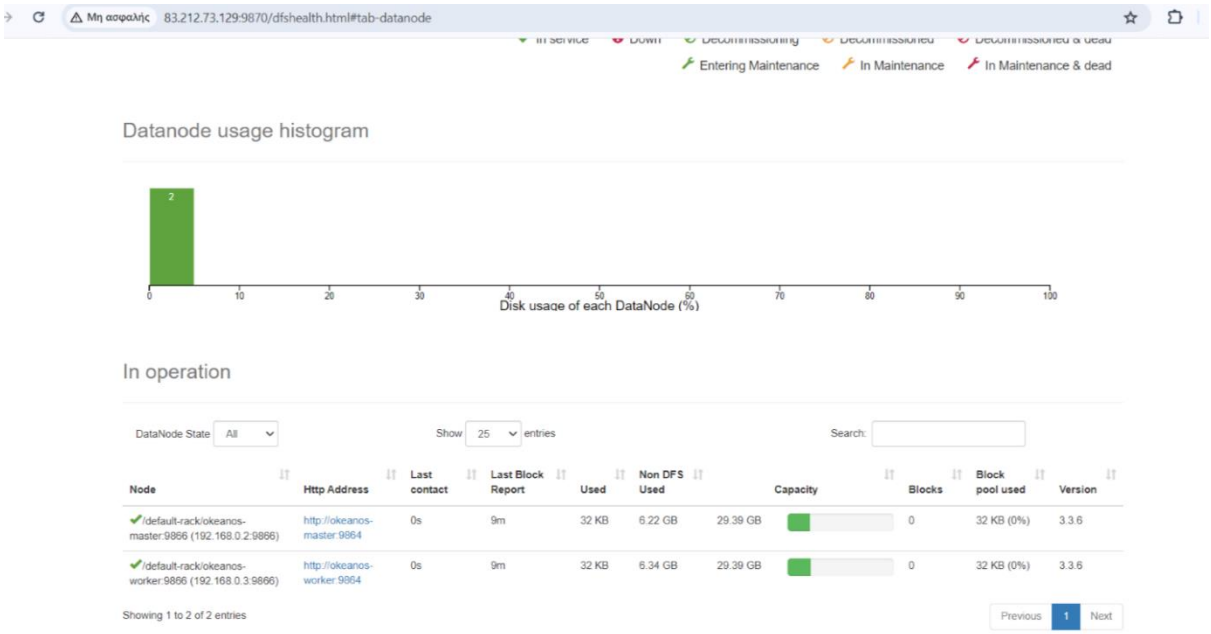
ΕΞΑΜΗΝΙΑΙΑ ΕΡΓΑΣΙΑ

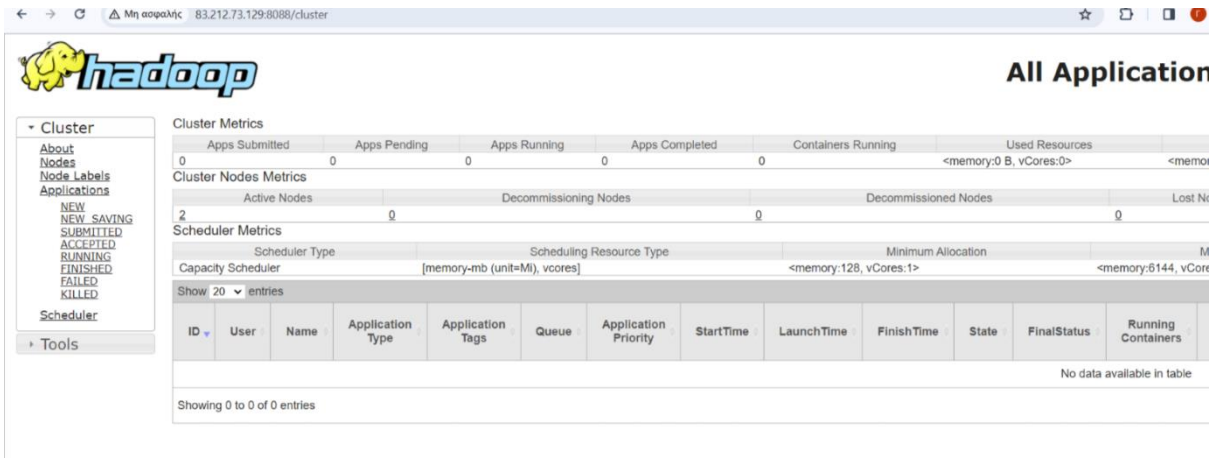
Ομάδα 23
Ονοματεπώνυμο: Καλαράς Γεώργιος **Α.Μ.:** 03118047
Ονοματεπώνυμο: Βαρής Αθανάσιος **Α.Μ.:** 03119606

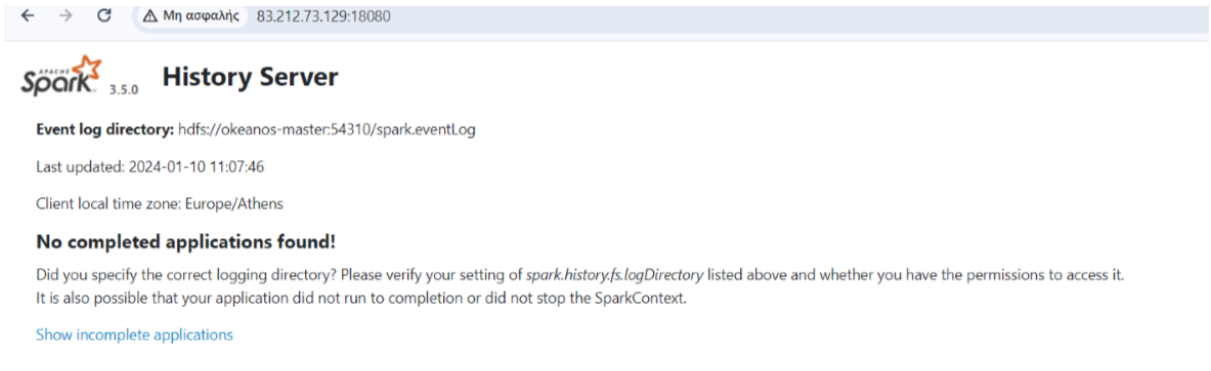
Ζητούμενο 1 :

