ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΒΑΣΕΩΝ ΓΝΩΣΕΩΝ ΚΑΙ ΔΕΔΟΜΕΝΩΝ

Προχωρημένα Θέματα Βάσεων Δεδομένων

Ακαδημαϊκό έτος 2023-24, 9ο Εξάμηνο

Διδάσκων: Δημήτριος Τσουμάκος
Υπεύθυνος Εργαστηρίου: Νικόλαος Χαλβαντζής

Εξαμηνιαία Εργασία

Μέλη ομάδας: Γεωργίου Χαρίδημος (el19027), Κιούρα Ιωάννα (el19102)
GitHub repo: https://github.com/ChGeorgiou/Advanced_DB_NTUA

Ζητούμενα

1. Η εγκατάσταση του συστήματος έγινε στους πόρους του okeanos-knossos. Ενδεικτικά παρουσιάζονται τα παρακάτω screenshots από τις οι web εφαρμογές των HDFS, YARN και Spark History Server, στα οποία φαίνεται ότι στο σύστημα υπάρχουν πράγματι 2 ενεργοί κόμβοι:

- HDFS (http://83.212.74.36:9870/dfshealth.html):

Summary

Security is off.

Safemode is off.

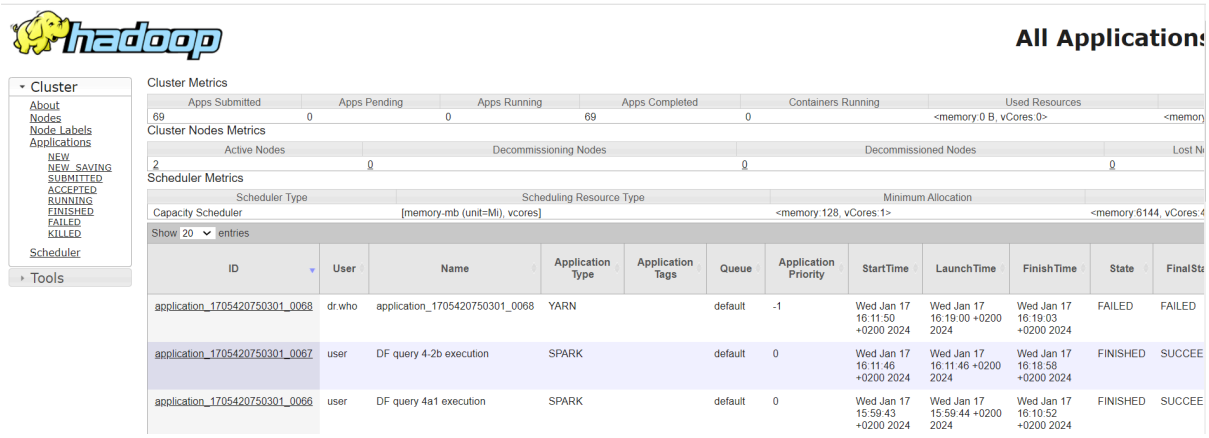
150 files and directories, 133 blocks (133 replicated blocks, 0 erasure coded block groups) = 283 total filesystem object(s).

Heap Memory used 77.71 MB of 278 MB Heap Memory. Max Heap Memory is 1.94 GB.

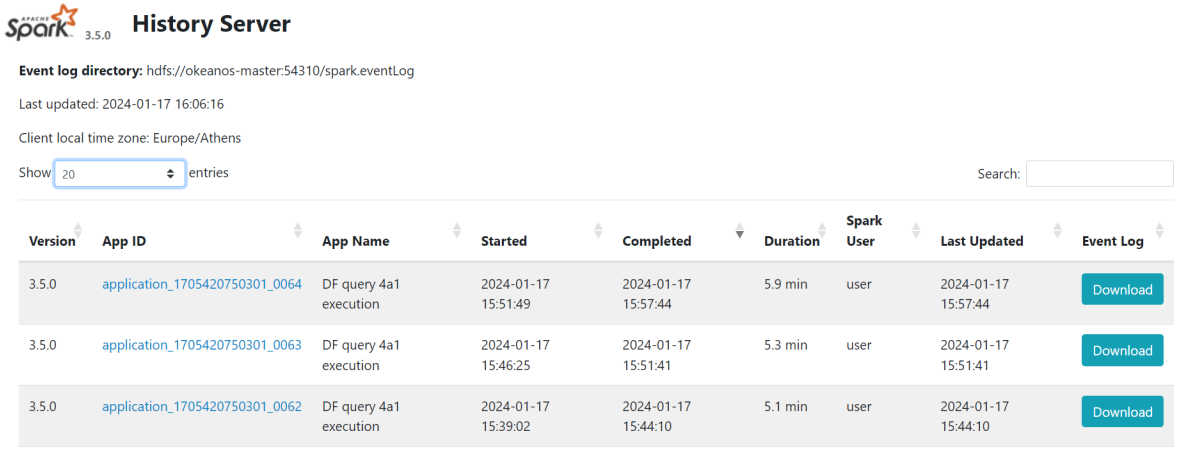
Non Heap Memory used 76.41 MB of 79.06 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	29.39 GB
Configured Remote Capacity:	0 B
DFS Used:	934.18 MB (3.1%)
Non DFS Used:	21.62 GB
DFS Remaining:	5.22 GB (17.75%)
Block Pool Used:	934.18 MB (3.1%)
DataNodes usages% (Min/Median/Max/stdDev):	3.10% / 3.10% / 3.10% / 0.00%
Live Nodes	1 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	1 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	2
Number of Blocks Pending Deletion (including replicas)	0
Block Deletion Start Time	Tue Jan 16 17:58:26 +0200 2024
Last Checkpoint Time	Wed Jan 17 15:59:58 +0200 2024
Enabled Erasure Coding Policies	RS-6-3-1024k

- YARN (http://83.212.74.36:8088/cluster):



- Spark History Server (http://http://83.212.74.36:18080/):



2. Ο κώδικάς μας για αυτό το ζητούμενο είναι ο εξής:

```python
import pandas as pd
d1 = pd.read_csv('Crime_Data_from_2010_to_2019.csv')
d2 = pd.read_csv('Crime_Data_from_2020_to_Present.csv')
d1.rename(columns={"AREA ": "AREA"}, inplace=True)
d2.rename(columns={"AREA ": "AREA"}, inplace=True)
data = pd.concat([d1, d2], ignore_index=True, sort=False).drop_duplicates()
data = data.astype({"Date Rptd": 'datetime64[ns]', "DATE OCC": 'datetime64[ns]',
    "Vict Age": int, "LAT": float, "LON": float})
print("Total number of rows: " + str(len(data)))
print()
print("Data types of all columns:")
print(data.dtypes)
data.to_csv('data.csv', index=False)

import subprocess
hdfs_command = f'hdfs dfs -copyFromLocal -f data.csv /user/user/csv_files/'
subprocess.run(hdfs_command, shell=True)

from pyspark.sql import SparkSession
import time, datetime
spark = SparkSession \
    .builder \
    .appName("SQL query 3 execution") \
    .config("spark.executor.instances", 4) \
    .getOrCreate()

data_df = spark.read.csv("hdfs://okeanos-master:54310/user/user/csv_files/data.csv",
    header=True, inferSchema=True)
data_df.printSchema()
```

