**ΠΡΟΗΓΜΕΝΑ ΘΕΜΑΤΑ ΒΑΣΕΩΝ ΔΕΔΟΜΕΝΩΝ**

**ΑΝΑΦΟΡΑ ΕΞΑΜΗΝΙΑΙΑΣ ΕΡΓΑΣΙΑΣ**

Δελατόλας Γρηγόρης (03119202)

Παπανικόλας Εμμανουήλ (03119063)

ΙΑΝΟΥΑΡΙΟΣ 2024

Contents

[ΣΧΟΛΙΑ 3](#_Toc155911608)

[ΖΗΤΟΥΜΕΝΟ 1 3](#_Toc155911609)

[ΖΗΤΟΥΜΕΝΟ 2 4](#_Toc155911610)

[ΖΗΤΟΥΜΕΝΟ 3 5](#_Toc155911611)

[Dataframe 6](#_Toc155911612)

[Σύγκριση 8](#_Toc155911613)

[ΖΗΤΟΥΜΕΝΟ 4 8](#_Toc155911614)

[ΖΗΤΟΥΜΕΝΟ 5 10](#_Toc155911615)

[Αποτέλεσμα 11](#_Toc155911616)

[2 Executors 11](#_Toc155911617)

[4 Executors 12](#_Toc155911618)

[Ανάλυση 12](#_Toc155911619)

[ΖΗΤΟΥΜΕΝΟ 6 13](#_Toc155911620)

[Ερώτημα α 13](#_Toc155911621)

[1ο Μέρος - Dataframe 13](#_Toc155911622)

[2ο Μέρος – Dataframe 13](#_Toc155911623)

[1ο Μέρος – SQL 13](#_Toc155911624)

[2o Μέρος – SQL 14](#_Toc155911625)

[Ερώτημα β 1ο μέρος - Dataframe 2ο μέρος - Dataframe 15](#_Toc155911626)

[1ο Μέρος – SQL 16](#_Toc155911627)

[2ο Μέρος – SQL 16](#_Toc155911628)

[Σχόλιο 17](#_Toc155911629)

[ΖΗΤΟΥΜΕΝΟ 7 17](#_Toc155911630)

[BROADCAST 17](#_Toc155911631)

[MERGE 17](#_Toc155911632)

[SHUFFLE\_HASH 17](#_Toc155911633)

[SHUFFLE\_REPLICATE\_NL 17](#_Toc155911634)

[QUERY 4 - Μέρος 1 17](#_Toc155911635)

[QUERY 4 - Μέρος 2 18](#_Toc155911636)

