



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ

Αναφορά εξαμηνιαίας εργασίας για το μάθημα
«Προχωρημένα Θέματα Βάσεων Δεδομένων»

Ομάδα 46

Βασιλική Μισύρη 03119123
Δημήτριος Βασιλείου 03119830

Ερώτημα 1

Ακολουθώντας τα βήματα εγκατάστασης, έχουν δημιουργηθεί δύο εικονικές μηχανές στην υπηρεσία Okeanos-Knossos και έχουν επιτυχώς εγκατασταθεί και διαμορφωθεί τα Apache Spark, Hadoop Distributed File System και Apache Yarn. Για να επιβεβαιώσουμε τη σωστή λειτουργία του cluster μας εκτελούμε αρχικά τις παρακάτω εντολές:

- `ssh snf-39522.ok-kno.grnetcloud.net` για να συνδεθούμε στον master node.
- Έχοντας συνδεθεί επιτυχώς στον master, εκτελούμε `start-dfs.sh` για να εκκινήσουμε το HDFS και στους δύο κόμβους.
- Εκτελούμε `start-yarn.sh` για να εκκινήσουμε το Yarn.
- Τέλος, εκτελούμε `$SPARK_HOME/sbin/start-history-server.sh` για να εκκινήσουμε τον Spark History Server.

Έχοντας εκτελέσει τις παραπάνω εντολές συνεχίζουμε ανοίγοντας τις web εφαρμογές των HDFS, Hadoop Yarn και Spark UI ακολουθώντας τους συνδέσμους <http://83.212.73.13:9870>, <http://83.212.73.13:8088/cluster> και <http://83.212.73.13:8088/cluster> αντίστοιχα.

Ακολουθώντας τον πρώτο σύνδεσμο βλέπουμε 2 διαθέσιμους nodes οπότε το HDFS λειτουργεί σωστά.

Configured Capacity:	58.78 GB
Configured Remote Capacity:	0 B
DFS Used:	1.18 GB (2%)
Non DFS Used:	11.68 GB
DFS Remaining:	42.89 GB (72.97%)
Block Pool Used:	1.18 GB (2%)
DataNodes usages% (Min/Median/Max/stdDev):	0.00% / 4.00% / 4.00% / 2.00%
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)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion (including replicas)	0
Block Deletion Start Time	Sun Jan 07 00:47:55 +0200 2024
Last Checkpoint Time	Sun Jan 07 00:47:57 +0200 2024
Enabled Erasure Coding Policies	RS-6-3-1024k

Ακολουθώντας τον δεύτερο σύνδεσμο βλέπουμε 2 διαθέσιμους nodes οπότε το Hadoop Yarn λειτουργεί σωστά.



Cluster

[About](#)
[Nodes](#)
[Node Labels](#)
[Applications](#)
[NEW](#)
[NEW SAVING](#)
[SUBMITTED](#)
[ACCEPTED](#)
[RUNNING](#)
[FINISHED](#)
[FAILED](#)
[KILLED](#)
[Scheduler](#)

Tools

Cluster Metrics

Apps Submitted	Apps F
0	0

Cluster Nodes Metrics

Active Nodes
2

Scheduler Metrics

Scheduler Type
Capacity Scheduler

Show 20 entries

ID	User	Name	Applica Type
Showing 0 to 0 of 0 entries			

Τέλος, ακολουθώντας τον τρίτο σύνδεσμο βλέπουμε το ιστορικό των υποβληθέντων εφαρμογών στο Spark.

spark 3.5.0

History Server

Event log directory: hdfs://master-42:54310/spark.eventLog

Last updated: 2024-01-07 00:53:31

Client local time zone: Europe/Athens

Show 20 entries

Search:

Version	App ID	App Name	Attempt ID	Started	Completed	Duration	Spark User	Last Updated	Event Log
3.5.0	application_1704569987208_0003	Dataframe query 3 execution 2 executors final		2024-01-06 21:44:28	2024-01-06 21:45:35	1.1 min	user	2024-01-06 21:45:35	Download
3.5.0	application_1704569987208_0002	Dataframe query 3 execution 3 executors final		2024-01-06 21:42:53	2024-01-06 21:44:01	1.1 min	user	2024-01-06 21:44:01	Download
3.5.0	application_1704569987208_0001	Dataframe query 3 execution 4 executors final		2024-01-06 21:41:14	2024-01-06 21:42:29	1.3 min	user	2024-01-06 21:42:29	Download
3.5.0	application_1704566151047_0011	Dataframe query 3 execution 4 executors		2024-01-06 21:03:35	2024-01-06 21:05:15	1.7 min	user	2024-01-06 21:05:15	Download

Συμπεραίνουμε λοιπόν, ότι το cluster μας έχει διαμορφωθεί σωστά.

Ερώτημα 2

Παραθέτουμε τον κώδικα για την εκτέλεση του ερωτήματος:

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

spark = SparkSession \
    .builder \
    .appName("Dataframe Creation") \
    .getOrCreate()

crimes_df1 = spark.read.csv("Crime_Data_from_2010_to_2019.csv", header=True, inferSchema=True)
crimes_df2 = spark.read.csv("Crime_Data_from_2020_to_Present.csv", header=True,
inferSchema=True)
crimes_df = crimes_df1.union(crimes_df2)

crimes_df = crimes_df.withColumn("Date Rptd", col("Date Rptd").cast(DateType()))
crimes_df = crimes_df.withColumn("DATE OCC", col("DATE OCC").cast(DateType()))
crimes_df = crimes_df.withColumn("Vict Age", col("Vict Age").cast(IntegerType()))
crimes_df = crimes_df.withColumn("LAT", col("LAT").cast(DoubleType()))
crimes_df = crimes_df.withColumn("LON", col("LON").cast(DoubleType()))

crimes_df.printSchema()

print(f"Total number of lines: {crimes_df.count()}")

print("Column data types:")
print(crimes_df.dtypes)
```

Παραθέτουμε το output του ερωτήματος:

```

root
|-- 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: 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 Cd 1: integer (nullable = true)
|-- Crm Cd 2: integer (nullable = true)
|-- Crm Cd 3: integer (nullable = true)
|-- Crm Cd 4: integer (nullable = true)
|-- LOCATION: string (nullable = true)
|-- Cross Street: string (nullable = true)
|-- LAT: double (nullable = true)
|-- LON: double (nullable = true)

```

Total number of lines: 2988445

Column data types:

```

[('DR_NO', 'int'), ('Date Rptd', 'date'), ('DATE OCC', 'date'), ('TIME OCC',
'int'), ('AREA ', 'int'), ('AREA NAME', 'string'), ('Rpt Dist No', 'int'), ('Part
1-2', 'int'), ('Crm Cd', 'int'), ('Crm Cd Desc', 'string'), ('Mocodes', 'string'),
('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 Cd 1', 'int'),
('Crm Cd 2', 'int'), ('Crm Cd 3', 'int'), ('Crm Cd 4', 'int'), ('LOCATION',
'string'), ('Cross Street', 'string'), ('LAT', 'double'), ('LON', 'double')]

```

Ερώτημα 3

Εκτέλεση με Dataframe API

Παραθέτουμε τον κώδικα για την εκτέλεση του ερωτήματος χρησιμοποιώντας το Dataframe API:

```

from pyspark.sql import SparkSession
from pyspark.sql.window import Window
from pyspark.sql import functions as F
from pyspark.sql.types import IntegerType, DoubleType

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

crimes_df1 = spark.read.csv("Crime_Data_from_2010_to_2019.csv", header=True, inferSchema=True)
crimes_df2 = spark.read.csv("Crime_Data_from_2020_to_Present.csv", header=True,
inferSchema=True)
crimes_df = crimes_df1.union(crimes_df2)

crimes_df = crimes_df.withColumn("Date Rptd", F.to_date(F.col("Date Rptd"), "MM/dd/yyyy hh:mm:ss
a"))
crimes_df = crimes_df.withColumn("DATE OCC", F.to_date(F.col("DATE OCC"), "MM/dd/yyyy hh:mm:ss
a"))

```

```

crimes_df = crimes_df.withColumn("Vict Age", F.col("Vict Age").cast(IntegerType()))
crimes_df = crimes_df.withColumn("LAT", F.col("LAT").cast(DoubleType()))
crimes_df = crimes_df.withColumn("LON", F.col("LON").cast(DoubleType()))