Η προεπεξεργασία των δεδομένων (συνένωση datasets, αφαίρεση διπλότυπων γραμμών, αλλαγή τύπων) έγινε τοπικά στον master και στη συνέχεια το τελικό αρχείο CSV αποθηκεύτηκε στο HDFS. Συγκεκριμένα, ένα από τα δύο crime data dataset είχε ονοματισμένη την στήλη “AREA” με ένα κενό μετά, δηλαδή “AREA ”, οπότε την μετονομάσαμε για εξαλειφτεί το κενό. Υπήρχαν επίσης διπλότυπες γραμμές στο dataset μετά την συνένωση των δύο αρχικών, τις οποίες κάναμε drop. Τέλος, αλλάξαμε τον τύπο για όσες στήλες μας ζητήθηκε, τυπώσαμε τον αριθμό σειρών και τους τύπους όλων των στηλών και αποθηκεύσαμε τοπικά το τελικό αρχείο. Για να ανεβάσουμε έπειτα το αρχείο στο hdfs, τρέξαμε την εντολή copyFromLocal μέσα από το python script χρησιμοποιώντας το module subprocess. Για να ελέγξουμε ότι όλα έγιναν όπως τα περιμέναμε, ξεκινήσαμε ένα Spark Session για να κατεβάσουμε το αρχείο από το hdfs και να τυπώσουμε ξανά τους τύπους δεδομένων όλων των στηλων. Ο συνολικός αριθμός γραμμών είναι:

```
Total number of rows: 2335238
```

Και οι τύποι των στηλών:

```
|-- DR_NO: integer (nullable = true)
|-- Date Rptd: date (nullable = true)
|-- DATE OCC: date (nullable = true)
|-- TIME OCC: integer (nullable = true)
|-- AREA: integer (nullable = true)
|-- AREA NAME: string (nullable = true)
|-- Rpt Dist No: integer (nullable = true)
|-- Part 1-2: integer (nullable = true)
|-- Crm Cd: integer (nullable = true)
|-- Crm Cd Desc: string (nullable = true)
|-- Mocodes: string (nullable = true)
|-- Vict Age: integer (nullable = true)
|-- Vict Sex: string (nullable = true)
|-- Vict Descent: string (nullable = true)
|-- Premis Cd: double (nullable = true)
|-- Premis Desc: string (nullable = true)
|-- Weapon Used Cd: double (nullable = true)
|-- Weapon Desc: string (nullable = true)
|-- Status: string (nullable = true)
|-- Status Desc: string (nullable = true)
|-- Crm Cd 1: double (nullable = true)
|-- Crm Cd 2: double (nullable = true)
|-- Crm Cd 3: double (nullable = true)
|-- Crm Cd 4: double (nullable = true)
|-- LOCATION: string (nullable = true)
|-- Cross Street: string (nullable = true)
|-- LAT: double (nullable = true)
|-- LON: double (nullable = true)
```

Οι χρόνοι εκτέλεσης για όλα τα επόμενα ερωτήματα αναφέρονται στον χρόνο που πέρασε ξεκινώντας ακριβώς μετά το διάβασμα του .csv αρχείου με τα δεδομένα και έως την εκτέλεση του query, πριν την εκτύπωση του αποτελέσματος.

3. Για αυτό το ζητούμενο ο κώδικάς μας για την εκτέλεση σε DataFrame API είναι:

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, year, month, dayofmonth, count, dense_rank, desc
from pyspark.sql.window import Window
import time, datetime

spark = SparkSession \
    .builder \
    .appName("DF query 1 execution") \
    .config("spark.executor.instances", 4) \
    .getOrCreate()

data_df = spark.read.csv("hdfs://okeanos-master:54310/user/user/csv_files/data.csv",
                         header=True, inferSchema=True)

start_time = time.time()

d = data_df.withColumn("year", year(col("DATE OCC"))).withColumn("month", month(col("DATE OCC")))
crime_counts = d.groupBy("year", "month").agg(count("*").alias("crime_count"))
window_spec = Window.partitionBy("year").orderBy(desc("crime_count"))
ranked_crimes = crime_counts.withColumn("rank", dense_rank().over(window_spec))
top3_months_per_year = ranked_crimes.filter(col("rank") <= 3)
result = top3_months_per_year.orderBy("year", desc("crime_count"))

end_time = time.time()

result.show(result.count(), truncate=False)
result.write.csv("/user/user/csv_files/Query1_DF", header=True, mode="overwrite")

print('Total time for DF: ', end_time - start_time, 'sec')
```

Αφού δημιουργήσουμε ένα Spark Session και κατεβάσουμε τα δεδομένα μας από το hdfs ως DataFrame, ξεκινάμε την επεξεργασία τους δημιουργώντας 2 επιπλέον στήλες: 1 στήλη με το έτος που συνέβη το κάθε έγκλημα και 1 με τον μήνα που συνέβη. Έπειτα ομαδοποιούμε τις σειρές με βάση αυτές τις νέες στήλες, αριθμώντας τον αριθμό των εγκλημάτων σε κάθε ομαδοποίηση. Τέλος χρησιμοποιούμε dense_rank για να αριθμήσουμε για κάθε χρόνο τους

μήνες του, ανάλογα με τον αριθμό εγκλημάτων που διαπράχτηκαν σε αυτούς, με το 1 να αντιστοιχεί στον μέγιστο. Επιλέχτηκε το dense_rank ώστε να διατηρούνται οι ισοπαλίες ως ισοπαλίες (δεν υπήρχε κάποια συγκεκριμένη καθοδήγηση). Τέλος, κρατάμε μόνο τις ομαδοποιήσεις με rank μεγαλύτερο ή ίσο του 3, ώστε να διατηρηθούν οι μήνες με τους 3 υψηλότερους καταγεγραμμένους αριθμούς εγκλημάτων για κάθε έτος και ταξινομούμε αυτούς τους μήνες με βάση τον αριθμό εγκλημάτων τους για κάθε έτος.

Ο κώδικάς μας για την εκτέλεση σε SQL API:

```
from pyspark.sql import SparkSession
import time, datetime
spark = SparkSession.builder.appName("SQL query 1 execution").config("spark.executor.instances", 4) \
    .getOrCreate()
data_df = spark.read.csv("hdfs://okeanos-master:54310/user/user/csv_files/data.csv",
                         header=True, inferSchema=True)

start_time = time.time()

data_df.createOrReplaceTempView("data")
query = "WITH ranked_crimes AS (\
    SELECT \
        YEAR(`DATE OCC`) AS year, \
        MONTH(`DATE OCC`) AS month, \
        COUNT(*) AS crime_total, \
        DENSE_RANK() OVER (PARTITION BY YEAR(`DATE OCC`) ORDER BY COUNT(*) DESC) AS crime_rank \
    FROM data \
    GROUP BY YEAR(`DATE OCC`), MONTH(`DATE OCC`)) \
SELECT year, month, crime_total, crime_rank \
FROM ranked_crimes \
WHERE crime_rank <= 3 \
ORDER BY year ASC, crime_total DESC "
result = spark.sql(query)

end_time = time.time()

