



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

**ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ**

ΟΜΑΔΑ 12

ΕΛΕΥΘΕΡΙΑ ΑΡΚΑΔΟΠΟΥΛΟΥ 03119442

ΔΕΣΠΟΙΝΑ ΒΙΔΑΛΗ 03119111

Προχωρημένα Θέματα Βάσεων Δεδομένων

Αναφορά Εξαμηνιαίας Εργασίας 2023-2024

Repo Link:

<https://github.com/despoinavdl/AdvancedDatabasesNTUA23/tree/main>

Ζητούμενο 1

Εγκαθιστούμε και διαμορφώνουμε κατάλληλα την πλατφόρμα Apache Spark ώστε να εκτελείται πάνω από τον διαχειριστή πόρων του Apache Hadoop YARN και μορφοποιούμε κατάλληλα το Apache Hadoop Distributed File System, σύμφωνα με τις οδηγίες. Το κατανεμημένο περιβάλλον εργασίας περιλαμβάνει 2 κόμβους (master-worker).

Στον master node εκκινούμε τις web εφαρμογές HDFS, Yarn και τον Spark History Server με χρήση των εντολών:

```
start-dfs.sh  
start-yarn.sh  
$SPARK_HOME/sbin/start-history-server.sh
```

Για το HDFS, επιβεβαιώνουμε ότι η διαδικασία έχει πετύχει προσπελαύνοντας με έναν browser την <http://83.212.81.189:9870> και επιβεβαιώνοντας ότι έχουμε 2 διαθέσιμους live nodes. Για το YARN, ελέγχουμε προσπελαύνοντας την

<http://83.212.81.189:8088/cluster> ότι έχουμε 2 διαθέσιμους live nodes. Τα σχετικά στιγμιότυπα:

Configured Capacity:	58.78 GB
Configured Remote Capacity:	0 B
DFS Used:	2.04 GB (3.48%)
Non DFS Used:	12.82 GB
DFS Remaining:	40.88 GB (69.56%)
Block Pool Used:	2.04 GB (3.48%)
DataNodes usages% (Min/Median/Max/stdDev):	2.17% / 4.78% / 4.78% / 1.31%
Live Nodes	2 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Used
0	0	0	0	0	<memory:0 B, vC

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes
2	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation
Capacity Scheduler	[memory-mb (unit=Mi), vcores]	<memory:128, vCores:1>

Ζητούμενο 2

Δημιουργούμε ένα directory /data στο hdfs, και κατεβάζουμε τα datasets της εκφώνησης στον master με χρήση της εντολής *wget*.

Ανεβάζουμε τα datasets στο hdfs με χρήση της εντολής:

```
hdfs dfs -put /path/to/local/directory hdfs://oceanos-master:54310/data
```

Ο κώδικας για τη δημιουργία του dataframe βρίσκεται στο αρχείο [df.py](#).

Τα αποτελέσματα:

Number of rows: 2973193

Column	Type
DR_NO	string
Date Rptd	date
DATE OCC	date
TIME OCC	string
AREA	string
AREA NAME	string
Rpt Dist No	string
Part 1-2	string
Crm Cd	string
Crm Cd Desc	string
Mocodes	string
Vict Age	int
Vict Sex	string
Vict Descent	string
Premis Cd	string
Premis Desc	string
Weapon Used Cd	string
Weapon Desc	string
Status	string
Status Desc	string
Crm Cd 1	string
Crm Cd 2	string
Crm Cd 3	string
Crm Cd 4	string
LOCATION	string
Cross Street	string
LAT	double
LON	double

Ζητούμενο 3

Για την εκτέλεση με 4 Spark executors, χρησιμοποιούμε το flag `--num-executors 4`.

Η υλοποίηση του Query 1 με SQL API βρίσκεται στο αρχείο [query_1_sql.py](#).

Η υλοποίηση του Query 1 με DataFrame API βρίσκεται στο αρχείο [query_1_df.py](#).