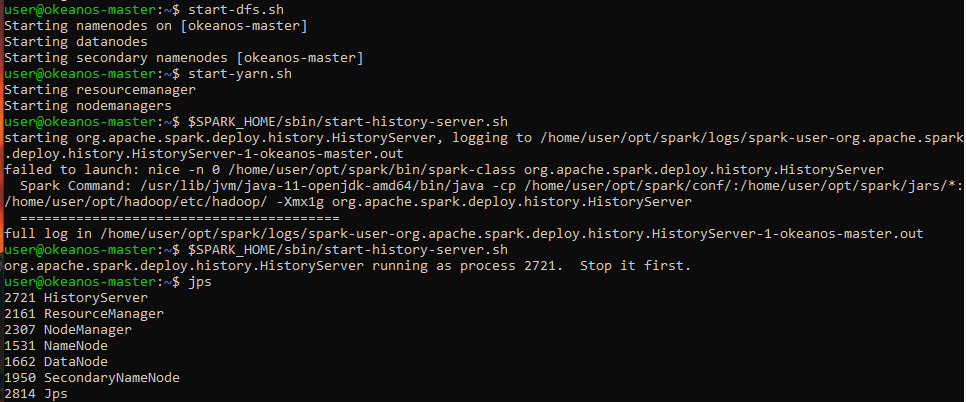
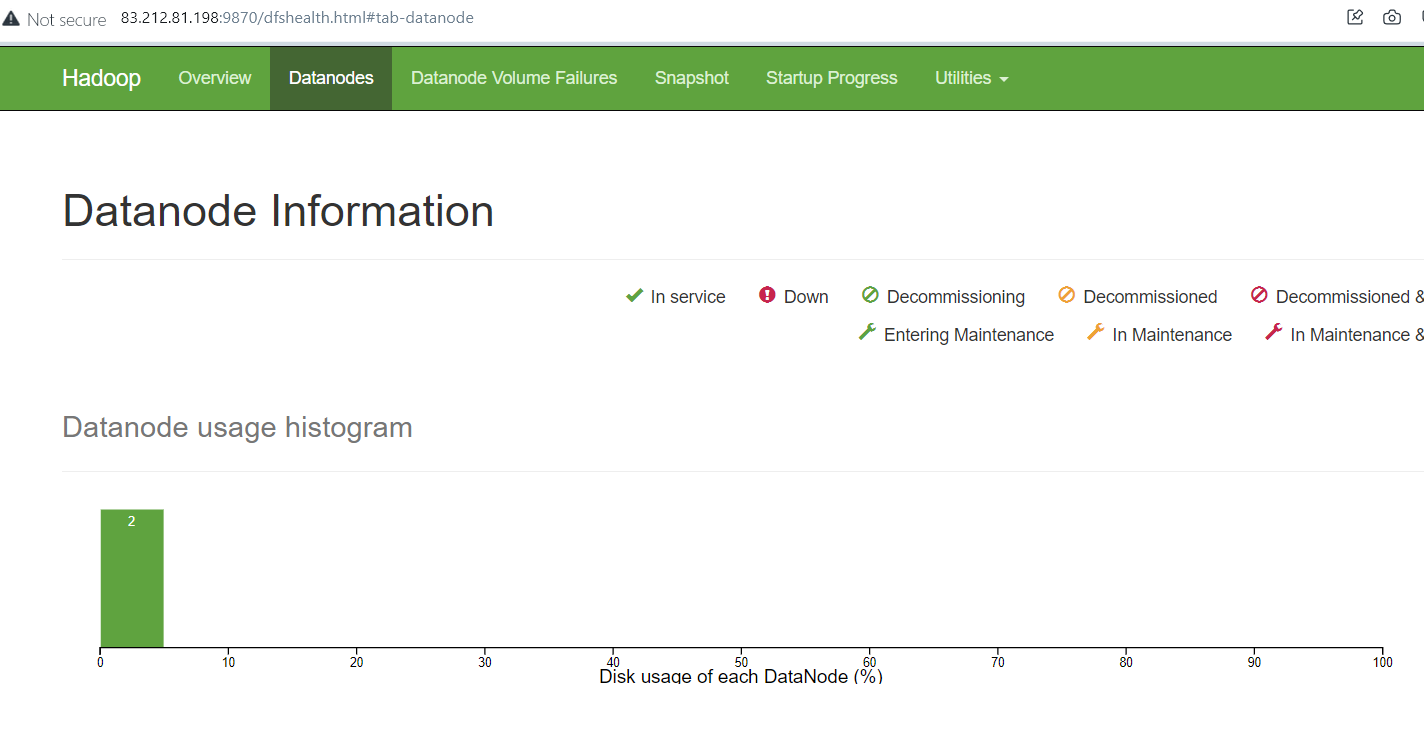
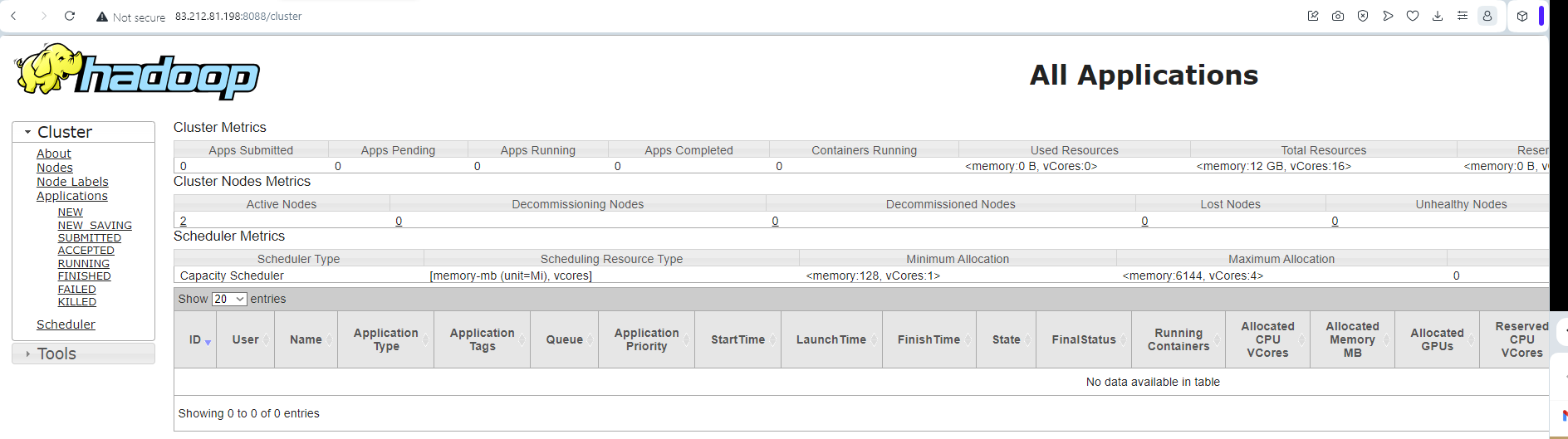
[Query 3 19](#_Toc155911637)