Έγινε εγκατάσταση και διαμόρφωση του περιβάλλοντος με βάση και τις οδηγίες σε πόρους του okeanos. Οπότε βλέπουμε τα αναμενόμενα στοιχεία για τους 2 κόμβους στις κατάλληλες IP μέσα από έναν browser:




Μη ασφαλής
83.212.73.129:9870/dfshealth.html#tab-datanode
Entering Maintenance
In Maintenance
In Maintenance & dead
Datanode usage histogram
2
0
10
20
30
40
50
60
70
80
90
100
Disk usage of each DataNode (%)
In operation
DataNode State All
Show 25 entries
Search:
Node
Http Address
Last contact
Last Block Report
Used
Non DFS Used
Capacity
Blocks
Block pool used
Version
/default-rack/okeanos-master:9866 (192.168.0.2:9866)
http://okeanos-master:9864
0s
9m
32 KB
6.22 GB
29.39 GB
0
32 KB (0%)
3.3.6
/default-rack/okeanos-worker:9866 (192.168.0.3:9866)
http://okeanos-worker:9864
0s
9m
32 KB
6.34 GB
29.39 GB
0
32 KB (0%)
3.3.6
Showing 1 to 2 of 2 entries
Previous
1
Next


Μη ασφαλής
83.212.73.129:8088/cluster
hadoop
All Application
Cluster
About
Nodes
Node Labels
Applications
NEW
NEW_SAVING
SUBMITTED
ACCEPTED
RUNNING
FINISHED
FAILED
KILLED
Scheduler
Tools
Cluster Metrics
Apps Submitted
Apps Pending
Apps Running
Apps Completed
Containers Running
Used Resources
0
0
0
0
0
<memory:0 B, vCores:0>
Cluster Nodes Metrics
Active Nodes
Decommissioning Nodes
Decommissioned Nodes
Lost N
2
0
0
0
Scheduler Metrics
Scheduler Type
Scheduling Resource Type
Minimum Allocation
Capacity Scheduler
[memory-mb (unit=Mi), vcores]
<memory:128, vCores:1>
<memory:6144, vCore
Show 20 entries
ID
User
Name
Application Type
Application Tags
Queue
Application Priority
StartTime
LaunchTime
FinishTime
State
FinalStatus
Running Containers
No data available in table
Showing 0 to 0 of 0 entries


Μη ασφαλής
83.212.73.129:18080
Spark 3.5.0
History Server
Event log directory: hdfs://okeanos-master:54310/spark.eventLog
Last updated: 2024-01-10 11:07:46
Client local time zone: Europe/Athens
No completed applications found!
Did you specify the correct logging directory? Please verify your setting of spark.history.fs.logDirectory listed above and whether you have the permissions to access it.
It is also possible that your application did not run to completion or did not stop the SparkContext.
Show incomplete applications

Ζητούμενο 2 :

Στο ερώτημα αυτό έπρεπε να δημιουργήσουμε ένα DataFrame από το βασικό σύνολο δεδομένων, διατηρώντας τα αρχικά ονόματα στηλών και προσαρμόζοντας κάποιους τύπους δεδομένων σύμφωνα με την εκφώνηση και τελικά να τυπώσουμε τον συνολικό αριθμό γραμμών του συνόλου δεδομένων και τον τύπο της κάθε στήλης. Παρακάτω φαίνεται ο κώδικας που γράψαμε για την υλοποίηση αυτού του ερωτήματος καθώς επίσης και το αποτέλεσμα που λάβαμε :

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
from pyspark.sql.types import DateType, IntegerType, DoubleType

spark = SparkSession.builder.appName("LosAngelesCrime").getOrCreate()

# Define the file path of your main dataset CSV file
main_dataset_path = "hdfs://okeanos-master:54310/user/user/ALLDATA/Crime_Data_from_2010_to_2019.csv"
main_dataset_path_sec = "hdfs://okeanos-master:54310/user/user/ALLDATA/Crime_Data_from_2020_to_Present.csv"

# Schema of Dataframe
main_dataset_schema = ("DR_NO INT, Date_Rptd STRING, DATE_OCC STRING, TIME_OCC STRING, AREA INT, "
                       "AREA_NAME STRING, RPTDISTNO INT, PART12 INT, Crm_Cd INT, CrmCd_Desc STRING, Mocodes INT, "
                       "Vict_Age INT, Vict_Sex STRING, Vict_Descent STRING, Premis_Cd INT, "
                       "Premis_Desc STRING, Weapon_Used_Cd INT, Weapon_Desc STRING, "
                       "Status STRING, Status_Desc STRING, Crm_Cd1 INT, Crm_Cd2 INT, Crm_Cd3 INT, Crm_Cd4 INT, "
                       "LOCATION STRING, Cross_Street STRING, LAT FLOAT, LON FLOAT"
                       )

# Creation of Dataframes (wiht the desired schemas) from CSVs
main_dataset = spark.read.csv(main_dataset_path, header=True, sep=',', schema=main_dataset_schema)
main_dataset_sec = spark.read.csv(main_dataset_path_sec, header=True, sep=',', schema=main_dataset_schema)
main_dataset_df = main_dataset.union(main_dataset_sec)