Τα αποτελέσματα:

Total time for SQL: 38.8721
Total time for DF: 39.3836

DataFrame

year	month	crime_total	#
2010	7	15632	1
2010	3	15460	2
2010	6	15335	3
2011	8	17956	1
2011	7	17693	2
2011	10	17508	3
2012	8	22966	1
2012	10	22442	2
2012	5	22338	3
2013	1	9495	1
2013	8	9307	2
2013	7	9160	3
2014	7	11998	1
2014	10	11621	2
2014	8	11621	3
2015	8	13864	1
2015	10	13314	2
2015	9	13289	3
2016	8	14886	1
2016	7	14350	2
2016	10	14152	3
2017	8	33938	1
2017	10	33885	2
2017	9	33174	3
2018	10	15497	1
2018	11	15350	2
2018	8	14897	3
2019	7	19334	1
2019	8	19085	2
2019	3	18951	3
2020	1	7978	1
2020	2	6656	2
2020	3	5819	3
2021	7	34929	1
2021	8	33769	2
2021	10	33601	3
2022	8	17305	1
2022	7	16849	2
2022	5	16328	3
2023	8	38140	1
2023	10	37974	2
2023	7	37814	3

SQL

year	month	crime_total	rank
2010	7	15632	1
2010	3	15460	2
2010	6	15335	3
2011	8	17956	1
2011	7	17693	2
2011	10	17508	3
2012	8	22966	1
2012	10	22442	2
2012	5	22338	3
2013	1	9495	1
2013	8	9307	2
2013	7	9160	3
2014	7	11998	1
2014	10	11621	2
2014	8	11621	2
2015	8	13864	1
2015	10	13314	2
2015	9	13289	3
2016	8	14886	1
2016	7	14350	2
2016	10	14152	3
2017	8	33938	1
2017	10	33885	2
2017	9	33174	3
2018	10	15497	1
2018	11	15350	2
2018	8	14897	3
2019	7	19334	1
2019	8	19085	2
2019	3	18951	3
2020	1	7978	1
2020	2	6656	2
2020	3	5819	3
2021	7	34929	1
2021	8	33769	2
2021	10	33601	3
2022	8	17305	1
2022	7	16849	2
2022	5	16328	3
2023	8	38140	1
2023	10	37974	2
2023	7	37814	3

Εκτελώντας και τις δύο υλοποιήσεις παρατηρούμε ότι οι χρόνοι εκτέλεσης έχουν πολύ μικρή απόκλιση μεταξύ τους. Αυτό ενδέχεται να οφείλεται στο γεγονός ότι το Spark χρησιμοποιεί το Catalyst Optimizer ώστε να βελτιστοποιήσει το logical plan των προγραμμάτων και έτσι οι δύο υλοποιήσεις καταλήγουν να έχουν πολύ παρόμοιο logical plan.

Ζητούμενο 4

Η υλοποίηση του Query 2 με SQL API βρίσκεται στο αρχείο [query_2_sql.py](#).

Η υλοποίηση του Query 2 με RDD API βρίσκεται στο αρχείο [query_2_rdd.py](#).

Τα αποτελέσματα:

Total time for RDD: 55.1153

Total time for SQL: 37.4206

RDD				SQL			
period	count	#		part_of_day	crime_count	rank	
night	240393	1		night	240393	1	
evening	188522	2		evening	188522	2	
afternoon	149713	3		afternoon	149713	3	
morning	125500	4		morning	125500	4	

Συγκρίνοντας τα αποτελέσματα για τους χρόνους εκτέλεσης, παρατηρούμε ότι η υλοποίηση με RDD είναι σημαντικά πιο αργή από την SQL. Αυτό συμβαίνει επειδή το RDD διαθέτει low-level abstraction και δεν βελτιστοποιείται από το Spark, σε αντίθεση με την υλοποίηση μέσω SQL.

Ζητούμενο 5

Για την υλοποίηση του Query 3, ως παραδοχή, παίρνουμε τα δεδομένα για το median household income από το αρχείο LA_income_2015.csv, αφού ζητείται η καταγωγή των θυμάτων για το έτος 2015.

Η υλοποίηση του Query 3 με DataFrame API βρίσκεται στο αρχείο [query_3_df.py](#).