result.show(result.count(), truncate=False)
result.write.csv("/user/user/csv_files/Query1_SQL", header=True, mode="overwrite")
print('Total time for SQL: ', end_time - start_time, 'sec')
```

Σε SQL API ακολουθούμε ακριβώς την ίδια λογική. Αρχικά δημιουργούμε τις νέες στήλες για το έτος και τον μήνα και ομαδοποιούμε βάσει αυτών, υπολογίζοντας παράλληλα και το άθροισμα των εγκλημάτων για κάθε ομαδοποίηση. Εφαρμόζουμε dense_rank για κάθε χρόνο ξεχωριστά και τέλος κρατάμε τους μήνες με τα 3 μεγαλύτερα αθροίσματα για κάθε έτος.

Το αποτέλεσμα του **Query 1** και στις δύο περιπτώσεις είναι:

year	month	crime_count	rank
2010	1	13409	1
2010	3	13334	2
2010	7	13281	3
2011	1	17346	1
2011	7	16562	2
2011	3	16471	3
2012	1	17943	1
2012	8	17661	2
2012	5	17502	3
2013	8	9112	1
2013	7	8956	2
2013	1	8682	3
2014	7	12106	1
2014	8	11990	2
2014	12	11969	3
2015	12	13177	1
2015	10	13171	2
2015	8	12872	3
2016	10	13879	1
2016	8	13713	2
2016	12	13657	3
2017	10	20433	1
2017	7	20193	2
2017	1	19833	3
2018	10	15320	1
2018	12	15087	2
2018	8	14920	3
2019	7	19122	1
2019	8	18979	2
2019	3	18856	3
2020	1	6090	1
2020	2	5869	2
2020	5	5530	3
2021	10	19309	1
2021	7	18661	2
2021	8	18376	3
2022	5	13965	1
2022	7	13824	2
2022	8	13757	3
2023	8	19875	1
2023	7	19814	2
2023	10	19698	3

Ο χρόνος εκτέλεσης για το SQL API είναι 0.467 sec. Η τιμή αυτή αποτελεί τον μέσο όρο 10 εκτελέσεων από 0.39s έως και 0.52s (μια τιμή 1.32s δεν ελήφθη υπόψιν ως πολύ ακραία).

Ο χρόνος εκτέλεσης για το Dataframe API είναι 0.324 sec. Η τιμή αυτή αποτελεί τον μέσο όρο 10 εκτελέσεων από 0.24s έως και 0.45s (μια τιμή 0.59s δεν ελήφθη υπόψιν ως πολύ ακραία).

Σχολιασμός και αιτιολόγηση:

Όπως παρατηρούμε, το DataFrame API είναι λίγο ταχύτερο του SQL API. Η διαφορά αυτή δεν είναι πολύ σημαντική (σίγουρα σε σχέση με αυτή που θα δούμε στο επόμενο ερώτημα). Το γεγονός αυτό μπορεί να εξηγηθεί εύκολα, καθώς τα SQL και DataFrame APIs είναι υψηλού επιπέδου APIs που χρησιμοποιούν τον Catalyst Optimizer για την βελτιστοποίηση των ερωτημάτων αποφεύγοντας περιττά κόστη και μειώνοντας σημαντικά τον χρόνο εκτέλεσης. Το προτεινόμενο execution plan μπορεί βέβαια να διαφέρει και για αυτό να παρατηρούνται διαφορές στην απόδοση. Στην παραπάνω περίπτωση, το DataFrame API ίσως ήταν λίγο αποδοτικότερο, λόγω της τμηματοποίησης του κώδικα, σε αντίθεση με το SQL API όπου γράφτηκε για ολόκληρο το ερώτημα ένα query. Αυτό που εν τέλει οδηγεί σε επιλογή ενός εκ των 2 θα είναι η εξοικείωση του προγραμματιστή με την SQL ή την python για την επιλογή ενός εκ των 2 APIs. Για αυτό και εμείς, μοιράζοντας τα queries, εκτελέσαμε το 3 μέσω SQL API και το 4 μέσω DataFrame, καθώς ο καθένας μας επέλεξε με βάση τις δικές του προτιμήσεις.

4. Ο κώδικάς μας για το SQL API:

```
from pyspark.sql import SparkSession
import time, datetime
spark = SparkSession.builder.appName("SQL query 2 execution").config("spark.executor.instances", 4) \
    .getOrCreate()

data_df = spark.read.csv("hdfs://okeanos-master:54310/user/user/csv_files/data.csv",
                         header=True, inferSchema=True)

start_time = time.time()

data_df.createOrReplaceTempView("data")
query = "SELECT COUNT(*) AS crime_count, \
    CASE \
        WHEN `TIME OCC` BETWEEN 500 AND 1159 THEN 'Morning' \
        WHEN `TIME OCC` BETWEEN 1200 AND 1659 THEN 'Afternoon' \
        WHEN `TIME OCC` BETWEEN 1700 AND 2059 THEN 'Evening' \
        ELSE 'Night' \
    END AS time_of_day \
FROM data \
WHERE `Premis Desc` = 'STREET' \
GROUP BY time_of_day \
ORDER BY crime_count DESC "
result = spark.sql(query)

end_time = time.time()

result.show(result.count(), truncate=False)
result.write.csv("/user/user/csv_files/Query2_SQL", header=True, mode="overwrite")
print('Total time for SQL: ', end_time - start_time, 'sec')
```

Στον κώδικα κρατάμε από το αρχικό σύνολο δεδομένων μόνο όσα εγκλήματα συνέβησαν στον δρόμο (STREET) και δημιουργούμε την στήλη time_of_day, η οποία προκύπτει από την στήλη TIME_OCC που δείχνει τι ώρα διαπράχτηκε το κάθε έγκλημα. Η στήλη time_of_day μπορεί να πάρει μία από 4 τιμές ανάλογα με το range στο οποίο ανήκει η τιμή της στήλη TIME_OCC. Ομαδοποιούμε τα εγκλήματα με βάση την τιμή τους στην στήλη time_of_day και παράλληλα μετράμε πόσα ανήκουν στην κάθε ομάδα. Το τελικό αποτέλεσμα ταξινομείται με βάση τον αριθμό εγκλημάτων για την κάθε ομάδα.

Ο κώδικάς μας για το RDD API:

```python
from pyspark.sql import SparkSession
import time, datetime
spark = SparkSession \
    .builder \
    .appName("RDD query 2 execution") \
    .config("spark.executor.instances", 4) \
    .getOrCreate()

data = spark.read.csv("hdfs://okeanos-master:54310/user/user/csv_files/data.csv",
                      header=True, inferSchema=True)
data_rdd = data.rdd

start_time = time.time()

street = data_rdd.filter(lambda x: x["Premis Desc"] == "STREET")
time_of_day = street.map(lambda x: "Morning" if (int(x[3]) >= 500 and int(x[3]) <= 1159) \
else ("Afternoon" if (int(x[3]) >= 1200 and int(x[3]) <= 1659) \
else ("Evening" if (int(x[3]) >= 1700 and int(x[3]) <= 2059) \
else "Night"))).map(lambda x: (x, 1)).reduceByKey(lambda x,y: x+y).sortBy(lambda x: x[1], False)

end_time = time.time()

output_path = "hdfs://okeanos-master:54310/user/user/csv_files/Query2_RDD"
time_of_day.coalesce(1).saveAsTextFile(output_path)

print(time_of_day.collect())

print('Total time for RDD: ', end_time - start_time, 'sec')
```

Εδώ ξεκινάμε κρατώντας και πάλι μόνο τις σειρές που έχουν εγκλήματα που συνέβησαν στον δρόμο. Έπειτα, κατασκευάζουμε ένα rdd με μία μόνο στήλη που θα περιέχει τις τιμές Morning, Afternoon, Evening, Night ανάλογα με την τιμή της στήλης 3, δηλαδή της TIME_OCC. Για αυτό το rdd εφαρμόζουμε μία συνάρτηση που προσθέτει μία παραπάνω στήλη στην οποία όλες οι γραμμές έχουν τιμή 1. Στην συνέχεια, εφαρμόζουμε μία συνάρτηση η οποία ομαδοποιεί το rdd για όλες τις τιμές της πρώτης στήλης, αθροίζοντας τις τιμές της δεύτερης για κάθε ομδαοποίηση. Τέλος, γίνεται ταξινόμηση με βάση την δεύτερη στήλη (άθροισμα).

Το αποτέλεσμα του **Query 2** (μέσω SQL API και RDD API αντίστοιχα):

crime_count	time_of_day
186375	Night
145617	Evening
115214	Afternoon
96440	Morning

Ο χρόνος εκτέλεσης για το SQL API είναι 0.368 sec. Η τιμή αυτή αποτελεί τον μέσο όρο 10 εκτελέσεων από 0.28s έως και 0.42s.