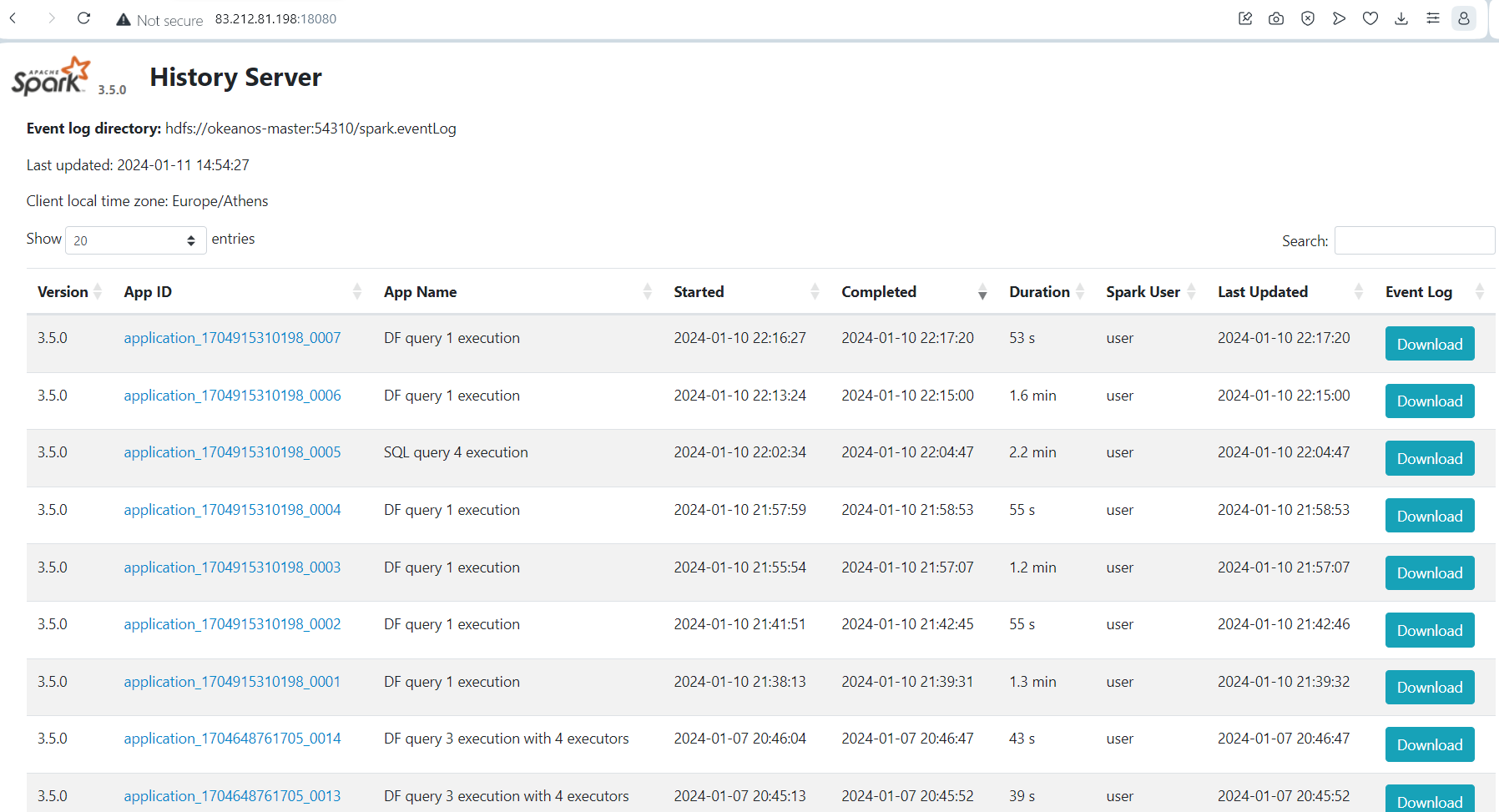
# ΣΧΟΛΙΑ

Όλοι οι κώδικες είναι διαθέσιμοι στο github της ομάδας:

# ΖΗΤΟΥΜΕΝΟ 1

Ακολουθήσαμε τις οδηγίες του εργαστηριακού οδηγού και εγκαταστήσαμε και διαμορφώσαμε κατάλληλα την πλατφόρμα εκτέλεσης Apache Spark.

  
  
Παρατηρούμε την ορθή λειτουργία μέσω των UIs των DFS, Yarn kai History Server αντίστοιχα:



# ΖΗΤΟΥΜΕΝΟ 2

Κατεβάζουμε τα 2 datasets στο local directory. Τρέχουμε scp στο terminal του local pc, δηλαδή πριν συνδεθούμε σε κάποιο VM:

scp C:/temp/crime\_dataset.csv user@snf-39594.ok-kno.grnetcloud.net:/home/user/data

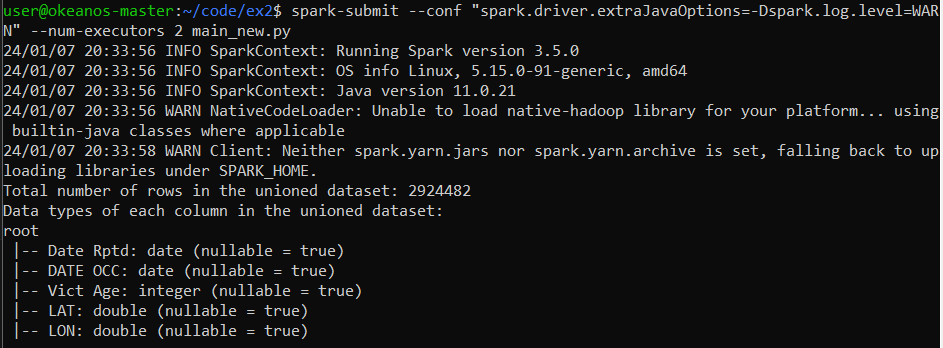
-Έπειτα φτιάχνουμε φάκελο στο hadoop

hdfs dfs -mkdir /crime\_data

-Ανεβάζουμε τα αρχεία στο hadoop.

hdfs dfs -copyFromLocal /home/user/data/ crime\_dataset.csv /main\_dataset

-Δημιουργώντας ένα αρχείο py τρέχουμε τα παρακάτω και παίρνουμε το ζητούμενο output:



Για τη δημιουργία του unioned\_df χρησιμοποιήσαμε τον κώδικα:

# Read the CSV files

df\_2010\_to\_2019 = spark.read.csv("hdfs://okeanos-master:54310/main\_dataset/2010\_to\_2019.csv", header=False, schema=crime\_schema)

df\_2020\_to\_present = spark.read.csv("hdfs://okeanos-master:54310/main\_dataset/2020\_to\_present.csv", header=False, schema=crime\_schema)

# Union the two datasets and Rename the 'AREA ' column in df2010 to 'AREA'

df\_2010\_to\_2019= df\_2010\_to\_2019.withColumnRenamed('AREA ', 'AREA')