Τα αποτελέσματα της εκτέλεσης για 2, 3, 4 Spark executors αντίστοιχα (παρατίθενται τα αποτελέσματα σε πίνακα μία φορά, αφού είναι ίδια για διαφορετικό αριθμό Spark executors, και στη συνέχεια παρατίθενται μόνο οι χρόνοι εκτέλεσης):

Υψηλότερο Εισόδημα

Vict Descent	#
White	8
Other	3
Black	2
Hispanic/Latin/Me...	2

Χαμηλότερο Εισόδημα

Vict Descent	#
Hispanic/Latin/Me...	1025
Black	986
White	621
Other	314
Other Asian	86
NULL	7
Japanese	3
American Indian/A...	3
Chinese	2
Filipino	1

Total time for 2 Spark executors: 86.6731
 Total time for 3 Spark executors: 79.9424
 Total time for 4 Spark executors: 56.6077

Παρατηρούμε ότι, όσο αυξάνεται ο αριθμός Spark executors που χρησιμοποιούμε, μειώνεται ο χρόνος εκτέλεσης. Οι executors επιτρέπουν την παράλληλη επεξεργασία δεδομένων σε ένα Spark cluster. Περισσότεροι executors εκτελούν πιο γρήγορα διαδικασίες όπως filtering, aggregating, joining και transforming πάνω στα δεδομένα.

Ζητούμενο 6

Η υλοποίηση του Query 4.1 με DataFrame API βρίσκεται στο αρχείο [query_4_1_df.py](#).

Η υλοποίηση του Query 4.2 με DataFrame API βρίσκεται στο αρχείο [query_4_2_df.py](#).

Τα αποτελέσματα της εκτέλεσης:

Query 4.1

(a)			(b)		
year	average_distance	#	DIVISION	average_distance	#
2010	2.743966826763732	6762	77TH STREET	2.6371111285758633	17383
2011	2.7181861910254472	7797	SOUTHEAST	2.1049342137476397	13691
2012	2.90838317905297	8600	NEWTON	2.0273503785367826	9884
2013	2.68513467421713	2981	SOUTHWEST	2.697177074490011	8625
2014	2.730827096395304	3411	HOLLENBECK	2.650709744978364	6100
2015	2.6251270594574954	4557	HARBOR	4.065393973480676	5798
2016	2.6797074676042545	5389	RAMPART	1.5773847552918816	4990
2017	2.7177048271584496	13592	NORTHEAST	3.8627692721174705	4320
2018	2.675418337304041	5776	OLYMPIC	1.8218143052858153	4251
2019	2.738825331613662	7129	HOLLYWOOD	1.4412522937053858	4158
2020	2.411219772922354	2492	MISSION	4.715367423109495	4003
2021	2.6620316456405053	18240	FOOTHILL	3.772710087851956	3536
2022	2.471510160332706	7660	WILSHIRE	2.3971524583774366	3519
2023	2.6666512273361125	17572	NORTH HOLLYWOOD	2.6769083300722123	3507
			CENTRAL	1.1354176934445663	3469
			WEST VALLEY	3.438406078300398	3194
			PACIFIC	3.749110149209904	2677
			VAN NUYS	2.2204182322386727	2670
			DEVONSHIRE	3.979222156574428	2557
			TOPANGA	3.4448701509474904	2072
			WEST LOS ANGELES	4.195647332866541	1554

Query 4.2

(a)

year	average_distance	#
2010	2.3850816628208786	6762
2011	2.4172515869079376	7797
2012	2.5433010739197646	8600
2013	2.4382969832639074	2981
2014	2.4290693589518733	3411
2015	2.4107651169604467	4557
2016	2.461490737788886	5389
2017	2.359936539680825	13592
2018	2.390266123862842	5776
2019	2.4291624775464227	7129
2020	2.2026511955244548	2492
2021	2.343222240539347	18240
2022	2.1680429944575397	7660
2023	2.371109196835763	17572

(b)

DIVISION	average_distance	#
77TH STREET	1.7013668577005783	14449
SOUTHEAST	2.1964630483847487	12543
SOUTHWEST	2.296840137442361	11176
NEWTON	1.574683193297174	7255
WILSHIRE	2.4626724975877416	6263
HOLLENBECK	2.6447647300930015	6169
HOLLYWOOD	1.9694193430036948	6114
HARBOR	3.876940896631801	5666
OLYMPIC	1.6586570751497223	5332
RAMPART	1.4220718575522295	4752
VAN NUYS	2.9561059243596177	4579
FOOTHILL	3.555559924969752	4219
CENTRAL	1.026796653556548	3707
NORTH HOLLYWOOD	2.7087276241861207	3526
NORTHEAST	3.6918118436208154	3438
WEST VALLEY	2.7755357919291526	3098
MISSION	3.79131313053364	2556
PACIFIC	3.719391310781052	2547
TOPANGA	2.9994294543420814	2176
DEVONSHIRE	3.0508628410640033	1350
WEST LOS ANGELES	2.6690709558302768	1043