```
[('Night', 186375), ('Evening', 145617), ('Afternoon', 115214), ('Morning', 96440)]
```

Ο χρόνος εκτέλεσης για το RDD API είναι 17.48 sec. Η τιμή αυτή αποτελεί τον μέσο όρο 10 εκτελέσεων από 16.78s έως και 18.62s.

Σχολιασμός και αιτιολόγηση:

Σε αυτό το ερώτημα, παρατηρούμε πολύ μεγάλη διαφορά στην απόδοση των 2 APIs. Αυτό ήταν αναμενόμενο. Το SQL API είναι υψηλότερου επιπέδου από το RDD API. Έτσι, πριν την εκτέλεση του query λαμβάνει χώρα μια σειρά από βελτιστοποιήσεις, με αποτέλεσμα ο χρόνος εκτέλεσης να μειώνεται πολύ. Αντίθετα, το RDD API είναι χαμηλότερου επιπέδου. Εκτελεί τα ερωτήματα με τη σειρά που τα βλέπει στον κώδικα. Βέβαια, βελτιστοποίηση μπορεί να συμβεί και εδώ αλλά από τον ίδιο τον προγραμματιστή, στα ερωτήματα που θέτει. Αυτό σημαίνει πως από έμπειρους προγραμματιστές, και ανάλογα με το task, το RDD API θα μπορούσε να είναι το ίδιο, αν όχι και περισσότερο, αποδοτικό των SQL/DataFrame APIs. Στην προκειμένη περίπωστη, ωστόσο, δεν ισχύει αυτό. Γενικά, λοιπόν, το SQL (και DataFrame) API είναι πιο εύχρηστο από το RDD και δεν χρειάζεται βελτιστοποίηση στον κώδικα στον ίδιο βαθμό.

5. Ο κώδικάς μας για το Query 3 σε SQL API:

```
1   from pyspark.sql import SparkSession
2   import time, datetime
3   from io import StringIO
4   from contextlib import redirect_stdout
5
6   spark = SparkSession \
7       .builder \
8       .appName("SQL query 3 execution") \
9       .config("spark.executor.instances", 2) \
10      .getOrCreate()
11
12  data_df = spark.read.csv("hdfs://okeanos-master:54310/user/user/csv_files/data.csv",
13                           header=True, inferSchema=True)
14  revgecoding_df = spark.read.csv("hdfs://okeanos-master:54310/user/user/csv_files/revgecoding.csv",
15                                  header=True, inferSchema=True)
16  LA_income_2015_df = spark.read.csv("hdfs://okeanos-master:54310/user/user/csv_files/income/LA_income_2015.
17                                     header=True, inferSchema=True)
18
19  start_time = time.time()
20
21  data_df.createOrReplaceTempView("data")
22  revgecoding_df.createOrReplaceTempView("revgecoding")
23  LA_income_2015_df.createOrReplaceTempView("LA_income_2015")
```

```
query = """
    SELECT CASE
        WHEN d.`Vict Descent` = 'W' THEN 'White'
        WHEN d.`Vict Descent` = 'B' THEN 'Black'
        WHEN d.`Vict Descent` = 'A' THEN 'Other Asian'
        WHEN d.`Vict Descent` = 'C' THEN 'Chinese'
        WHEN d.`Vict Descent` = 'D' THEN 'Cambodian'
        WHEN d.`Vict Descent` = 'F' THEN 'Filipino'
        WHEN d.`Vict Descent` = 'G' THEN 'Guamanian'
        WHEN d.`Vict Descent` = 'H' THEN 'Hispanic/Latin/Mexican'
        WHEN d.`Vict Descent` = 'I' THEN 'American Indian/Alaskan Native'
        WHEN d.`Vict Descent` = 'J' THEN 'Japanese'
        WHEN d.`Vict Descent` = 'K' THEN 'Korean'
        WHEN d.`Vict Descent` = 'L' THEN 'Laotian'
        WHEN d.`Vict Descent` = 'O' THEN 'Other'
        WHEN d.`Vict Descent` = 'P' THEN 'Pacific Islander'
        WHEN d.`Vict Descent` = 'S' THEN 'Samoan'
        WHEN d.`Vict Descent` = 'U' THEN 'Hawaiian'
        WHEN d.`Vict Descent` = 'V' THEN 'Vietnamese'
        WHEN d.`Vict Descent` = 'Z' THEN 'Asian Indian'
        WHEN d.`Vict Descent` = 'X' THEN 'Unknown'
        ELSE d.`Vict Descent`
    END AS Race,
    COUNT(*) AS victim_count
FROM data as d
INNER JOIN revgecoding as r
ON d.LAT=r.LAT AND d.LON=r.LON
WHERE d.`Vict Descent` IS NOT NULL
AND YEAR(d.`DATE OCC`) = 2015
AND r.ZIPCode IN (
        (SELECT `Zip Code`
        FROM LA_income_2015
        WHERE `Zip Code` IN (SELECT DISTINCT ZIPCode FROM revgecoding)
        ORDER BY CAST(regexp_replace(`Estimated Median Income`, '[^0-9.]', '') AS DECIMAL(18,2)) DESC
        LIMIT 3)
    UNION ALL
        (SELECT `Zip Code`
        FROM LA_income_2015
        WHERE `Zip Code` IN (SELECT DISTINCT ZIPCode FROM revgecoding)
        ORDER BY CAST(regexp_replace(`Estimated Median Income`, '[^0-9.]', '') AS DECIMAL(18,2)) ASC
        LIMIT 3)
    )
GROUP BY d.`Vict Descent`
ORDER BY victim_count DESC
"""
```

```
result = spark.sql(query)

end_time = time.time()

result.show(result.count(), truncate=False)
result.write.csv("/user/user/csv_files/Query3_SQL", header=True,
        mode="overwrite")