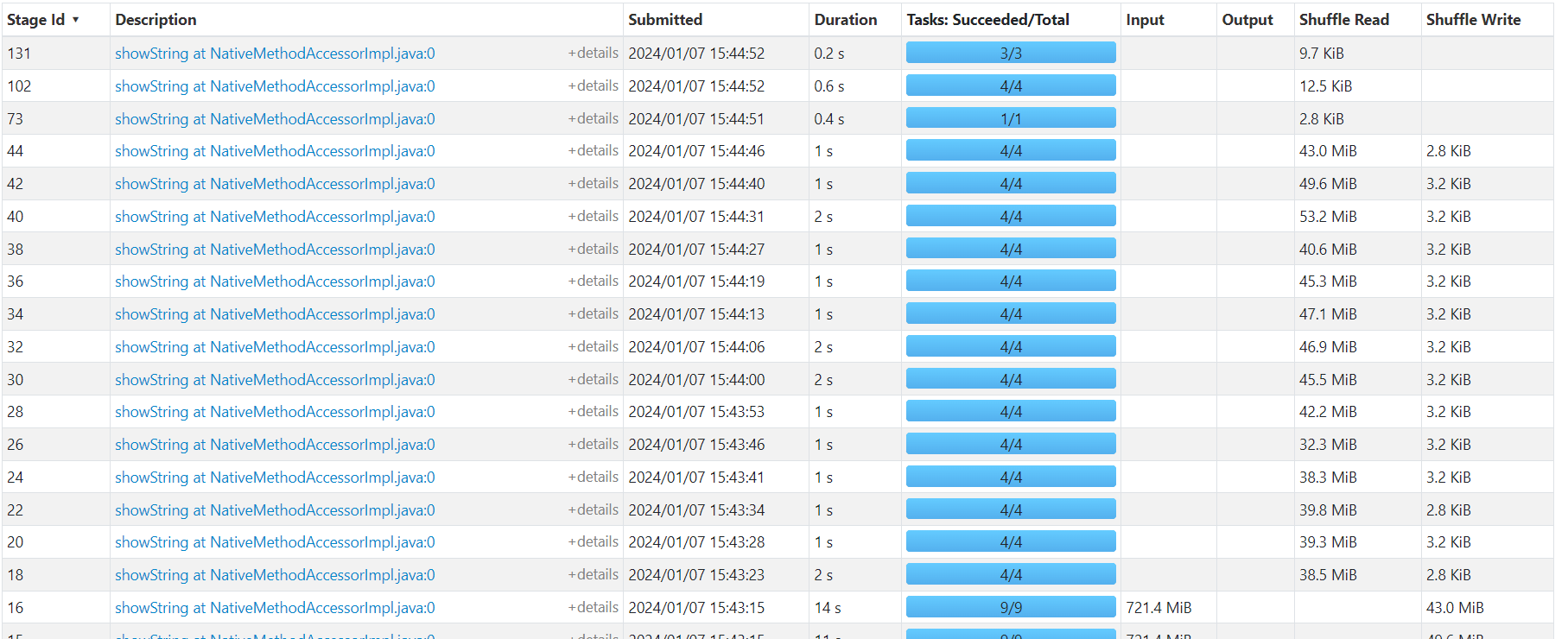
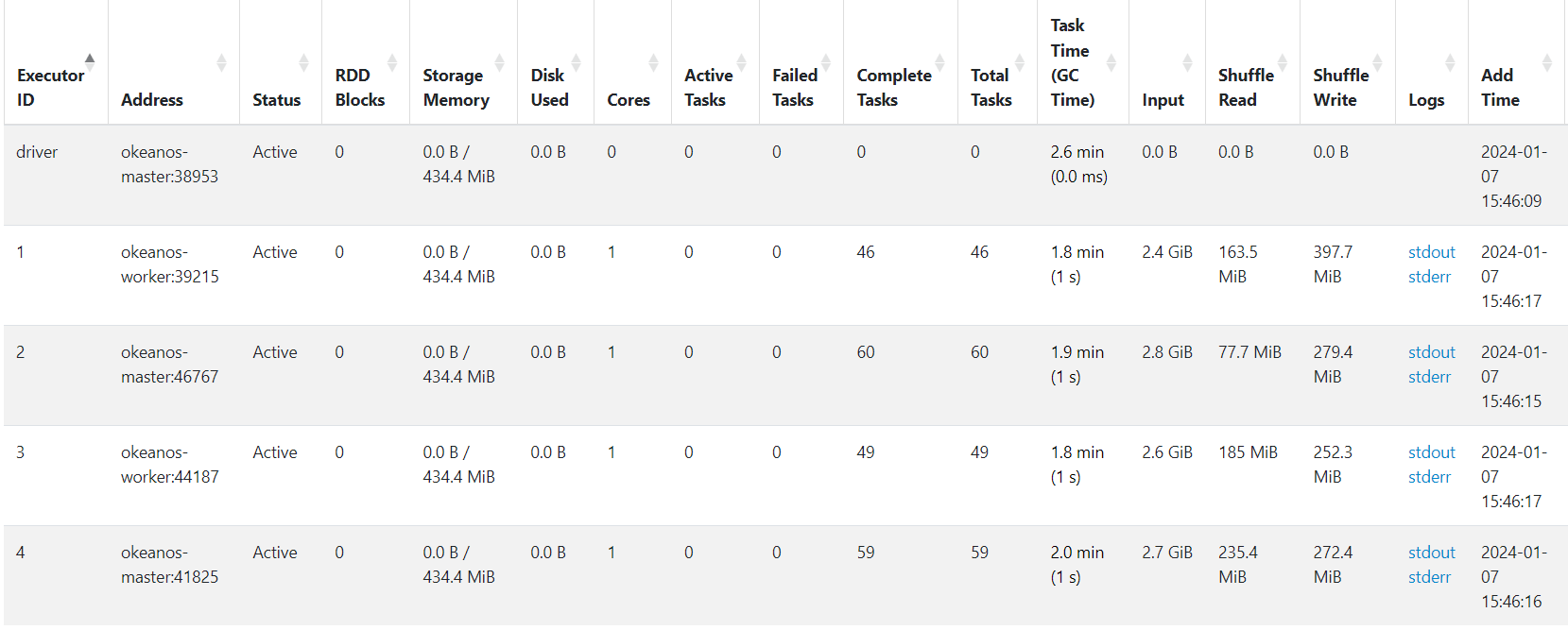
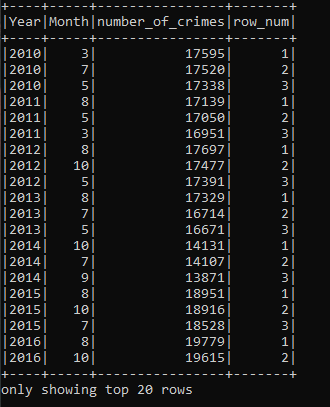
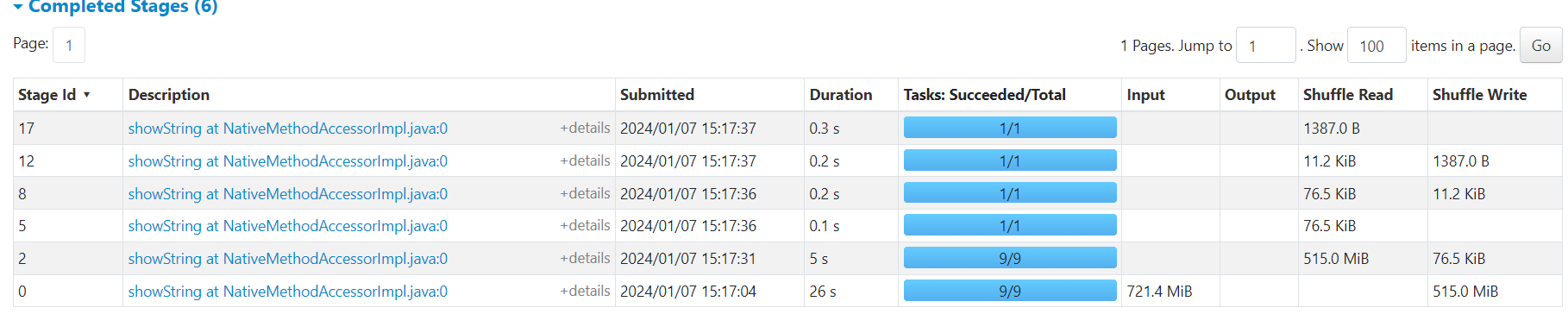