# Using DataFrame API for the query
window_spec = Window.partitionBy("year").orderBy(F.desc("crime_total"))

result_df = crimes_df.groupBy(F.year("DATE OCC").alias("year"), F.month("DATE
OCC").alias("month")) \
    .agg(F.count("*").alias("crime_total")) \
    .withColumn("#", F.row_number().over(window_spec)) \
    .filter(F.col("#") <= 3) \
    .orderBy("year", F.desc("crime_total"))

result_df.show(result_df.count(), truncate=False)

```

Παραθέτουμε το output του ερωτήματος:

year	month	crime_total	#
2010	1	19515	1
2010	3	18131	2
2010	7	17856	3
2011	1	18134	1
2011	7	17283	2
2011	10	17034	3
2012	1	17943	1
2012	8	17661	2
2012	5	17502	3
2013	8	17440	1
2013	1	16820	2
2013	7	16644	3
2014	7	12196	1
2014	10	12133	2
2014	8	12028	3
2015	10	19219	1
2015	8	19011	2
2015	7	18709	3
2016	10	19659	1
2016	8	19490	2
2016	7	19448	3
2017	10	20431	1
2017	7	20192	2
2017	1	19833	3
2018	5	19972	1
2018	7	19875	2
2018	8	19761	3
2019	7	19121	1
2019	8	18979	2
2019	3	18854	3
2020	1	18496	1
2020	2	17255	2
2020	5	17204	3
2021	10	26676	1
2021	12	26317	2
2021	11	25715	3
2022	5	20418	1
2022	10	20274	2
2022	6	20201	3
2023	8	19743	1
2023	7	19697	2
2023	1	19633	3

Ο χρόνος εκτέλεσης στην υλοποίηση με Dataframe είναι 1.1 min όπως βλέπουμε από τον Spark History Server:

3.5.0	application_1704999680291_0002	Dataframe query 1 execution final	2024-01-11 21:05:28	2024-01-11 21:06:32	1.1 min	user	2024-01-11 21:06:33	Download
-------	--------------------------------	-----------------------------------	---------------------	---------------------	---------	------	---------------------	--------------------------

Εκτέλεση με SQL API

Παραθέτουμε τον κώδικα για την εκτέλεση του ερωτήματος χρησιμοποιώντας το SQL API:

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

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

crimes_df1 = spark.read.csv("Crime_Data_from_2010_to_2019.csv", header=True, inferSchema=True)
crimes_df2 = spark.read.csv("Crime_Data_from_2020_to_Present.csv", header=True,
inferSchema=True)
crimes_df = crimes_df1.union(crimes_df2)

crimes_df = crimes_df.withColumn("Date Rptd", to_date(col("Date Rptd"), "MM/dd/yyyy hh:mm:ss
a"))
crimes_df = crimes_df.withColumn("DATE OCC", to_date(col("DATE OCC"), "MM/dd/yyyy hh:mm:ss a"))
crimes_df = crimes_df.withColumn("Vict Age", col("Vict Age").cast(IntegerType()))
crimes_df = crimes_df.withColumn("LAT", col("LAT").cast(DoubleType()))
crimes_df = crimes_df.withColumn("LON", col("LON").cast(DoubleType()))

crimes_df.createOrReplaceTempView("crimes")

max_crimes_per_month_and_year_query = """SELECT year, month, crime_total, `#`
FROM (
SELECT
YEAR(`DATE OCC`) AS year,
MONTH(`DATE OCC`) AS month,
COUNT(*) AS crime_total,
ROW_NUMBER() OVER (PARTITION BY YEAR(`DATE OCC`)

ORDER BY COUNT(*) DESC) as `#`

FROM crimes
GROUP BY YEAR(`DATE OCC`), MONTH(`DATE OCC`)
) ranked
WHERE `#` <= 3
ORDER BY year ASC, crime_total DESC"""

result = spark.sql(max_crimes_per_month_and_year_query)
result.show(result.count(), truncate=False)
```

Παραθέτουμε το output του ερωτήματος:

year	month	crime_total	#
2010	1	19515	1
2010	3	18131	2
2010	7	17856	3
2011	1	18134	1
2011	7	17283	2
2011	10	17034	3
2012	1	17943	1
2012	8	17661	2
2012	5	17502	3
2013	8	17440	1
2013	1	16820	2
2013	7	16644	3
2014	7	12196	1
2014	10	12133	2
2014	8	12028	3
2015	10	19219	1

2015 8	19011	2	
2015 7	18709	3	
2016 10	19659	1	
2016 8	19490	2	
2016 7	19448	3	
2017 10	20431	1	
2017 7	20192	2	
2017 1	19833	3	
2018 5	19972	1	
2018 7	19875	2	
2018 8	19761	3	
2019 7	19121	1	
2019 8	18979	2	
2019 3	18854	3	
2020 1	18496	1	
2020 2	17255	2	
2020 5	17204	3	
2021 10	26676	1	
2021 12	26317	2	
2021 11	25715	3	
2022 5	20418	1	
2022 10	20274	2	
2022 6	20201	3	
2023 8	19743	1	
2023 7	19697	2	
2023 1	19633	3	

+-----+-----+-----+-----+

Ο χρόνος εκτέλεσης στην υλοποίηση με SQL είναι 1.1 min όπως βλέπουμε από τον Spark History Server:

3.5.0	application_1704999680291_0003	SQL query 1 execution final	2024-01-11 21:07:09	2024-01-11 21:08:15	1.1 min	user	2024-01-11 21:08:16	Download
-------	--------------------------------	-----------------------------	------------------------	------------------------	---------	------	------------------------	--------------------------

Σχολιασμός επίδοσης των APIs

Σημειώνουμε ότι και οι δύο υλοποιήσεις εκτελέστηκαν με 4 spark executors.

Παρατηρούμε ότι τόσο η υλοποίηση με Dataframe API όσο και αυτή με SQL API οδηγούν σε ίδιο χρόνο εκτέλεσης (αθροίζοντας τους ακριβείς χρόνους εκτέλεσης των επιμέρους jobs σε κάθε περίπτωση φαίνεται πως υπάρχει μια διαφορά περίπου 1 s η οποία είναι αμελητέα και ίσως οφείλεται και στο διαφορετικό επεξεργαστικό φορτίο των virtual machines κατά την διάρκεια των δύο εκτελέσεων). Και στις δύο υλοποιήσεις, κατά την εκτέλεση, το Spark επεξεργάζεται το logical plan που δημιουργείται από τις εντολές Dataframe ή SQL και το μετατρέπει σε physical plan που μπορεί να εκτελεστεί στο κατανεμημένο περιβάλλον του Spark Cluster. Το physical plan αυτό περιγράφει τις πραγματικές λειτουργίες και φυσικές ενέργειες που πρέπει να γίνουν για να εκτελεστεί ο Dataframe ή SQL κώδικας. Παρατηρώντας τα δύο physical plans που προκύπτουν κατά τις υλοποιήσεις με τα δύο διαφορετικά APIs από τον Spark History Server, βλέπουμε πως αυτά είναι ακριβώς ίδια, κάτι που αιτιολογεί τους ίσους χρόνους εκτέλεσής τους.

Ερώτημα 4

Εκτέλεση με RDD API

Παραθέτουμε τον κώδικα για την εκτέλεση του ερωτήματος χρησιμοποιώντας το RDD API:

```
from pyspark.sql import SparkSession
import csv

sc = SparkSession \
    .builder \
    .appName("RDD query 2 execution final") \
    .config("spark.executor.instances", "4") \
    .getOrCreate() \
```

```

.sparkContext

def is_morning(str_time):
    time = int(str_time)
    return 500 <= time and time <= 1159

def is_noon(str_time):
    time = int(str_time)
    return 1200 <= time and time <= 1659

def is_evening(str_time):
    time = int(str_time)
    return 1700 <= time and time <= 2059

def is_night(str_time):
    time = int(str_time)
    return ((2100 <= time and time <= 2359) or (time >= 0 and time <= 459))

def split_csv(line):
    return next(csv.reader([line]))

file_paths = ["Crime_Data_from_2010_to_2019.csv", "Crime_Data_from_2020_to_Present.csv"]
crimes_street_day_periods = sc.textFile(",".join(file_paths)) \
    .map(split_csv) \
    .filter(lambda x: x[15] == "STREET") \
    .map(lambda x: ("morning", 1) if is_morning(x[3]) else ("noon", 1) if is_noon(x[3]) else
    ("evening", 1) if is_evening(x[3]) else ("night", 1))) \
    .reduceByKey(lambda x, y: x + y) \
    .sortBy(lambda x: x[1], ascending=False) \

print(crimes_street_day_periods.collect())

```

Παραθέτουμε το output του ερωτήματος:

```
[('night', 237137), ('evening', 186896), ('noon', 148077), ('morning', 123748)]
```

Ο χρόνος εκτέλεσης στην υλοποίηση με RDD είναι 40 s όπως βλέπουμε από τον Spark History Server:

3.5.0	application_1704581292398_0002	RDD query 2 execution final	2024-01-07 01:08:13	2024-01-07 01:08:53	40 s
-------	--------------------------------	-----------------------------	---------------------	---------------------	------

Εκτέλεση με Dataframe API

Παραθέτουμε τον κώδικα για την εκτέλεση του ερωτήματος χρησιμοποιώντας το Dataframe API:

```

from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when

spark = SparkSession \
    .builder \
    .appName("Dataframe query 2 execution final") \
    .config("spark.executor.instances", "4") \
    .getOrCreate() \

crimes_df1 = spark.read.csv("Crime_Data_from_2010_to_2019.csv", header=True, inferSchema=True)
crimes_df2 = spark.read.csv("Crime_Data_from_2020_to_Present.csv", header=True,
inferSchema=True)
crimes_df = crimes_df1.union(crimes_df2)

street_crimes_df = crimes_df.filter(crimes_df["Premis Desc"] == "STREET")
street_crimes_with_day_period_df = street_crimes_df.withColumn("Day Period",
when(street_crimes_df["TIME OCC"].between(500, 1159), "morning") \
    .when(street_crimes_df["TIME OCC"].between(1200, 1659), "noon") \
    .when(street_crimes_df["TIME OCC"].between(1700, 2059), "evening") \
    .when(street_crimes_df["TIME OCC"].between(2100, 2359), "night").when(street_crimes_df["TIME
OCC"].between(0, 459), "night"))

street_crimes_with_day_period_df.groupBy("Day Period").count().select("Day Period",
"count").orderBy(col("count").desc()).show()

```


Παραθέτουμε το output του ερωτήματος:

```
+-----+-----+
|Day Period| count|
+-----+-----+
|      night|237137|
|    evening|186896|
|       noon|148077|
|    morning|123748|
+-----+-----+
```

Ο χρόνος εκτέλεσης στην υλοποίηση με Dataframe είναι 53 s όπως βλέπουμε από τον Spark History Server:

3.5.0	application_1704581292398_0003	Dataframe query 2 execution final	2024-01-07 01:15:41	2024-01-07 01:16:34	53 s
-------	--------------------------------	-----------------------------------	---------------------	---------------------	------

Σχολιασμός επίδοσης των APIs

Σημειώνουμε ότι και οι δύο υλοποιήσεις εκτελέστηκαν με 4 spark executors.