print('Total time for SQL: ', end_time - start_time, 'sec')
```

Εδώ χρησιμοποιούμε inner join για να συνενώσουμε το σύνολο δεδομένων με τα εγκλήματα με το σύνολο δεδομένων με τους κωδικούς ZIP. Η ένωση γίνεται με βάση τις στήλες των συντεταγμένων. Επίσης φιλτράρουμε τα δεδομενα ώστε τα εγκλήματα να διαπράχτηκαν το 2015, να μην υπάρχουν σειρές όπου η καταγωγή των θυμάτων είναι κενή και ο κωδικός ZIP να είναι ένας εκ των 3 κωδικών περιοχών με το υψηλότερο εισόδημα ή των 3 κωδικών περιοχών με το χαμηλότερο εισόδημα. Τους 6 αυτούς κωδικούς τους υπολογίζουμε από το

σύνολο δεδομένων με τα εισοδήματα για το 2015 που περιέχει και τους ZIP κωδικούς των περιοχών αυτών. Σύμφωνα με απάντηση στο φόρουμ απορίων για την εργασία, το σύνολο δεδομένων revgecoding κατασκευάστηκε περνώντας ένα-ένα τα ζεύγη (lat, long) σε έναν reverse geo-coder - επομένως όποιο ZIP Code δε βρίσκεται στο συγκεκριμένο αρχείο δεν αντιστοιχεί σε κάποιο ζεύγος συντεταγμένων του βασικού συνόλου δεδομένων με τα εγκλήματα. Έτσι, καθώς το σύνολο δεδομένων με τα εισοδήματα (LA_income_2015) περιέχει και περιοχές για τις οποίες δεν έχουμε δεδομένα εγκλημάτων, για να βρούμε τους 6 κωδικούς που θέλουμε βάζουμε ως προϋπόθεση να υπάρχουν στο revgecoding. Στην συνέχεια, ταξινομούμε από το μεγαλύτερο στο μικρότερο εισόδημα και βάζουμε όριο 3 για να βρούμε τους 3 κωδικούς περιοχών με τα μεγαλύτερα εισοδήματα για το 2015 και ομοίως από το μικρότερο στο μεγαλύτερο με όριο 3 για τους 3 κωδικούς με τα μικρότερα εισοδήματα το 2015. Από το αποτέλεσμα επιλέγουμε την στήλη με την καταγωγή των θυμάτων, εκτελώντας παράλληλα μία αντιστοίχηση των κωδικών γραμμάτων σε λέξεις. Επίσης ομαδοποιούμε με βάση την καταγωγή των θυμάτων, υπολογίζοντας και το άθροισμα των εγκλημάτων που διαπράχθηκαν εις βάρος κάθε καταγωγής. Το τελικό αποτέλεσμα ταξινομείται με βάση το άθροισμα αυτό, από το μεγαλύτερο στο μικρότερο.

Το αποτέλεσμα του **Query 3**:

Race	victim_count
Hispanic/Latin/Mexican	1032
Black	983
White	652
Other	323
Other Asian	89
Unknown	62
Korean	8
Japanese	3
American Indian/Alaskan Native	3
Chinese	2
Filipino	1

Σχόλιο για την ερμηνεία του Tip «Οι περιοχές που καλύπτονται στο Median Household Income by Zip Code σύνολο δεδομένων αφορούν την ευρύτερη περιοχή της Κομητείας του Los Angeles»:

Στο συγκεκριμένο ερώτημα το παραπάνω tip θα μπορούσε να ερμηνευθεί ως «Μας ενδιαφέρουν μόνο Zip Codes εντός της πόλης του Los Angeles και όχι της ευρύτερης περιοχής της Κομητείας. Αυτές οι περιοχές είναι όσες ξεκινάνε με Los Angeles στο πεδίο Community.». Μετά από ανάγνωση της απάντησης στο συγκεκριμένο ερώτημα https://helios.ntua.gr/mod/forum/discuss.php?d=29333 θεωρήσαμε ότι αυτό που εννοείται είναι ότι μας ενδιαφέρουν τα Zip Codes για τα οποία έχουμε δεδομένα στο LAPD dataset και όχι μόνο αυτά των οποίων τα Communities ξεκινάνε με «Los Angeles». Στην δεύτερη περίπτωση θα έπρεπε να προστεθεί και ένα «Community like “Los Angeles%”» στο SQL query.

Χρόνοι εκτέλεσης με το SQL API (μ.ό. 10 προσπαθειών):

Πλήθος Executors	Χρόνος Εκτέλεσης (sec)
2	1.05
3	1.00
4	0.51

Σχολιασμός:
Προφανώς, η αύξηση των executors (κυρίως από 3 σε 4) οδηγεί σε υψηλότερη απόδοση του συστήματός μας για το συγκεκριμένο query. Αυτό συμβαίνει γιατί με την αύξηση των executors, αυξάνεται η παραλληλοποίηση των διεργασιών, αξιοποιούνται καλύτερα οι πόροι του συστήματος (κάθε executor έχει έναν πυρήνα και 1GB μνήμης με βάσης το configuration file οπότε υπήρχαν κι άλλοι διαθέσιμοι στο σύστημα) και γίνεται καλύτερη κατανομή των tasks. Βέβαια, δεν είναι ο κανόνας ότι αύξηση των executors θα φέρει και αύξηση απόδοσης, λόγω των trade-offs που συμβαίνουν σε περιπτώσεις αυξημένης παραλληλοποίησης, όπως π.χ. το overhead κάθε executor (JVM startup και allocation πόρων) ή το scheduling overhead.

6. Ο κώδικάς μας για το Query 4 σε DataFrame API:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, year, row_number, month, dayofmonth, count, dense_rank, desc, avg, asc, min, m
from pyspark.sql.window import Window
import time, datetime
import geopy.distance
from pyspark.sql.functions import udf
from pyspark.sql.types import FloatType
from io import StringIO
from contextlib import redirect_stdout

@udf(FloatType())
def get_distance(lat1 , long1 , lat2 , long2):
    return geopy.distance.geodesic((lat1 , long1), (lat2 , long2)).km

spark = SparkSession \
    .builder \
    .appName("DF query 4 execution") \
    .config("spark.executor.instances", 4) \
    .getOrCreate()

data_df = spark.read.csv("hdfs://okeanos-master:54310/user/user/csv_files/data.csv", header=True, inferSchema=True)
LAPD_Police_Stations_df = spark.read.csv("hdfs://okeanos-master:54310/user/user/csv_files/LAPD_Police_Stations.csv",
                                         header=True, inferSchema=True)

start_time = time.time()

filtered_data = data_df.filter((col("LAT") != 0.0) & (col("LON") != 0.0) & col("Weapon Used Cd").isNotNull() &
                               (col("Weapon Used Cd")<=199.9) & (col("Weapon Used Cd")>=100.0))

```

```
# 4.1a
data_a = filtered_data.withColumn("year", year(col("DATE OCC")))
data_1a = data_a.select(["year", "LAT", "LON", "AREA"])
lapd_a = LAPD_Police_Stations_df.select(["X", "Y", "PREC"])
joined1 = data_1a.join(lapd_a, data_1a["AREA"] == lapd_a["PREC"], "inner")
result1 = joined1.withColumn("dist", get_distance(col("LAT"), col("LON"), col("Y"), col("X")))\
    .groupBy("year").agg(avg("dist").alias("average_distance"), count("*").alias("crime_count"))\
    .orderBy(asc("year"))

# 4.1b
data_b = filtered_data.select(["LAT", "LON", "AREA"]).withColumnRenamed("AREA", "PREC")
lapd_b = LAPD_Police_Stations_df.select(["X", "Y", "PREC", "DIVISION"])
joined2 = data_b.join(lapd_b, on="PREC", how="inner").withColumnRenamed("DIVISION", "division")
result2 = joined2.withColumn("dist", get_distance(col("LAT"), col("LON"), col("Y"), col("X")))\
    .groupBy("division").agg(avg("dist").alias("average_distance"), count("*").alias("crime_count"))\
    .orderBy(desc("crime_count"))

#4.2a
data_2a = data_a.select(["year", "LAT", "LON"]).withColumn("ID", monotonically_increasing_id())
joined3 = data_2a.crossJoin(lapd_a)
result3 = joined3.withColumn("dist", get_distance(col("LAT"), col("LON"), col("Y"), col("X")))\
    .groupBy("ID", "year").agg(min("dist").alias("min_distance"))\
    .groupBy("year").agg(avg("min_distance").alias("average_min_distance"), count("*").alias("crime_count"))\
    .orderBy(asc("year"))

```

```
#4.2b
data_2b = data_b.select(["LAT", "LON"]).withColumn("ID", monotonically_increasing_id())
joined4 = data_2b.crossJoin(lapd_b).withColumn("dist", get_distance(col("LAT"), col("LON"), col("Y"), col("X")))
window_spec = Window.partitionBy("ID").orderBy("dist")
ranked_df = joined4.withColumn("rank", row_number().over(window_spec))
closest_div = ranked_df.filter(col("rank") == 1).drop("rank")
result4 = closest_div.groupBy("DIVISION").agg(avg("dist").alias("average_min_distance"), count("*").alias("crime_count"))\
    .orderBy(desc("crime_count"))

end_time = time.time()

result1.show(result1.count(), truncate=False)
result2.show(result2.count(), truncate=False)
result3.show(result3.count(), truncate=False)
result4.show(result4.count(), truncate=False)

result1.write.csv("/user/user/csv_files/Query4-1a_DF", header=True, mode="overwrite")
result2.write.csv("/user/user/csv_files/Query4-1b_DF", header=True, mode="overwrite")
result3.write.csv("/user/user/csv_files/Query4-2a_DF", header=True, mode="overwrite")
result4.write.csv("/user/user/csv_files/Query4-2b_DF", header=True, mode="overwrite")

print('Total time for DF: ', end_time - start_time, 'sec')
```

Αρχικά κρατάμε από το αρχικό σύνολο δεδομένων μόνο τα εγκλήματα που δεν αναφέρονται στο Null Island, έχουν καταγεγραμμένο ένα όπλο που χρησιμοποιήθηκε και το όπλο αυτό είναι πυροβόλο οποιασδήποτε μορφής.