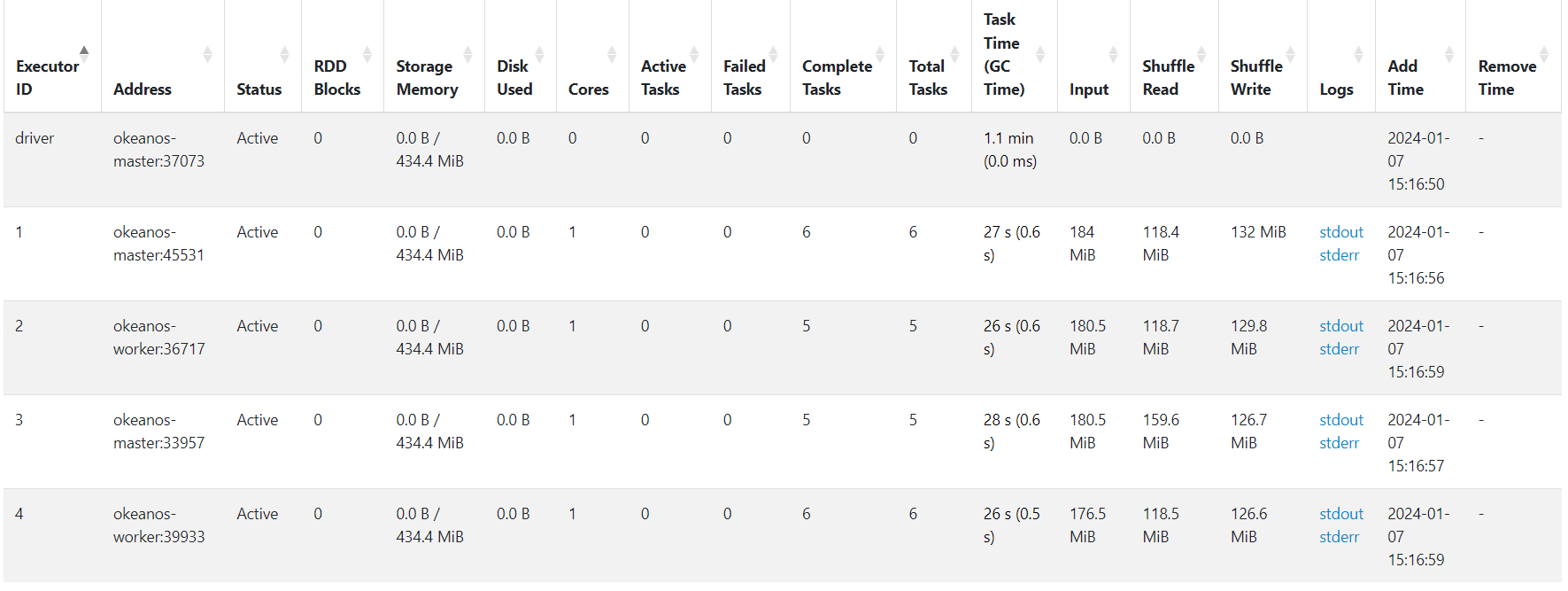
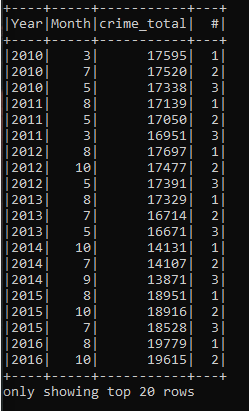
unioned\_df = df\_2010\_to\_2019.unionByName(df\_2020\_to\_present).distinct()