Παρατηρούμε ότι η εκτέλεση με RDD API είναι γρηγορότερη από αυτή με Dataframe API κατά 13 seconds, χρόνος δηλαδή που δεν μπορεί να θεωρηθεί αμελητέος. Αυτό μπορεί να συμβαίνει για διάφορους λόγους.

- ➔ Το RDD επεξεργάζεται τα δεδομένα ως λίστες από strings δίχως να σχηματίζει κάποιο σχήμα για αυτά όπως κάνει το Dataframe. Η δημιουργία σχήματος από το dataset και η εξαγωγή τύπων για κάθε στήλη κοστίζει σε χρόνο, συνεπώς το RDD ξεκινάει ταχύτερα την επεξεργασία.
- ➔ Το RDD χρησιμοποιεί low-level συναρτήσεις όπως οι flat, flatMap και reduceByKey των οποίων η χρήση κρίνεται περισσότερο αποδοτική σε συγκεκριμένες περιπτώσεις σε αντίθεση με την περισσότερο high-level λογική του Dataframe. Το δοθέν ερώτημα μπορεί εύκολα και γρήγορα να υλοποιηθεί με RDD καθώς είναι απλό, δεν απαιτεί περίπλοκες ενέργειες όπως joins, ενώ αποτελείται απλώς από λίγες και διαδοχικές ενέργειες πάνω στο ίδιο dataset.
- ➔ Το Dataframe API χρησιμοποιεί optimizations σε αντίθεση με το RDD. Για ένα τόσο απλό ερώτημα ο χρόνος που προστίθεται για υλοποιηθούν τα optimizations προσθέτει επιπλέον overhead, καθιστώντας έτσι την υλοποίηση με Dataframe πιο αργή.

Ερώτημα 5

Παραθέτουμε τον κώδικα για την εκτέλεση του ερωτήματος χρησιμοποιώντας το Dataframe API:

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField, StructType, IntegerType, DoubleType, StringType,
DateType
from pyspark.sql.functions import col, when, regexp_extract, regexp_replace, year, to_date

spark = SparkSession \
    .builder \
    .appName("Dataframe query 3 execution 2 executors final") \
    .config("spark.executor.instances", "2") \
    .getOrCreate() \

crimes_df = spark.read.csv("Crime_Data_from_2010_to_2019.csv", header=True, inferSchema=True)

crimes_df = crimes_df.withColumn("Date Rptd", to_date(col("Date Rptd"), "MM/dd/yyyy hh:mm:ss a"))
crimes_df = crimes_df.withColumn("DATE OCC", to_date(col("DATE OCC"), "MM/dd/yyyy hh:mm:ss a"))
crimes_df = crimes_df.withColumn("Vict Age", col("Vict Age").cast(IntegerType()))
crimes_df = crimes_df.withColumn("LAT", col("LAT").cast(DoubleType()))
crimes_df = crimes_df.withColumn("LON", col("LON").cast(DoubleType()))

# Filter year 2015 on crimes and filter nulls
```

```

excluded_null_crimes = crimes_df.filter((year(crimes_df["Date Rptd"]) == 2015) & crimes_df["Vict
Descent"].isNotNull())

# Alias names for user-friendly results
victims_alias_crimes_df = excluded_null_crimes.withColumn("Vict Descent", when(crimes_df["`Vict
Descent`"] == "A", "Other Asian")
    .when(crimes_df["`Vict Descent`"] == "B", "Black")
    .when(crimes_df["`Vict Descent`"] == "C", "Chinese")
    .when(crimes_df["`Vict Descent`"] == "D", "Cambodian")
    .when(crimes_df["`Vict Descent`"] == "F", "Filipino")
    .when(crimes_df["`Vict Descent`"] == "G", "Guamanian")
    .when(crimes_df["`Vict Descent`"] == "H", "Hispanic/Latin/Mexican")
    .when(crimes_df["`Vict Descent`"] == "I", "American Indian/Alaskan Native")
    .when(crimes_df["`Vict Descent`"] == "J", "Japanese")
    .when(crimes_df["`Vict Descent`"] == "K", "Korean")
    .when(crimes_df["`Vict Descent`"] == "L", "Laotian")
    .when(crimes_df["`Vict Descent`"] == "O", "Other")
    .when(crimes_df["`Vict Descent`"] == "P", "Pacific Islander")
    .when(crimes_df["`Vict Descent`"] == "S", "Samoan")
    .when(crimes_df["`Vict Descent`"] == "U", "Hawaiian")
    .when(crimes_df["`Vict Descent`"] == "V", "Vietnamese")
    .when(crimes_df["`Vict Descent`"] == "W", "White")
    .when(crimes_df["`Vict Descent`"] == "X", "Unknown")
    .when(crimes_df["`Vict Descent`"] == "Z", "Asian Indian"))

# Read income and filter out $ and , in the income column. Keep records that are located on LA
income_df = spark.read.csv("LA_income_2015.csv", header=True, inferSchema=True)
income_df = income_df.withColumn("Estimated Median Income", regexp_replace("Estimated Median
Income", '\\$', ''))
income_df = income_df.withColumn("Estimated Median Income", regexp_replace("Estimated Median
Income", ',, ', '' ).cast("int"))
income_df = income_df.filter(col("Community").contains("Los Angeles"))

# Read geocoding and if a columnn has two zip codes, always keep the first
rev_geocoding_df = spark.read.csv("revgeocoding.csv", header=True, inferSchema=True)
rev_geocoding_df = rev_geocoding_df.withColumn("ZIPcode", regexp_extract("ZIPcode", r"(\d+)",
1).cast("int"))

# Filter out zip codes where none crime occurred
rev_geocoding_zips_df = rev_geocoding_df.select("ZIPcode").distinct()
income_with_crimes_df = income_df.join(rev_geocoding_zips_df, income_df["Zip Code"] ==
rev_geocoding_zips_df["ZIPcode"], "inner")

# Find highest 3 and lowest 3 zip codes based on income
highest_3_zip_codes = income_with_crimes_df.orderBy(col("Estimated Median
Income").desc()).limit(3)
lowest_3_zip_codes = income_with_crimes_df.orderBy(col("Estimated Median
Income").asc()).limit(3)

# Get zip code on crimes by joining crimes with geocoding on LAT, LON
joined1 = victims_alias_crimes_df.alias("victims_alias_crimes").join(
    rev_geocoding_df.alias("geocoding"),
    (col("victims_alias_crimes.LAT") == col("geocoding.LAT")) & (col("victims_alias_crimes.LON")
== col("geocoding.LON")),
    "inner"
)

# Join the above with highest 3 and lowest 3 zip codes to get the result
print("Highest 3")
joined2 = joined1.join(highest_3_zip_codes, highest_3_zip_codes["Zip Code"] ==
joined1["ZIPcode"], "inner")
joined2 = joined2.groupBy("Vict Descent").count().select("Vict Descent",
"count").orderBy(col("count").desc())
joined2.show()

print()