Για το **Query 4-1a** προσθέτουμε στα φιλτραρισμένα δεδομένα μας μία επιπλέον στήλη που δίνει το έτος στο οποίο διαπράχτηκε το κάθε έγκλημα. Έπειτα ξεχωρίζουμε τις στήλες με τις συντεταγμένες, την νέα στήλη με το έτος και την στήλη που διευκρινίζει το αστυνομικό τμήμα που ανέλαβε την υπόθεση με έναν αριθμό. Από το σύνολο δεδομένων για τα αστυνομικά τμήματα του Los Angeles επιλέγουμε τις στήλες με τις συντεταγμένες τους και την αντίστοιχη στήλη με τον αριθμό που αντιστοιχεί στο συγκεκριμένο αστυνομικό τμήμα. Χρησιμοποιούμε inner join για να ενώσουμε αυτά τα δύο σύνολα δεδομένων με βάση την στήλη που δίνει το αριθμό του αστυνομικού τμήματος. Το αποτέλεσμα δηλαδή περιέχει τον χρόνο που διαπράχτηκε το έγκλημα, τις συντεταγμένες του εγκλήματος, τον αριθμό του αστυνομικού τμήματος και τις συντεταγμένες του αστυνομικού τμήματος. Έτσι δημιουργούμε μία επιπλέον στήλη που υπολογίζει την απόσταση του εγκλήματος από το αστυνομικό τμήμα με βάση τις συντεταγμένες τους. Έπειτα ομαδοποιούμε με βάση τον

χρόνο υπολογίζοντας δηλαδή για κάθε έτος την μέση τιμή της στήλης με την απόσταση και μετρώντας τον αριθμό εγκλημάτων που συνέβησαν σε κάθε έτος. Τέλος, ταξινομούμε με βάση το έτος, από το μικρότερο στο μεγαλύτερο.

Για το **Query 4-1b** αρχικά ξεχωρίζουμε τις στήλες με τις συντεταγμένες και την στήλη που διευκρινίζει το αστυνομικό τμήμα που ανέλαβε την υπόθεση με έναν αριθμό. Έπειτα μετονομάζουμε την στήλη με τον αριθμό του αστυνομικού τμήματος στα φιλτραρισμένα δεδομένα μας ώστε να έχει το ίδιο όνομα με την αντίστοιχη στα δεδομένα των αστυνομικών τμημάτων του LA. Από το σύνολο δεδομένων για τα αστυνομικά τμήματα του Los Angeles επιλέγουμε τις στήλες με τις συντεταγμένες τους, την αντίστοιχη στήλη με τον αριθμό που αντιστοιχεί στο συγκεκριμένο αστυνομικό τμήμα και το όνομα του αστυνομικού τμήματος. Χρησιμοποιούμε inner join για να ενώσουμε αυτά τα δύο σύνολα δεδομένων με βάση την στήλη που δίνει το αριθμό του αστυνομικού τμήματος. Το αποτέλεσμα δηλαδή περιέχει τις συντεταγμένες του εγκλήματος, τον αριθμό του αστυνομικού τμήματος, το όνομα του αστυνομικού τμήματος και τις συντεταγμένες του αστυνομικού τμήματος. Έτσι δημιουργούμε μία επιπλέον στήλη που υπολογίζει την απόσταση του εγκλήματος από το αστυνομικό τμήμα με βάση τις συντεταγμένες τους. Έπειτα ομαδοποιούμε με βάση το όνομα του αστυνομικού τμήματος, υπολογίζοντας για κάθε αστυνομικό τμήμα την μέση τιμή της στήλης με την απόσταση και μετρώντας τον αριθμό εγκλημάτων που ανέλαβε το κάθε ένα. Τέλος, ταξινομούμε με βάση τον αριθμό εγκλημάτων, από τον μεγαλύτερο στον μικρότερο.

Για το **Query 4-2a** χρησιμοποιούμε το DataFrame που δημιουργήσαμε στο Query 4-1a με την στήλη με το έτος. Από αυτή επιλέγουμε τις στήλες με τις συντεταγμένες, την στήλη με το έτος και προσθέτουμε μία στήλη που δίνει έναν μοναδικό αναγνωριστικό αριθμό ID για κάθε row, ώστε να μπορούμε να ξεχωρίσουμε τα διαφορετικά εγκλήματα τώρα που δεν έχουμε όλες τις στήλες. Χρησιμοποιούμε επίσης το DataFrame με τις λιγότερες στήλες (year, LAT, LON, AREA) που κατασκευάσαμε στο Query 4-1a από το σύνολο δεδομένων για τα αστυνομικά τμήματα του Los Angeles. Με cross join μεταξύ αυτών των δύο DataFrames δημιουργούμε ουσιαστικά το καρτεσιανό τους γινόμενο, ώστε για κάθε έγκλημα να έχουμε 21 σειρές που διαφέρουν καθώς έχουν συνενωθεί με τις πληροφορίες για διαφορετικό αστυνομικό τμήμα. Έτσι δημιουργούμε μία επιπλέον στήλη που υπολογίζει την απόσταση του εγκλήματος από κάθε αστυνομικό τμήμα με βάση τις συντεταγμένες τους. Έπειτα ομαδοποιούμε με βάση τον μοναδικό αναγνωριστικό ID, υπολογίζοντας δηλαδή για κάθε έγκλημα την ελάχιστη τιμή της στήλης με την απόσταση. Γίνεται παράλληλα ομαδοποίηση και με βάση τον χρόνο, που δεν έχει καμία επιρροή καθώς όλες οι σειρές με ίδιο ID έχουν και ίδια τιμή στην στήλη year. Ωστόσο, είναι σκόπιμο να γίνει ώστε να διατηρηθεί και η στήλη year, αφού στην συνέχεια ομαδοποιούμε το αποτέλεσμα με βάση μόνο εκείνη, υπολογίζοντας την μέση τιμή της ελάχιστης απόστασης που βρήκαμε για κάθε έτος, καθώς και τον αριθμό των εγκλημάτων για κάθε έτος. Το τελικό αποτέλεσμα ταξινομείται με βάση το έτος, από το μικρότερο στο μεγαλύτερο.

Για το **Query 4-2b** επιλέγουμε από το φιλτραρισμένο σύνολο δεδομένων μόνο τις στήλες με τις συντεταγμένες του εγκλήματος και προσθέτουμε μία στήλη που δίνει έναν μοναδικό αναγνωριστικό αριθμό ID για κάθε row, ώστε να μπορούμε να ξεχωρίσουμε τα διαφορετικά

εγκλήματα τώρα που δεν έχουμε όλες τις στήλες. Εκτελούμε cross join αυτού του DataFrame με το DataFrame για τα αστυνομικά τμήματα του Los Angeles με τις επιλεγμένες στήλες από το Query 4-1b. Έτσι για κάθε έγκλημα έχουμε 21 σειρές όπου η καθεμία το αντιστοιχίζει σε διαφορετικό αστυνομικό τμήμα. Στην συνέχεια δημιουργούμε μία επιπλέον στήλη που υπολογίζει την απόσταση του εγκλήματος από κάθε αστυνομικό τμήμα με βάση τις συντεταγμένες τους. Έπειτα αριθμούμε για κάθε έγκλημα τις 21 αυτές σειρές με βάση την τιμή της στήλης με την απόσταση. Συγκεκριμένα, τις ταξινομούμε από την σειρά με την μικρότερη τιμή στην σειρά με την μεγαλύτερη και τις αριθμούμε ξεκινώντας από το 1. Από αυτές, κρατάμε μόνος τις σειρές με αριθμό 1, δηλαδή κρατάμε για κάθε έγκλημα την σειρά με την μικρότερη απόσταση. Ομαδοποιούμε το αποτέλεσμα με βάση το όνομα του αστυνομικού τμήματος, υπολογίζοντας για κάθε αστυνομικό τμήμα την μέση τιμή της στήλης με την απόσταση και μετρώντας τον αριθμό εγκλημάτων που αντιστοιχεί στο κάθε ένα. Τέλος, ταξινομούμε με βάση τον αριθμό εγκλημάτων, από τον μεγαλύτερο στον μικρότερο.