Παρατηρήσαμε τη διαφορά στο όνομα της στήλης AREA στα δυο dataframes και το αλλάξαμε και χρησιμοποιήσαμε unionByName για να σιγουρευτούμε πως δε παίζει ρόλο η σειρά των στηλών, ενώ το distinct() οδηγεί στην αποφυγή duplicate σειρών κατά το union. Την ίδια λογική ακολουθήσαμε και στα union των υπόλοιπων ασκήσεων

# ΖΗΤΟΥΜΕΝΟ 3

Χρησιμοποιούμε την εντολή   
*spark-submit --conf "spark.driver.extraJavaOptions=-Dspark.log.level=WARN" --num-executors X filename.py*

Dataframe  
  
  
  
SQL



## Σύγκριση

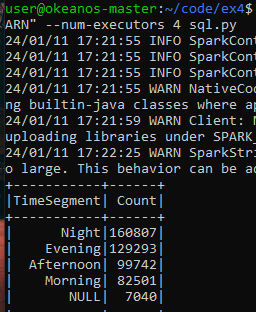
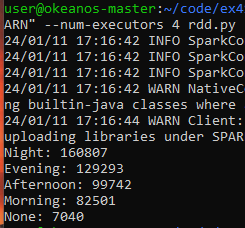
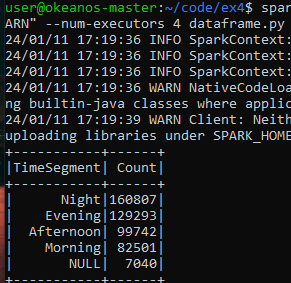
Αναλύουμε τις πληροφορίες του Spark UI για τις δυο εκτελέσεις του Query 1 με 4 executors σε dataframe και SQL API αντίστοιχα. Βλέπουμε πως δεν υπάρχει μεγάλη διαφορά στο Memory Usage, ωστόσο ο χρόνος εκτέλεσης του DataFrame API είναι σταθερά χαμηλότερος από τον χρόνο εκτέλεσης της SQL. Αυτό υποδεικνύει ότι το DataFrame API εκτελεί καλύτερα το συγκεκριμένο query.

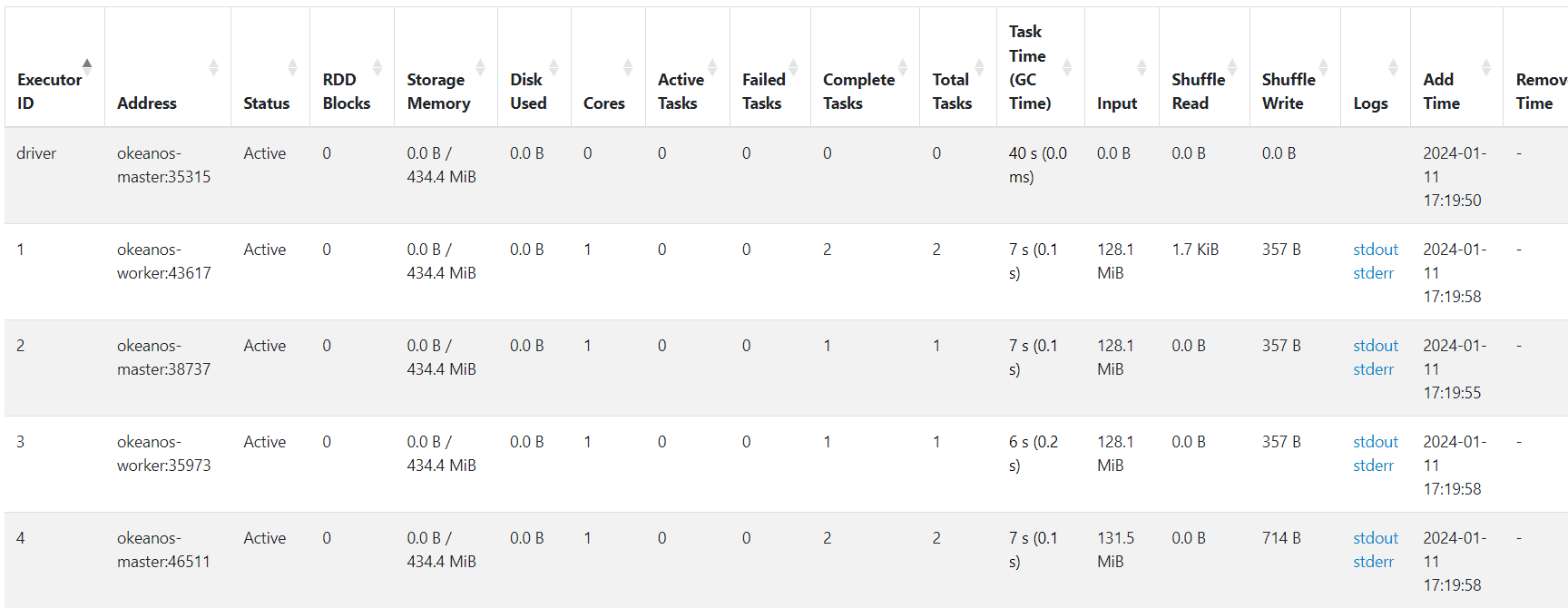
Παρατηρώντας τα Stages για το κάθε API στο Spark UI, βλέπουμε πως το SQL API εκτελεί πολύ περισσότερα stages. Αυτό μπορεί να αποδοθεί στις στρατηγικές βελτιστοποίησης που χρησιμοποιούνται από το Spark Catalyst, το οποίο είναι υπεύθυνο για τη βελτιστοποίηση και την εκτέλεση ερωτημάτων Spark SQL. Η εκτέλεση του SQL API περιλαμβάνει πολλαπλά ενδιάμεσα στάδια και κυρίως λειτουργίες showString που συνδέονται συνήθως με την εμφάνιση του αποτελέσματος ενός DataFrame. Οι πολλαπλές εμφανίσεις του showString υποδηλώνουν ότι υπάρχουν πολλά βήματα που εμπλέκονται στην εμφάνιση του αποτελέσματος, συμβάλλοντας στον αυξημένο αριθμό σταδίων.  
  