print("Lowest 3")
joined3 = joined1.join(lowest_3_zip_codes, lowest_3_zip_codes["Zip Code"] == joined1["ZIPcode"],
"inner")
joined3 = joined3.groupBy("Vict Descent").count().select("Vict Descent",
"count").orderBy(col("count").desc())
joined3.show()

```

Σημειώνουμε ότι για την εκτέλεση με 2 και 3 executors απλώς αλλάζουμε την γραμμή `.config("spark.executor.instances", "2") \` βάζοντας 2 ή 3

Παραθέτουμε το output του ερωτήματος:

Highest 3

	Vict	Descent	count
		White	714
		Other	297
	Hispanic/Latin/Me...		119
		Black	37
		Unknown	36
		Other Asian	24
		Chinese	1

Lowest 3

	Vict	Descent	count
	Hispanic/Latin/Me...		1521
		Black	1079
		White	692
		Other	389
		Other Asian	101
		Unknown	64
		Korean	7
	American Indian/A...		3
		Japanese	3
		Chinese	2
		Filipino	2

Χρόνος εκτέλεσης με 4 executors

Είναι 1.1 min όπως βλέπουμε από τον Spark History Server:

3.5.0	application_1704919127361_0007	Dataframe query 3 execution 4 executors Final	2024-01-10 23:57:28	2024-01-10 23:58:35	1.1 min
-------	--------------------------------	---	---------------------	---------------------	---------

Χρόνος εκτέλεσης με 3 executors

Είναι 1.0 min όπως βλέπουμε από τον Spark History Server:

3.5.0	application_1704919127361_0006	Dataframe query 3 execution 3 executors Final	2024-01-10 23:55:56	2024-01-10 23:56:57	1.0 min
-------	--------------------------------	---	---------------------	---------------------	---------

Χρόνος εκτέλεσης με 2 executors

Είναι 1.1 min όπως βλέπουμε από τον Spark History Server:

3.5.0	application_1704919127361_0003	Dataframe query 3 execution 2 executors final	2024-01-10 23:25:40	2024-01-10 23:26:46	1.1 min
-------	--------------------------------	---	---------------------	---------------------	---------

Σχολιασμός χρόνων εκτέλεσης

Γενικά όσο αυξάνονται οι executors για ένα ερώτημα, αυξάνονται και οι εργασίες που μπορούν να εκτελεστούν παράλληλα. Ωστόσο, ενδέχεται να υπάρξουν χρονικές καθυστερήσεις στην ανταλλαγή δεδομένων μεταξύ των executors. Επίσης, με περισσότερους executors, αυξάνονται οι υπολογισμοί που

πρέπει να κάνει ο driver, καθώς αυτός είναι που αποφασίζει με ποιον τρόπο θα κατανεμηθούν οι εργασίες που πρέπει να γίνουν στους executors, ζητάει πόρους (CPU, μνήμη) από τον cluster manager για τους executors και γενικά επικοινωνεί μαζί τους. Συνεπώς, μπορεί μερικές φορές η αύξηση του αριθμού από executors να μην οδηγήσει απαραίτητα σε βελτίωση του χρόνου εκτέλεσης.

Αυτό φαίνεται να συμβαίνει στην περίπτωση του παρόντος ερωτήματος. Παρατηρούμε ότι οι χρόνοι εκτέλεσης είναι πολύ κοντινοί, συγκεκριμένα η εκτέλεση με 3 executors είναι γρηγορότερες κατά 0.1 sec από αυτές με 1 και 4. Αυτό δηλώνει ότι τελικά η καλύτερη λύση είναι κάπου ενδιάμεσα καθώς κερδίζεται χρόνος τόσο από την αύξηση των executors (από 2 σε 3) και κατά συνέπεια του μεγαλύτερου παραλληλισμού αλλά και από τη μείωσή τους (από 4 σε 3) λόγω μικρότερου απαιτούμενου συντονισμού.

Ερώτημα 6

Παραθέτουμε τον κώδικα για την εκτέλεση του ερωτήματος χρησιμοποιώντας το SQL API:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, to_date, udf
from pyspark.sql.types import DoubleType, IntegerType
from math import radians, cos, sin, asin, sqrt

def get_distance(longit_a, latit_a, longit_b, latit_b):
    longit_a, latit_a, longit_b, latit_b = map(radians, [longit_a, latit_a, longit_b, latit_b])
    dist_longit = longit_b - longit_a
    dist_latit = latit_b - latit_a

    area = sin(dist_latit/2)**2 + cos(latit_a) * cos(latit_b) * sin(dist_longit/2)**2

    central_angle = 2 * asin(sqrt(area))
    radius = 6371

    distance = central_angle * radius
    return abs(round(distance, 4))

spark = SparkSession.builder.appName("SQL query 4 execution
final").config("spark.executor.instances", "4").getOrCreate()
get_distance_udf = udf(lambda lat1, lon1, lat2, lon2: get_distance(lat1, lon1, lat2, lon2))
spark.udf.register("get_distance", get_distance_udf)

crimes_df1 = spark.read.csv("Crime_Data_from_2010_to_2019.csv", header=True, inferSchema=True)
crimes_df2 = spark.read.csv("Crime_Data_from_2020_to_Present.csv", header=True,
inferSchema=True)
crimes_df = crimes_df1.union(crimes_df2)

crimes_df = crimes_df.withColumn("Date Rptd", to_date(col("Date Rptd"), "MM/dd/yyyy hh:mm:ss
a"))
crimes_df = crimes_df.withColumn("DATE OCC", to_date(col("DATE OCC"), "MM/dd/yyyy hh:mm:ss a"))
crimes_df = crimes_df.withColumn("Vict Age", col("Vict Age").cast(IntegerType()))
crimes_df = crimes_df.withColumn("LAT", col("LAT").cast(DoubleType()))
crimes_df = crimes_df.withColumn("LON", col("LON").cast(DoubleType()))

police_departments_df = spark.read.csv("LAPD_Police_Stations.csv", header=True,
inferSchema=True)
police_departments_df = police_departments_df.withColumn("X", col("X").cast(DoubleType()))
police_departments_df = police_departments_df.withColumn("Y", col("Y").cast(DoubleType()))

crimes_df.createOrReplaceTempView("crimes")
police_departments_df.createOrReplaceTempView("departments")

##### AVERAGE DISTANCE FROM DEPARTMENT THAT UNDERTOOK THE INVESTIGATION FOR
FIREARM CRIMES PER YEAR #####

firearm_crimes_query = """SELECT * FROM crimes
WHERE (lat <> 0 or lon <> 0) and `Weapon Used Cd` LIKE '1__'"""

firearm_crimes = spark.sql(firearm_crimes_query)
firearm_crimes.createOrReplaceTempView("firearm_crimes")

firearm_crimes_distance_query = """SELECT YEAR(`DATE OCC`) AS year, CAST(get_distance(c.`LAT`,
c.`LON`, d.`Y`, d.`X`) as DOUBLE) AS distance
```

```

FROM firearm_crimes c
JOIN departments d ON c.`AREA` = d.`PREC`""

firearm_crimes_distance = spark.sql(firearm_crimes_distance_query)
firearm_crimes_distance.createOrReplaceTempView("firearm_crimes_distance")

firearm_crimes_distance_query = ""SELECT year, AVG(`distance`) as average_distance, COUNT(*) as
`#`
FROM firearm_crimes_distance
GROUP BY year
ORDER BY year ASC""

firearm_crimes_distance = spark.sql(firearm_crimes_distance_query)

##### AVERAGE DISTANCE FROM DEPARTMENT THAT UNDERTOOK THE INVESTIGATION FOR
WEAPON CRIMES PER DEPARTMENT #####

weapon_crimes_query = ""SELECT * FROM crimes
WHERE (lat <> 0 or lon <> 0) and `Weapon Used Cd` IS NOT NULL""

weapon_crimes = spark.sql(weapon_crimes_query)
weapon_crimes.createOrReplaceTempView("weapon_crimes")

weapon_crimes_distance_query = ""SELECT `DIVISION` as division, get_distance(c.`LAT`, c.`LON`,
d.`Y`, d.`X`) AS distance
FROM weapon_crimes c
JOIN departments d ON c.`AREA` = d.`PREC`""

weapon_crimes_distance = spark.sql(weapon_crimes_distance_query)
weapon_crimes_distance.createOrReplaceTempView("weapon_crimes_distance")

weapon_crimes_distance_query = ""SELECT division, AVG(`distance`) as average_distance, COUNT(*)
as `#`
FROM weapon_crimes_distance
GROUP BY division
ORDER BY `#` DESC""

weapon_crimes_distance = spark.sql(weapon_crimes_distance_query)

##### AVERAGE DISTANCE FROM CLOSEST DEPARTMENT FOR FIREARM CRIMES PER YEAR
#####

firearm_crimes_min_distance_query = ""WITH ranked_distances as (
SELECT `DR_NO` as crime,
YEAR(`DATE OCC`) as year,
CAST(get_distance(c.`LAT`, c.`LON`, d.`Y`, d.`X`) as
DOUBLE) AS distance,
ROW_NUMBER() OVER (PARTITION BY `DR_NO` ORDER BY
CAST(get_distance(c.`LAT`, c.`LON`, d.`Y`, d.`X`) as DOUBLE) ASC) as distance_rank
FROM firearm_crimes c
JOIN departments d)
SELECT crime, year, distance
FROM ranked_distances
WHERE distance_rank = 1""

firearm_crimes_min_distance = spark.sql(firearm_crimes_min_distance_query)
firearm_crimes_min_distance.createOrReplaceTempView("firearm_crimes_min_distance")

firearm_crimes_avg_distance_query = ""SELECT year, AVG(`distance`) as average_distance,
COUNT(*) as `#`
FROM firearm_crimes_min_distance
GROUP BY year
ORDER BY year ASC""

firearm_crimes_avg_distance = spark.sql(firearm_crimes_avg_distance_query)

##### AVERAGE DISTANCE FROM CLOSEST DEPARTMENT FOR WEAPON CRIMES PER DEPARTMENT
#####

weapon_crimes_min_distance_query = ""WITH ranked_distances as (
SELECT `DR_NO` as crime,
`DIVISION` as division,
CAST(get_distance(c.`LAT`, c.`LON`, d.`Y`, d.`X`) as
DOUBLE) AS distance,

```

```

            ROW_NUMBER() OVER (PARTITION BY `DR_NO` ORDER BY
CAST(get_distance(c.`LAT`, c.`LON`, d.`Y`, d.`X`) as DOUBLE) ASC) as distance_rank
            FROM weapon_crimes c
            JOIN departments d)
        SELECT crime, division, distance
        FROM ranked_distances
        WHERE distance_rank = 1"""