Ζητούμενο 7

Εκτελούμε τα Query 3, Query 4, με τις μεθόδους `hint&explain` για τους 4 διαφορετικούς τρόπους εκτέλεσης `join`, και παίρνουμε το πλάνο μέσω κειμένου και στη συνέχεια γραφικά μέσω του Spark History UI.

Από το Spark History UI μπορούμε για κάθε διαφορετικό `execution` του κάθε `query` να αντιστοιχίσουμε τα Job IDs στα διαφορετικά κομμάτια του κώδικα που εκτελούνται. Από τα `descriptions` για τα `operations`, κάθε διαφορετικό `operation` `showString` αντιστοιχεί και σε ένα διαφορετικό `dataframe` που σχηματίζεται από `join` και τυπώνεται (διαφορετικό κομμάτι-ροή του DAG diagram). Από τη γραφική αναπαράσταση απομονώνουμε τους χρόνους εκτέλεσης των `showString` για κάθε διαφορετικό τρόπο εκτέλεσης και παραθέτουμε τα αποτελέσματα στον πίνακα. Στα ερωτήματα όπου τυπώνουμε πάνω από ένα `dataframe`, επιλέξαμε αυτό με τον μεγαλύτερο χρόνο εκτέλεσης.

	broadcast	merge	shuffle_hash	shuffle_replicate_nl
Query 3	15 sec	14 sec	11 sec	7.2 min
Query 4.1	7 sec	8 sec	7 sec	8 sec
Query 4.2	10 sec	10 sec	10 sec	10 sec

Παρατηρούμε ότι, στην περίπτωση του Query 3 (inner left join με “=” condition), οι μέθοδοι `broadcast`, `merge`, `shuffle hash` παίρνουν περίπου την ίδια ώρα, με την `shuffle hash` να είναι η γρηγορότερη. Η `shuffle replicate nested loop` παίρνει αισθητά περισσότερη ώρα να εκτελεστεί.

Το αποτέλεσμα αυτό είναι αναμενόμενο, καθώς η μέθοδος `shuffle replicate nested loop` κάνει `replicate` σε κάθε `node` το μικρότερο σε διαστάσεις `dataframe` του `join` και χρησιμοποιεί `nested loop`. Στην περίπτωση του Query 3, το μικρότερο σε διαστάσεις `dataframe` είναι το `revgecoding`, το οποίο είναι αρκετά μεγάλο, και έτσι η διαδικασία καθιστάται χρονοβόρα. Το ότι η `shuffle hash` είναι η πιο αποδοτική μέθοδος πιθανό να αποδίδεται στο ότι το `join` βασίζεται σε “=” condition: τα δεδομένα με ίδιο `join key` να καταλήξουν στα ίδια `nodes`, γεγονός που καθιστά πιο εύκολη την επεξεργασία τους.

Στην περίπτωση των Query 4.1 (inner left join με “=” condition), όλες οι μέθοδοι παίρνουν περίπου τον ίδιο χρόνο. Στην περίπτωση αυτή, η μέθοδος `shuffle replicate nested loop` δεν φαίνεται να παίρνει περισσότερη ώρα για να εκτελεστεί, γεγονός που πιθανώς αποδίδεται στο ότι το μικρότερο σε μέγεθος `dataframe` είναι το `police_stations` (21 γραμμές), που το `replication` του παίρνει σαφώς λιγότερο χρόνο από το `revgecoding`.

Στην περίπτωση των Query 4.2 (cross join), όλες οι μέθοδοι παίρνουν περίπου τον ίδιο χρόνο για να εκτελεστούν.