Τα αποτελέσματα του **Query 4**:
Query 4-1a:

year	average_distance	crime_count
2010	2.713129694813762	6177
2011	2.760297296122407	7015
2012	2.8357548501766847	6532
2013	2.685644123880696	2981
2014	2.731252108440917	3411
2015	2.661203488790034	4245
2016	2.703749775898177	5120
2017	2.72395610209098	7786
2018	2.680475441109129	5687
2019	2.7394299042987393	7129
2020	2.411321190939274	2492
2021	2.63975204844821	9745
2022	2.4674702548078384	7035
2023	2.555912857713699	9202

Query 4-1b:

division	average_distance	crime_count
77TH STREET	2.661302632669427	10123
SOUTHEAST	2.1016176899231316	8921
SOUTHWEST	2.7007557001501876	8623
NEWTON	2.00444284446561	6972
HOLLENBECK	2.6489972790526073	6100
HARBOR	4.079479650843425	5428
RAMPART	1.577304070976699	4990
CENTRAL	1.1360874348576455	3469
HOLLYWOOD	1.4519702485332784	3268
MISSION	4.696967346132304	3084
OLYMPIC	1.8354079508081123	3047
NORTHEAST	3.895878465626484	2730
WILSHIRE	2.3263709350913637	2617
FOOTHILL	3.7486430046488395	2538
NORTH HOLLYWOOD	2.6968818865271325	2427
PACIFIC	3.7483714068372005	2023
WEST VALLEY	3.452494794739544	1991
VAN NUYS	2.2147655784626914	1957
TOPANGA	3.41008351486434	1590
DEVONSHIRE	3.962621290310503	1565
WEST LOS ANGELES	4.217406558264489	1094

Query 4-2a:

year	average_min_distance	crime_count
2010	2.368641061685112	6177
2011	2.439865634941409	7015
2012	2.5055255745226757	6532
2013	2.43824231121108	2981
2014	2.429036603057203	3411
2015	2.4358989892371694	4245
2016	2.473738255165517	5120
2017	2.391618931052631	7786
2018	2.3918210680147656	5687
2019	2.4294088128732088	7129
2020	2.2020454725017395	2492
2021	2.3527163780992493	9745
2022	2.169733275069611	7035
2023	2.273820758078039	9202

Query 4-2b:

```
+----------------+--------------------+-----------+
|DIVISION        |average_min_distance|crime_count|
+----------------+--------------------+-----------+
|SOUTHWEST       |2.1655091872159895  |8729       |
|77TH STREET     |1.7077101040938787  |8669       |
|SOUTHEAST       |2.217804564847498   |8151       |
|HOLLENBECK      |2.6302994180129065  |6139       |
|WILSHIRE        |2.473077329463466   |5406       |
|HARBOR          |3.891790823937586   |5305       |
|NEWTON          |1.5683781813559128  |5196       |
|HOLLYWOOD       |1.9488415514896167  |4677       |
|RAMPART         |1.3652501212211796  |4598       |
|OLYMPIC         |1.6888869547340741  |4288       |
|CENTRAL         |0.9972380407551311  |3399       |
|VAN NUYS        |2.933667868285249   |3292       |
|FOOTHILL        |3.5483540951015113  |3079       |
|NORTH HOLLYWOOD |2.707046522883924   |2421       |
|NORTHEAST       |3.755100033299768   |2199       |
|WEST VALLEY     |2.772503767332064   |1978       |
|MISSION         |3.8062795031532737  |1948       |
|PACIFIC         |3.7129527499877946  |1922       |
|TOPANGA         |2.989080034859173   |1585       |
|DEVONSHIRE      |3.0087140132779813  |829        |
|WEST LOS ANGELES|2.726491570303159   |747        |
+----------------+--------------------+-----------+
```

7. Για τα joins των Query 3 και Query 4, χρησιμοποιούμε την μέθοδο hint ώστε να εκτελεστούν με διαφορετικό τρόπο (BROADCAST, MERGE, SHUFFLE_HASH, SHUFFLE_REPLICATE_NL) και την μέθοδο explain ώστε να πάρουμε το πλάνο εκτέλεσης μέσω κειμένου. Χρησιμοποιούμε επίσης το Spark History UI για να πάρουμε το πλάνο εκτέλεσης και γραφικά.

Συγκεκριμένα, στον κώδικα του Query 3 προσθέτουμε τις παρακάτω γραμμές, ακριβώς μετά τον ορισμό του query:

```
query_broadcast = query.replace("SELECT CASE", "SELECT /*+ BROADCAST(r) */ CASE")
query_merge = query.replace("SELECT CASE", "SELECT /*+ MERGE(r) */ CASE")
query_shuffle_hash = query.replace("SELECT CASE", "SELECT /*+ SHUFFLE_HASH(r) */ CASE")
query_shuffle_replicate_nl = query.replace("SELECT CASE", "SELECT /*+ SHUFFLE_REPLICATE_NL(r) */ CASE")
```

Εκτελούμε με spark.sql() κάθε ένα από αυτά τα τροποποιημένα queries και με τον παρακάτω κώδικα αποθηκεύουμε τοπικά το αποτέλεσμα του explain:

```
with StringIO() as buffer:
    with redirect_stdout(buffer):
        result.explain()
    physical_plan_str = buffer.getvalue()

with open("./explain/query3/explain_<query_type>.txt", "w") as explain_file:
    explain_file.write(physical_plan_str)
```

Για το Query 4 προσθέτουμε τις εξής γραμμές μετά από κάθε join:

```
#joined1 = data_1a.join(lapd_a.hint("BROADCAST"), data_1a["AREA"] == lapd_a["PREC"], "inner")
#joined1 = data_1a.join(lapd_a.hint("MERGE"), data_1a["AREA"] == lapd_a["PREC"], "inner")
#joined1 = data_1a.join(lapd_a.hint("SHUFFLE_HASH"), data_1a["AREA"] == lapd_a["PREC"], "inner")
#joined1 = data_1a.join(lapd_a.hint("SHUFFLE_REPLICATE_NL"), data_1a["AREA"] == lapd_a["PREC"], "inner")
```

```
#joined2 = data_b.join(lapd_b.hint("BROADCAST"), on="PREC", how="inner").withColumnRenamed("DIVISION", "division")
#joined2 = data_b.join(lapd_b.hint("MERGE"), on="PREC", how="inner").withColumnRenamed("DIVISION", "division")
#joined2 = data_b.join(lapd_b.hint("SHUFFLE_HASH"), on="PREC", how="inner").withColumnRenamed("DIVISION", "division")
#joined2 = data_b.join(lapd_b.hint("SHUFFLE_REPLICATE_NL"), on="PREC", how="inner").withColumnRenamed("DIVISION", "division")
```

```
#joined3 = data_2a.crossJoin(lapd_a.hint("BROADCAST"))
#joined3 = data_2a.crossJoin(lapd_a.hint("MERGE"))
#joined3 = data_2a.crossJoin(lapd_a.hint("SHUFFLE_HASH"))
#joined3 = data_2a.crossJoin(lapd_a.hint("SHUFFLE_REPLICATE_NL"))
```

```
#joined4 = data_2b.crossJoin(lapd_b.hint("BROADCAST")).withColumn("dist",
#           get_distance(col("LAT"), col("LON"), col("Y"), col("X")))
#joined4 = data_2b.crossJoin(lapd_b.hint("MERGE")).withColumn("dist",
#           get_distance(col("LAT"), col("LON"), col("Y"), col("X")))
#joined4 = data_2b.crossJoin(lapd_b.hint("SHUFFLE_HASH")).withColumn("dist",
#           get_distance(col("LAT"), col("LON"), col("Y"), col("X")))
#joined4 = data_2b.crossJoin(lapd_b.hint("SHUFFLE_REPLICATE_NL")).withColumn("dist",
#           get_distance(col("LAT"), col("LON"), col("Y"), col("X")))
```