```

```

weapon_crimes_min_distance = spark.sql(weapon_crimes_min_distance_query)
weapon_crimes_min_distance.createOrReplaceTempView("weapon_crimes_min_distance")

```

```

weapon_crimes_avg_distance_query = """SELECT division, avg(distance) as average_distance,
COUNT(*) as `#`
        FROM weapon_crimes_min_distance
        GROUP BY division
        ORDER BY `#` DESC"""

```

```

weapon_crimes_avg_distance = spark.sql(weapon_crimes_avg_distance_query)

```

```

##### SHOW RESULTS #####

```

```

print("Distance calculated from the police department that undertook the investigation:\n")
firearm_crimes_distance.show(firearm_crimes_distance.count(), truncate=False)
weapon_crimes_distance.show(weapon_crimes_distance.count(), truncate=False)

```

```

print("Distance calculated from the closest police department:\n")
firearm_crimes_avg_distance.show(firearm_crimes_avg_distance.count(), truncate=False)
weapon_crimes_avg_distance.show(weapon_crimes_avg_distance.count(), truncate=False)

```

Η συνάρτηση `get_distance()` για τον υπολογισμό της γεωγραφικής απόστασης δύο σημείων, βάσει του γεωγραφικού μήκους και πλάτους τους, χρησιμοποιεί την haversine formula.
Παραθέτουμε το output του ερωτήματος:

Distance calculated from the police department that undertook the investigation:

year	average_distance	#
2010	2.4247349975645287	8212
2011	2.427409471792032	7232
2012	2.488719519289648	6532
2013	2.5018637375813504	5838
2014	2.4411897326709204	4227
2015	2.3917836758834774	6763
2016	2.3632303086419735	8100
2017	2.3941536090418705	7786
2018	2.3796064751112875	7413
2019	2.3877205919483844	7129
2020	2.343789407328843	8487
2021	2.326585719911779	12696
2022	2.2616833815461366	10025
2023	2.211905399839828	8741

division	average_distance	#
77TH STREET	2.6085129823670288	94482
SOUTHEAST	1.929796472086767	77723
SOUTHWEST	2.6875844734374197	72508
CENTRAL	0.9232786509744385	63216
NEWTON	1.5730249722022316	61156
RAMPART	1.309771455297707	55590
OLYMPIC	1.3714763875466698	52773
HOLLYWOOD	1.4761946101435202	50929
PACIFIC	3.5000874878539285	44253
MISSION	3.0641245906163745	43480

NORTH HOLLYWOOD	1.9799575848256274	42440
HOLLENBECK	1.9139651735641234	41368
HARBOR	2.506908753504846	40658
WILSHIRE	1.9867997242768844	37719
NORTHEAST	4.014050938417195	37137
VAN NUYS	1.6530086292446273	36075
TOPANGA	2.4245588028168985	34648
FOOTHILL	3.998242012585891	34642
WEST VALLEY	3.3527219585185177	33750
DEVONSHIRE	3.9620965112509046	30842
WEST LOS ANGELES	3.8141525862068946	26912
+-----+-----+-----+		

Distance calculated from the closest police department:

year	average_distance	#	
+-----+-----+-----+			
2010	1.8791257428153918	8212	
2011	1.891918376659294	7232	
2012	1.9531837415799163	6532	
2013	1.9122739465570393	5838	
2014	1.7730363851431292	4227	
2015	1.8657328700280944	6763	
2016	1.8610920000000017	8100	
2017	1.8588549704597992	7786	
2018	1.8649509375421607	7413	
2019	1.8788348997054265	7129	
2020	1.8574277365382417	8487	
2021	1.8332026680348925	9745	
2022	1.7858388229426438	10025	
2023	1.759313041986042	8741	
+-----+-----+-----+			

division	average_distance	#	
+-----+-----+-----+			
SOUTHWEST	1.726212021427455	82511	
OLYMPIC	1.4585148341177043	77133	
77TH STREET	1.2061523648648593	74888	
WILSHIRE	2.150778470731135	70792	
SOUTHEAST	1.8598909343430048	60481	
CENTRAL	0.6903291462910852	58076	
RAMPART	0.960673237547218	56654	
HOLLENBECK	2.437910282167657	55747	
HOLLYWOOD	1.5463916420367267	53458	
VAN NUYS	2.033527408095908	52718	
NORTH HOLLYWOOD	2.1621879561559267	45160	
NEWTON	1.456476531463029	40778	
HARBOR	2.4059155485258574	39515	
TOPANGA	2.2225614986648936	35952	
WEST VALLEY	2.3015525370281846	31395	
PACIFIC	2.693196166470172	31381	
WEST LOS ANGELES	2.5395088119064253	31378	
MISSION	2.481874293374688	30957	
FOOTHILL	2.0954996544097573	28936	
DEVONSHIRE	2.4835230117300338	16283	
NORTHEAST	3.2320515700062624	14363	
+-----+-----+-----+			

Ερώτημα 7

Query 3

Στην υλοποίηση του query 3 γίνονται συνολικά 4 joins. Αρχικά, θα δούμε ποια στρατηγική είναι καλύτερη για τα joins με βάση τη θεωρία, βασιζόμενοι στα μεγέθη των dataframes που γίνονται κάθε φορά join καθώς και στο αν είναι ταξινομημένα ή όχι, και έπειτα θα επαληθεύσουμε τα θεωρητικά αποτελέσματα δοκιμάζοντας διαφορετικές στρατηγικές join στην υλοποίηση. Σημειώνουμε ότι για δηλώσουμε στο Spark τον τρόπο με τον οποίο θέλουμε να γίνει το join χρησιμοποιούμε την εντολή hint, δίνοντας ως παράμετρο τον τρόπο. Για παράδειγμα

```
income_with_crimes_df = income_df.join(rev_geocoding_zips_df.hint("merge"),
income_df["Zip Code"] == rev_geocoding_zips_df["ZIPcode"], "inner")
```

Για να δούμε το πλάνο εκτέλεσης χρησιμοποιούμε την εντολή explain στο dataframe, πχ

```
income_with_crimes_df.explain()
```

Πρώτο join:

```
income_with_crimes_df = income_df.join(rev_geocoding_zips_df, income_df["Zip Code"]
== rev_geocoding_zips_df["ZIPcode"], "inner")
```

Βρίσκουμε τα μεγέθη των δύο dataframes, εφαρμόζοντας την μέθοδο count() σε αυτά (για λόγους απλότητας παραλείπονται οι γραμμές που μετράνε τα μεγέθη, καθώς επιτέλεσαν απλώς βοηθητικό ρόλο). Βρίσκουμε ότι το income_df περιέχει 284 γραμμές, ενώ το rev_geocoding_zips_df περιέχει 181. Βλέπουμε ότι τα δύο dataframes είναι μικρά και το μέγεθός τους δεν διαφέρει και πολύ, συνεπώς δεν παίζει μεγάλο ρόλο ποια στρατηγική θα χρησιμοποιήσουμε.

Δεύτερο join:

```
joined1 = victims_alias_crimes_df.alias("victims_alias_crimes").join(
    rev_geocoding_df.alias("geocoding"),
    (col("victims_alias_crimes.LAT") == col("geocoding.LAT")) &
    (col("victims_alias_crimes.LON") == col("geocoding.LON")),
    "inner"
)
```

Βρίσκουμε ότι το victims_alias_crimes_df περιέχει 190439 εγγραφές ενώ το rev_geocoding_df περιέχει 37781 εγγραφές.

- **Broadcast join:** Το δεύτερο dataframe είναι μικρό και μπορεί να χωρέσει στη μνήμη κάθε executor συνεπώς θα μπορούσε να γίνει broadcasted. Οπότε δοκιμάζουμε με Broadcast join. Παραθέτουμε το συνολικό χρόνο εκτέλεσης καθώς και το πλάνο εκτέλεσης:

3.5.0	application_1704919127361_0009	Dataframe query 3 join2 broadcast Final	2024-01-11 00:07:02	2024-01-11 00:08:07	1.1 min
-------	--------------------------------	---	---------------------	---------------------	---------

```
== Physical Plan ==
AdaptiveSparkPlan (23)
+- == Final Plan ==
   * HashAggregate (13)
   +- ShuffleQueryStage (12), Statistics(sizeInBytes=80.0 B, rowCount=5)
      +- Exchange (11)
         +- * HashAggregate (10)
            +- * Project (9)
               +- * BroadcastHashJoin Inner BuildRight (8)
                  :- * Project (3)
                  :  +- * Filter (2)
                  :    +- Scan csv (1)
               +- BroadcastQueryStage (7), Statistics(sizeInBytes=6.0 MiB, rowCount=3.78E+4)
                  +- BroadcastExchange (6)
```



```
+-- * Filter (5)
+-- Scan csv (4)
```

- **Merge join:** Για να εκτελεστεί το Merge join καλό θα ήταν και τα δύο dataframes να είναι μεγάλα με πάνω από 1 εκατομμύριο εγγραφές το καθένα, ωστόσο επειδή κάνουμε join σε κλειδιά που είναι αριθμοί, και μπορούν σχετικά εύκολα να ταξινομηθούν, δοκιμάζουμε με Merge join. Παραθέτουμε το συνολικό χρόνο εκτέλεσης καθώς και το πλάνο εκτέλεσης:

3.5.0	application_1704919127361_0010	Dataframe query 3 join2 merge Final	2024-01-11 00:09:33	2024-01-11 00:10:38	1.1 min
-------	--------------------------------	-------------------------------------	---------------------	---------------------	---------

```
== Physical Plan ==
AdaptiveSparkPlan (32)
+- == Final Plan ==
  * HashAggregate (19)
  +- ShuffleQueryStage (18), Statistics(sizeInBytes=16.0 B, rowCount=1)
    +- Exchange (17)
      +- * HashAggregate (16)
        +- * Project (15)
          +- * SortMergeJoin Inner (14)
            :- * Sort (7)
              : +- AQEShuffleRead (6)
                : +- ShuffleQueryStage (5), Statistics(sizeInBytes=4.4 MiB,
rowCount=1.90E+5)
                : +- Exchange (4)
                  : +- * Project (3)
                    : +- * Filter (2)
                      : +- Scan csv (1)
            +- * Sort (13)
              +- AQEShuffleRead (12)
                +- ShuffleQueryStage (11), Statistics(sizeInBytes=885.5 KiB,
rowCount=3.78E+4)
                  +- Exchange (10)
                    +- * Filter (9)
                      +- Scan csv (8)
```

- **Shuffle Hash join:** Το Shuffle Hash Join, θεωρητικά είναι περισσότερο ακριβό σε χρόνο καθώς χρειάζεται αρκετή μεταφορά δεδομένων, ώστε εγγραφές με το ίδιο κλειδί να βρίσκονται στον ίδιο executor, ενώ επίσης απαιτεί και τη δημιουργία και διατήρηση ενός hash table. Ωστόσο, δοκιμάζουμε και εκτέλεση με Shuffle Hash Join. Παραθέτουμε το συνολικό χρόνο εκτέλεσης καθώς και το πλάνο εκτέλεσης:

3.5.0	application_1704919127361_0011	Dataframe query 3 join2 shuffle hash Final	2024-01-11 00:11:29	2024-01-11 00:12:33	1.1 min
-------	--------------------------------	--	---------------------	---------------------	---------

```
== Physical Plan ==
AdaptiveSparkPlan (28)
+- == Final Plan ==
  * HashAggregate (17)
  +- ShuffleQueryStage (16), Statistics(sizeInBytes=16.0 B, rowCount=1)
    +- Exchange (15)
      +- * HashAggregate (14)
        +- * Project (13)
          +- * ShuffledHashJoin Inner BuildRight (12)
            :- AQEShuffleRead (6)
              : +- ShuffleQueryStage (5), Statistics(sizeInBytes=4.4 MiB, rowCount=1.90E+5)
                : +- Exchange (4)
                  : +- * Project (3)
                    : +- * Filter (2)
                      : +- Scan csv (1)
            +- AQEShuffleRead (11)
              +- ShuffleQueryStage (10), Statistics(sizeInBytes=885.5 KiB,
rowCount=3.78E+4)
                +- Exchange (9)
                  +- * Filter (8)
                    +- Scan csv (7)
```

- Shuffle Replicate Nested Loop Join:** Το Shuffle Replicate Nested Loop Join είναι θεωρητικά η πιο αργή τεχνική, καθώς αποτελεί υπολογισμό του καρτεσιανού γινομένου των δύο dataframes. Εκτελώντας το query με αυτή τη στρατηγική, ο χρόνος εκτέλεσης ξεπέρασε τα 5 λεπτά δίχως καν να ολοκληρωθεί η εκτέλεση.

Παρατηρούμε ότι και οι 3 υλοποιήσεις (Broadcast, Merge, Shuffle Hash) παρέχουν ίδιους χρόνους εκτέλεσης. Βλέποντας και τα πλάνα εκτέλεσης, διαπιστώνουμε ότι τα βήματα που ακολουθεί κάθε στρατηγική επαληθεύονται στην πράξη.

Τρίτο και Τέταρτο join:

```
joined2 = joined1.join(highest_3_zip_codes, highest_3_zip_codes["Zip Code"] ==
joined1["ZIPcode"], "inner")
```

```
joined3 = joined1.join(lowest_3_zip_codes, lowest_3_zip_codes["Zip Code"] ==
joined1["ZIPcode"], "inner")
```

Επειδή τα δύο παραπάνω joins είναι παρόμοια (στην πρώτη περίπτωση γίνεται join ένα dataframe με 3 zip codes, ενώ στη δεύτερη γίνεται join το ίδιο dataframe με 3 άλλα zip codes) θα αναλύσουμε τις διάφορες στρατηγικές join μόνο για το ένα join. Βρίσκουμε ότι το `joined1` περιέχει 1190226 ενώ το `highest_3_zip_codes` μόνο 3.

- **Broadcast join:** Το δεύτερο dataframe περιέχει μόνο 3 εγγραφές οπότε ενδείκνυται να χρησιμοποιήσουμε Broadcast join. Παραθέτουμε το συνολικό χρόνο εκτέλεσης καθώς και το πλάνο εκτέλεσης:

3.5.0	application_1704919127361_0012	Dataframe query 3 join3 broadcast Final	2024-01-11 00:14:33	2024-01-11 00:15:44	1.2 min
-------	--------------------------------	---	---------------------	---------------------	---------

[illegible]

```

+- * HashAggregate (19)
+- * Project (18)
+- * Filter (17)
+- Scan csv (16)

```

- **Merge join:** Η εκτέλεση με Merge join ενδέχεται να αργήσει λίγο παραπάνω λόγω της τεράστιας διαφοράς σε μέγεθος των dataframes. Επίσης, σίγουρα θα χαθεί χρόνος στην ταξινόμηση των εγγραφών του μεγάλου dataframe. Παραθέτουμε το συνολικό χρόνο εκτέλεσης:

3.5.0	application_1704919127361_0013	Dataframe query 3 join3 merge Final	2024-01-11 00:16:34	2024-01-11 00:17:46	1.2 min
-------	--------------------------------	-------------------------------------	---------------------	---------------------	---------

```

== Physical Plan ==
AdaptiveSparkPlan (71)
+- == Final Plan ==
    TakeOrderedAndProject (42)
      +- * HashAggregate (41)
        +- AQEShuffleRead (40)
          +- ShuffleQueryStage (39), Statistics(sizeInBytes=248.0 B, rowCount=7)
            +- Exchange (38)
              +- * HashAggregate (37)
                +- * Project (36)
                  +- * SortMergeJoin Inner (35)
                    :- * Sort (14)
                    : +- AQEShuffleRead (13)
                    :   +- ShuffleQueryStage (12), Statistics(sizeInBytes=6.9 MiB,
rowCount=1.89E+5)
                    :
                    :   +- Exchange (11)
                    :     +- * Project (10)
                    :       +- * BroadcastHashJoin Inner BuildRight (9)
                    :         :- * Project (3)
                    :         : +- * Filter (2)
                    :         :   +- Scan csv (1)
                    :         +- BroadcastQueryStage (8), Statistics(sizeInBytes=6.0
MiB, rowCount=3.74E+4)
                    :
                    :   +- BroadcastExchange (7)
                    :     +- * Project (6)
                    :       +- * Filter (5)
                    :       +- Scan csv (4)
                  +- * Sort (34)
                    +- AQEShuffleRead (33)
                      +- ShuffleQueryStage (32), Statistics(sizeInBytes=48.0 B,
rowCount=3)
                      +- Exchange (31)
                        +- TakeOrderedAndProject (30)
                          +- * Project (29)
                            +- * BroadcastHashJoin Inner BuildLeft (28)
                              :- BroadcastQueryStage (19),
Statistics(sizeInBytes=1028.0 KiB, rowCount=107)
                              : +- BroadcastExchange (18)
                              :   +- * Project (17)
                              :     +- * Filter (16)
                              :     +- Scan csv (15)
                            +- * HashAggregate (27)
                              +- AQEShuffleRead (26)
                                +- ShuffleQueryStage (25),
Statistics(sizeInBytes=2.8 KiB, rowCount=180)
                                +- Exchange (24)
                                  +- * HashAggregate (23)
                                  +- * Project (22)
                                  +- * Filter (21)
                                  +- Scan csv (20)

```

- **Shuffle Hash join:** Παραθέτουμε το συνολικό χρόνο εκτέλεσης καθώς και το πλάνο εκτέλεσης:

3.5.0	application_1704919127361_0014	Dataframe query 3 join3 shuffle hash Final	2024-01-11 00:18:23	2024-01-11 00:19:35	1.2 min
-------	--------------------------------	--	---------------------	---------------------	---------

```

== Physical Plan ==
AdaptiveSparkPlan (67)
+- == Final Plan ==
    TakeOrderedAndProject (40)