Αντίθετα η εκτέλεση του DataFrame API φαίνεται να έχει λιγότερα ενδιάμεσα στάδια, πιθανώς λόγω διαφορών στον τρόπο με τον οποίο το Spark Catalyst βελτιστοποιεί και εκτελεί λειτουργίες DataFrame. Οι λειτουργίες του DataFrame API ενδέχεται να μεταφραστούν πιο άμεσα σε μια ακολουθία physical stages, με αποτέλεσμα ένα πιο συνοπτικό σχέδιο εκτέλεσης.

Όσον αφορά τους 4 executors, η εκτέλεση του DataFrame API φαίνεται να έχει λιγότερα ενδιάμεσα στάδια και έτσι με 4 executors, οι λειτουργίες μπορεί να κατανεμηθούν σε αυτούς πιο αποτελεσματικά. Επίσης οι λειτουργίες union και distinct μπορούν να επωφεληθούν από την παράλληλη επεξεργασία με πολλούς εκτελεστές.  
  
Από την άλλη, το SQL query περιλαμβάνει ένα GROUP BY, που μπορεί να οδηγήσει σε shuffling δεδομένων, εάν δεν βρίσκονται στον ίδιο εκτελεστή. Με 4 executors ο βαθμός παραλληλισμού στο shuffling δεδομένων εξαρτάται από το partition των δεδομένων και το κλειδί που χρησιμοποιείται για την τυχαία αναπαραγωγή. Περισσότεροι εκτελεστές μπορούν ενδεχομένως να βελτιώσουν την αποτελεσματικότητα της ανταλλαγής δεδομένων, αλλά ο υπερβολικός παραλληλισμός μπορεί να οδηγήσει σε αύξηση του overhead, δηλαδή επιπλέον χρόνο και πόρους για την ανακατανομή και ανταλλαγή δεδομένων μεταξύ των κόμβων του cluster κατά τη διάρκεια μιας λειτουργίας shuffle.

ΖΗΤΟΥΜΕΝΟ 4  
  
Χρησιμοποιούμε την εντολή   
*spark-submit --conf "spark.driver.extraJavaOptions=-Dspark.log.level=WARN" --num-executors 4 filename.py  
για να εκτελέσουμε με 4 executors*

Αντιμετωπίσαμε ένα θέμα με το union στο RDD και για αυτό το υλοποιήσαμε μόνο για το dataset 2010-2019 και στα 3 APIs με 4 εκτελεστές.



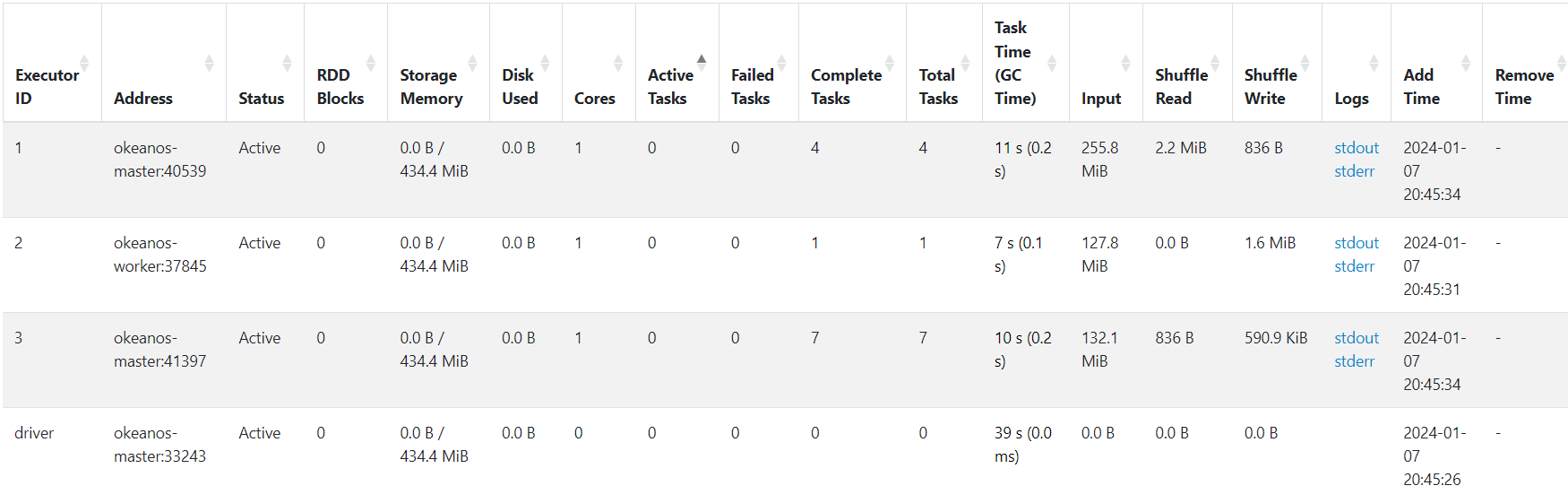
 RDD:  
  
DataFrame:  
SQL:  
  