Python scripts

df.py

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

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

crime_df = spark.read.csv("hdfs://oceanos-master:54310/data/crime_data_2010_2019" \
                           ,header=True)

crime2_df = spark.read.csv("hdfs://oceanos-master:54310/data/crime_data_2020_present" \
                            ,header=True)

df = crime_df.union(crime2_df)

#specified columns
df = df.withColumn("Date Rptd", to_date(df["Date Rptd"], "MM/dd/yyyy hh:mm:ss a")) \
        .withColumn("DATE OCC", to_date(df["DATE OCC"], "MM/dd/yyyy hh:mm:ss a")) \
        .withColumn("Vict Age", df["Vict Age"].cast(IntegerType())) \
        .withColumn("LAT", df["LAT"].cast(DoubleType())) \
        .withColumn("LON", df["LON"].cast(DoubleType()))

row_count = df.count()
print("Number of rows:", row_count)

column_types = df.dtypes
print("Column Types:")
for col_name, col_type in column_types:
    print(f"{col_name}: {col_type}")

#create new csv: total_crime.csv and save it
path = "hdfs://oceanos-master:54310/data/total_crime.csv"
df.write.csv(path, header=True, mode="overwrite")
```

query_1_sql.py

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField, StructType, TimestampType, DoubleType,
StringType, IntegerType
from pyspark.sql.functions import year, month, count, dense_rank, col, to_date
from pyspark.sql.window import Window
import time, datetime

start_time = time.time()

spark = SparkSession \
    .builder \
    .appName("SQL query 1") \
    .getOrCreate()

df = spark.read.csv("hdfs://oceanos-master:54310/data/total_crime.csv" \
                    ,header=True)
```

```

df.createOrReplaceTempView("crime_records")

query_string = "SELECT * FROM(\n
    SELECT \n
        YEAR(`Date Rptd`) AS year, \n
        MONTH(`Date Rptd`) AS month, \n
        COUNT(*) AS crime_total, \n
        RANK() OVER (PARTITION BY YEAR(`Date Rptd`) ORDER BY COUNT(*) DESC) AS rank \n
    FROM \n
        crime_records \n
    GROUP BY \n
        YEAR(`Date Rptd`), \n
        MONTH(`Date Rptd`) \n
    ) AS ranked_data \n
    WHERE \n
        rank <= 3 \n
    ORDER BY \n
        year, \n
        rank;"

final_query = spark.sql(query_string)
final_query.show(100, truncate=False)
print('Total time for SQL: ',time.time() - start_time , 'sec')

```

query_1_df.py

```

from pyspark.sql import SparkSession
from pyspark.sql.types import StructField, StructType, TimestampType, DoubleType,
StringType, IntegerType
from pyspark.sql.functions import year, month, count, dense_rank, col, to_date,
row_number
from pyspark.sql.window import Window
import time, datetime

start_time = time.time()

spark = SparkSession \
    .builder \
    .appName("DF query 1") \
    .getOrCreate()

df = spark.read.csv("hdfs://oceanos-master:54310/data/total_crime.csv" \
    ,header=True)

df.createOrReplaceTempView("crime_records")

#add year, month, crime_counts as columns
df = df.withColumn("year", year("Date Rptd"))
df = df.withColumn("month", month("Date Rptd"))
crime_counts = df.groupBy("year", "month").\
    agg(count("*").alias("crime_total"))

#window function partitioned by year to rank the months
window_spec = Window.partitionBy("year").orderBy(col("crime_total").desc())

#add rank
crime_counts_with_rank = crime_counts.withColumn(
    "#",

```

```

    row_number().over(window_spec))

top_3_months = crime_counts_with_rank.where(col("#") <= 3)

top_3_months.show(100, truncate=False)

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

