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

1ο Μέλος Ομάδας: Γεωργία Μπουσμπουκέα (03119059)

2ο Μέλος Ομάδας: Ιωάννα Χουρδάκη (03119208)

ID Ομάδας: 9

Ιανουάριος 2024

Πίνακας Περιεχομένων

Απαντήσεις στα Ζητούμενα

Ζητούμενο 1

Για τις ανάγκες της παρούσας εργασίας, δημιουργήθηκαν δύο κόμβοι (Master – Worker) κάνοντας χρήση πόρων από την υπηρεσία Okeanos – Knossos. Στη συνέχεια, έγινε εγκατάσταση του Apache Spark και στους δύο κόμβους, καθώς και κατάλληλη διαμόρφωση περιβάλλοντος των Hadoop Distributed File System (HDFS) και YARN. Με την ολοκλήρωση του παρόντος βήματος, τα HDFS, YARN και Spark History Server έγιναν διαθέσιμα και προσβάσιμα από τα παρακάτω links:

- HDFS: {our_ip_address}:9870
- YARN: {our_ip_address}:8088
- Spark History Server: {our_ip_address}:18080

Σχόλιο: Για την εκτέλεση των queries σε κάθε βήμα χρησιμοποιούμε την εντολή "**`spark-submit --num-executors <n> --conf spark.log.level=WARN <filename.py>`**" και παρακολουθούμε την εκτέλεσή της από το YARN και τον Sparl History Server.

Ζητούμενο 2

Αρχικά, αναρτήθηκαν τα αρχεία .csv που δίνονται για την ανάλυση (Los Angeles Crime Data, LA Police Stations, Median Household Income by Zip Code, Reverse Geocoding) στο HDFS. Έπειτα, χρησιμοποιώντας το βασικό Σετ Δεδομένων (Los Angeles Crime Data), δημιουργήσαμε ένα DataFrame και μετά την αφαίρεση των διπλότυπων (drop duplicates) προέκυψαν το παρακάτω συνολικό πλήθος γραμμών και οι παρακάτω τύποι δεδομένων:

```
Total number of rows df: 2933545
Schema:
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)
```

Στη συνέχεια, αποθηκεύουμε το DataFrame σε .parquet file για γρήγορη ανάγνωση. Παραθέτουμε παρακάτω τον κώδικα που χρησιμοποιήθηκε.

Κώδικας

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField, StructType, IntegerType,
DoubleType, StringType
from pyspark.sql.functions import col, to_date
import datetime

# Imports for Query1
from pyspark.sql.functions import month, year, count, desc, rank,
                                  countDistinct, regexp_replace
from pyspark.sql.window import Window

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

# Set the legacy time parser policy
spark.conf.set("spark.sql.legacy.timeParserPolicy", "LEGACY")

LA_Crime_Data = StructType([
    StructField("DR_NO", IntegerType()),
    StructField("Date Rptd", StringType()),
    StructField("DATE OCC", StringType()),
    StructField("TIME OCC", IntegerType()),
    StructField("AREA", IntegerType()),
    StructField("AREA NAME", StringType()),
    StructField("Rpt Dist No", IntegerType()),
    StructField("Part 1-2", IntegerType()),
    StructField("Crm Cd", IntegerType()),
    StructField("Crm Cd Desc", StringType()),
    StructField("Mocodes", StringType()),
    StructField("Vict Age", IntegerType()),
    StructField("Vict Sex", StringType()),
    StructField("Vict Descent", StringType()),
    StructField("Premis Cd", IntegerType()),
    StructField("Premis Desc", StringType()),
    StructField("Weapon Used Cd", IntegerType()),
    StructField("Weapon Desc", StringType()),
    StructField("Status", StringType()),
    StructField("Status Desc", StringType()),
```

```
    StructField("Crm Cd 1", IntegerType()),
    StructField("Crm Cd 2", IntegerType()),
    StructField("Crm Cd 3", IntegerType()),
    StructField("Crm Cd 4", IntegerType()),
    StructField("LOCATION", StringType()),
    StructField("Cross Street", StringType()),
    StructField("LAT", DoubleType()),
    StructField("LON", DoubleType())
])

crime_df1 = spark.read.csv("hdfs:///user/user/Crime_Data_from_2010_to_2019.csv",
                           header=True, schema=LA_Crime_Data)
print("Total number of rows df1:", crime_df1.count())

crime_df2 = spark.read.csv("hdfs:///use/user/Crime_Data_from_2020_to_Present.csv",
                           header=True, schema=LA_Crime_Data)
print("Total number of rows df2:", crime_df2.count())

# Concatenate DataFrames
crime_df = crime_df1.union(crime_df2)
crime_df=crime_df.withColumn("Date Rptd", to_date(col("Date Rptd"),"MM/dd/yyyy"))
crime_df=crime_df.withColumn("DATE OCC", to_date(col("DATE OCC"),"MM/dd/yyyy"))
crime_df = crime_df.dropDuplicates()

# Print the total number of rows
print("Total number of rows df:", crime_df.count())
# Prin data type of each column
print("Schema:")
crime_df.printSchema()

crime_df.write.parquet("output/crime_df.parquet")
```

Ζητούμενο 3

Σε αυτό το βήμα υλοποιείται το Query 1 με χρήση DataFrame και SQL APIs και εκτέλεση με 4 Spark executors.

- Για το SQL API θα ορίσουμε ένα εσωτερικό subquery, το οποίο, με χρήση της συνάρτησης **`RANK()`** πάνω σε partition στο έτος, θα κατατάξει τους μήνες κάθε έτους ανάλογα με το πλήθος των εγκλημάτων. Με το εξωτερικό subquery διαλέγουμε τους τρεις κορυφαίους μήνες κάθε έτους.

- Για το DataFrame API θα ορίσουμε ένα **`window`**, που θα κάνει **`partition`** με βάση το έτος και θα ταξινομεί με βάση το πλήθος εγκλημάτων. Θα το εφαρμόσουμε στο DataFrame μαζί με την συνάρτηση **`rank()`**, που θα εκτελέσει την ίδια λειτουργία με πριν.

Τα αποτελέσματα που παίρνουμε είναι:

SQL API

```
For each year the months with the most crimes are:
With SQL API:
+----+-----+-----------+----------------+
|year|month|crime_total|rank_within_year|
+----+-----+-----------+----------------+
|2010|1    |19515      |1               |
|2010|3    |18131      |2               |
|2010|7    |17856      |3               |
|2011|1    |18135      |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    |13531      |1               |
|2014|10   |13362      |2               |
|2014|8    |13317      |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    |19973      |1               |
|2018|7    |19875      |2               |
|2018|8    |19761      |3               |
|2019|7    |19121      |1               |
|2019|8    |18979      |2               |
|2019|3    |18856      |3               |
|2020|1    |18498      |1               |
|2020|2    |17256      |2               |
|2020|5    |17205      |3               |
|2021|10   |19306      |1               |
|2021|7    |18659      |2               |
|2021|8    |18375      |3               |
|2022|5    |20419      |1               |
|2022|10   |20276      |2               |
|2022|6    |20204      |3               |
|2023|8    |19772      |1               |
|2023|7    |19709      |2               |
|2023|1    |19637      |3               |
```

DataFrame API

```
With DF API:
+----+-----+-----------+----------------+
|year|month|crime_total|rank_within_year|
+----+-----+-----------+----------------+
|2010|1    |19515      |1               |
|2010|3    |18131      |2               |
|2010|7    |17856      |3               |
|2011|1    |18135      |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    |13531      |1               |
|2014|10   |13362      |2               |
|2014|8    |13317      |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    |19973      |1               |
|2018|7    |19875      |2               |
|2018|8    |19761      |3               |
|2019|7    |19121      |1               |
|2019|8    |18979      |2               |
|2019|3    |18856      |3               |
|2020|1    |18498      |1               |
|2020|2    |17256      |2               |
|2020|5    |17205      |3               |
|2021|10   |19306      |1               |
|2021|7    |18659      |2               |
|2021|8    |18375      |3               |
|2022|5    |20419      |1               |
|2022|10   |20276      |2               |
|2022|6    |20204      |3               |
|2023|8    |19772      |1               |
|2023|7    |19709      |2               |
|2023|1    |19637      |3               |
+----+-----+-----------+----------------+
```

Χρόνοι εκτέλεσης

Μετράμε τον χρόνο που χρειάζονται οι δύο διαφορετικές υλοποιήσεις. Κάνουμε αρκετές μετρήσεις, σε τακτά διαστήματα της ημέρας, αφαιρούμε τις τιμές outliers και παίρνουμε τον μέσο όρο. Οι τιμές χρόνων εκτέλεσης, τελικά, είναι οι εξής:

- Για το SQL API: **25.23s**
- Για το DataFrame API: **6.19s**

Η διαφορά αυτή οφείλεται στο επιπλέον parsing και στην πολυπλοκότητα του SQL Query, που ενδεχομένως να περιορίζει τις βελτιστοποιήσεις που μπορεί να κάνει ο Catalyst Optimizer του Spark.

Κώδικας

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, to_date
import datetime

# Imports for Query 1
from pyspark.sql.functions import month, year, count, desc, rank,
                                  countDistinct, regexp_replace
from pyspark.sql.window import Window

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

crime_df = spark.read.parquet("output/crime_df.parquet")

## SQL API ###
start_time_sql = datetime.datetime.now()

# Remove the rows with Null values in DATE OCC (there are none)
crime_df_na_DateOCC=crime_df.na.drop(subset=["DATE OCC"])
print("Total number of rows df after na:", crime_df_na_DateOCC.count())
crime_df_na_DateOCC.createOrReplaceTempView("crimes_q1")

query1 = """
    SELECT year, month, crime_total, rank_within_year
    FROM (
        SELECT year, month, crime_total,
               RANK() OVER (PARTITION BY year ORDER BY crime_total DESC) AS
              rank_within_year
```

```
        FROM (
            SELECT year(`DATE OCC`) AS year, month(`DATE OCC`) AS month,
                COUNT(*) AS crime_total
            FROM crimes_q1
            GROUP BY year(`DATE OCC`), month(`DATE OCC`)
        ) ranked
    ) ranked_with_ranks
    WHERE rank_within_year <= 3
    ORDER BY year
"""

result1_SQL=spark.sql(query1)
print("For each year the months with the most crimes are:")
print("With SQL API:")

result1_SQL.show(result1_SQL.count(), False)

end_time_sql = datetime.datetime.now()

### DataFrame API ###

start_time_df = datetime.datetime.now()

result1_DF = crime_df.groupBy(year('DATE OCC').alias('year'), month('DATE OCC') \
                    .alias('month')).agg(count('*').alias('crime_total'))

# Define a Window partitioned by year and ordering by the number of crimes per
month in descending order
window_spec = Window.partitionBy('year').orderBy(desc('crime_total'))

result1_DF.withColumn('rank_within_year', rank().over(window_spec))
result1_DF = result1_DF.where(col('rank_within_year') <= 3)
result1_DF = result1_DF.orderBy('year', desc('crime_total'))

print("With DF API:")
result1_DF.show(result1_DF.count(), False)

end_time_df = datetime.datetime.now()
execution_time_sql = end_time_sql - start_time_sql
execution_time_df = end_time_df - start_time_df

# Display execution times
print(f"SQL Execution Time: {execution_time_sql} seconds")
print(f"DataFrame Execution Time: {execution_time_df} seconds")
spark.stop()
```

Ζητούμενο 4

Σε αυτό το βήμα υλοποιείται το Query 2 με χρήση DataFrame/SQL και RDD APIs και εκτέλεση με 4 Spark executors.

- Για το SQL API χωρίζουμε τον πίνακα σε 4 χρονικές ζώνες βάσει της ώρας που διαπράχθηκε το κάθε έγκλημα (**`TIME OCC`**), με χρήση των **`CASE`** και **`GROUP BY`**, κρατώντας μόνο τα εγκλήματα που έγιναν στον δρόμο (**`Premis Desc = STREET`**). Έπειτα, κατατάσσουμε τις ζώνες ανάλογα με το πλήθος εγκλημάτων (χρήση συνάρτησης **`COUNT()`**).

- Για το RDD API, αρχικά, διαβάζουμε και διαμορφώνουμε κατάλληλα τα αρχεία, ώστε να μπορέσουμε να τα επεξεργαστούμε (διαχωρισμός σε στήλες, διαγραφή διπλότυπων). Στη συνέχεια, με χρήση της συνάρτησης **`map()`**:

    - Διατηρούμε μόνο τα εγκλήματα που διαπράχθηκαν στον δρόμο (**`x[15] == "STREET"`**)
    - Δημιουργούμε 4 πίνακες χρονικών ζωνών, βάσει της ώρας που διαπράχθηκε το κάθε έγκλημα (**`x[3]`**)

  Έπειτα, με χρήση της συνάρτησης **`count()`** υπολογίζουμε το πλήθος των εγκλημάτων κάθε χρονικής ζώνης και με τη συνάρτηση **`sorted()`** τις ταξινομούμε σε φθίνουσα σειρά.

Παρακάτω παραθέτουμε τα αποτελέσματα που προκύπτουν:

SQL API

```
+------------+------+
|Day Segments| Count|
+------------+------+
|       Night|232961|
|     Evening|183392|
|   Afternoon|144790|
|     Morning|121062|
+------------+------+
```

RDD API

```
[('Night', 232961), ('Evening', 183392), ('Afternoon', 144790), ('Morning', 121062)]
```

Χρόνοι εκτέλεσης

Με σκοπό την ορθή σύγκριση των χρόνων εκτέλεσης, μετράμε μόνο την εκτέλεση του query και στις δύο περιπτώσεις και όχι τη δημιουργία των πινάκων, καθώς για το RDD API τα δεδομένα υπόκεινται στην αρχή σε επεξεργασία, ενώ για το SQL API ο πίνακας αντλείται κατευθείαν από το HDFS.

Μετά από 10 μετρήσεις σε διαφορετικά χρονικά διαστήματα της μέρας και αφαίρεση των outliers, προκύπτουν οι παρακάτω μέσες τιμές χρόνων εκτέλεσης:

- Για το SQL API: **8.84s**
- Για το RDD API: **297.65s**

Η μεγάλη αυτή απόκλιση των δύο APIs, οφείλεται κυρίως σε βελτιστοποιήσεις του Apache Spark που εφαρμόζονται στο SQL API, από τις οποίες το RDD δεν επωφελείται στον ίδιο βαθμό.