Κάθε φορά που θέλουμε να τρέξουμε με ένα συγκεκριμένο hint όλο το query, το βγάζουμε από τα σχόλια για κάθε subquery και βάζουμε όλα τα υπόλοιπα σε σχόλια. Στην συνέχεια χρησιμοποιούμε τον παρακάτω κώδικα για να αποθηκεύσουμε το αποτέλεσμα του explain σε κατάλληλο φάκελο, ανάλογα με τον τύπο του hint:

```
with StringIO() as buffer:
    with redirect_stdout(buffer):
        result1.explain()
        result2.explain()
        result3.explain()
        result4.explain()
    physical_plan_str = buffer.getvalue()

with open("./explain/query4/explain_<query_type>.txt", "w") as explain_file:
    explain_file.write(physical_plan_str)
```

Τα πλάνα εκτέλεσης που παράγονται είναι πολύ παρόμοια, με κάποιες γραμμές μόνο διαφορετικές. Αυτές είναι οι γραμμές που περιγράφουν τις μεθόδους join. Για αυτές γνωρίζουμε τα εξής:

BROADCAST:
Χρησιμοποιείται για το broadcasting μικρότερων tables σε όλους τους worker nodes. Μπορεί να είναι αποτελεσματικό όταν το broadcasted table είναι μικρό και χωράει στη μνήμη των worker nodes, καθώς και όταν θέλουμε να εκτελέσουμε μόνο ένα equi-join για να συνδυάσουμε δύο σύνολα δεδομένων με βάση τα ίδια μη ταξινομημένα κλειδιά. Η default τιμή του spark.sql.autoBroadcastJoinThreshold είναι 10MB.

MERGE:
Το merge join είναι γενικά αποτελεσματικό όταν και τα δύο tables που συζεύγνονται είναι μεγάλα, διασκορπισμένα σε πολλούς κόμβους και ταξινομημένα με βάση το κλειδί του join (ή η ταξινόμηση είναι φτηνή). Χρησιμοποιείται όταν τα δεδομένα είναι μεγάλα, αλλά όχι τόσο μεγάλα ώστε να απαιτείται shuffle_join.

SHUFFLE_JOIN:

Χρησιμοποιείται γενικά για το join δύο μεγάλων tables, όταν αυτά είναι πολύ μεγάλα και δεν χωράνε στην μνήμη ενός κόμβου. Περιλαμβάνει την ανακατανομή δεδομένων σε όλο το cluster με βάση την τιμή κατακερματισμού του join key. Μπορεί να είναι αποτελεσματική όταν η κατανομή των δεδομένων δεν είναι καλή και θέλουμε να εξισορροπήσουμε το φορτίο στους κόμβους. Μπορεί επίσης να είναι μια κατάλληλη επιλογή εάν ούτε το broadcasting ούτε το merge είναι ιδανικά για μία συγκεκριμένη περίπτωσή.

SHUFFLE_REPLICATE_NL (Cartesian Product Join (CPJ)):
Το Cartesian Product Join (CPJ) είναι μια μέθοδος που χρησιμοποιείται στο Spark για την ένωση δύο συνόλων δεδομένων δημιουργώντας όλους τους δυνατούς συνδυασμούς των εγγραφών και στα δύο σύνολα δεδομένων. Στην CPJ, κάθε εγγραφή στο ένα σύνολο δεδομένων αντιστοιχίζεται με κάθε εγγραφή στο άλλο σύνολο δεδομένων, με αποτέλεσμα ένα καρτεσιανό γινόμενο των δύο συνόλων δεδομένων. Με αυτήν την στρατηγική το μικρότερο table αναπαράγεται σε όλους τους worker κόμβους πριν από το join. Είναι χρήσιμη όταν το μικρότερο table είναι αρκετά μικρό σε σχέση με το μεγαλύτερο και θέλουμε να αποφύγουμε την ανακατανομή μεγάλου όγκου δεδομένων. Για μεγάλα tables, είναι ιδιαίτερα “κοστοβόρα” διαδικασία και γενικά συνίσταται να χρησιμοποιείται μόνο όταν δεν μπορούν να εφαρμοστούν οι υπόλοιπες.

Για το Query 3:
Εδώ κάνουμε join το table με τα δεδομένα όλων των εγκλημάτων που έχει 2.335.238 σειρές με το table με τα zip codes που έχει 37.781 σειρές. Το δεύτερο δηλαδή table είναι πολύ μικρότερο από το πρώτο και έχει συγκεκριμένα μέγεθος 877 KB. Η καλύτερη έτσι στρατηγική φαίνεται να είναι η BROADCAST και μάλιστα είναι αυτή που χρησιμοποιείται αυτόματα όταν δεν χρησιμοποιούμε την μέθοδο hint.

Για το Query 4:
Για κάθε ένα από τα 4 αποτελέσματα που παράγονται γίνεται ένα join. Όλα αυτά τα joins είναι πάντα μεταξύ του LAPD_Police_Stations και των αρχικών μας δεδομένων για τα εγκλήματα, φιλτραρισμένα ώστε να παραμένουν μόνο όσα αφορούν χρήση πυροβόλων όπλων. Το πρώτο table έχει μόλις 23 σειρές, ενώ το δεύτερο έχει 84.557 σειρές. Για τα δύο πρώτα subqueries, όπου θέλουμε να βρούμε την μέση απόσταση των εγκλημάτων από το αστυνομικό τμήμα που τα χειριστηκε, έχουμε χρησιμοποιήσει inner joins. Για αυτά καταλληλότερη μέθοδος είναι η BROADCAST, αφού επρόκειτο για equi-joins και το ένα table είναι σημαντικά μικρότερο από το άλλο. Για τα δύο επόμενα subqueries, όπου ψάχνουμε την μέση απόσταση των εγκλημάτων από το κοντινότερο σε αυτά αστυνομικό τμήμα, έχουμε χρησιμοποιήσει cross-join, δεν έχουμε δηλαδή equi-join. Επομένως, θα λέγαμε ότι η καλύτερη στρατηγική θα ήταν η SHUFFLE_REPLICATE_NL. Χωρίς να δώσουμε hints, βλέπουμε ότι για τα πρώτα δύο subqueries χρησιμοποιείται όντως BROADCAST μέθοδος, ενώ για τα άλλα δύο χρησιμοποιείται η μέθοδος Broadcast Nested Loop Join. Η τελευταία είναι μία μέθοδος παρόμοια με την BROADCAST αλλά κατάλληλη για πιο περίπλοκα join conditions ή για όταν έχουμε non-equality checks, όπως στο cross join, όπου δεν δίνονται συγκεκριμένα conditions.