# Convert columns to the specified types
main_dataset_df = main_dataset_df.withColumn("Date_Rptd", col("Date_Rptd").cast(DateType()))
main_dataset_df = main_dataset_df.withColumn("DATE_OCC", col("DATE_OCC").cast(DateType()))
main_dataset_df = main_dataset_df.withColumn("Vict_Age", col("Vict_Age").cast(IntegerType()))
main_dataset_df = main_dataset_df.withColumn("LAT", col("LAT").cast(DoubleType()))
main_dataset_df = main_dataset_df.withColumn("LON", col("LON").cast(DoubleType()))

# Show the total number of rows and data types of each column
main_dataset_df.printSchema()
print("Total number of rows: ", main_dataset_df.count())
```

```
root
 |-- DR_NO: integer (nullable = true)
 |-- Date_Rptd: date (nullable = true)
 |-- DATE_OCC: date (nullable = true)
 |-- TIME_OCC: string (nullable = true)
 |-- AREA: integer (nullable = true)
 |-- AREA_NAME: string (nullable = true)
 |-- RPTDISTNO: integer (nullable = true)
 |-- PART12: integer (nullable = true)
 |-- Crm_Cd: integer (nullable = true)
 |-- CrmCd_Desc: string (nullable = true)
 |-- Mocodes: integer (nullable = true)
 |-- Vict_Age: integer (nullable = true)
 |-- Vict_Sex: string (nullable = true)
 |-- Vict_Descent: string (nullable = true)
 |-- Premis_Cd: integer (nullable = true)
 |-- Premis_Desc: string (nullable = true)
 |-- Weapon_Used_Cd: integer (nullable = true)
 |-- Weapon_Desc: string (nullable = true)
 |-- Status: string (nullable = true)
 |-- Status_Desc: string (nullable = true)
 |-- Crm_Cd1: integer (nullable = true)
 |-- Crm_Cd2: integer (nullable = true)
 |-- Crm_Cd3: integer (nullable = true)
 |-- Crm_Cd4: integer (nullable = true)
 |-- LOCATION: string (nullable = true)
 |-- Cross_Street: string (nullable = true)
 |-- LAT: double (nullable = true)
 |-- LON: double (nullable = true)
```

```
24/01/15 10:00:08 INFO DAGScheduler: Job 1 finished: count at NativeMethodAccessorImpl.java:0, took 0.326222 s
Total number of rows:  2769681
24/01/15 10:00:08 INFO SparkContext: Invoking stop() from shutdown hook
```

Ζητούμενο 3 :

Στο ερώτημα αυτό υλοποιήσαμε το **Query 1** χρησιμοποιώντας DataFrame και SQL APIs. Και οι δύο υλοποιήσεις εκτελέστηκαν με 4 Spark executors. Παρακάτω βλέπουμε τους κώδικες των δυο υλοποιήσεων και τα αποτελέσματα που λάβαμε :

DataFrame:

```
 from pyspark.sql.window import Window
from pyspark.sql import SparkSession
from pyspark.sql.functions import col,month,year,count,to_timestamp,row_number,desc
from pyspark.sql.types import DateType, IntegerType, DoubleType

# Create a Spark session
spark = SparkSession.builder.appName("Query1").getOrCreate()

# Define the file path of your main dataset CSV file
main_dataset_path = "hdfs://okeanos-master:54310/user/user/ALLDATA/Crime_Data_from_2010_to_2019.csv"
main_dataset_path_sec = "hdfs://okeanos-master:54310/user/user/ALLDATA/Crime_Data_from_2020_to_Present.csv"

column_name = ("DATE OCC")

# Creation of Dataframes (wiht the desired schemas) from CSVs
main_dataset = spark.read.csv(main_dataset_path, header=True).select(to_timestamp(col(column_name), 'MM/dd/yyyy hh:mm:ss a').alias('DATE_OCC'))
main_dataset_sec = spark.read.csv(main_dataset_path_sec, header=True).select(to_timestamp(col(column_name), 'MM/dd/yyyy hh:mm:ss a').alias('DATE_OCC'))
main_dataset_df = main_dataset.union(main_dataset_sec)
#main_dataset_df.show()
#main_dataset_df.printSchema

# New DF with year and month columns from the "DATE_OCC" column
grouped_df = main_dataset_df.withColumn("Year", year("DATE_OCC"))\
                            .withColumn("Month", month("DATE_OCC"))
#grouped_df.show()
#grouped_df.printSchema()

# Group by year and month, and count occurrences
result_df = grouped_df.groupBy("Year", "Month").agg(count("*").alias("Occurrence/Crime total"))

#result_df.show()
#result_df.printSchema()

# Define a window specification for each year, ordered by the count in descending order
window_spec = Window.partitionBy("Year").orderBy(desc("Occurrence/Crime total"))

# Rank the rows within each year based on the count row_number => arithmei kathe row tou window spec
result_df = result_df.withColumn("Rank", row_number().over(window_spec))

#Keep only top 3
result_df = result_df.filter(col("Rank") <= 3)

result_df.show(50)
result_df.printSchema()
```

Εκτελέστηκε με την εξής εντολή :

```
user@okeanos-master:~/opt/scripts$ spark-submit --num-executors 4 query1_Dataframe.py
```

Και τα αποτελέσματα που λάβαμε :

year	month	crime_total	#
2010	1	12930	1
2010	3	12873	2
2010	4	12301	3
2011	1	21321	1
2011	3	20987	2
2011	10	20866	3
2012	1	31423	1
2012	8	31041	2
2012	10	30921	3
2013	1	8691	1
2013	8	8008	2
2013	12	8001	3
2014	5	5296	1
2014	6	5248	2
2014	7	4830	3
2015	3	10200	1
2015	5	10018	2
2015	7	9785	3
2016	12	16670	1
2016	10	16616	2
2016	1	16430	3
2017	1	27731	1
2017	3	27560	2
2017	7	27150	3
2018	1	6259	1
2018	8	5815	2
2018	7	5632	3
2019	7	19122	1
2019	8	18979	2
2019	3	18858	3
2020	1	5259	1
2020	2	5132	2
2020	5	4891	3
2021	7	28315	1
2021	10	28257	2
2021	8	27674	3
2022	7	31647	1
2022	8	31448	2
2022	10	30880	3
2023	8	25665	1
2023	7	25642	2
2023	1	25536	3

Από το web UI λάβαμε τις παρακάτω πληροφορίες, όπου παρατηρούμε ότι το query εκτελέστηκε σε 1 λεπτό και 3 δευτερόλεπτα :

Application application_1705318973565_0069

User:	user
Name:	Query1
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Tue Jan 16 09:46:21 +0200 2024
Launched:	Tue Jan 16 09:46:21 +0200 2024
Finished:	Tue Jan 16 09:47:25 +0200 2024
Elapsed:	1mins, 3sec
Tracking URL:	History
Log Aggregation Status:	DISABLED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

SQL API:

```
from pyspark.sql.window import Window
from pyspark.sql import SparkSession
from pyspark.sql.functions import col,month,year,count,to_timestamp,row_number,desc
from pyspark.sql.types import DateType, IntegerType, DoubleType

# Create a Spark session
spark = SparkSession.builder.appName("Query1").getOrCreate()

# Define the file path of your main dataset CSV file
main_dataset_path = "hdfs://okeanos-master:54310/user/user/ALLDATA/Crime_Data_from_2010_to_2019.csv"
main_dataset_path_sec = "hdfs://okeanos-master:54310/user/user/ALLDATA/Crime_Data_from_2020_to_Present.csv"

column_name = ("DATE OCC")

# Creation of Dataframes (wiht the desired schemas) from CSVs
main_dataset = spark.read.csv(main_dataset_path, header=True).select(to_timestamp(col(column_name), 'MM/dd/yyyy hh:mm:ss a').alias('DATE_OCC'))
main_dataset_sec = spark.read.csv(main_dataset_path_sec, header=True).select(to_timestamp(col(column_name), 'MM/dd/yyyy hh:mm:ss a').alias('DATE_OCC'))
main_dataset_df = main_dataset.union(main_dataset_sec)

main_dataset_df.createOrReplaceTempView("mainDF")

query = "SELECT EXTRACT(YEAR FROM DATE_OCC) AS year, EXTRACT(MONTH FROM DATE_OCC) AS month FROM mainDF"
help_df = spark.sql(query)

help_df.registerTempTable("help")

query2 = "select year,month,count(*) as crime_total FROM help group by year,month having count(*) > 2 order by year,month"

help_df2 = spark.sql(query2)
help_df2.registerTempTable("help2")

query3 = "SELECT year,month,crime_total,count \
        FROM (Select year,month,crime_total, ROW_NUMBER() OVER (PARTITION BY year ORDER BY crime_total DESC) as rank, ROW_NUMBER() OVER (PARTITION BY year ORDER BY crime_total DESC) % 4 as count \
        FROM help2) ranked \
        WHERE rank <=3"

spark.sql(query3).show(50)
```

Εκτελέστηκε με την εξής εντολή :

```
user@okeanos-master:~/opt/scripts$ spark-submit --num-executors 4 query1_SQL.py
```

Τα αποτελέσματα που λάβαμε ήταν ίδια με αυτά της υλοποίησης με το DataFrame (Δεν παρουσιάζονται ξανά για λόγους ευανάγνωστης αναφοράς).

Από το web UI λάβαμε τις παρακάτω πληροφορίες, όπου παρατηρούμε ότι το query εκτελέστηκε σε 1 λεπτό και 10 δευτερόλεπτα :

Application application_1705318973565_0074

User:	user
Name:	Query1
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Tue Jan 16 10:01:24 +0200 2024
Launched:	Tue Jan 16 10:01:25 +0200 2024
Finished:	Tue Jan 16 10:02:35 +0200 2024
Elapsed:	1mins, 10sec
Tracking URL:	History
Log Aggregation Status:	DISABLED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Από τους χρόνους που λάβαμε δεν μπορούμε σχετικά με το αν υπάρχει ουσιώδης διαφορά στην απόδοση των δύο API. Θεωρούμε ότι η διαφορά των 7 δευτερολέπτων για την ολοκλήρωση των 2 εκτελέσεων είναι αμελητέα και δεν αρκεί για να χαρακτηρίσει αποδοτικότερο το API DataFrame. Από την άλλη μία διαφορά που παρατηρήθηκε έγκειται στην ταχύτητα παραγωγής κώδικα. Ειδικότερα, ο εξοικειωμένος με την Python προγραμματιστής θα επωφεληθεί περισσότερο από το DataFrame API (Αντίστοιχα και ο εξοικειωμένος με την SQL προγραμματιστής με το SQL API).

Ζητούμενο 4 :

Στο ερώτημα αυτό υλοποιήσαμε το **Query 2** χρησιμοποιώντας DataFrame και RDD APIs. Και οι δύο υλοποιήσεις εκτελέστηκαν με 4 Spark executors. Παρακάτω βλέπουμε τους κώδικες των δυο υλοποιήσεων και τα αποτελέσματα που λάβαμε :

DataFrame:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col,month,year,count,desc,when
from pyspark.sql.types import IntegerType
import time

# Create a Spark session
spark = SparkSession.builder.appName("Query2_Dataframe").getOrCreate()

# Define the file path of your main dataset CSV file
main_dataset_path = "hdfs://okeanos-master:54310/user/user/ALLDATA/Crime_Data_from_2010_to_2019.csv"
main_dataset_path_sec = "hdfs://okeanos-master:54310/user/user/ALLDATA/Crime_Data_from_2020_to_Present.csv"

main_dataset = spark.read.csv(main_dataset_path, header=True).select("TIME OCC","Premis Desc")
main_dataset_sec = spark.read.csv(main_dataset_path_sec, header=True).select("TIME OCC","Premis Desc")
main_dataset_df = main_dataset.union(main_dataset_sec)

def categorize_time(ts_column):
    return when((ts_column > '500') & (ts_column < '1159'), 'Morning')\
        .when((ts_column > '1200') & (ts_column < '1659'), 'Afternoon')\
        .when((ts_column > '1700') & (ts_column < '2059'), 'Night')\
        .otherwise('Late_night')

#Start time measurment
start_time = time.time()

filtered_df = main_dataset_df.filter(main_dataset_df["Premis Desc"]=="STREET")
filtered_df = filtered_df.withColumn("TIME OCC",col("TIME OCC").cast(IntegerType()))


filtered_df = filtered_df.withColumn('Time_Category', categorize_time(col('TIME OCC')))

result_df = (
    filtered_df.groupBy('Time_Category')
    .agg(count('*').alias('Occurrences'))
    .orderBy(desc('Occurrences'))
)

end_time = time.time()
result_df.show()
print ("Elapsed time only for the query calculation: ",end_time - start_time ,"seconds")
```