Κώδικας

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField, StructType, IntegerType,
DoubleType, StringType, TimestampType
from pyspark.sql.functions import col, to_date, when, substring
from pyspark.sql import functions as F
from io import StringIO
import csv
import time

### DataFrame/SQL API ###

spark = SparkSession.builder \
       .appName("Query 2 using SQL") \
       .config("spark.executor.instances", 4) \
       .getOrCreate()

# Set the legacy time parser policy
spark.conf.set("spark.sql.legacy.timeParserPolicy", "LEGACY")

crime_df = spark.read.parquet("output/crime_df.parquet")

crime_df.createOrReplaceTempView("CrimeReports")
```

```
query2 = """
    SELECT
      CASE
       WHEN `TIME OCC` BETWEEN 0500 AND 1159 THEN 'Morning'
       WHEN `TIME OCC` BETWEEN 1200 AND 1659 THEN 'Afternoon'
       WHEN `TIME OCC` BETWEEN 1700 AND 2059 THEN 'Evening'
       WHEN `TIME OCC` BETWEEN 0000 AND 0459 OR `TIME OCC` BETWEEN 2100 AND
       2359 THEN 'Night'
     END AS `Day Segments`, COUNT(*) AS Count
   FROM CrimeReports
   WHERE `Premis Desc` = 'STREET'
   GROUP BY `Day Segments`
   ORDER BY Count DESC
"""

# Start timer
start_time = time.time()
# Execute Query 2
query2_df = spark.sql(query2)
query2_df.show()
# Stop timer
end_time = time.time()
# Calculate the runtime
print(f"DataFrame/SQL API runtime: {end_time - start_time} seconds")
# Stop the Spark session when done
spark.stop()


### RDD API ###

spark = SparkSession.builder \
       .appName("Query 2 using RDD") \
       .config("spark.executor.instances", 4) \
       .getOrCreate()

# Get the SparkContext
sc = spark.sparkContext

# Open 1st file
data1 = sc.textFile("hdfs:///user/user/Crime_Data_from_2010_to_2019.csv") \
          .map(lambda x: next(csv.reader(StringIO(x))))
headers = data1.first()
# Delete column names
data1 = data1.filter(lambda x: x != headers)
```

```
# Open 2nd file
data2 = sc.textFile("hdfs:///user/user/Crime_Data_from_2020_to_Present.csv") \
          .map(lambda x: next(csv.reader(StringIO(x))))
headers = data2.first()

# Delete column names
data2 = data2.filter(lambda x: x != headers)

# Merge the 2 files
data = data1.union(data2)
# Keep only the Date part (not Time) from DATE OCC and Date Rptd columns
data = data.map(lambda x: [element.split(" ")[0] if (i == 1 or i == 2) else
               element for i, element in enumerate(x)])
headers = data.first()
# Delete column names
data = data.filter(lambda x: x != headers)

# Delete duplicates
data = data.map(lambda x: (x[0], x))
data = data.groupByKey().map(lambda x: (x[0], list(x[1])[0]))
data = data.map(lambda x: x[1])

# Start timer
start_time = time.time()


# Find STREET
Premis_Street = data.map(lambda x: x if (x[15] == "STREET") else None) \
                    .filter(lambda x: x != None)

# Separate the day in 4 segments
Morning = Premis_Street.map(lambda x: x if ( (int(x[3]) >= 500) and (int(x[3])
<= 1159) ) else None).filter(lambda x: x != None)

Afternoon = Premis_Street.map(lambda x: x if ( (int(x[3]) >= 1200) and
(int(x[3]) <= 1659) ) else None).filter(lambda x: x != None)

Evening = Premis_Street.map(lambda x: x if ( (int(x[3]) >= 1700) and
(int(x[3]) <= 2059) ) else None).filter(lambda x: x != None)

Night = Premis_Street.map(lambda x: x if ( ( (int(x[3]) >= 0) and (int(x[3])
<= 459) ) or ( (int(x[3]) >= 2100) and (int(x[3]) <= 2359) ) ) else
None).filter(lambda x: x != None)

# Create a list of tuples containing the Day Segment and its count
counts = [("Morning", Morning.count()),
```

```
          ("Afternoon", Afternoon.count()),
          ("Evening", Evening.count()),
          ("Night", Night.count())]

# Sort the list of tuples by count in descending order
sorted_counts = sorted(counts, key=lambda x: x[1], reverse=True)
print(sorted_counts)

# Stop timer
end_time = time.time()
# Calculate the runtime
print(f"RDD API runtime: {end_time - start_time} seconds")

# Stop the Spark session when done
spark.stop()
```

Ζητούμενο 5

Σε αυτό το βήμα υλοποιείται το Query 3 με χρήση DataFrame/SQL API και εκτέλεση με 2-4 Spark executors.

Αρχικά διαβάζουμε τα δύο νέα .csv αρχεία και κάνουμε την απαραίτητη επεξεργασία (αφαίρεση duplicates, Nulls). Έπειτα θα εκτελέσουμε ένα **`join`** μεταξύ του **`crime_df`** και του **`revgecoding_df`**, στα **`LAT`** και **`LON`**, ώστε να ενσωματώσουμε την πληροφορία του **`Zip Code`** στο **`crime_df`**. Στην συνέχεια θα βρούμε όλα τα μοναδικά **`Zip Codes`** που έχουμε στο **`crimes_df`**, και ανάμεσά τους θα αναζητήσουμε εκείνα, για τα οποία εμφανίζονται τα τρία **`max`** και **`min incomes`** στο **`income_df`** (μας ενδιαφέρουν οι περιοχές για τις οποίες έχουμε δεδομένα εγκλημάτων). Τέλος, εκτελούμε τα queries και τα αποτελέσματα που λαμβάνουμε είναι:

```
Number of rows in revgecoding_df: 37781
Number of rows in income_df: 284
Num of instances with multiple zipcodes:  0
Victim descent of minimum income ZipCode areas:
+-----------------------------+-----+
|Victim Descent               |Count|
+-----------------------------+-----+
|Hispanic/Latin/Mexican       |1531 |
|Black                        |1093 |
|White                        |703  |
|Other                        |393  |
|Other Asian                  |103  |
|Unknown                      |64   |
|Korean                       |9    |
|Japanese                     |3    |
|American Indian/Alaskan Native|3    |
|Chinese                      |2    |
|Filipino                     |1    |
+-----------------------------+-----+

Victim descent of maximum income ZipCode areas:
+---------------------+-----+
|Victim Descent       |Count|
+---------------------+-----+
|White                |338  |
|Other                |106  |
|Hispanic/Latin/Mexican|52   |
|Unknown              |27   |
|Black                |16   |
|Other Asian          |16   |
+---------------------+-----+
```

Χρόνοι εκτέλεσης

Θα συγκρίνουμε τους χρόνους εκτέλεσης για 2, 3, 4 Spark executors. Ακολουθώντας την μεθοδολογία του Query 1, προκύπτουν χρόνοι εκτέλεσης:

- Για 2 executors: **32.71s**
- Για 3 executors: **34.16s**
- Για 4 executors: **40.52s**

Αυτό αρχικά φαίνεται παράδοξο, όμως εξηγείται από την έλλειψη παραλληλισμού των tasks. Συγκεκριμένα τα jobs του Spark χωρίζονται σε stages και κάθε stage αποτελείται από κάποια tasks, τα οποία μπορούν να εκτελεστούν παράλληλα σε διαφορετικούς executors, βελτιώνοντας τον χρόνο εκτέλεσης (γίνεται και stage parallelism, αλλά διέπεται από

περισσότερες αλληλεξαρτήσεις). Στην περίπτωσή μας, βλέπουμε στον Spark History Server ότι σχεδόν όλα τα stages αποτελούνται από **ένα** task το καθένα, μειώνοντας την δυνατότητα εκτέλεσής τους παράλληλα:

Spark History Server

Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total
46	showString at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:42	0.2 s	1/1
44	showString at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:41	1.0 s	5/5
43	$anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	+details	2024/01/09 19:44:40	0.7 s	1/1
42	$anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	+details	2024/01/09 19:44:40	0.2 s	1/1
41	count at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:40	0.1 s	1/1
38	count at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:39	0.3 s	1/1
36	count at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:38	1 s	5/5
35	$anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	+details	2024/01/09 19:44:37	0.4 s	1/1
34	$anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	+details	2024/01/09 19:44:36	0.5 s	1/1
33	showString at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:34	0.2 s	1/1
31	showString at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:32	2 s	5/5
30	$anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	+details	2024/01/09 19:44:31	0.6 s	1/1
29	$anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	+details	2024/01/09 19:44:31	0.3 s	1/1
28	count at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:30	0.3 s	1/1
25	count at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:30	0.3 s	1/1
23	count at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:28	1 s	5/5
22	$anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	+details	2024/01/09 19:44:27	0.5 s	1/1
21	$anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	+details	2024/01/09 19:44:26	0.7 s	1/1
20	collect at /home/user/query3.py:106	+details	2024/01/09 19:44:22	2 s	1/1
18	javaToPython at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:17	5 s	5/5
17	$anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	+details	2024/01/09 19:44:15	1 s	1/1
16	count at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:14	0.1 s	1/1
12	count at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:13	0.4 s	1/1
9	count at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:12	1 s	1/1
7	count at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:08	3 s	1/1
6	count at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:05	2 s	1/1
4	count at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:05	0.2 s	1/1
3	count at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:04	0.4 s	1/1
1	count at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:44:00	4 s	1/1
0	parquet at NativeMethodAccessorImpl.java:0	+details	2024/01/09 19:43:49	2 s	1/1

Ακόμα ένας παράγοντας που επηρεάζει την εκτέλεση είναι το data skew. Συγκεκριμένα, σε ένα stage που αποτελείται από περισσότερα tasks θα παρατηρήσουμε την έλλειψη ισορροπίας μεταξύ τους. Για παράδειγμα, στην ακόλουθη περίπτωση, ακόμη κι αν τα tasks εκτελέστηκαν παράλληλα, το μικρότερο από αυτά διήρκησε **87ms** και το μεγαλύτερο **0.9s**, δηλαδή 10 φορές πιο χρονοβόρο:

Spark History Server

Summary Metrics for 5 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	87.0 ms	0.4 s	0.6 s	0.6 s	0.9 s
GC Time	10.0 ms	10.0 ms	15.0 ms	16.0 ms	96.0 ms
Input Size / Records	628.8 KiB / 103017	3.8 MiB / 703366	3.8 MiB / 703465	3.9 MiB / 704952	3.9 MiB / 718745
Shuffle Write Size / Records	282 B / 4	426 B / 6	426 B / 6	426 B / 6	426 B / 6

Τελικά, καταλήγουμε στην διαπίστωση ότι, για να βελτιωθεί ο χρόνος εκτέλεσης με χρήση περισσότερων executors το πρόγραμμά μας θα πρέπει να διέπεται από ανάλογο βαθμό παραλληλισμού, κάτι που δεν ισχύει στην συγκεκριμένη περίπτωση, οπότε οι περισσότεροι executors απλά καταναλώνουν τα resources του συστήματος και αυξάνεται το I/O.

Κώδικας

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField, StructType, IntegerType,
DoubleType, StringType
from pyspark.sql.functions import col, to_date
import datetime

# Imports for Query3
from pyspark.sql.functions import year, count, desc, countDistinct,
                            regexp_replace

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

spark.conf.set("spark.sql.debug.maxToStringFields", "100")

crime_df = spark.read.parquet("output/crime_df.parquet")

LA_income_2015=StructType([
    StructField("Zip Code", IntegerType()),
    StructField("Community", StringType()),
    StructField("Estimated Median Income", StringType())
])
income_df= spark.read.csv("hdfs:///user/user/LA_income_2015.csv", header=True,
schema= LA_income_2015)

# We read 'Estimated Median Income' as string bc there are the '$', ','
characters, we remove them and we convert the type to integer,
# bc we will use this columnn in orderBy later
income_df = income_df.withColumn("Estimated Median Income",
      regexp_replace(col("Estimated Median Income"), "[\$,]", ""))
income_df = income_df.withColumn("Estimated Median Income", col("Estimated
      Median Income").cast("integer"))

Revgecoding=StructType([
    StructField("LAT", DoubleType()),
    StructField("LON", DoubleType()),
    StructField("ZIPcode", IntegerType())
])
revgecoding_df= spark.read.csv("hdfs:///user/user/revgecoding.csv",
                           header=True, schema= Revgecoding)

print("Number of rows in revgecoding_df:", revgecoding_df.count())
```

```
print("Number of rows in income_df:", income_df.count())

start_time = datetime.datetime.now()

revgecoding_df=revgecoding_df.dropna(subset=['LAT', 'LON', 'ZIPcode'])
# We check if there are multiple zip codes for a pair of LAT, LON values ->
there are non
zipcode_counts = revgecoding_df.groupBy('LAT', 'LON') \
    .agg(countDistinct('ZIPCode').alias('num_zipcodes')) \
    .filter(col('num_zipcodes') > 1) \
    .count()
print("Num of instances with multiple zipcodes: ", zipcode_counts)

# We drop rows that are irrelevant (null or not in 2015)
crime_df_na_vDesc=crime_df.na.drop(subset=["Vict Descent"])
crime_df_na_vDesc_2015 = crime_df_na_vDesc.filter(year('DATE OCC') == 2015)
crime_df_trunc = crime_df_na_vDesc_2015.select("DR_NO", "LAT", "LON", "Vict
Descent")

crimes_zip_joined=crime_df_trunc.join(revgecoding_df, ['LAT', 'LON'], 'inner')

unique_zip_codes = crimes_zip_joined.select('ZIPcode').distinct() \
                                    .rdd.flatMap(lambda x: x).collect()
filtered_income_df = income_df.filter(income_df['Zipcode'].isin(unique_zip_codes))

min_income = filtered_income_df.select('Zip Code','Estimated Median Income') \
                               .orderBy('Estimated Median Income') \
                               .limit(3)

max_income = filtered_income_df.select('Zip Code','Estimated Median Income') \
                               .orderBy(desc('Estimated Median Income')) \
                               .limit(3)

# Execute the query
min_income.createOrReplaceTempView("min_income_view")
max_income.createOrReplaceTempView("max_income_view")
crimes_zip_joined.createOrReplaceTempView("crimes_zip_joined_view")


query3a = """
    SELECT  CASE `Vict Descent`
        WHEN 'W' THEN 'White'
        WHEN 'B' THEN 'Black'
        WHEN 'H' THEN 'Hispanic/Latin/Mexican'
        WHEN 'A' THEN 'Other Asian'
        WHEN 'C' THEN 'Chinese'
```

```
        WHEN 'D' THEN 'Cambodian'
        WHEN 'F' THEN 'Filipino'
        WHEN 'G' THEN 'Guamanian'
        WHEN 'I' THEN 'American Indian/Alaskan Native'
        WHEN 'J' THEN 'Japanese'
        WHEN 'K' THEN 'Korean'
        WHEN 'L' THEN 'Laotian'
        WHEN 'O' THEN 'Other'
        WHEN 'P' THEN 'Pacific Islander'
        WHEN 'S' THEN 'Samoan'
        WHEN 'U' THEN 'Hawaiian'
        WHEN 'V' THEN 'Vietnamese'
        WHEN 'X' THEN 'Unknown'
        WHEN 'Z' THEN 'Asian Indian'
        ELSE `Vict Descent`
    END AS `Victim Descent`,
    COUNT(*) AS Count
    FROM crimes_zip_joined_view
    WHERE ZIPCode IN (SELECT `Zip Code` FROM min_income_view)
    GROUP BY `Vict Descent`
    ORDER BY Count DESC
"""

print("Victim descent of minimum income ZipCode areas:")
result3a=spark.sql(query3a)
result3a.show(result3a.count(), False)


query3b = """
    SELECT CASE `Vict Descent`
        WHEN 'W' THEN 'White'
        WHEN 'B' THEN 'Black'
        WHEN 'H' THEN 'Hispanic/Latin/Mexican'
        WHEN 'A' THEN 'Other Asian'
        WHEN 'C' THEN 'Chinese'
        WHEN 'D' THEN 'Cambodian'
        WHEN 'F' THEN 'Filipino'
        WHEN 'G' THEN 'Guamanian'
        WHEN 'I' THEN 'American Indian/Alaskan Native'
        WHEN 'J' THEN 'Japanese'
        WHEN 'K' THEN 'Korean'
        WHEN 'L' THEN 'Laotian'
        WHEN 'O' THEN 'Other'
        WHEN 'P' THEN 'Pacific Islander'
        WHEN 'S' THEN 'Samoan'
```

```
        WHEN 'U' THEN 'Hawaiian'
        WHEN 'V' THEN 'Vietnamese'
        WHEN 'X' THEN 'Unknown'
        WHEN 'Z' THEN 'Asian Indian'
    ELSE `Vict Descent`
    END AS `Victim Descent`,
    COUNT(*) AS Count
    FROM crimes_zip_joined_view
    WHERE ZIPCode IN (SELECT `Zip Code` FROM max_income_view)
    GROUP BY `Vict Descent`
    ORDER BY Count DESC
"""

print("Victim descent of maximum income ZipCode areas:")
result3b=spark.sql(query3b)
result3b.show(result3b.count(), False)

end_time= datetime.datetime.now()

execution_time = end_time - start_time
print("Execution Time:", execution_time)

spark.stop()
```