```

query_2_sql.py

```

from pyspark.sql import SparkSession
from pyspark.sql.types import StructField, StructType, TimestampType, DoubleType,
StringType, IntegerType
from pyspark.sql.functions import year, month, count, dense_rank, col, to_date
from pyspark.sql.window import Window
import time, datetime

start_time = time.time()

spark = SparkSession \
    .builder \
    .appName("SQL query 2") \
    .getOrCreate()

df = spark.read.csv("hdfs://okeanos-master:54310/data/total_crime.csv" \
                    ,header=True)

df.createOrReplaceTempView("crime_records")

query_string = """
SELECT
    part_of_day,
    COUNT(*) AS crime_count,
    DENSE_RANK() OVER (ORDER BY COUNT(*) DESC) AS rank
FROM (
    SELECT
        CASE
            WHEN CAST(`TIME OCC` AS INT) >= 500 AND CAST(`TIME OCC` AS INT) < 1200 THEN
'morning'
            WHEN CAST(`TIME OCC` AS INT) >= 1200 AND CAST(`TIME OCC` AS INT) < 1700 THEN
'afternoon'
            WHEN CAST(`TIME OCC` AS INT) >= 1700 AND CAST(`TIME OCC` AS INT) < 2100 THEN
'evening'
            ELSE 'night'
        END AS part_of_day
    FROM
        crime_records
    WHERE
        `Premis Desc` = 'STREET'
) tmp
GROUP BY
    part_of_day
ORDER BY
    rank;
"""

final_query = spark.sql(query_string)
final_query.show()
print('Total time for SQL: ',time.time() - start_time , 'sec')

```

query_2_rdd.py

```
from pyspark.sql import SparkSession
from operator import add
import csv
import time
from io import StringIO

start_time = time.time()

sc = SparkSession \
    .builder \
    .appName("RDD query ") \
    .getOrCreate() \
    .sparkContext

# CSV parser (for rows with multiple ',')
def parse_csv(line):
    reader = csv.reader(StringIO(line))
    return next(reader)

crime_records = sc.textFile("hdfs://okeanos-master:54310/data/total_crime.csv") \
    .map(parse_csv)

def get_part_of_day(time_occ):
    time_occ = int(time_occ)
    if 500 <= time_occ < 1200:
        return 'morning'
    elif 1200 <= time_occ < 1700:
        return 'afternoon'
    elif 1700 <= time_occ < 2100:
        return 'evening'
    else:
        return 'night'

part_of_day_rdd = crime_records.filter(lambda x: x[15] == 'STREET') \
    .map(lambda x: get_part_of_day(x[3]))

#count occurrences of each part of the day
part_of_day_counts = part_of_day_rdd.map(lambda x: (x, 1)).reduceByKey(add)

#sort by count in descending order
sorted_part_of_day = part_of_day_counts.map(lambda x: (x[1], x[0])) \
    .sortByKey(ascending=False)

#add rank
ranked_part_of_day = sorted_part_of_day.zipWithIndex() \
    .map(lambda x: (x[0][1], x[0][0], x[1] + 1))

for result in ranked_part_of_day.collect():
    print(result)

print('Total time for RDD: ', time.time() - start_time, 'sec')
sc.stop()
```