Εκτελέστηκε με την εξής εντολή :

```
user@okeanos-master:~/opt/scripts$ spark-submit --num-executors 4 query2_Dataframe.py
```

Τα αποτελέσματα που λάβαμε είναι τα παρακάτω:

```
+-------------+-----------+
|Time_Category|Occurrences|
+-------------+-----------+
|   Late_night|     264531|
|        Night|     153689|
|    Afternoon|     122338|
|      Morning|     112947|
+-------------+-----------+

Elapsed time only for the query calculation:  0.8892695903778076 seconds
```

Αξίζει να σημειωθεί ότι ο χρόνος που φαίνεται κάτω από τα αποτελέσματα αφορά μόνο τον χρόνο που χρειάστηκε ο υπολογισμός του query και όχι ολόκληρη η εκτέλεση του προγράμματος.

Ο συνολικός χρόνος εκτέλεσης του προγράμματος φαίνεται από τις πληροφορίες που μας παρέχει το web UI:

Application application_1705318973565_0138

User:	user
Name:	Query2_Dataframe
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Wed Jan 17 10:36:33 +0200 2024
Launched:	Wed Jan 17 10:36:34 +0200 2024
Finished:	Wed Jan 17 10:37:20 +0200 2024
Elapsed:	46sec
Tracking URL:	History
Log Aggregation Status:	DISABLED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

RDD API:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession
import time

# Create a Spark session
spark = SparkSession.builder.appName("Query2_RDD").getOrCreate()

# Define the file path of your main dataset CSV file
main_dataset_path = "hdfs://okeanos-master:54310/user/user/ALLDATA/Crime_Data_from_2010_to_2019.csv"
main_dataset_path_sec = "hdfs://okeanos-master:54310/user/user/ALLDATA/Crime_Data_from_2020_to_Present.csv"

# Read CSV files and select required columns
main_dataset_rdd = spark.read.csv(main_dataset_path, header=True).select("TIME OCC", "Premis Desc").rdd
main_dataset_sec_rdd = spark.read.csv(main_dataset_path_sec, header=True).select("TIME OCC", "Premis Desc").rdd
main_dataset_rdd = main_dataset_rdd.union(main_dataset_sec_rdd)

# Define function to categorize time
def categorize_time(ts_column):
    return (
        "Morning" if 500 < ts_column < 1159 else
        "Afternoon" if 1200 < ts_column < 1659 else
        "Night" if 1700 < ts_column < 2059 else
         "Late_night"
    )

#Start time
start_time = time.time()

filtered_rdd = main_dataset_rdd.filter(lambda row: row["Premis Desc"] == "STREET")
filtered_rdd = filtered_rdd.map(lambda row: (int(row["TIME OCC"]), row["Premis Desc"]))
filtered_rdd = filtered_rdd.map(lambda row: (row[0], row[1], categorize_time(row[0])))
# Count occurrences for each time category
result_rdd = filtered_rdd.map(lambda row: (row[2], 1)).reduceByKey(lambda x, y: x + y)
result_rdd = result_rdd.sortBy(lambda x: x[1], ascending=False)

end_time = time.time()

# Display the final result
print("Result RDD:")
print(result_rdd.collect())
print("Elapsed time only for the query calculation: ",end_time - start_time ,"seconds")
```

Εκτελέστηκε με την εξής εντολή :

```
user@okeanos-master:~/opt/scripts$ spark-submit --num-executors 4 query2_RDD.py
```

Τα αποτελέσματα που λάβαμε είναι τα παρακάτω:

```
[('Late_night', 264531), ('Night', 153689), ('Afternoon', 122338), ('Morning', 112947)]
Elapsed time only for the query calculation:  25.47447633743286 seconds
24/01/17 11:02:35 INFO SparkContext: Invoking stop() from shutdown hook
```

Όπως και στην Dataframe υλοποίηση ο χρόνος που αναγράφεται εδώ αφορά αποκλειστικά τον υπολογισμό του query και δεν αναφέρεται στο συνολικό χρόνο εκτέλεσης του προγράμματος ο οποίος φαίνεται παρακάτω :

Application application_1705318973565_0143

User:	user
Name:	Query2_RDD
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Wed Jan 17 10:57:46 +0200 2024
Launched:	Wed Jan 17 10:57:47 +0200 2024
Finished:	Wed Jan 17 10:59:10 +0200 2024
Elapsed:	1mins, 23sec
Tracking URL:	History
Log Aggregation Status:	DISABLED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Παρατηρώντας τους παραπάνω χρόνους παρατηρούμε και πειραματικά ότι γνωρίζουμε ήδη θεωρητικά ότι δηλαδή **η υλοποίηση με Dataframe είναι ταχύτερη/πιο αποδοτική σε σχέση με την υλοποίηση με RDD.**

Ένας από τους λόγους που ισχύει αυτό είναι η υλοποίηση με τα Dataframe χρησιμοποιεί τον Catalyst Optimizer ο οποίος επανατοποθετεί τις εντολές σε καλύτερη σειρά, «κλαδεύει» αχρείαστες πράξεις/εντολές και γενικά χρησιμοποιεί διάφορες τεχνικές με στόχο την ταχύτερη εκτέλεση του προγράμματος/query (Λειτουργεί ως ενός είδους compiler).

Από την άλλη η υλοποίηση με RDD API δεν χρησιμοποιεί καθόλου τον Catalyst Optimizer με συνέπεια να μην έχει πρόσβαση στις βελτιστοποιήσεις που αυτός προσφέρει.

Ζητούμενο 5 :

Εδώ, υλοποιήσαμε το **Query 3** με DataFrame API, για διαφορετικό πλήθος Spark executors (συγκεκριμένα για 2, 3 και 4). Ο κώδικας που χρησιμοποιήσαμε και τα αποτελέσματα που προέκυψαν φαίνονται παρακάτω.

Ο κώδικας, κοινός για όλες τις περιπτώσεις:

```
from pyspark.sql.window import Window
from pyspark.sql import SparkSession
from pyspark.sql.functions import col,month,year,count,to_timestamp,row_number,desc
from pyspark.sql.types import DateType, IntegerType, DoubleType

# Spark session
spark = SparkSession.builder.appName("Query3_DataFrame").getOrCreate()

# The file paths of the CSV files
main_dataset_path = "hdfs://okeanos-master:54310/user/user/ALLDATA/Crime_Data_from_2010_to_2019.csv"
median_household_path = "hdfs://okeanos-master:54310/user/user/ALLDATA/LA_income_2015.csv"
reverse_geocoding_path = "hdfs://okeanos-master:54310/user/user/ALLDATA/revgecoding.csv"


main_dataset_df = spark.read.csv(main_dataset_path, header=True).select("DATE OCC","Vict Descent","LAT","LON")
median_household_df = spark.read.csv(median_household_path, header=True).select("Zip Code","Estimated Median Income")
rev_geocoding_df = spark.read.csv(reverse_geocoding_path, header=True).select("LAT","LON","ZIPcode" )

# Only data for 2015
main_dataset_2015 = main_dataset_df.filter(col("DATE OCC").like("%2015%"))

# filtering for incorrect Victim Descent
filtered_dataset_df = main_dataset_2015.filter((col("Vict Descent").isNotNull()) & (col("Vict Descent") != "Unknown") & (col("Vict Descent") != 'O'))

# crime data with ZIP codes from reverse geocoding
dataset_with_zip = filtered_dataset_df.join(rev_geocoding_df, ["LAT", "LON"], "inner")

# adding income information for each ZIP code
dataset_with_income = dataset_with_zip.join(median_household_df, dataset_with_zip["ZIPcode"] == median_household_df["Zip Code"], "inner")

# Finding the desired ZIP codes from the ZIP codes under examination
dataset_by_income = dataset_with_income.groupBy("Zip Code", "Estimated Median Income").agg(count("*").alias("Number of Victims"))

# The 3 ZIP codes with the highest income
top_income_zipcodes = dataset_by_income.orderBy(desc("Estimated Median Income")).limit(3).select("Zip Code").collect()
top_income_zipcodes = [row["Zip Code"] for row in top_income_zipcodes]

# The 3 ZIP codes with the lowest income
bottom_income_zipcodes = dataset_by_income.orderBy("Estimated Median Income").limit(3).select("Zip Code").collect()
bottom_income_zipcodes = [row["Zip Code"] for row in bottom_income_zipcodes]

# All 6 ZIP codes
all_income_zipcodes = top_income_zipcodes + bottom_income_zipcodes
```

```
# Dataset with only the 6 ZIP codes (the 3 highest and the 3 lowest ZIP codes by income)
combined_dataset = dataset_with_income.filter(col("ZIPcode").isin(all_income_zipcodes))

# number of victims per ethnic group for each ZIP code
result_df = combined_dataset.groupBy("Vict Descent", "ZIPcode").agg(count("*").alias("Number of Victims"))

# results ordered by ZIP code
result_df = result_df.orderBy("ZIPcode", desc("Number of Victims"))

# showing the results
result_df.show(100)
```

Τα αποτελέσματα για τους 6 ζητούμενους ZIP codes που προέκυψαν, προφανώς τα ίδια και για τις τρεις υλοποιήσεις, είναι:

Vict Descent	ZIPcode	Number of Victims
W	90067	3
W	90094	124
B	90094	36
H	90094	31
X	90094	13
A	90094	10
F	90094	1
W	90292	636
H	90292	82
B	90292	75
X	90292	69
A	90292	27
F	90292	1
W	91214	11
H	91214	1
W	91326	94
H	91326	21
A	91326	15
B	91326	4
X	91326	3
W	91604	985
H	91604	164
B	91604	67
A	91604	35
X	91604	6
F	91604	4

Έτσι, τρέχοντας για 2 Spark executors την εντολή:

```
user@okeanos-master:~/opt/scripts$ spark-submit --num-executors 2 query3_Dataframe.py
```

Παίρνουμε ότι ο συνολικός χρόνος εκτέλεσης είναι 59 δευτερόλεπτα, όπως προκύπτει από το web UI:

Application application_1705318973565_0165

User:	user
Name:	Query3_DataFrame
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Wed Jan 17 13:19:30 +0200 2024
Launched:	Wed Jan 17 13:19:31 +0200 2024
Finished:	Wed Jan 17 13:20:30 +0200 2024
Elapsed:	59sec
Tracking URL:	History
Log Aggregation Status:	DISABLED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Για 3 Spark executors:

```
user@okeanos-master:~/opt/scripts$ spark-submit --num-executors 3 query3_Dataframe.py
```

Ο συνολικός χρόνος εκτέλεσης είναι 1 λεπτό και 9 δευτερόλεπτα:

Application application_1705318973565_0166

User:	user
Name:	Query3_DataFrame
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Wed Jan 17 13:23:11 +0200 2024
Launched:	Wed Jan 17 13:23:11 +0200 2024
Finished:	Wed Jan 17 13:24:21 +0200 2024
Elapsed:	1mins, 9sec
Tracking URL:	History
Log Aggregation Status:	DISABLED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Για 4 Spark executors:

```
user@okeanos-master:~/opt/scripts$ spark-submit --num-executors 4 query3_Dataframe.py
```

Συνολικός χρόνος εκτέλεσης 1 λεπτό και 7 δευτερόλεπτα:

Application application_1705318973565_0167

User:	user
Name:	Query3_DataFrame
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Wed Jan 17 13:27:22 +0200 2024
Launched:	Wed Jan 17 13:27:23 +0200 2024
Finished:	Wed Jan 17 13:28:30 +0200 2024
Elapsed:	1mins, 7sec
Tracking URL:	History
Log Aggregation Status:	DISABLED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Παρατηρούμε ότι και στις 3 περιπτώσεις έχουμε παρόμοιους χρόνους εκτέλεσης. Συγκεκριμένα, πρόκειται για περίπου 1 λεπτό σε κάθε περίπτωση με μικρές διαφοροποιήσεις, όπου για 2 Spark executors έχουμε λίγο γρηγορότερη εκτέλεση.

Ζητούμενο 6 :

Εδώ μας ζητείται να υλοποιήσουμε το query 4. Σε αυτό έχουμε 2 σκέλη.

Στο **πρώτο** υπολογίζουμε:

a) Τον αριθμό εγκλημάτων με χρήση πυροβόλων όπλων και την μέση απόστασή τους (σε km) από το αστυνομικό τμήμα που ανάλαβε την υπόθεση ανά έτος
b) Τον αριθμό εγκλημάτων με χρήση πυροβόλων όπλων και την μέση απόστασή τους (σε km) από το αστυνομικό τμήμα που ανάλαβε την υπόθεση ανά αστυνομικό τμήμα

Για να τα υλοποιήσουμε χρησιμοποιούμε τον παρακάτω κώδικα DataFrame API:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col,month,year,avg,count,to_timestamp,row_number,desc,when
from pyspark.sql.types import DateType, IntegerType, DoubleType
import time
#import geopy.distance
from math import radians, sin, cos, sqrt, atan2

# calculate the distance between two points [lat1, long1], [lat2, long2] in km
#def get_distance(lat1, long1, lat2, long2):
#    return geopy.distance.geodesic((lat1, long1), (lat2, long2)).km

# other ways to calculate distance
# calculating haversine distance (from coordinates to km)
def get_distance_2(lat1, long1, lat2, long2):
    # Convert latitude and longitude from degrees to radians
    lat1, long1, lat2, long2 = map(radians, [lat1, long1, lat2, long2])
    # Haversine formula
    dlat = lat2 - lat1
    dlon = long2 - long1
    a = sin(dlat / 2) ** 2 + cos(lat1) * cos(lat2) * sin(dlon / 2) ** 2
    c = 2 * atan2(sqrt(a), sqrt(1 - a))
    # Radius of the Earth in kilometers (mean value) : radius = 6371.0
    # Calculate and return the distance
    return 6371.0 * c


# Create a Spark session
spark = SparkSession.builder.appName("Query4_First_Dataframe").getOrCreate()

# File paths of the CSV files
main_dataset_path = "hdfs://okeanos-master:54310/user/user/ALLDATA/Crime_Data_from_2010_to_2019.csv"
main_dataset_path_sec = "hdfs://okeanos-master:54310/user/user/ALLDATA/Crime_Data_from_2020_to_Present.csv"
police_station_path = "hdfs://okeanos-master:54310/user/user/ALLDATA/LAPD_Police_Stations.csv"

main_dataset = spark.read.csv(main_dataset_path, header=True).select(to_timestamp(col("DATE OCC"), 'MM/dd/yyyy hh:mm:ss a').alias('DATE_OCC'),"AREA ","Weapon Used Cd","LAT","LON")
main_dataset_sec = spark.read.csv(main_dataset_path_sec, header=True).select(to_timestamp(col("DATE OCC"), 'MM/dd/yyyy hh:mm:ss a').alias('DATE_OCC'),"AREA","Weapon Used Cd","LAT","LON")

# Renaming the column in the first dataset to match the second dataset
main_dataset = main_dataset.withColumnRenamed("AREA ", "AREA")

main_dataset_df = main_dataset.union(main_dataset_sec)
police_station_df = spark.read.csv(police_station_path, header=True).select("X","Y","DIVISION","PREC")
```