Ζητούμενο 6

Σε αυτό το βήμα υλοποιείται το Query 4 με χρήση DataFrame/SQL API. Για τις ανάγκες του παρόντος βήματος δημιουργήθηκαν οι παρακάτω τρεις βοηθητικοί πίνακες:

- Precincts: Πίνακας δημιουργημένος από το αρχείο "LAPD_Police_Stations.csv", ο οποίος περιλαμβάνει τα χαρακτηριστικά των αστυνομικών τμημάτων της πόλης.

- CrimeView1: Πίνακας View δημιουργημένος από τον αρχικό (CrimeReports) μετά την αφαίρεση εγκλημάτων που διαπράχθηκαν σε τοποθεσίες με μηδενικές συντεταγμένες (**`LAT <> 0 AND LON <> 0`**) και μη πολυβόλων όπλων (**`Weapon Used Cd BETWEEN 100 AND 199`**).

- CrimeView2: Πίνακας View δημιουργημένος από τον αρχικό (CrimeReports) μετά την αφαίρεση εγκλημάτων που διαπράχθηκαν σε τοποθεσίες με μηδενικές συντεταγμένες (**`LAT <> 0 AND LON <> 0`**) και όπλων με τιμή NULL (**`Weapon Used Cd IS NOT NULL`**).

Επιπλέον, δημιουργήθηκε η συνάρτηση **`get_distance()`**, η οποία υπολογίζει την απόσταση δύο σημείων πάνω στη Γη με χρήση του τύπου Haversine, βάσει των συντεταγμένων τους.

Στη συνέχεια, θα αναλύσουμε τις υλοποιήσεις των queries που ζητούνται.

- Query4a1: Με χρήση του **`INNER JOIN _ ON`** ενώνουμε τους πίνακες CrimeView1 και Precincts πάνω στους μοναδικούς κωδικούς αστυνομικών τμημάτων (**`AREA – PREC`**). Στη συνέχεια, χρησιμοποιώντας τις συναρτήσεις **`AVG()`** και **`get_distance()`** υπολογίζουμε ανά έτος τη μέση απόσταση των σημείων όπου διαπράχθηκαν τα εγκλήματα από το αστυνομικό τμήμα που τα ανέλαβε και χρησιμοποιώντας τη συνάρτηση **`COUNT()`** βρίσκουμε ανά έτος το συνολικό πλήθος εγκλημάτων. Τέλος, ταξινομούμε ανά έτος με αύξουσα σειρά (**`ORDER BY Year`**). Λόγω της αξιοποίησης του CrimeView1, εξετάζουμε μόνο τα εγκλήματα που έγινε χρήση *πυροβόλου* όπλου.

- Query4b1: Με χρήση του **`INNER JOIN _ ON`** ενώνουμε τους πίνακες CrimeView2 και Precincts πάνω στους μοναδικούς κωδικούς αστυνομικών τμημάτων (**`AREA – PREC`**). Στη συνέχεια, χρησιμοποιώντας τις συναρτήσεις **`AVG()`** και **`get_distance()`** υπολογίζουμε ανά αστυνομικό τμήμα τη μέση απόσταση των τόπων εγκλήματος από το τμήμα που ανέλαβε την έρευνα και χρησιμοποιώντας τη συνάρτηση **`COUNT()`** βρίσκουμε το συνολικό πλήθος εγκλημάτων που ανέλαβε κάθε τμήμα. Τέλος, ταξινομούμε ανά αριθμό περιστατικών με φθίνουσα σειρά (**`ORDER BY Count DESC`**). Λόγω της αξιοποίησης του CrimeView2, εξαιρούμε τα εγκλήματα με τιμή όπλου NULL.

- Query4a2: Ορίζουμε, αρχικά, ένα subquery, το οποίο βρίσκει ανά έγκλημα το κοντινότερο αστυνομικό τμήμα. Αυτό επιτυγχάνεται, δημιουργώντας έναν πίνακα με όλα τα πιθανά ζεύγη εγκλήματος – αστυνομικού τμήματος (**`CrimeView1 INNER JOIN Precincts`**), από τον οποίο, με χρήση των συναρτήσεων **`MIN()`** και **`get_distance()`**, υπολογίζεται η ελάχιστη απόσταση κάθε εγκλήματος από όλα τα αστυνομικά τμήματα.

Στη συνέχεια, αξιοποιούμε τα αποτελέσματα του subquery, για να βρούμε ανά έτος τη μέση απόσταση των σημείων όπου διαπράχθηκαν τα εγκλήματα από το κοντινότερο αστυνομικό τμήμα (χρήση συνάρτησης **`AVG()`**) και το πλήθος των εγκλημάτων (χρήση συνάρτησης **`COUNT()`**). Τέλος, ταξινομούμε ανά έτος με αύξουσα σειρά (**`ORDER BY Year`**). Όμοια με το Query4a1, λόγω του βοηθητικού πίνακα CrimeView1, θεωρούμε μόνο τα εγκλήματα που έγινε χρήση *πυροβόλου* όπλου.

- Query4b2: Αρχικά, κατασκευάζουμε δύο subqueries. Το πρώτο υπολογίζει για όλα τα πιθανά ζεύγη εγκλήματος – αστυνομικού τμήματος (**`CrimeView2 INNER JOIN Precincts`**) τη μεταξύ τους απόσταση (**`get_distance()`**). Το δεύτερο υπολογίζει για κάθε έγκλημα την απόσταση του πλησιέστερου αστυνομικού τμήματος, όμοια με το Query4a2 (**`MIN(get_distance())`**). Με χρήση **`INNER JOIN _ ON`** ενώνουμε τα αποτελέσματα των subqueries πάνω στους κωδικούς εγκλήματος (**`DR_NO`**) και στις αποστάσεις (**`Distance = Min_Distance`**). Με αυτόν τον τρόπο, επιτυγχάνουμε να αντιστοιχίσουμε το κάθε περιστατικό σε ένα μόνο τμήμα, το οποίο μάλιστα είναι το κοντινότερο. Τέλος, υπολογίζουμε για κάθε αστυνομικό τμήμα τη μέση τιμή των αποστάσεων των *κοντινών* σε αυτό περιστατικών (**`AVG(Min_Distance)`**) και ταξινομούμε βάσει του πλήθους περιστατικών με φθίνουσα σειρά (**`ORDER BY Count DESC`**). Λόγω της αξιοποίησης του CrimeView2, εξαιρούμε τα εγκλήματα με τιμή όπλου NULL.

Τα αποτελέσματα που προκύπτουν είναι τα εξής:

Query4a1

Year	Average Distance	Count
2010	2.784	8212
2011	2.793	7232
2012	2.836	6532
2013	2.826	5838
2014	2.773	4526
2015	2.706	6763
2016	2.718	8100
2017	2.724	7786
2018	2.733	7413
2019	2.74	7129
2020	2.69	8487
2021	2.64	9745
2022	2.609	10025
2023	2.548	8896

Query4b1

Division	Average Distance	Count
77TH STREET	2.645	94513
SOUTHEAST	2.092	72851
SOUTHWEST	2.608	72499
CENTRAL	1.007	63298
NEWTON	2.055	61188
RAMPART	1.532	55620
HOLLYWOOD	1.433	50914
OLYMPIC	1.759	48913
PACIFIC	3.877	42764
HOLLENBECK	2.595	41391
MISSION	4.71	40874
HARBOR	3.941	40659
NORTH HOLLYWOOD	2.548	39895
WILSHIRE	2.403	37739
NORTHEAST	3.991	37122
VAN NUYS	2.136	36075
FOOTHILL	4.237	34532
WEST VALLEY	3.385	33709
TOPANGA	3.513	32377
DEVONSHIRE	3.979	30249

Query4a2

Year	Average Distance	Count
2010	2.435	8212
2011	2.462	7232
2012	2.506	6532
2013	2.456	5838
2014	2.389	4526
2015	2.388	6763
2016	2.429	8100
2017	2.392	7786
2018	2.409	7413
2019	2.43	7129
2020	2.384	8487
2021	2.353	9745
2022	2.313	10025
2023	2.267	8896

Query4b2

Division	Average Distance	Count
77TH STREET	1.674	78938
SOUTHWEST	2.161	78152
HOLLYWOOD	1.92	70742
SOUTHEAST	2.222	66782
OLYMPIC	1.666	60668
CENTRAL	0.867	59535
WILSHIRE	2.478	58124
RAMPART	1.361	56399
VAN NUYS	2.808	55356
NEWTON	1.6	45461
HOLLENBECK	2.587	42073
NORTH HOLLYWOOD	2.605	40817
PACIFIC	3.846	40428
FOOTHILL	3.98	40280
HARBOR	3.687	39487
WEST VALLEY	2.852	34934
TOPANGA	3.045	33078
MISSION	3.775	27370
NORTHEAST	3.766	27327
WEST LOS ANGELES	2.713	22412

Κώδικας

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField, StructType, IntegerType,
                          DoubleType, StringType, TimestampType
from pyspark.sql.functions import col, to_date, when, substring
from pyspark.sql import functions as F
from pyspark.sql.functions import udf
import math
import time

# Function for diastance calculation
def get_distance(lat1, lon1, lat2, lon2):
    R = 6371  # Earth radius in kilometers

    # Convert latitude and longitude from degrees to radians
    lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])

    # Differences in coordinates
    dlat = lat2 - lat1
    dlon = lon2 - lon1

    # Haversine formula
    a = math.sin(dlat / 2) ** 2 + math.cos(lat1) * math.cos(lat2) *
      math.sin(dlon / 2) ** 2
```

```
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    distance = R * c  # Distance in kilometers
    return distance


# Create Spark session
spark = SparkSession.builder.appName("Query 4").getOrCreate()
# Register the UDF
spark.udf.register("get_distance", get_distance, DoubleType())

crime_df = spark.read.parquet("output/crime_df.parquet")

# Precincts data
precincts_schema = StructType([
    StructField("X", DoubleType()),
    StructField("Y", DoubleType()),
    StructField("FID", IntegerType()),
    StructField("DIVISION", StringType()),
    StructField("LOCATION", StringType()),
    StructField("PREC", IntegerType())
])

Precincts_df = spark.read.csv("LAPD_Police_Stations.csv", header=True,
schema=precincts_schema)

# Create Tables
crime_df.createOrReplaceTempView("CrimeReports")
Precincts_df.createOrReplaceTempView("Precincts")

# Create useful views

view1  = """
CREATE OR REPLACE TEMPORARY VIEW CrimeView1 AS
SELECT * FROM CrimeReports
WHERE LAT <> 0.0 AND LON <> 0.0 AND `Weapon Used Cd` BETWEEN 100 AND 199
"""

view2  = """
CREATE OR REPLACE TEMPORARY VIEW CrimeView2 AS
SELECT * FROM CrimeReports
WHERE LAT <> 0.0 AND LON <> 0.0 AND `Weapon Used Cd` IS NOT NULL
"""

spark.sql(view1)
spark.sql(view2)
```

```
### Query 4a (1) ###

query4a1 = """
SELECT YEAR(c.`DATE OCC`) AS Year, AVG(get_distance(c.LAT, c.LON, p.Y, p.X))
AS `Average Distance`, COUNT(*) AS Count
FROM CrimeView1 AS c
INNER JOIN Precincts p ON c.AREA = p.PREC
GROUP BY Year
ORDER BY Year
"""

query4a1_df = spark.sql(query4a1)
query4a1_df.show()


### Query 4b (1) ###

query4b1 = """
SELECT p.DIVISION AS Division, AVG(get_distance(c.LAT, c.LON, p.Y, p.X)) AS
`Average Distance`, COUNT(*) AS Count
FROM CrimeView2 AS c
INNER JOIN Precincts p ON c.AREA = p.PREC
GROUP BY Division
ORDER BY Count DESC;
"""

query4b1_df = spark.sql(query4b1)
query4b1_df.show()