query_3_df.py

```
from pyspark.sql import SparkSession, functions as F
from pyspark.sql.window import Window
from pyspark.sql.functions import count, desc, year, row_number, col, regexp_replace
import time

start_time = time.time()

spark = SparkSession \
    .builder \
    .appName("DF query 3") \
    .getOrCreate()

income = spark.read.csv("hdfs://oceanos-master:54310/data/income/LA_income_2015.csv",
header=True)
df = spark.read.csv("hdfs://oceanos-master:54310/data/total_crime.csv" \
    ,header=True)
revgecoding = spark.read.csv("hdfs://oceanos-master:54310/data/revgecoding.csv",
header=True)

#filter data to include only 2015, and remove victimless crimes
df = df.filter(year(df["Date Rptd"]) == 2015)
df = df.filter(df["Vict Descent"] != "X")

#join based on longitude and latitude
joined_df = df.join(revgecoding.hint("broadcast"), (df.LON == revgecoding.LON) & (df.LAT
== revgecoding.LAT), "left")
joined_df.explain(extended=True)

#join with revgeocoding
joined_df = joined_df.select(df["*"], revgecoding["ZIPcode"].alias("Joined_ZIP_Code"))

#join with income
joined_income_df = joined_df.join(income, joined_df["Joined_ZIP_Code"] == income["Zip
Code"], "inner")

#remove dolar signs
joined_income_df = joined_income_df.withColumn(
    "Estimated Median Income",
    regexp_replace(col("Estimated Median Income"), "[^\\d.]", ""))
)
joined_income_df = joined_income_df.withColumn(
    "Estimated Median Income",
    col("Estimated Median Income").cast("float"))
)

final_df = joined_income_df.select(joined_df["Vict Descent"],
joined_df["Joined_ZIP_Code"], \
    joined_income_df["Estimated Median Income"])

grouped_final_df = final_df.groupBy('Vict Descent', 'Joined_ZIP_Code', 'Estimated Median
Income') \
    .agg(F.count('*').alias('#'))

sorted_grouped_final_df = grouped_final_df.orderBy(desc("Estimated Median Income"))

unique_income_df = sorted_grouped_final_df.dropDuplicates(["Joined_ZIP_Code", "Estimated
Median Income"])
sorted_unique_income_df_desc = unique_income_df.orderBy(desc("Estimated Median Income"))

#top 3
```

```

top_3_zip_codes = sorted_unique_income_df_desc.select("Joined_ZIP_Code")
top_3_zip_codes = top_3_zip_codes.limit(3)

#bottom 3
sorted_unique_income_df_asc = unique_income_df.orderBy("Estimated Median Income")
bot_3_zip_codes = sorted_unique_income_df_asc.select("Joined_ZIP_Code")
bot_3_zip_codes = bot_3_zip_codes.limit(3)

#filter for rows corresponding to the top 3 zip codes
filtered_top_3_df = grouped_final_df.join(
    top_3_zip_codes,
    grouped_final_df["Joined_ZIP_Code"] == top_3_zip_codes["Joined_ZIP_Code"],
    'inner'
)

#filter for rows corresponding to the bottom 3 zip codes
filtered_bot_3_df = grouped_final_df.join(
    bot_3_zip_codes,
    grouped_final_df["Joined_ZIP_Code"] == bot_3_zip_codes["Joined_ZIP_Code"],
    'inner'
)

top_result = filtered_top_3_df.groupBy('Vict
Descent').agg(F.sum('#').alias('#')).orderBy(desc('#'))

bot_result = filtered_bot_3_df.groupBy('Vict
Descent').agg(F.sum('#').alias('#')).orderBy(desc('#'))

#reform
map_descent = {
    "W": "White",
    "O": "Other",
    "B": "Black",
    "H": "Hispanic/Latin/Mexican",
    "A": "Other Asian",
    "C": "Chinese",
    "D": "Cambodian",
    "F": "Filipino",
    "G": "Guamanian",
    "I": "American Indian/Alaskan Native",
    "J": "Japanese",
    "L": "Laotian",
    "P": "Pacific Islander",
    "S": "Samoan",
    "U": "Hawaiian",
    "V": "Vietnamese",
    "Z": "Asian Indian"
}

map_function = F.udf(lambda x: map_descent.get(x))
top_result = top_result.withColumn("Vict Descent", map_function(df["Vict Descent"]))
bot_result = bot_result.withColumn("Vict Descent", map_function(df["Vict Descent"]))
top_result.show()
bot_result.show()

print('Total time: ',time.time() - start_time , 'sec')

```

query_4_1_df.py

```
from pyspark.sql import SparkSession
from pyspark.sql.types import IntegerType
from pyspark.sql.functions import year, count, col, mean, cos, asin, sqrt
import time

start_time = time.time()

spark = SparkSession \
    .builder \
    .appName("DF query 4") \
    .getOrCreate()

#spark.sparkContext.addPyFile("/home/user/geopy-2.4.1/geopy")
#import distance

df = spark.read.csv("hdfs://oceanos-master:54310/data/total_crime.csv" \
    ,header=True)

df = df.withColumn("AREA", col("AREA").cast(IntegerType()))

police_stations = spark.read.csv("hdfs://oceanos-master:54310/data/la_police_stations" \
    ,header=True)

police_stations = police_stations.withColumn("PREC", col("PREC").cast(IntegerType()))

# calculate the distance between two points [lat1, lon1], [lat2, lon2] in km
def get_distance(lat1, lon1, lat2, lon2):
    r = 6371 # km
    p = 3.14 / 180.0

    a = 0.5 - cos((lat2-lat1)*p)/2 + cos(lat1*p) * cos(lat2*p) *
(1-cos((lon2-lon1)*p))/2
    return 2 * r * asin(sqrt(a))

df = df.select(df["LAT"], df["LON"], df["DATE OCC"], df["AREA"], df["Weapon Used Cd"])

firearm_crimes = df.filter(df["Weapon Used Cd"].like("1__"))

joined_df = firearm_crimes.join(
    #police_stations.hint("shuffle_replicate_nl"),
    police_stations,
    firearm_crimes["AREA"] == police_stations["PREC"],
    "left"
)
#joined_df.explain(extended=True)

#filter out NULL
filtered_df = joined_df.filter(((col("LAT") != 0.0) & (col("LON") != 0.0)) &
    (col("X").isNotNull()) &
    (col("Y").isNotNull())
)

#distance_udf = udf(get_distance, FloatType())
distance_df = filtered_df.withColumn("distance", get_distance(col("LAT"), col("LON"),
col("Y"), col("X")))
distance_df = distance_df.withColumn("year", year("DATE OCC"))

final = distance_df.groupBy("year").agg(
    count("*").alias("#"),
```



```

        mean("distance").alias("average_distance")
    ).orderBy("year")
    final = final.select("year", "average_distance", "#")
    final.show()

#4_1_b
distance_df_b = filtered_df.withColumn("distance", get_distance(col("LAT"), col("LON"),
col("Y"), col("X")))

final_b = distance_df_b.groupBy("DIVISION").agg(
    count("*").alias("#"),
    mean("distance").alias("average_distance")
).orderBy(col("#").cast("int").desc())

final_b = final_b.select("DIVISION", "average_distance", "#")
final_b.show(21, truncate=False)