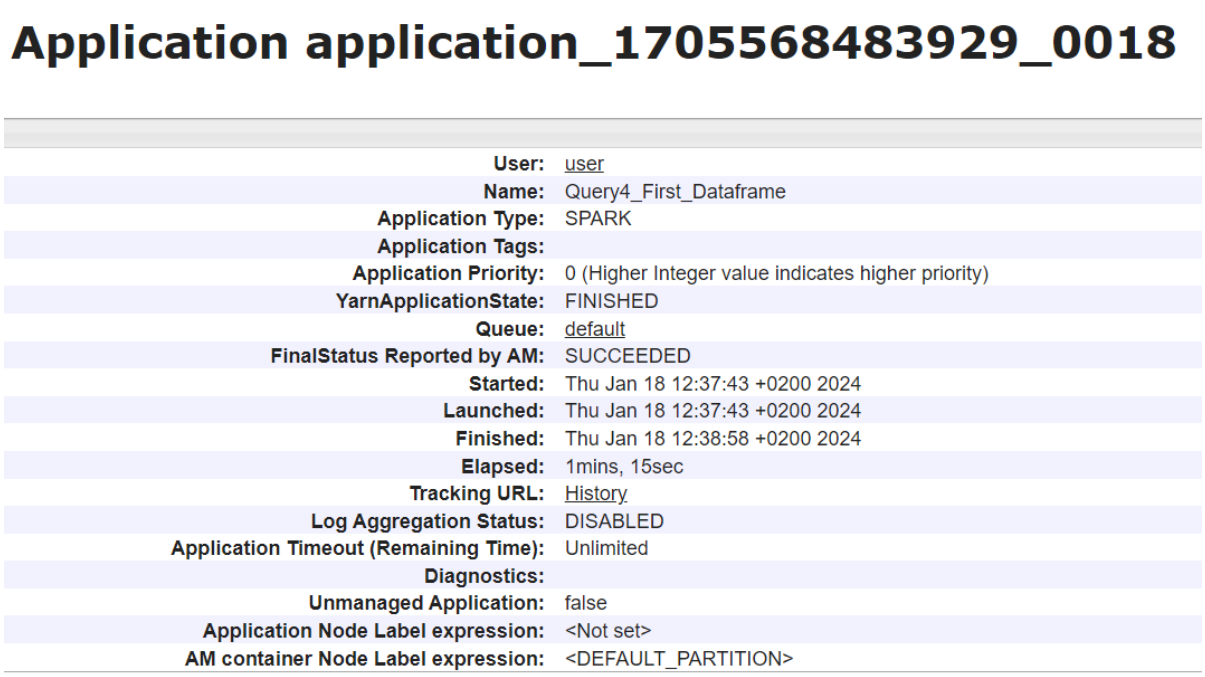
```
# Casting coordinates to DoubleType
main_dataset_df = main_dataset_df.withColumn("LAT",col("LAT").cast(DoubleType()))
main_dataset_df = main_dataset_df.withColumn("LON",col("LON").cast(DoubleType()))
police_station_df = police_station_df.withColumn("X",col("X").cast(DoubleType()))
police_station_df = police_station_df.withColumn("Y",col("Y").cast(DoubleType()))

# Getting Year column from the "DATE_OCC" column
main_dataset_df = main_dataset_df.withColumn("Year", year("DATE_OCC"))

# Filter out Null Island records
main_dataset_df = main_dataset_df.filter((col("LAT") != 0) & (col("LON") != 0))

# Only incidents with firearms (Weapon Used Cd of the form '1xx')
firearm_crimes = main_dataset_df.filter(col("Weapon Used Cd").rlike("[1].*"))

# Registering our own User Defined Function (UDF) to calculate distance
#get_distance_udf = spark.udf.register("get_distance", get_distance)
get_distance_2_udf = spark.udf.register("get_distance_2", get_distance_2)

# Joining firearm_crimes with police_station correctly ("AREA" from main_dataset is the equivalent to "PREC" from police_station)
combined_dataset = firearm_crimes.join(police_station_df, firearm_crimes["AREA"] == police_station_df["PREC"], "inner")

# Calculating distance for each crime from police station
#result = combined_dataset.withColumn("distance", get_distance_udf(col("LAT"), col("LON"), col("X"), col("Y")))
result = combined_dataset.withColumn("distance", get_distance_2_udf(col("LAT"), col("LON"), col("X"), col("Y")))

# Average distance and number of incidents per year
result_a = result.groupBy("Year").agg(avg("distance").alias("average_distance"), count("*").alias("number_of_incidents"))
result_a = result_a.orderBy("Year")

# Display the results
result_a.show()

# Average distance and number of incidents per police station
result_b = result.groupBy("DIVISION").agg(avg("distance").alias("average_distance"), count("*").alias("number_of_incidents"))
result_b = result_b.orderBy(desc("number_of_incidents"))

# Display the results
result_b.show()
```

Από αυτό παίρνουμε τα παρακάτω αποτελέσματα για τα ερωτήματα (α) και (β) αντίστοιχα:

Year	average_distance	number_of_incidents
2010	10932.579224107678	4763
2011	10931.000027341988	11754
2012	10931.113339205027	20506
2013	10923.11318178909	624
2014	10934.79255796961	2703
2015	10929.61745173897	3571
2016	10930.908916500157	6093
2017	10930.669408943533	14906
2018	10943.480320815295	120
2019	10930.42847476975	10451
2021	10930.884232171524	15347
2022	10930.744530612725	20691
2023	10929.906697142114	16721

DIVISION	average_distance	number_of_incidents
77TH STREET	10931.056821356187	27924
SOUTHEAST	10928.51684558062	17266
NEWTON	10936.193218969107	16497
OLYMPIC	10936.774883961521	9499
NORTHEAST	10946.106880950945	9413
WEST VALLEY	10917.214266226902	8846
MISSION	10932.168440005918	7819
NORTH HOLLYWOOD	10935.578025755782	7510
FOOTHILL	10939.393262873306	6858
PACIFIC	10914.42910191871	5959
DEVONSHIRE	10923.57819688867	5590
TOPANGA	10910.2069382779	5069

Για την εκτέλεση αυτού, παίρνουμε και τον χρόνο ολικής εκτέλεσης από το web UI, που είναι 1 λεπτό και 15 δευτερόλεπτα, όπως φαίνεται:



Application application_1705568483929_0018

User:	user
Name:	Query4_First_Dataframe
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Thu Jan 18 12:37:43 +0200 2024
Launched:	Thu Jan 18 12:37:43 +0200 2024
Finished:	Thu Jan 18 12:38:58 +0200 2024
Elapsed:	1mins, 15sec
Tracking URL:	History
Log Aggregation Status:	DISABLED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

(Όλοι οι κώδικες που φαίνονται παραπάνω υπάρχουν και στο παρακάτω GitHub repository)