### Query 4a (2) ###

query4a2 = """
SELECT Year, AVG(`Min Distance`) AS `Average Distance`, COUNT(*) AS Count
FROM (
        SELECT c.DR_NO, YEAR(c.`DATE OCC`) AS Year, MIN(get_distance(c.LAT,
c.LON, p.Y, p.X)) AS `Min Distance`
        FROM CrimeView1 AS c
        CROSS JOIN Precincts p
        GROUP BY c.DR_NO, Year
    )
GROUP BY Year
ORDER BY Year
"""
```

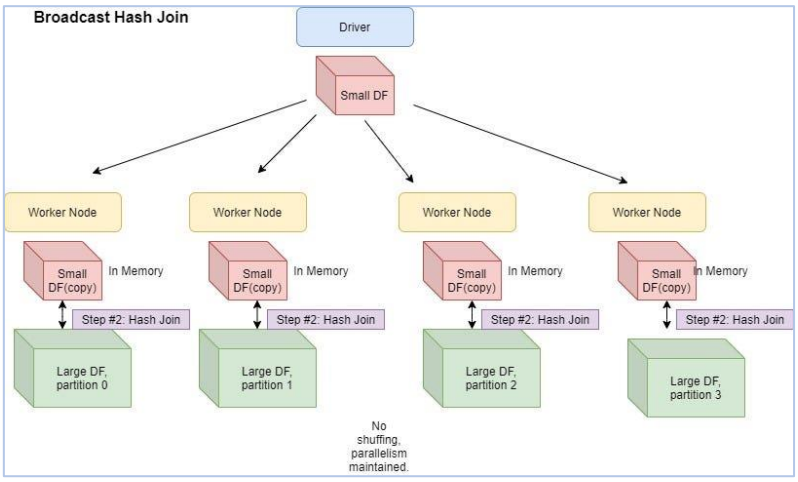
```
query4a2_df = spark.sql(query4a2)
query4a2_df.show()


### Query 4b (2) ###

query4b2 = """
SELECT t1.DIVISION AS Division, AVG(t2.Distance) AS `Average Distance`,
COUNT(*) AS Count
FROM (
        SELECT c.DR_NO, p.DIVISION, get_distance(c.LAT, c.LON, p.Y, p.X) AS
Distance
        FROM CrimeView2 AS c
        CROSS JOIN Precincts p
    ) AS t1
INNER JOIN (
        SELECT c.DR_NO, MIN(get_distance(c.LAT, c.LON, p.Y, p.X)) AS Distance
        FROM CrimeView2 AS c
        CROSS JOIN Precincts p
        GROUP BY DR_NO
    ) AS t2 ON t1.DR_NO = t2.DR_NO AND t1.Distance = t2.Distance
GROUP BY Division
ORDER BY Count DESC;
"""

query4b2_df = spark.sql(query4b2)
query4b2_df.show()
```

Ζητούμενο 7

Σε αυτό το βήμα, θα χρησιμοποιήσουμε **4** Spark executors.

QUERY 3

- **Broadcast Join:** Ιδανικό σε περιπτώσεις που το ένα DataFrame είναι μικρό, οπότε συμφέρει να κάνουμε ένα in-memory copy αυτού του DataFrame σε κάθε κόμβο και να γίνουν εκεί τα joins (για να έχει νόημα θα πρέπει να είναι μικρότερο από την διαθέσιμη μνήμη). Αν το DataFrame είναι κατάλληλου μεγέθους, αυτό το είδος join είναι το γρηγορότερο, καθώς γλιτώνει από πολύ shuffling.



Το **`crime_df`** έχει πιο πολλές σειρές και στήλες από το **`revgecoding_df`** (το ελέγχουμε), οπότε θα χρησιμοποιήσουμε το δεύτερο ως το μικρό DataFrame.

Γίνεται η εξής τροποποίηση στον κώδικα:

```
crimes_zip_joined=crime_df_trunc.join(broadcast(revgecoding_df),['LAT','LON'],
                                      'inner')
crimes_zip_joined.explain()
```

<u>Σχόλιο:</u> Ο χρόνος εκτέλεσης είναι **36.44 s.**

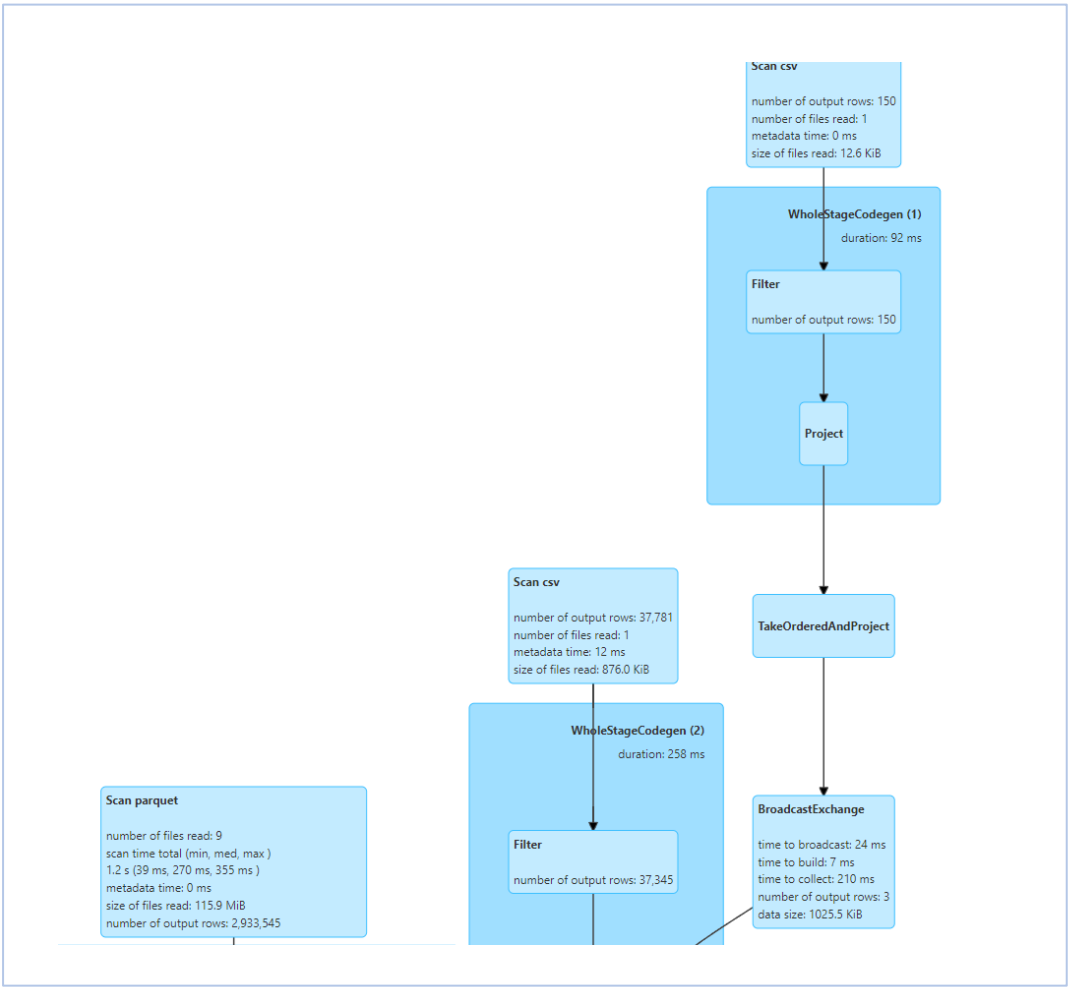
Με την μέθοδο **`explain()`** αμέσως μετά το join λαμβάνουμε το εξής Physical Plan:

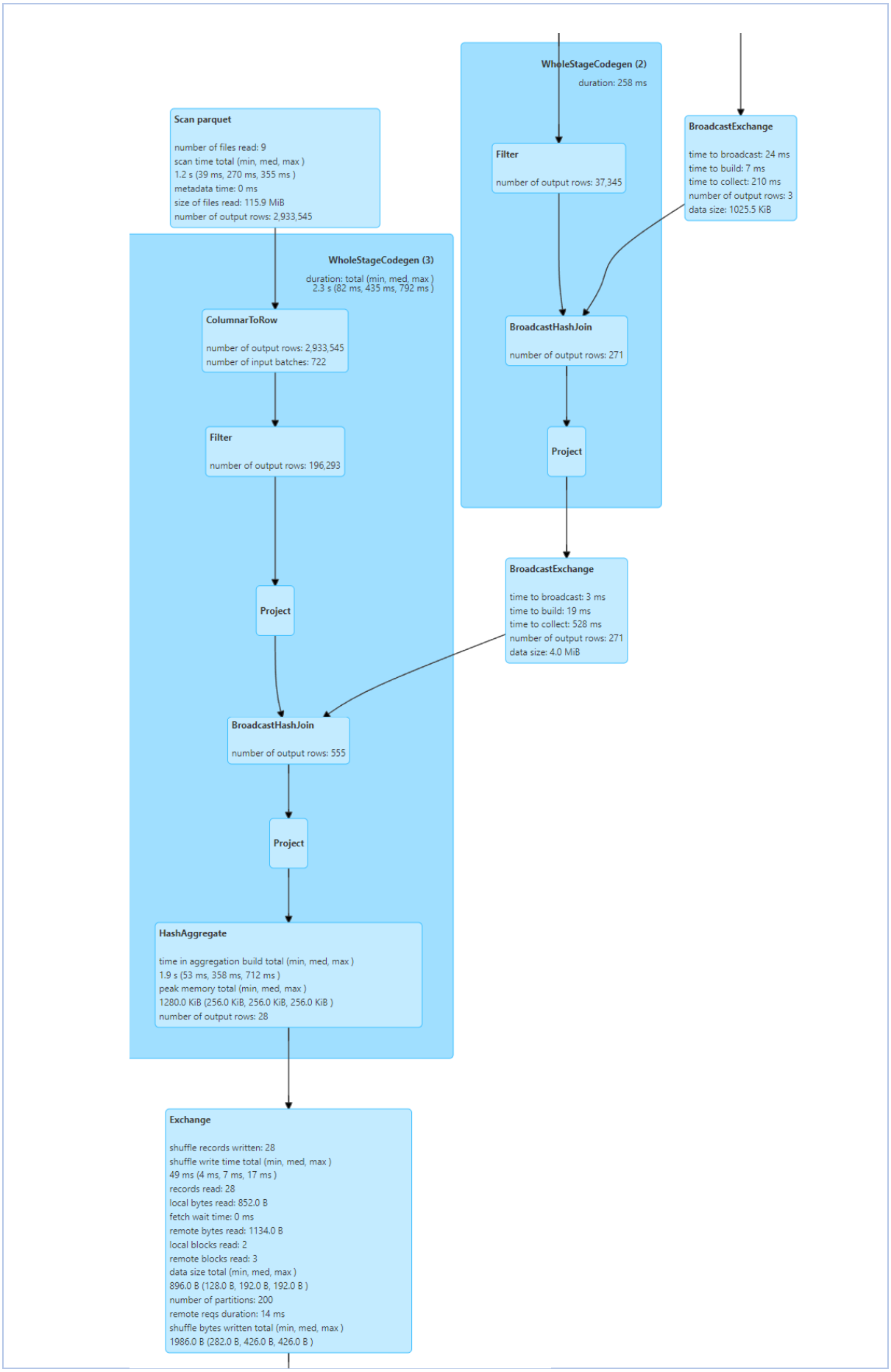
```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Project [LAT#26, LON#27, DR_NO#0, Vict Descent#13, ZIPcode#73]
   +- BroadcastHashJoin [knownfloatingpointnormalized(normalizenanandzero(LAT#26)), knownfloatingpointnormalized(normali
zenanandzero(LON#27))], [knownfloatingpointnormalized(normalizenanandzero(LAT#71)), knownfloatingpointnormalized(normali
zenanandzero(LON#72))], Inner, BuildRight, false
      :- Project [DR_NO#0, LAT#26, LON#27, Vict Descent#13]
      :  +- Filter ((((isnotnull(DATE OCC#2) AND atleastnnonnulls(1, Vict Descent#13)) AND (year(DATE OCC#2) = 2015)) AN
D isnotnull(LAT#26)) AND isnotnull(LON#27))
      :     +- FileScan parquet [DR_NO#0,DATE OCC#2,Vict Descent#13,LAT#26,LON#27] Batched: true, DataFilters: [isnotnul
l(DATE OCC#2), atleastnnonnulls(1, Vict Descent#13), (year(DATE OCC#2) = 2015), isnotnul..., Format: Parquet, Location:
InMemoryFileIndex(1 paths)[hdfs://okeanos-master:54310/user/user/output/crime_df.parquet], PartitionFilters: [], PushedF
ilters: [IsNotNull(`DATE OCC`), IsNotNull(LAT), IsNotNull(LON)], ReadSchema: struct<DR_NO:int,DATE OCC:date,Vict Descent
:string,LAT:double,LON:double>
      +- BroadcastExchange HashedRelationBroadcastMode(List(knownfloatingpointnormalized(normalizenanandzero(input[0, do
uble, false])), knownfloatingpointnormalized(normalizenanandzero(input[1, double, false]))),false), [plan_id=26]
         +- Filter ((atleastnnonnulls(3, LAT#71, LON#72, ZIPcode#73) AND isnotnull(LAT#71)) AND isnotnull(LON#72))
            +- FileScan csv [LAT#71,LON#72,ZIPcode#73] Batched: false, DataFilters: [atleastnnonnulls(3, LAT#71, LON#72,
 ZIPcode#73), isnotnull(LAT#71), isnotnull(LON#72)], Format: CSV, Location: InMemoryFileIndex(1 paths)[hdfs://okeanos-ma
ster:54310/user/user/revgecoding.csv], PartitionFilters: [], PushedFilters: [IsNotNull(LAT), IsNotNull(LON)], ReadSchema
: struct<LAT:double,LON:double,ZIPcode:int>
```

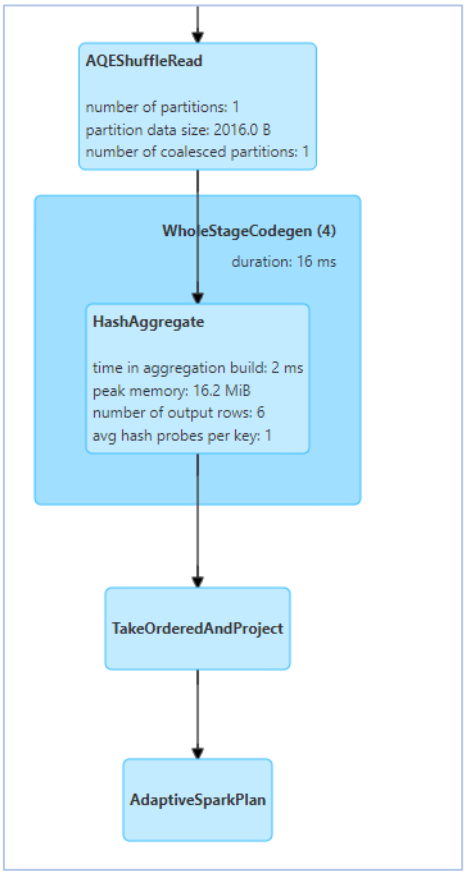
(Διαβάζουμε το Physical Plan bottom-to-top.)

Spark History Server

Στον Spark History Server λαμβάνουμε το SQL/DataFrame Visualisation για το συγκεκριμένο application:




WholeStageCodegen (2)
duration: 258 ms
Scan parquet
number of files read: 9
scan time total (min, med, max)
1.2 s (39 ms, 270 ms, 355 ms)
metadata time: 0 ms
size of files read: 115.9 MiB
number of output rows: 2,933,545
Filter
number of output rows: 37,345
BroadcastExchange
time to broadcast: 24 ms
time to build: 7 ms
time to collect: 210 ms
number of output rows: 3
data size: 1025.5 KiB
WholeStageCodegen (3)
duration: total (min, med, max)
2.3 s (82 ms, 435 ms, 792 ms)
ColumnarToRow
number of output rows: 2,933,545
number of input batches: 722
BroadcastHashJoin
number of output rows: 271
Filter
number of output rows: 196,293
Project
Project
BroadcastExchange
time to broadcast: 3 ms
time to build: 19 ms
time to collect: 528 ms
number of output rows: 271
data size: 4.0 MiB
BroadcastHashJoin
number of output rows: 555
Project
HashAggregate
time in aggregation build total (min, med, max)
1.9 s (53 ms, 358 ms, 712 ms)
peak memory total (min, med, max)
1280.0 KiB (256.0 KiB, 256.0 KiB, 256.0 KiB)
number of output rows: 28
Exchange
shuffle records written: 28
shuffle write time total (min, med, max)
49 ms (4 ms, 7 ms, 17 ms)
records read: 28
local bytes read: 852.0 B
fetch wait time: 0 ms
remote bytes read: 1134.0 B
local blocks read: 2
remote blocks read: 3
data size total (min, med, max)
896.0 B (128.0 B, 192.0 B, 192.0 B)
number of partitions: 200
remote reqs duration: 14 ms
shuffle bytes written total (min, med, max)
1986.0 B (282.0 B, 426.0 B, 426.0 B)

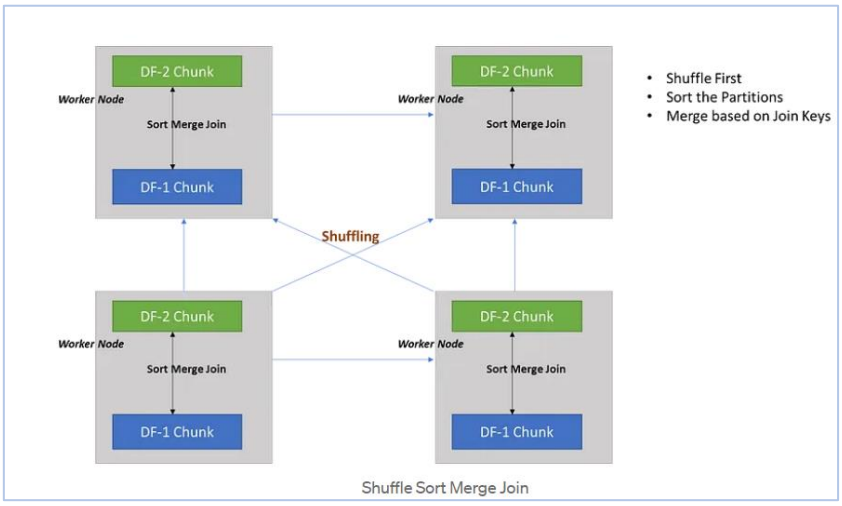


Από τα δύο αυτά στοιχεία, αρχικά, παρατηρούμε το διάβασμα των 2 .csv αρχείων και του .parquet αρχείου. Επιπλέον, βλέπουμε ότι γίνεται **`broadcast join`** και για τα **`revgecoding_df`**, **`income_df`** (για την εκτέλεση του ***`filtered_income_df = income_df.filter(income_df['Zip code'].isin(unique_zip_codes))`***). Σημαντικό είναι να σημειώσουμε, ότι το Spark εκτελεί τα **`filter`** πριν από τα **`join`** για οικονομία. Επιπλέον, κάνει αυτόματα και κάποιες βελτιστοποιήσεις (WholeStageCodegen, AQEShuffleRead). Παρατηρούμε, ακόμα, το Broadcast Join μεταξύ του **`crime_df_trunc`** και του **`revgecoding_df`**, που ακολουθείται από **`shuffle`** (exchange), που αφορά την συνένωση των δεδομένων για την εκτέλεση των queries (για το **`join`** δεν γίνεται κανένα **`shuffle`**). Έπεται η εκτέλεση των queries: το TakeOrderedAndProject αντιστοιχεί στα **`sort`** και **`select`** και το HashAggregate στο **`group by`**.

Είναι σημαντικό να σημειώσουμε, ότι το Physical Plan επιλέγεται *κατά την διάρκεια* του Runtime, οπότε το Physical Plan του Spark History Server μπορεί τελικά να διαφέρει από αυτό που παίρνουμε με την **`explain()`** (**`isFinalPlan = False`**).

Σχόλιο: Στη συνέχεια, θα παραλείψουμε τα αρχικά διαβάσματα των αρχείων .csv και τα τελικά βήματα (Exchange, AQEShuffleRead, HashAggregate, TakeOrderAndProject και AdaptiveSparkPlan), καθώς είναι **κοινά** για όλα τα Joins.

- **Merge Join:** Γίνεται shuffle των δεδομένων στους κόμβους, τα δύο DataFrames ταξινομούνται με βάση το join key (**`LAT`**, **`LON`**) και έπειτα, γίνεται το merge, οπότε συνδυάζονται οι γραμμές με το ίδιο key, δουλεύει καλά όταν τα keys είναι sorted ή indexed.



Shuffle Sort Merge Join

Η αλλαγή στον κώδικα είναι:

```
crimes_zip_joined=crime_df_trunc.hint("MERGE").join(revgecoding_df, ['LAT','LON'],
                                                    'inner')
crimes_zip_joined.explain()
```

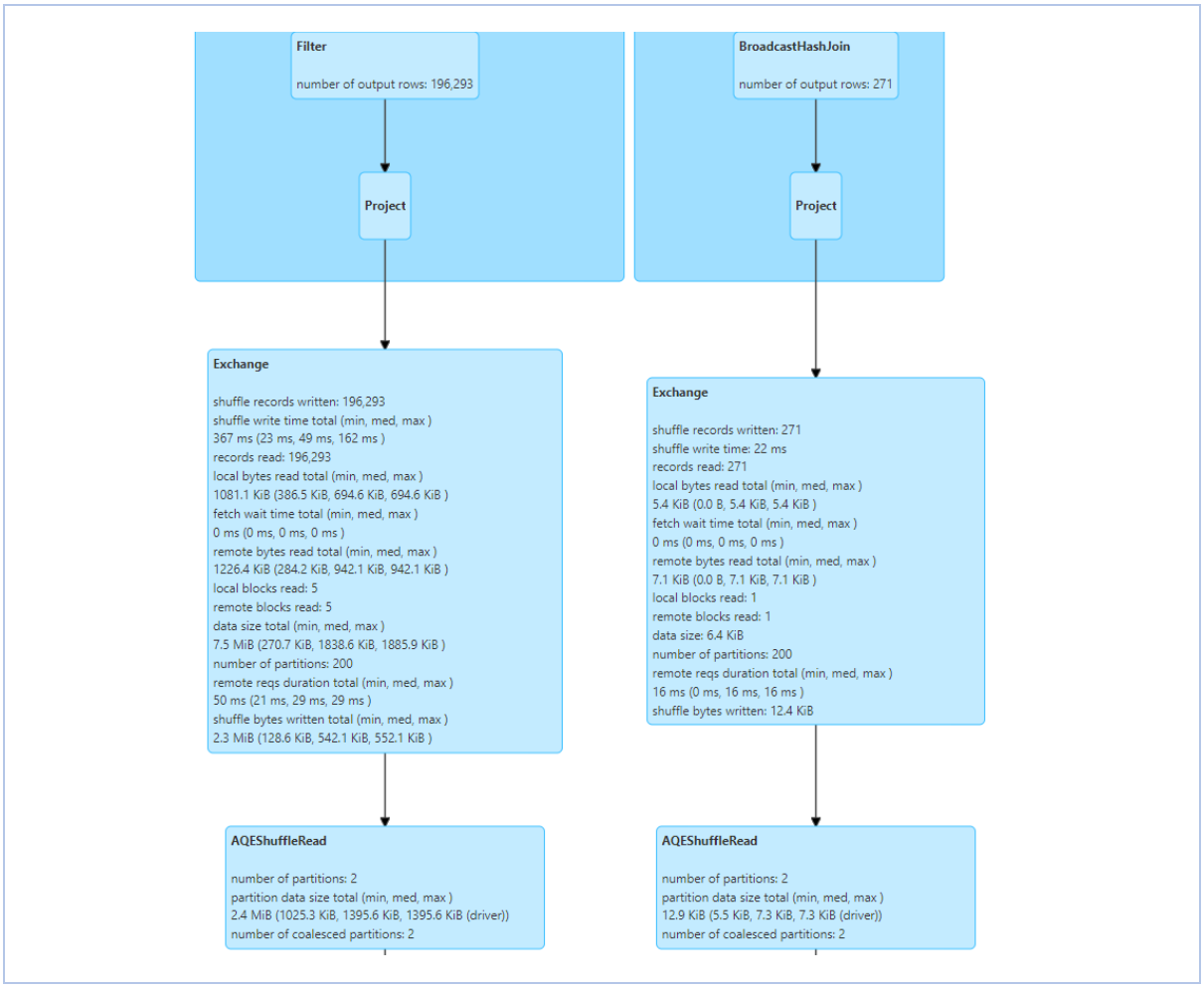
<u>Σχόλιο:</u> Ο χρόνος εκτέλεσης είναι **40.58s**.

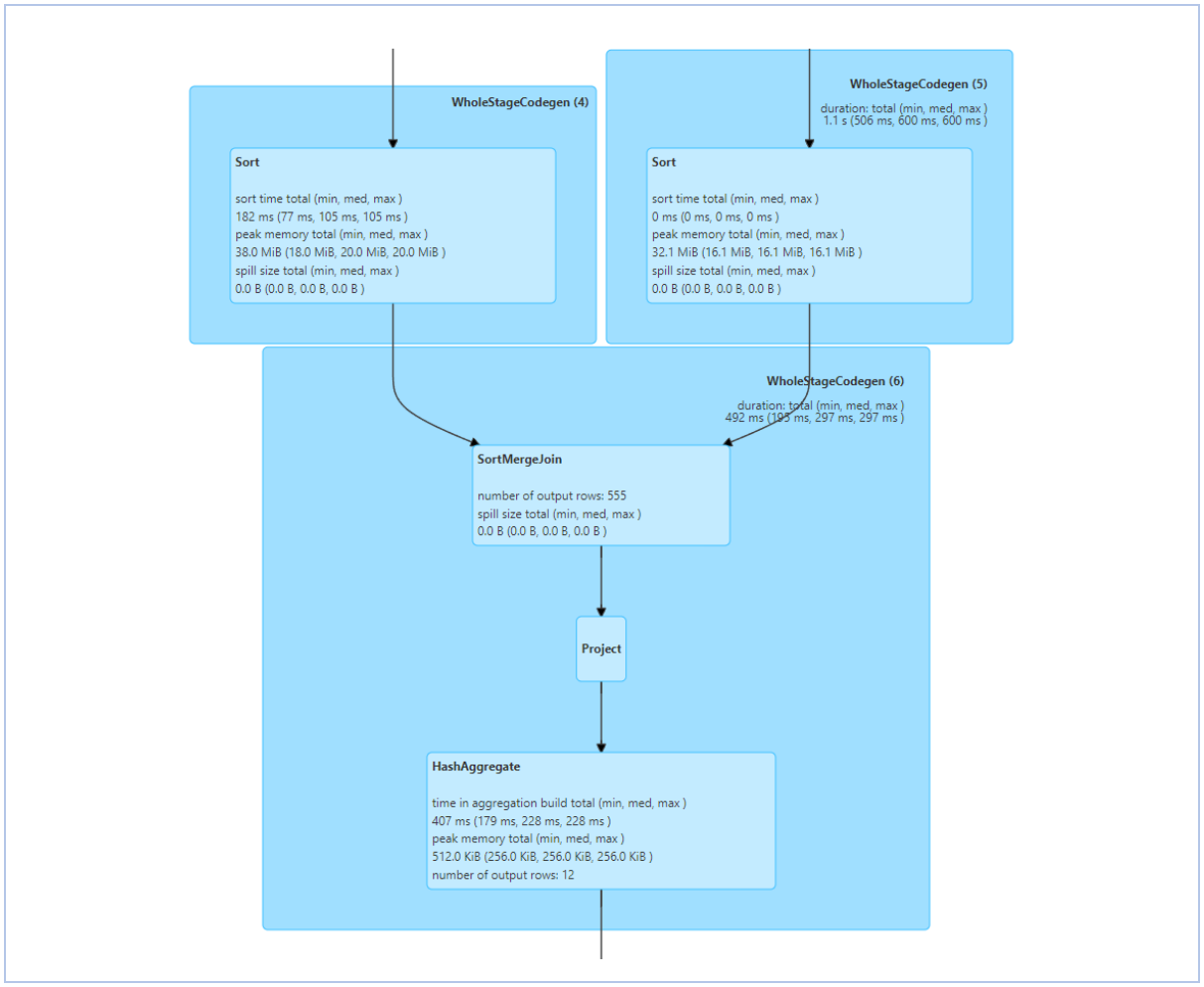
Η μέθοδος **`explain()`** δίνει:

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Project [LAT#26, LON#27, DR_NO#0, Vict Descent#13, ZIPcode#73]
   +- SortMergeJoin [knownfloatingpointnormalized(normalizenanandzero(LAT#26)), knownfloatingpointnormalized(normalizena
nandzero(LON#27))], [knownfloatingpointnormalized(normalizenanandzero(LAT#71)), knownfloatingpointnormalized(normalizena
nandzero(LON#72))], Inner
      :- Sort [knownfloatingpointnormalized(normalizenanandzero(LAT#26)) ASC NULLS FIRST, knownfloatingpointnormalized(n
ormalizenanandzero(LON#27)) ASC NULLS FIRST], false, 0
      :  +- Exchange hashpartitioning(knownfloatingpointnormalized(normalizenanandzero(LAT#26)), knownfloatingpointnorma
lized(normalizenanandzero(LON#27)), 200), ENSURE_REQUIREMENTS, [plan_id=27]
      :     +- Project [DR_NO#0, LAT#26, LON#27, Vict Descent#13]
      :        +- Filter ((((isnotnull(DATE OCC#2) AND atleastnnonnulls(1, Vict Descent#13)) AND (year(DATE OCC#2) = 201
5)) AND isnotnull(LAT#26)) AND isnotnull(LON#27))
      :           +- FileScan parquet [DR_NO#0,DATE OCC#2,Vict Descent#13,LAT#26,LON#27] Batched: true, DataFilters: [is
notnull(DATE OCC#2), atleastnnonnulls(1, Vict Descent#13), (year(DATE OCC#2) = 2015), isnotnul..., Format: Parquet, Loca
tion: InMemoryFileIndex(1 paths)[hdfs://okeanos-master:54310/user/user/output/crime_df.parquet], PartitionFilters: [], P
ushedFilters: [IsNotNull(`DATE OCC`), IsNotNull(LAT), IsNotNull(LON)], ReadSchema: struct<DR_NO:int,DATE OCC:date,Vict D
escent:string,LAT:double,LON:double>
      +- Sort [knownfloatingpointnormalized(normalizenanandzero(LAT#71)) ASC NULLS FIRST, knownfloatingpointnormalized(n
ormalizenanandzero(LON#72)) ASC NULLS FIRST], false, 0
         +- Exchange hashpartitioning(knownfloatingpointnormalized(normalizenanandzero(LAT#71)), knownfloatingpointnorma
lized(normalizenanandzero(LON#72)), 200), ENSURE_REQUIREMENTS, [plan_id=28]
            +- Filter ((atleastnnonnulls(3, LAT#71, LON#72, ZIPcode#73) AND isnotnull(LAT#71)) AND isnotnull(LON#72))
               +- FileScan csv [LAT#71,LON#72,ZIPcode#73] Batched: false, DataFilters: [atleastnnonnulls(3, LAT#71, LON#
72, ZIPcode#73), isnotnull(LAT#71), isnotnull(LON#72)], Format: CSV, Location: InMemoryFileIndex(1 paths)[hdfs://okeanos
-master:54310/user/user/revgecoding.csv], PartitionFilters: [], PushedFilters: [IsNotNull(LAT), IsNotNull(LON)], ReadSch
ema: struct<LAT:double,LON:double,ZIPcode:int>
```