```

query_4_2_df.py

```

from pyspark.sql import SparkSession
from pyspark.sql.types import IntegerType
from pyspark.sql import functions as F
from pyspark.sql.functions import year, count, col, mean, cos, asin, sqrt,
monotonically_increasing_id
from pyspark.sql.window import Window
import time

start_time = time.time()

spark = SparkSession \
    .builder \
    .appName("DF query 4b") \
    .getOrCreate()

df = spark.read.csv("hdfs://oceanos-master:54310/data/total_crime.csv" \
    ,header=True)

df = df.withColumn("AREA", col("AREA").cast(IntegerType()))

police_stations = spark.read.csv("hdfs://oceanos-master:54310/data/la_police_stations" \
    ,header=True)

police_stations = police_stations.withColumn("PREC",
    col("PREC").cast(IntegerType()))

# calculate the distance between two points [lat1, lon1], [lat2, lon2] in km
def get_distance(lat1, lon1, lat2, lon2):
    r = 6371 # km
    p = 3.14 / 180.0

    a = 0.5 - cos((lat2 - lat1) * p) / 2 + cos(lat1 * p) * cos(lat2 * p) * (1 - cos((lon2
    - lon1) * p)) / 2
    return 2 * r * asin(sqrt(a))

df = df.select(df["LAT"], df["LON"], df["DATE OCC"], df["AREA"],
    df["Weapon Used Cd"])

```

```

firearm_crimes = df.filter(df["Weapon Used Cd"].like("1__"))
firearm_crimes = firearm_crimes.withColumn("id", monotonically_increasing_id()) #to use
with window function

#cartesian product
#joined_df = firearm_crimes.crossJoin(police_stations.hint("shuffle_replicate_n1"))
joined_df = firearm_crimes.crossJoin(police_stations)
#joined_df.explain(extended=True)

#filter out NULL
filtered_df_a = joined_df.filter(((col("LAT") != 0.0) & (col("LON") != 0.0)) &
                                (col("X").isNotNull()) &
                                (col("Y").isNotNull())
)

distance_df = filtered_df_a.withColumn(
    "distance", get_distance(col("LAT"), col("LON"), col("Y"), col("X")))

window = Window.partitionBy("id").orderBy("distance")

closest_df = distance_df.withColumn("year", year("DATE OCC"))
closest_df = closest_df.withColumn(
    "rank",
    F.row_number().over(window))
closest_df = closest_df.filter(col("rank") == 1)

final = closest_df.groupBy("year").agg(
    count("*").alias("#"),
    mean("distance").alias("average_distance")).orderBy("year")

final = final.select("year", "average_distance", "#")
final.show()

#4_2_b
final_b = closest_df.groupBy("DIVISION").agg(
    count("*").alias("#"),
    mean("distance").alias("average_distance")).orderBy(
    col("#").cast("int").desc())

final_b = final_b.select("DIVISION", "average_distance", "#")
final_b.show(21, truncate=False)

```