```

```

+- * HashAggregate (39)
+- AQEShuffleRead (38)
+- ShuffleQueryStage (37), Statistics(sizeInBytes=248.0 B, rowCount=7)
+- Exchange (36)
+- * HashAggregate (35)
+- * Project (34)
+- * ShuffledHashJoin Inner BuildRight (33)
:- AQEShuffleRead (13)
: +- ShuffleQueryStage (12), Statistics(sizeInBytes=6.9 MiB,
rowCount=1.89E+5)
: +- Exchange (11)
: +- * Project (10)
: +- * BroadcastHashJoin Inner BuildRight (9)
: :- * Project (3)
: : +- * Filter (2)
: : +- Scan csv (1)
: +- BroadcastQueryStage (8), Statistics(sizeInBytes=6.0
MiB, rowCount=3.74E+4)
: +- BroadcastExchange (7)
: +- * Project (6)
: +- * Filter (5)
: +- Scan csv (4)
+- AQEShuffleRead (32)
+- ShuffleQueryStage (31), Statistics(sizeInBytes=48.0 B, rowCount=3)
+- Exchange (30)
+- TakeOrderedAndProject (29)
+- * Project (28)
+- * BroadcastHashJoin Inner BuildLeft (27)
:- BroadcastQueryStage (18),
Statistics(sizeInBytes=1028.0 KiB, rowCount=107)
: +- BroadcastExchange (17)
: +- * Project (16)
: +- * Filter (15)
: +- Scan csv (14)
+- * HashAggregate (26)
+- AQEShuffleRead (25)
+- ShuffleQueryStage (24),
Statistics(sizeInBytes=2.8 KiB, rowCount=180)
+- Exchange (23)
+- * HashAggregate (22)
+- * Project (21)
+- * Filter (20)
+- Scan csv (19)

```

- **Shuffle Replicate Nested Loop Join:** Επειδή το δεύτερο dataframe περιέχει μόλις 3 εγγραφές, το καρτεσιανό γινόμενο θα περιέχει εγγραφές τριπλάσιες από το μέγεθος του πρώτου dataframe. Όμως, ασυμπτωτικά ο χρόνος για το join σε αυτή την περίπτωση θα είναι $O(n)$ σε αντίθεση με $O(n^2)$ που είναι στη γενική περίπτωση. Σε αυτή τη στρατηγική join εξοικονομείται επίσης χρόνος, καθώς δεν χρειάζεται η δημιουργία και διατήρηση κάποιου hash table ούτε ταξινόμηση των εγγραφών. Παραθέτουμε το συνολικό χρόνο εκτέλεσης καθώς και το πλάνο εκτέλεσης:

3.5.0	application_1704919127361_0015	Dataframe query 3 join3 shuffle replicate nl Final	2024-01-11 00:20:06	2024-01-11 00:21:18	1.2 min
-------	--------------------------------	--	---------------------	---------------------	---------

```

== Physical Plan ==
AdaptiveSparkPlan (58)
+- == Final Plan ==
TakeOrderedAndProject (33)
+- * HashAggregate (32)
+- AQEShuffleRead (31)
+- ShuffleQueryStage (30), Statistics(sizeInBytes=576.0 B, rowCount=16)
+- Exchange (29)
+- * HashAggregate (28)
+- * Project (27)
+- CartesianProduct Inner (26)
:- * Project (10)
: +- * BroadcastHashJoin Inner BuildRight (9)
: :- * Project (3)
: : +- * Filter (2)
: : +- Scan csv (1)

```

```

:      +- BroadcastQueryStage (8), Statistics(sizeInBytes=6.0 MiB,
rowCount=3.74E+4)
:      +- BroadcastExchange (7)
:      +- * Project (6)
:      +- * Filter (5)
:      +- Scan csv (4)
+- TakeOrderedAndProject (25)
  +- * Project (24)
    +- * BroadcastHashJoin Inner BuildLeft (23)
      :- BroadcastQueryStage (15), Statistics(sizeInBytes=1028.0 KiB,
rowCount=107)
        : +- BroadcastExchange (14)
        : +- * Project (13)
        : +- * Filter (12)
        : +- Scan csv (11)
      +- * HashAggregate (22)
        +- ShuffleQueryStage (21), Statistics(sizeInBytes=2.8 KiB,
rowCount=180)
          +- Exchange (20)
            +- * HashAggregate (19)
              +- * Project (18)
                +- * Filter (17)
                  +- Scan csv (16)

```

Παρατηρούμε ότι και οι 4 στρατηγικές οδηγούν σε ίδιο χρόνο εκτέλεσης. Όπως εξηγήσαμε και παραπάνω κάθε στρατηγική έχει τα πλεονεκτήματά της, στο συγκεκριμένο μάλιστα join επειδή υπάρχει τεράστια διαφορά στο μέγεθος των δύο dataframes, δεν παίζει τελικά μεγάλο ρόλο ποια στρατηγική θα ακολουθηθεί. Για παράδειγμα για τη στρατηγική merge join υποθέσαμε ότι ενδέχεται να καθυστερήσει λόγω της διαφοράς μεγέθους στα dataframes και της ταξινόμησης, ωστόσο δεν φαίνεται να έγινε κάτι τέτοιο. Παρατηρώντας μάλιστα τα physical plans που μας δίνει ο history server για την εκτέλεση κάθε join, βλέπουμε ότι κάθε στρατηγική join επαληθεύεται στην πράξη.

Query 4

Στην υλοποίηση του query 4 γίνονται συνολικά 4 joins. Αυτά, ωστόσο, παρουσιάζουν ανά δύο ακριβώς την ίδια συμπεριφορά καθώς συμμετέχουν τα ίδια dataframes (στα join 1+3 τα firearm_crimes και departments και στα join 2+4 τα weapon_crimes και departments) και επομένως η θεωρητική ανάλυση και η πρακτική επιβεβαίωση ή μη των συμπερασμάτων της θα γίνει ενδεικτικά μόνο στα join 1+2. Εδώ σημειώνουμε πως όπου στην εκφώνηση ζητούσε να λάβουμε υπόψη μας όλα τα εγκλήματα που πραγματοποιήθηκαν με πυροβόλο όπλο, έγινε το filtering με βάση τον κωδικό 1xx, όπως υποδείχθηκε στα tips, ενώ στο δεύτερο σκέλος κάθε ερωτήματος, που ζητούσε όλα τα εγκλήματα που έγιναν με χρήση κάποιου όπλου, θεωρήθηκε πως ο κωδικός για το όπλο θα έπρεπε να είναι not null.

Για δηλώσουμε στο Spark τον τρόπο με τον οποίο θέλουμε να γίνει το join στην SQL χρησιμοποιούμε τα join hints, όπως φαίνεται στο παρακάτω παράδειγμα:

```

firearm_crimes_distance_query = """SELECT /*+ BROADCAST(c, d) */ YEAR(`DATE OCC`)
AS year, CAST(get_distance(c.`LAT`, c.`LON`, d.`Y`, d.`X`) as DOUBLE) AS distance
FROM firearm_crimes c
JOIN departments d ON c.`AREA` = d.`PREC`"""

```

Πρώτο join:

```

firearm_crimes_distance_query = """SELECT YEAR(`DATE OCC`) AS year,
CAST(get_distance(c.`LAT`, c.`LON`, d.`Y`, d.`X`) as DOUBLE) AS distance
FROM firearm_crimes c
JOIN departments d ON c.`AREA` = d.`PREC`"""

```

Με την συνάρτηση count() προκύπτει πως το firearm_crimes dataframe περιέχει 109181 εγγραφές, ενώ το departments μόλις 21.

- **Broadcast join:** Η διαφορά στο μέγεθος των δύο dataframes είναι εμφανής και ειδικά ο πολύ μικρός αριθμός εγγραφών στο δεύτερο dataframe ευνοεί την αποθήκευσή του σε όλους τους executors,

όπως ορίζει η διαδικασία του broadcast join. Παραθέτουμε το συνολικό χρόνο εκτέλεσης καθώς και το πλάνο εκτέλεσης:

3.5.0	application_1704999680291_0008	SQL query 4 join1 broadcast	2024-01-11 22:04:50	2024-01-11 22:05:54	1.1 min	user	2024-01-11 22:05:55	Download
-------	--	-----------------------------	------------------------	------------------------	---------	------	------------------------	--------------------------

[illegible]

- Merge join:** Καθώς κανένα από τα δύο dataframes δεν είναι ήδη ταξινομημένο, η μέθοδος αυτή δεν είναι η προτιμότερη. Ωστόσο, αν και θεωρητικά θα είναι πιο αργή λόγω του χρόνου που απαιτείται για την ταξινόμηση, επειδή τα μεγέθη των δύο dataframes δεν είναι τόσο μεγάλα ώστε η καθυστέρηση αυτή να είναι αισθητή, εν τέλει φαίνεται πως ο χρόνος εκτέλεσης δεν επηρεάζεται:

3.5.0	application_1704999680291_0011	SQL query 4 join1 merge	2024-01-11 22:14:41	2024-01-11 22:15:44	1.1 min	user	2024-01-11 22:15:44	Download
-------	--	-------------------------	------------------------	------------------------	---------	------	------------------------	--------------------------

[illegible]

```

:           : +- * Filter (2)
:           :   +- Scan csv (1)
:           +- * Project (6)
:           +- * Filter (5)
:           +- Scan csv (4)
+- * Sort (17)
    +- AQEShuffleRead (16)
      +- ShuffleQueryStage (15),
Statistics(sizeInBytes=336.0 B, rowCount=21)
    +- Exchange (14)
      +- * Filter (13)
        +- Scan csv (12)

```

- **Shuffle Hash join:** Αν και η μέθοδος αυτή ενδείκνυται για μη ταξινομημένα dataframes, λόγω της μεγάλης διαφοράς μεγέθους τους στο συγκεκριμένο join, ίσως αυτό να αντισταθμίζεται. Παραθέτουμε το συνολικό χρόνο εκτέλεσης καθώς και το πλάνο εκτέλεσης:

3.5.0	application_1704999680291_0012	SQL query 4 join1 shuffle hash	2024-01-11 22:18:20	2024-01-11 22:19:24	1.1 min	user	2024-01-11 22:19:25	Download
-------	--	--------------------------------	------------------------	------------------------	---------	------	------------------------	--------------------------

```

== Physical Plan ==
AdaptiveSparkPlan (43)
+- == Final Plan ==
   * HashAggregate (26)
   +- ShuffleQueryStage (25), Statistics(sizeInBytes=16.0 B, rowCount=1)
      +- Exchange (24)
         +- * HashAggregate (23)
            +- * HashAggregate (22)
               +- AQEShuffleRead (21)
                  +- ShuffleQueryStage (20), Statistics(sizeInBytes=224.0 B,
rowCount=14)
                     +- Exchange (19)
                        +- * HashAggregate (18)
                           +- * Project (17)
                              +- * ShuffledHashJoin Inner BuildRight (16)
                                 :- AQEShuffleRead (10)
                                    : +- ShuffleQueryStage (9),
Statistics(sizeInBytes=2.5 MiB, rowCount=1.09E+5)
                                     :   +- Exchange (8)
                                     :   +- Union (7)
                                     :   :- * Project (3)
                                     :   : +- * Filter (2)
                                     :   :   +- Scan csv (1)
                                     :   +- * Project (6)
                                     :   +- * Filter (5)
                                     :   +- Scan csv (4)
                                 +- AQEShuffleRead (15)
                                   +- ShuffleQueryStage (14),
Statistics(sizeInBytes=336.0 B, rowCount=21)
                                     +- Exchange (13)
                                       +- * Filter (12)
                                         +- Scan csv (11)

```

- **Shuffle Replicate Nested Loop Join:** Λόγω του πολύ μικρού μεγέθους του δεύτερου dataframe (μόλις 21 εγγραφές), η μέθοδος αυτή περιμένουμε να έχει καλή απόδοση, διότι το καρτεσιανό γινόμενο των δύο dataframes έχει επιτρεπτό μέγεθος. Παραθέτουμε το συνολικό χρόνο εκτέλεσης καθώς και το πλάνο εκτέλεσης:

3.5.0	application_1704999680291_0013	SQL query 4 join1 shuffle replicate nl	2024-01-11 22:21:11	2024-01-11 22:22:16	1.1 min	user	2024-01-11 22:22:17	Download
-------	--	---	------------------------	------------------------	---------	------	------------------------	--------------------------


```

: : +- * Filter (2)
: : +- Scan csv (1)
: +- * Project (6)
: +- * Filter (5)
: +- Scan csv (4)
+- BroadcastQueryStage (11),
Statistics(sizeInBytes=1024.2 KiB, rowCount=21)
+- BroadcastExchange (10)
+- * Filter (9)
+- Scan csv (8)

```

- **Merge join:** Δεδομένου του μεγαλύτερου μεγέθους του πρώτου dataframe, η διαδικασία της ταξινόμησης των dataframes αναμένεται να είναι ακόμη πιο χρονοβόρα, επηρεάζοντας και τον χρόνο εκτέλεσης ανάλογα. Παραθέτουμε το συνολικό χρόνο εκτέλεσης καθώς και το πλάνο εκτέλεσης:

3.5.0	application_1704999680291_0020	SQL query 4 join2 merge	2024-01-11 22:35:53	2024-01-11 22:37:07	1.2 min	user	2024-01-11 22:37:07	Download
-------	--------------------------------	-------------------------	------------------------	------------------------	---------	------	------------------------	--------------------------

```

== Physical Plan ==
AdaptiveSparkPlan (47)
+- == Final Plan ==
  * HashAggregate (28)
  +- ShuffleQueryStage (27), Statistics(sizeInBytes=16.0 B, rowCount=1)
    +- Exchange (26)
      +- * HashAggregate (25)
        +- * HashAggregate (24)
          +- AQEShuffleRead (23)
            +- ShuffleQueryStage (22), Statistics(sizeInBytes=584.0 B,
rowCount=21)
              +- Exchange (21)
                +- * HashAggregate (20)
                  +- * Project (19)
                    +- * SortMergeJoin Inner (18)
                      :- * Sort (11)
                        : +- AQEShuffleRead (10)
                          : +- ShuffleQueryStage (9),
Statistics(sizeInBytes=15.4 MiB, rowCount=1.01E+6)
                            +- Exchange (8)
                              +- Union (7)
                                :- * Project (3)
                                  : +- * Filter (2)
                                    : +- Scan csv (1)
                                      +- * Project (6)
                                        +- * Filter (5)
                                          +- Scan csv (4)
                                +- * Sort (17)
                                  +- AQEShuffleRead (16)
                                    +- ShuffleQueryStage (15),
Statistics(sizeInBytes=752.0 B, rowCount=21)
                                      +- Exchange (14)
                                        +- * Filter (13)
                                          +- Scan csv (12)

```

- **Shuffle Hash join:** Και πάλι η μέθοδος αυτή αναμένεται να είναι αποδοτική λόγω του ότι δεν χρειάζεται να ταξινομηθούν τα dataframes και επομένως δεν προστίθεται αυτή η χρονική καθυστέρηση. Παραθέτουμε το συνολικό χρόνο εκτέλεσης καθώς και το πλάνο εκτέλεσης:

3.5.0	application_1704999680291_0021	SQL query 4 join2 shuffle hash	2024-01-11 22:37:17	2024-01-11 22:38:29	1.2 min	user	2024-01-11 22:38:29	Download
-------	--------------------------------	-----------------------------------	------------------------	------------------------	---------	------	------------------------	--------------------------

```

== Physical Plan ==
AdaptiveSparkPlan (43)
+- == Final Plan ==
  * HashAggregate (26)

```

```

+- ShuffleQueryStage (25), Statistics(sizeInBytes=16.0 B, rowCount=1)
+- Exchange (24)
  +- * HashAggregate (23)
    +- * HashAggregate (22)
      +- AQEShuffleRead (21)
        +- ShuffleQueryStage (20), Statistics(sizeInBytes=584.0 B,
rowCount=21)
          +- Exchange (19)
            +- * HashAggregate (18)
              +- * Project (17)
                +- * ShuffledHashJoin Inner BuildRight (16)
                  :- AQEShuffleRead (10)
                    : +- ShuffleQueryStage (9),
Statistics(sizeInBytes=15.4 MiB, rowCount=1.01E+6)
                    : +- Exchange (8)
                    : +- Union (7)
                    : :- * Project (3)
                    : +- * Filter (2)
                    : +- Scan csv (1)
                    : +- * Project (6)
                    : +- * Filter (5)
                    : +- Scan csv (4)
                  +- AQEShuffleRead (15)
                    +- ShuffleQueryStage (14),
Statistics(sizeInBytes=752.0 B, rowCount=21)
                    +- Exchange (13)
                      +- * Filter (12)
                        +- Scan csv (11)

```

- **Shuffle Replicate Nested Loop Join:** Το μικρό μέγεθος του δεύτερου dataframe καθιστά και πάλι δυνατό την χρήση του καρτεσιανού γινομένου για τον υπολογισμό του join. Παραθέτουμε το συνολικό χρόνο εκτέλεσης καθώς και το πλάνο εκτέλεσης:

3.5.0	application_1704999680291_0022	SQL query 4 join2 shuffle replicate nl	2024-01-11 22:38:37	2024-01-11 22:39:47	1.2 min	user	2024-01-11 22:39:47	Download
-------	--------------------------------	---	------------------------	------------------------	---------	------	------------------------	--------------------------

```

== Physical Plan ==
AdaptiveSparkPlan (35)
+- == Final Plan ==
  * HashAggregate (20)
    +- ShuffleQueryStage (19), Statistics(sizeInBytes=16.0 B, rowCount=1)
      +- Exchange (18)
        +- * HashAggregate (17)
          +- * HashAggregate (16)
            +- AQEShuffleRead (15)
              +- ShuffleQueryStage (14), Statistics(sizeInBytes=5.1 KiB,
rowCount=189)
                +- Exchange (13)
                  +- * HashAggregate (12)
                    +- * Project (11)
                      +- CartesianProduct Inner (10)
                        :- Union (7)
                          : :- * Project (3)
                          : : +- * Filter (2)
                          : : +- Scan csv (1)
                          : +- * Project (6)
                          : +- * Filter (5)
                          : +- Scan csv (4)
                        +- * Filter (9)
                          +- Scan csv (8)

```

Παρατηρούμε ότι η αύξηση του αριθμού των εγγραφών του πρώτου dataframe σε σχέση με το πρώτο join έχει ως αποτέλεσμα να γίνει πιο εμφανής η μικρότερη αποδοτικότητα στρατηγικών, όπως το merge join και

το shuffle replicate nested loop join, λόγω της χρονικής καθυστέρησης που προσθέτουν η ταξινόμηση και ο υπολογισμός του καρτεσιανού γινομένου σε μεγαλύτερα dataframes. Η πιο αποδοτική και γρήγορη στρατηγική για το συγκεκριμένο join είναι θεωρητικά το broadcast join, λόγω του πολύ μικρού μεγέθους του δεύτερου dataframe, κάτι που επιβεβαιώνεται και πρακτικά από τον συνολικό χρόνο εκτέλεσης.

Github repository εργασίας

<https://github.com/JimV4/AdvancedDatabases-NTUA>