Spark History Server

Από τον Spark History Server, το αρχικό διάβασμα των .csv αρχείων είναι το ίδιο και στη συνέχεια παίρνουμε:

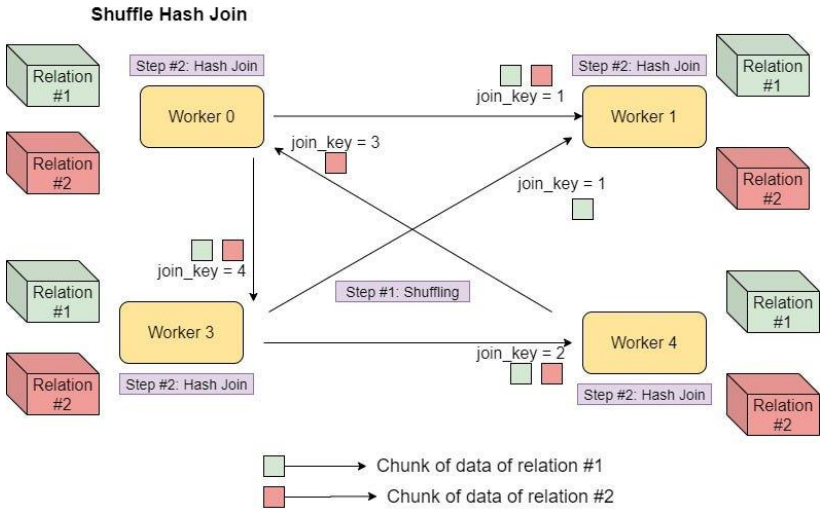




Για τα **revgecoding_df** και **income_df** γίνεται και πάλι **broadcast join**, καθώς το μικρό μέγεθος του **income_df** την καθιστά ως την πιο συμφέρουσα επιλογή.

Παρατηρώντας τις αλλαγές στο Physical Plan με το Merge Join, διακρίνουμε ότι εδώ έχουμε δύο **shuffles** για το **join**, για να διαμοιραστούν τα DataFrames στους δύο κόμβους. Ύστερα, ταξινομούνται στον καθένα ξεχωριστά και ακολουθεί το **join** για την συνένωσή τους.

- **Shuffle Hash Join:** Οι γραμμές με το ίδιο join key πάνε στον ίδιο κόμβο, όπου γίνεται τοπικά join και τελικά συνενώνονται τα αποτελέσματα.



Ο κώδικας γίνεται:

```
crimes_zip_joined=crime_df_trunc.hint("SHUFFLE_HUSH").join(revgecoding_df,['LAT',
                                                                    'LON'], 'inner')
crimes_zip_joined.explain()
```

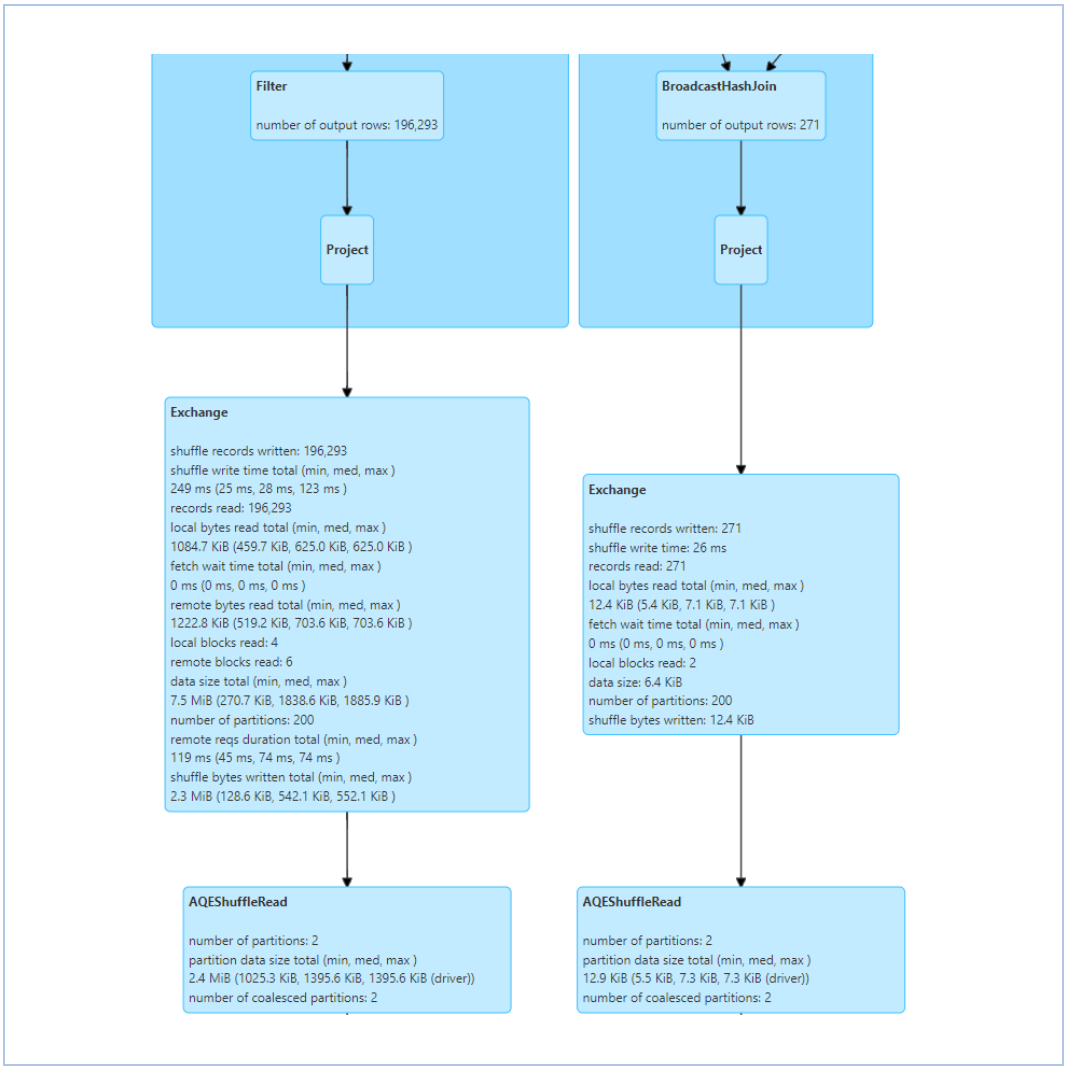
<u>Σχόλιο:</u> Ο χρόνος εκτέλεσης είναι **45.12s**.

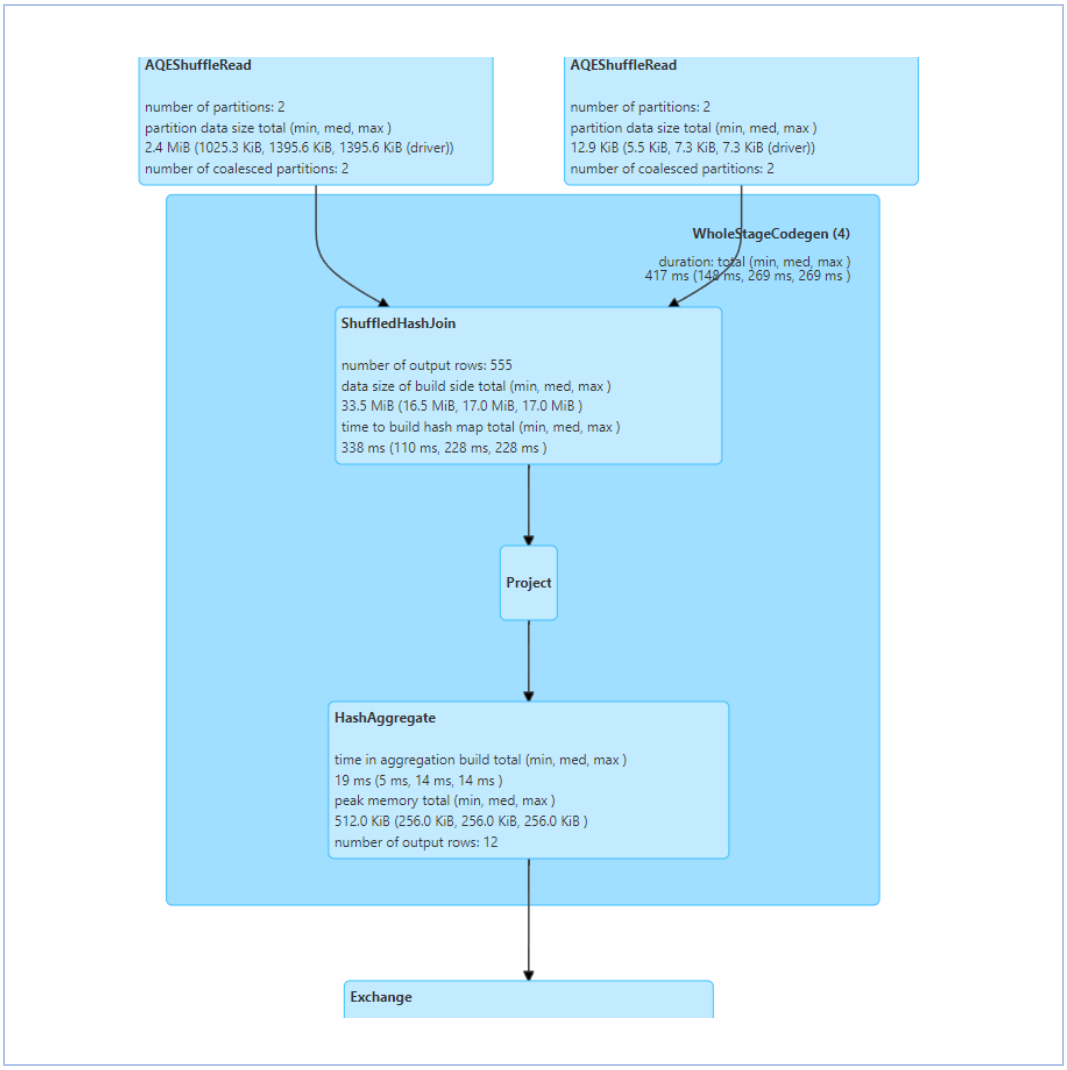
Η μέθοδος **`explain()`** δίνει:

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Project [LAT#26, LON#27, DR_NO#0, Vict Descent#13, ZIPcode#73]
   +- ShuffledHashJoin [knownfloatingpointnormalized(normalizenanandzero(LAT#26)), knownfloatingpointnormalized(normaliz
enanandzero(LON#27))], [knownfloatingpointnormalized(normalizenanandzero(LAT#71)), knownfloatingpointnormalized(normaliz
enanandzero(LON#72))], Inner, BuildLeft
      :- Exchange hashpartitioning(knownfloatingpointnormalized(normalizenanandzero(LAT#26)), knownfloatingpointnormaliz
ed(normalizenanandzero(LON#27)), 200), ENSURE_REQUIREMENTS, [plan_id=27]
      :  +- Project [DR_NO#0, LAT#26, LON#27, Vict Descent#13]
      :     +- Filter ((((isnotnull(DATE OCC#2) AND atleastnnonnulls(1, Vict Descent#13)) AND (year(DATE OCC#2) = 2015))
 AND isnotnull(LAT#26)) AND isnotnull(LON#27))
      :        +- FileScan parquet [DR_NO#0,DATE OCC#2,Vict Descent#13,LAT#26,LON#27] Batched: true, DataFilters: [isnot
null(DATE OCC#2), atleastnnonnulls(1, Vict Descent#13), (year(DATE OCC#2) = 2015), isnotnul..., Format: Parquet, Locatio
n: InMemoryFileIndex(1 paths)[hdfs://okeanos-master:54310/user/user/output/crime_df.parquet], PartitionFilters: [], Push
edFilters: [IsNotNull(`DATE OCC`), IsNotNull(LAT), IsNotNull(LON)], ReadSchema: struct<DR_NO:int,DATE OCC:date,Vict Desc
ent:string,LAT:double,LON:double>
      +- Exchange hashpartitioning(knownfloatingpointnormalized(normalizenanandzero(LAT#71)), knownfloatingpointnormaliz
ed(normalizenanandzero(LON#72)), 200), ENSURE_REQUIREMENTS, [plan_id=28]
         +- Filter ((atleastnnonnulls(3, LAT#71, LON#72, ZIPcode#73) AND isnotnull(LAT#71)) AND isnotnull(LON#72))
            +- FileScan csv [LAT#71,LON#72,ZIPcode#73] Batched: false, DataFilters: [atleastnnonnulls(3, LAT#71, LON#72,
 ZIPcode#73), isnotnull(LAT#71), isnotnull(LON#72)], Format: CSV, Location: InMemoryFileIndex(1 paths)[hdfs://okeanos-ma
ster:54310/user/user/revgecoding.csv], PartitionFilters: [], PushedFilters: [IsNotNull(LAT), IsNotNull(LON)], ReadSchema
: struct<LAT:double,LON:double,ZIPcode:int>
```

Spark History Server

Από τα γραφικά του Spark History Server παίρνουμε αρχικά το ίδιο διάβασμα των .csv αρχείων και ύστερα:





Καθώς οι executors είναι μοιρασμένοι σε δύο κόμβους, προκύπτει πανομοιότυπη εκτέλεση με πριν.

- **Shuffle Replicate NL:** Το μικρότερο DataFrame γίνεται broadcasted σε όλους τους κόμβους, ενώ το άλλο χωρίζεται σε partitions με βάση το join key και μοιράζεται στους κόμβους. Στη συνέχεια, κάθε κόμβος εκτελεί καρτεσιανό γινόμενο του partition του μεγαλύτερου dataset και ολόκληρου του μικρότερου dataset που είναι διαθέσιμο τοπικά και στο τέλος συνενώνονται τα αποτελέσματα. Η συγκεκριμένη μέθοδος υποστηρίζει και μη equi-joins.

```
for record_1 in relation_1:
  for record_2 in relation_2:
    # join condition is executed
```

Ο κώδικας γίνεται:

```
crimes_zip_joined=crime_df_trunc.hint("SHUFFLE_REPLICATE_NL").join(revgecoding_df,
                                        ['LAT', 'LON'], 'inner')
crimes_zip_joined.explain()
```

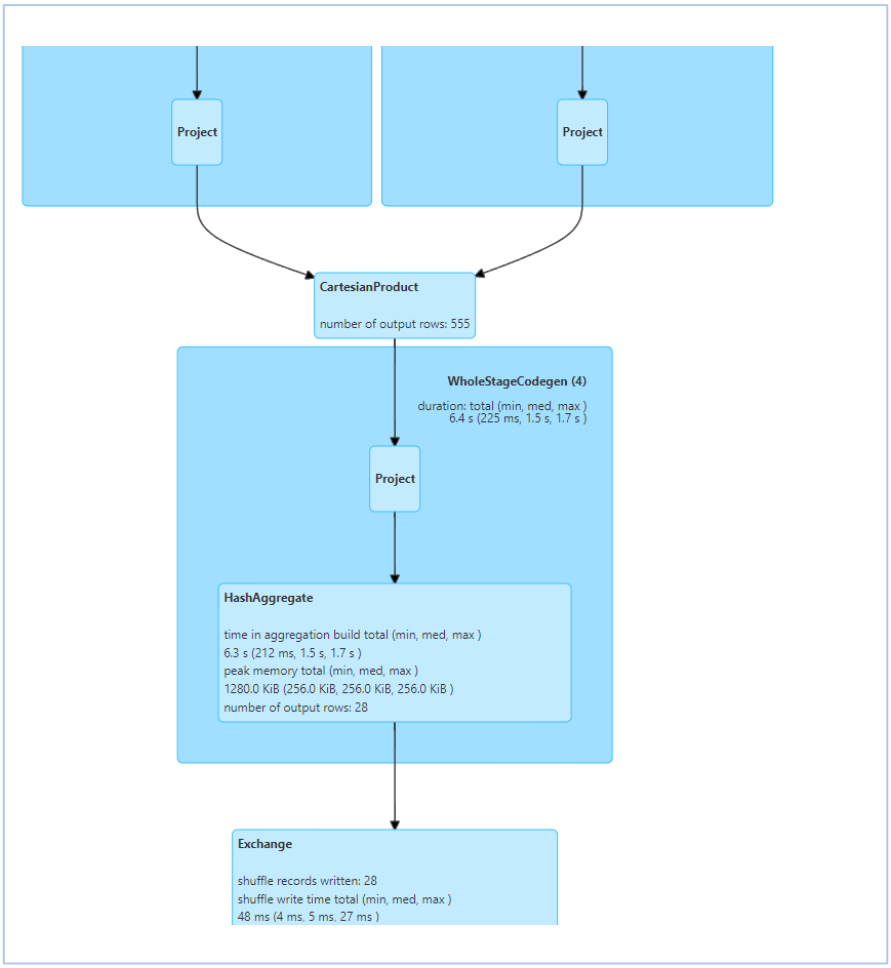
Σχόλιο: Ο χρόνος εκτέλεσης είναι **245s**.

Η μέθοδος **`explain()`** δίνει:

```
== Physical Plan ==
*(3) Project [LAT#26, LON#27, DR_NO#0, Vict Descent#13, ZIPcode#73]
+- CartesianProduct ((knownfloatingpointnormalized(normalizenanandzero(LAT#26)) = knownfloatingpointnormalized(normalize
nanandzero(LAT#71))) AND (knownfloatingpointnormalized(normalizenanandzero(LON#27)) = knownfloatingpointnormalized(norma
lizenanandzero(LON#72))))
   :- *(1) Project [DR_NO#0, LAT#26, LON#27, Vict Descent#13]
   :  +- *(1) Filter ((((isnotnull(DATE OCC#2) AND atleastnnonnulls(1, Vict Descent#13)) AND (year(DATE OCC#2) = 2015))
AND isnotnull(LAT#26)) AND isnotnull(LON#27))
   :     +- *(1) ColumnarToRow
   :        +- FileScan parquet [DR_NO#0,DATE OCC#2,Vict Descent#13,LAT#26,LON#27] Batched: true, DataFilters: [isnotnul
l(DATE OCC#2), atleastnnonnulls(1, Vict Descent#13), (year(DATE OCC#2) = 2015), isnotnul..., Format: Parquet, Location:
InMemoryFileIndex(1 paths)[hdfs://okeanos-master:54310/user/user/output/crime_df.parquet], PartitionFilters: [], PushedF
ilters: [IsNotNull(`DATE OCC`), IsNotNull(LAT), IsNotNull(LON)], ReadSchema: struct<DR_NO:int,DATE OCC:date,Vict Descent
:string,LAT:double,LON:double>
   +- *(2) Filter ((atleastnnonnulls(3, LAT#71, LON#72, ZIPcode#73) AND isnotnull(LAT#71)) AND isnotnull(LON#72))
      +- FileScan csv [LAT#71,LON#72,ZIPcode#73] Batched: false, DataFilters: [atleastnnonnulls(3, LAT#71, LON#72, ZIPco
de#73), isnotnull(LAT#71), isnotnull(LON#72)], Format: CSV, Location: InMemoryFileIndex(1 paths)[hdfs://okeanos-master:5
4310/user/user/revgecoding.csv], PartitionFilters: [], PushedFilters: [IsNotNull(LAT), IsNotNull(LON)], ReadSchema: stru
ct<LAT:double,LON:double,ZIPcode:int>
```

Spark History Server

Από τον Spark History Server παίρνουμε:



Παρατηρούμε ότι το Spark επιλέγει να κρατήσει τα δύο DataFrames στον ίδιο κόμβο και να εκτελέσει καρτεσιανό γινόμενο ως κάποιο optimization. Εφόσον τα DataFrames είναι ολόκληρα, χρειάζεται αρκετή ώρα.

<u>Συμπέρασμα:</u> Η πιο χρονοβόρα διεργασία του Spark είναι το ***shuffling***, δηλαδή ο διαμοιρασμός των δεδομένων στους κόμβους και στους executors, διότι αυξάνει το I/O. Επομένως, είναι λογικό η μέθοδος Broadcast Join, που δεν έχει επιπλέον shuffling, να είναι η πιο γρήγορη από τις μεθόδους join. Τα SortMerge και ShuffleHash έχουν παρόμοιο πλάνο εκτέλεσης. Ωστόσο, παρατηρούμε ότι το revgecoding.csv είναι ταξινομημένο με βάση το '**LAT**' σε αύξουσα σειρά και για ίδιες τιμές '**LAT**' είναι ταξινομημένο κατά '**LON**' σε φθίνουσα σειρά. Οπότε το sort, που γίνεται στην περίπτωση του SortMerge Join, εκτελείται γρήγορα, καθιστώντας το καταλληλότερο από το ShuffleHash. Τέλος, το ShuffleReplicateNL, δεν αξιοποιεί τα χαρακτηριστικά του συστήματος και των δεδομένων μας και κάνει ένα αχρείαστο καρτεσιανό γινόμενο, καταλήγοντας σε μεγάλο χρόνο εκτέλεσης.

QUERY 4

Θα εκτελέσουμε και θα συγκρίνουμε τα joins του πρώτου ερωτήματος μόνο, καθώς για το δεύτερο χρησιμοποιήσαμε **Cross Join**.

- **Broadcast Join**

Ο κώδικας και για τα δύο queries τροποποιείται ως εξής:

```
INNER JOIN (SELECT /*+ BROADCAST(p) */ * FROM Precincts p) p ON c.AREA = p.PREC
```

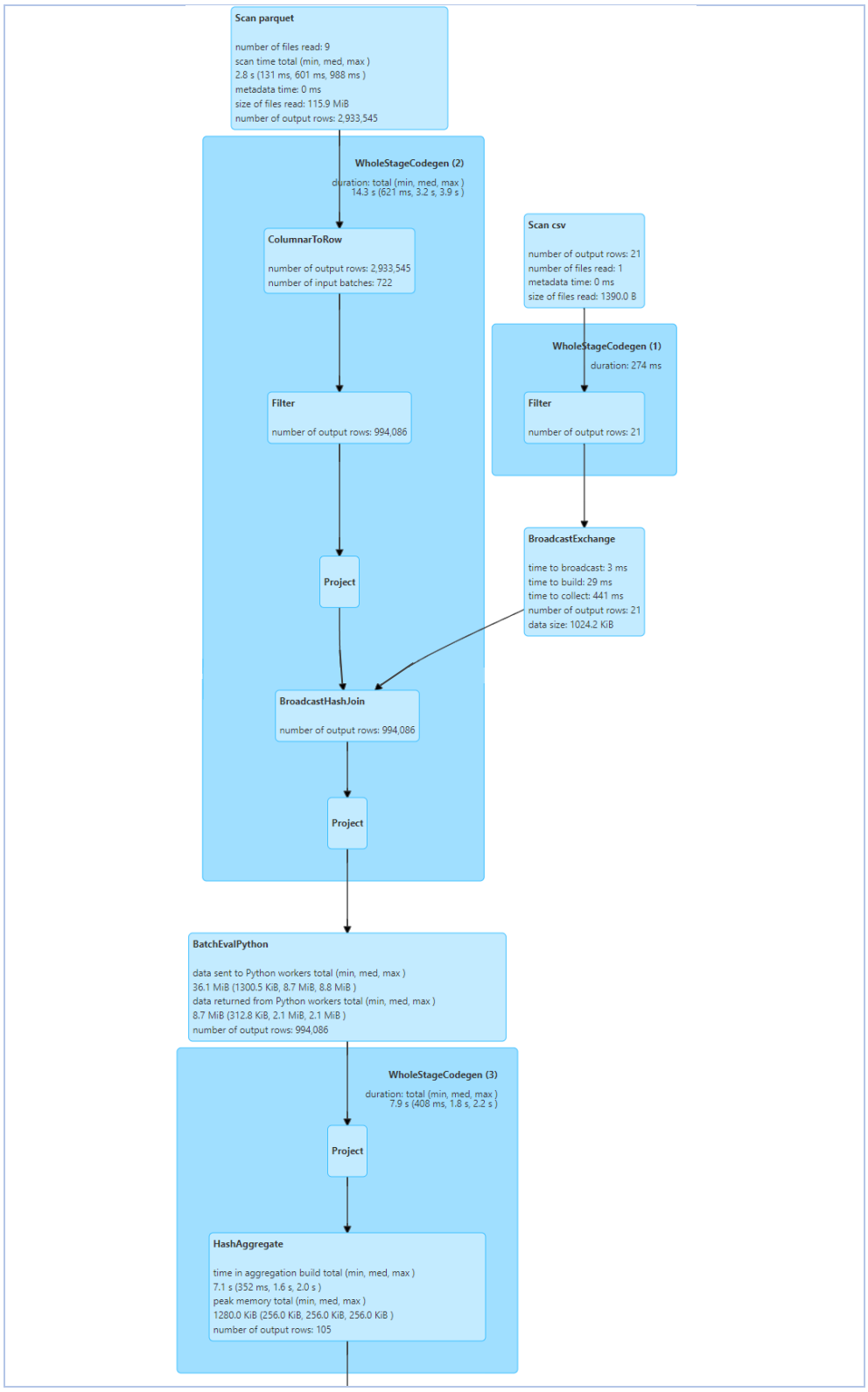
Σχόλιο: Ο χρόνος εκτέλεσης είναι **22.53s**.

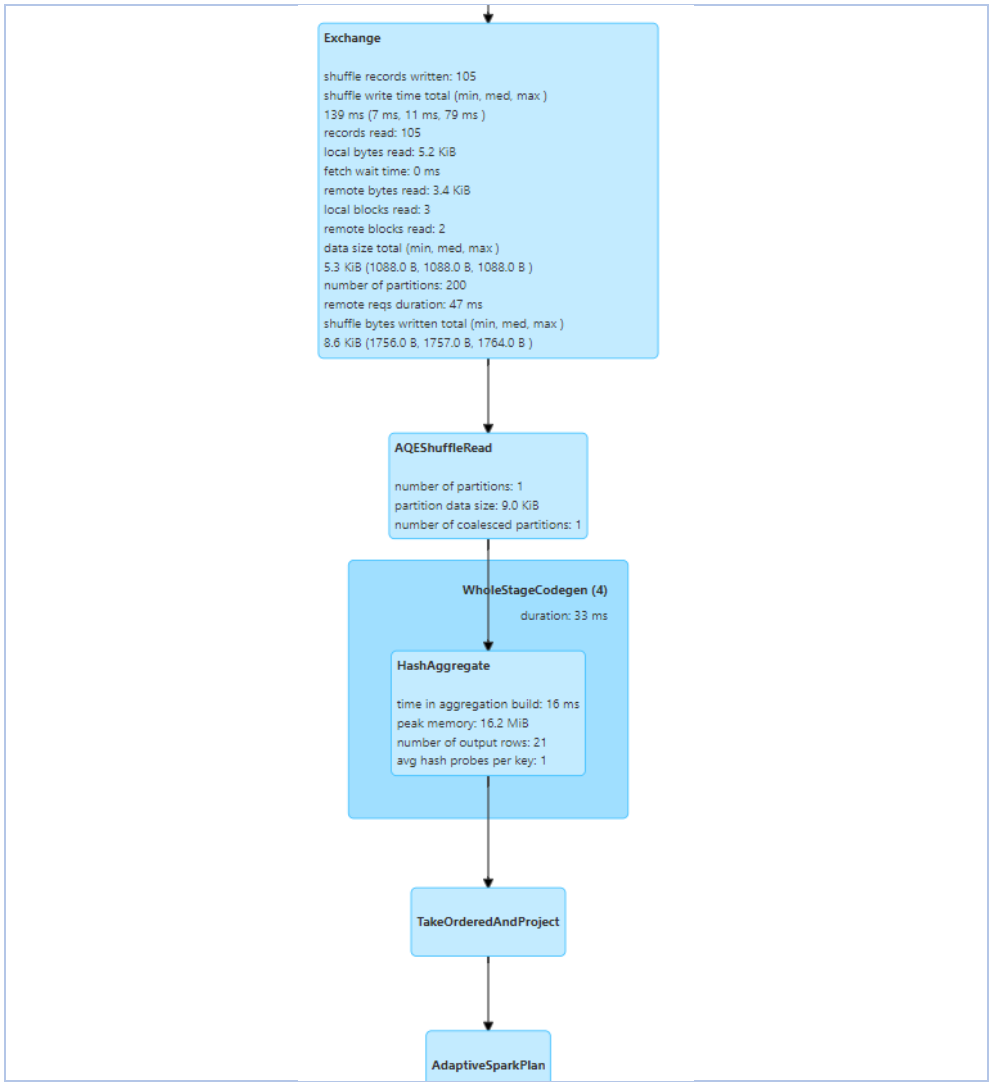
Η μέθοδος **`explain()`** δίνει:

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [Count#143L DESC NULLS LAST], true, 0
   +- Exchange rangepartitioning(Count#143L DESC NULLS LAST, 200), ENSURE_REQUIREMENTS, [plan_id=526]
      +- HashAggregate(keys=[Division#59], functions=[avg(pythonUDF0#208), count(1)])
         +- Exchange hashpartitioning(Division#59, 200), ENSURE_REQUIREMENTS, [plan_id=523]
            +- HashAggregate(keys=[Division#59], functions=[partial_avg(pythonUDF0#208), partial_count(1)])
               +- Project [DIVISION#59, pythonUDF0#208]
                  +- BatchEvalPython [get_distance(LAT#26, LON#27, Y#57, X#56)#172], [pythonUDF0#208]
                     +- Project [LAT#26, LON#27, X#56, Y#57, DIVISION#59]
                        +- BroadcastHashJoin [AREA#4], [PREC#61], Inner, BuildRight, false
                           :- Project [AREA#4, LAT#26, LON#27]
                           :  +- Filter (((((isnotnull(LAT#26) AND isnotnull(LON#27)) AND NOT (LAT#26 = 0.0)) AND NOT (L
ON#27 = 0.0)) AND isnotnull(Weapon Used Cd#16)) AND isnotnull(AREA#4))
                           :     +- FileScan parquet [AREA#4,Weapon Used Cd#16,LAT#26,LON#27] Batched: true, DataFilters
: [isnotnull(LAT#26), isnotnull(LON#27), NOT (LAT#26 = 0.0), NOT (LON#27 = 0.0), isnotnull(Weapon U..., Format: Parquet,
 Location: InMemoryFileIndex(1 paths)[hdfs://okeanos-master:54310/user/user/output/crime_df.parquet], PartitionFilters:
[], PushedFilters: [IsNotNull(LAT), IsNotNull(LON), Not(EqualTo(LAT,0.0)), Not(EqualTo(LON,0.0)), IsNotNull(`Weapon ...,
 ReadSchema: struct<AREA:int,Weapon Used Cd:int,LAT:double,LON:double>
                           +- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[3, int, false] as bigint)),f
alse), [plan_id=516]
                              +- Filter isnotnull(PREC#61)
                                 +- FileScan csv [X#56,Y#57,DIVISION#59,PREC#61] Batched: false, DataFilters: [isnotnull
(PREC#61)], Format: CSV, Location: InMemoryFileIndex(1 paths)[hdfs://okeanos-master:54310/user/user/LAPD_Police_Stations
.csv], PartitionFilters: [], PushedFilters: [IsNotNull(PREC)], ReadSchema: struct<X:double,Y:double,DIVISION:string,PREC
:int>
```

Spark History Server

Από το γραφικό περιβάλλον του Spark History Server παίρνουμε:





Όπως και πριν, έχουμε το αρχικό διάβασμα των αρχείων, το Broadcast Join του μικρότερου DataFrame, το τελικό **`shuffle`**, **`groupby`** και **`orderby`**. Παρόλο που έχουμε δύο queries, αφού αφορούν join των ίδιων DataFrames, αυτό γίνεται μόνο **μία** φορά.

Σχόλιο: Στα παρακάτω βήματα, θα παραλείπουμε το αρχικό διάβασμα των αρχείων και τα τελικά στάδια, καθώς είναι ίδια για όλα τα Joins.

- **Sort Merge Join**

Στον κώδικα προστίθεται:

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", "-1")
spark.conf.set("spark.sql.join.preferSortMergeJoin", "true")
```

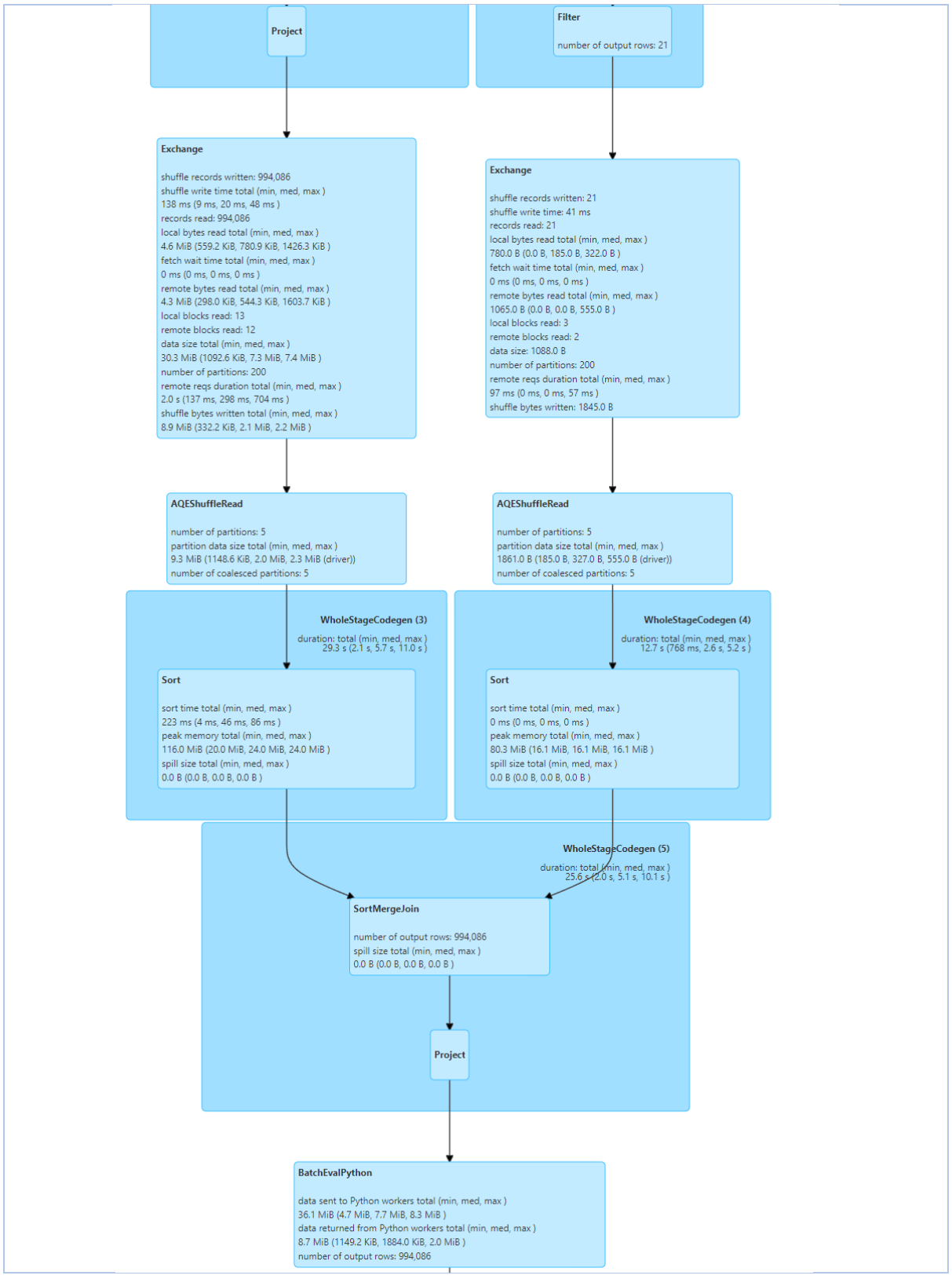
Σχόλιο: Ο χρόνος εκτέλεσης είναι **29.84s**.

Η μέθοδος **`explain()`** δίνει:

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [Count#137L DESC NULLS LAST], true, 0
   +- Exchange rangepartitioning(Count#137L DESC NULLS LAST, 200), ENSURE_REQUIREMENTS, [plan_id=656]
      +- HashAggregate(keys=[Division#59], functions=[avg(pythonUDF0#202), count(1)])
         +- Exchange hashpartitioning(Division#59, 200), ENSURE_REQUIREMENTS, [plan_id=653]
            +- HashAggregate(keys=[Division#59], functions=[partial_avg(pythonUDF0#202), partial_count(1)])
               +- Project [DIVISION#59, pythonUDF0#202]
                  +- BatchEvalPython [get_distance(LAT#26, LON#27, Y#57, X#56)#166], [pythonUDF0#202]
                     +- Project [LAT#26, LON#27, X#56, Y#57, DIVISION#59]
                        +- SortMergeJoin [AREA#4], [PREC#61], Inner
                           :- Sort [AREA#4 ASC NULLS FIRST], false, 0
                           :  +- Exchange hashpartitioning(AREA#4, 200), ENSURE_REQUIREMENTS, [plan_id=643]
                           :     +- Project [AREA#4, LAT#26, LON#27]
                           :        +- Filter (((((isnotnull(LAT#26) AND isnotnull(LON#27)) AND NOT (LAT#26 = 0.0)) AND
NOT (LON#27 = 0.0)) AND isnotnull(Weapon Used Cd#16)) AND isnotnull(AREA#4))
                           :           +- FileScan parquet [AREA#4,Weapon Used Cd#16,LAT#26,LON#27] Batched: true, DataF
ilters: [isnotnull(LAT#26), isnotnull(LON#27), NOT (LAT#26 = 0.0), NOT (LON#27 = 0.0), isnotnull(Weapon U..., Format: Pa
rquet, Location: InMemoryFileIndex(1 paths)[hdfs://okeanos-master:54310/user/user/output/crime_df.parquet], PartitionFil
ters: [], PushedFilters: [IsNotNull(LAT), IsNotNull(LON), Not(EqualTo(LAT,0.0)), Not(EqualTo(LON,0.0)), IsNotNull(`Weapo
n ..., ReadSchema: struct<AREA:int,Weapon Used Cd:int,LAT:double,LON:double>
                           +- Sort [PREC#61 ASC NULLS FIRST], false, 0
                              +- Exchange hashpartitioning(PREC#61, 200), ENSURE_REQUIREMENTS, [plan_id=644]
                                 +- Filter isnotnull(PREC#61)
                                    +- FileScan csv [X#56,Y#57,DIVISION#59,PREC#61] Batched: false, DataFilters: [isnotn
ull(PREC#61)], Format: CSV, Location: InMemoryFileIndex(1 paths)[hdfs://okeanos-master:54310/user/user/LAPD_Police_Stati
ons.csv], PartitionFilters: [], PushedFilters: [IsNotNull(PREC)], ReadSchema: struct<X:double,Y:double,DIVISION:string,P
REC:int>
```

Spark History Server

Από το γραφικό περιβάλλον του Spark History Server παίρνουμε:



Και πάλι παρατηρούμε τα δύο shuffles και τα τοπικά sorts, τα οποία ακολουθούνται από το ολικό Merge.

- **Shuffle Hash και Shuffle Replicate NL Joins:** Δεν καταφέραμε να εκτελέσουμε τα συγκεκριμένα Joins, διότι, παρά τις τροποποιήσεις στον κώδικα, το Spark εκτελεί για λόγους βελτιστοποίησης είτε Broadcast είτε SortMerge Join.

Συμπέρασμα: Το Broadcast Join δίνει και πάλι καλύτερο χρόνο εκτέλεσης, αποτέλεσμα που συνάδει με όσα είπαμε πριν. Συνεπώς, συνολικά μπορούμε να πούμε, πως στις παραπάνω περιπτώσεις η πιο κατάλληλη επιλογή είναι το **Broadcast Join**, καθώς το ένα DataFrame είναι αρκετά μικρό και χωράει στην μνήμη.

Github

Στον παρακάτω σύνδεσμο μπορείτε να βρείτε όλους τους κώδικες που υλοποιήσαμε:

https://github.com/georgia-bous/AdvancedDB.git