  
ΣΧΟΛΙΟ: Παρατηρούμε πως για 4 εκτελεστές, η απόδοση των Dataframe (40s) και SQL (43s) APIs είναι σημαντικά καλύτερη από του RDD API (57s). Αυτό οφείλεται στο γεγογονός ότι τα RDDs έχουν αργή επίδοση όταν χρησιμοποιούνται με non-JVM γλώσσες, όπως οι Python, Rust, C και λοιπές. Επίσης, τα RDDs δεν μπορούν να γίνουν optimized από το Spark, ενώ τα Datafrmaes/SQL APIs χρησιμοποιούν τον Spark Catalyst optimizer που μπορεί να εκτελέσει διάφορες βελτιστοποιήσεις και βοηθά στη βελτιστοποίηση του σχεδίου εκτέλεσης για καλύτερες επιδόσεις.  
Τέλος, τα DataFrames χρησιμοποιούν δημιουργία κώδικα για να μεταφράσουν λειτουργίες υψηλού επιπέδου σε κώδικα χαμηλού επιπέδου, γεγονός που μπορεί να οδηγήσει σε πιο αποτελεσματική εκτέλεση. Οι μετασχηματισμοί RDD πραγματοποιούνται κατά τον χρόνο εκτέλεσης, γεγονός που μπορεί να συνεπάγεται μεγαλύτερη επιβάρυνση.

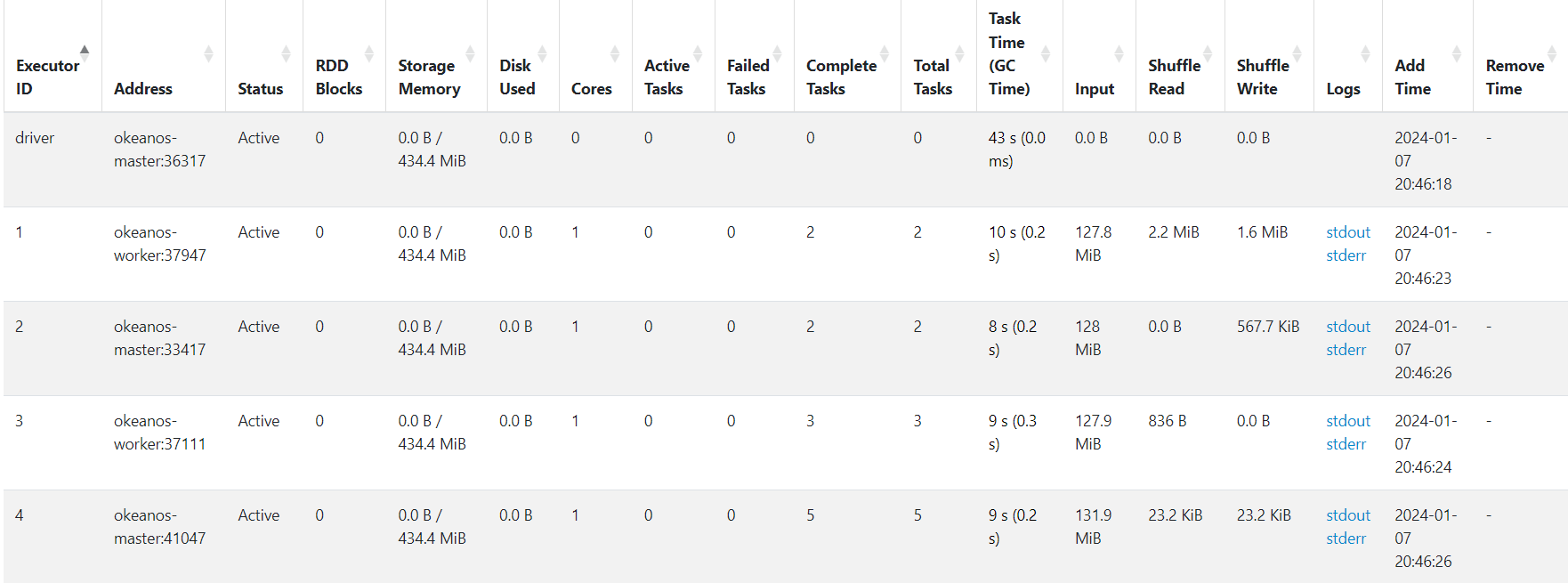
# ΖΗΤΟΥΜΕΝΟ 5

Υλοποιήθηκε με Dataframe API

## Αποτέλεσμα

## 2 Executors

  
  
3 Executors

4 Executors

Ανάλυση

Οι χρόνοι διάρκειας για 2, 3 και 4 executors αντίστοιχα (40s, 39s, 43s) δεν δείχνουν να εμφανίζουν σημαντική διαφορά.   
Παράλληλα παρατηρούμε ότι όπως είναι λογικό όσο λιγότεροι executors τόσο μεγαλύτερη χρήση μνήμης κάνει ο καθένας. Η χρήση πόρων (μνήμη και CPU) σε κάθε εκτελεστή είναι μέτρια, υποδεικνύοντας ότι το συγκεκριμένο query δεν απαιτεί μεγάλη χρήση πόρων.

Η αύξηση του αριθμού των εκτελεστών μπορεί να οδηγήσει σε καλύτερο παραλληλισμό των εργασιών, ιδίως σε μεγαλύτερα σύνολα δεδομένων.  
Εάν οι πόροι που είναι διαθέσιμοι σε κάθε executor είναι περιορισμένοι, η αύξηση του αριθμού των executors ενδέχεται να μην οδηγήσει σε σημαντικές βελτιώσεις  
Στην περίτωσή μας, η πολυπλοκότητα του query δεν βοηθάει να παρατηρήσουμε μεγάλες διαφορές μεταξύ των executors. Μπορούμε να υποθέσουμε από τα παραπάνω δεδομένα πως όσο και αν αυξήσουμε τον αριθμό τον executors πάνω από 4 δεν θα δούμε σημαντική μείωση στο χρόνο.

# ΖΗΤΟΥΜΕΝΟ 6

## Ερώτημα α

### C:\Users\30697\AppData\Local\Packages\Microsoft.Windows.Photos_8wekyb3d8bbwe\TempState\ShareServiceTempFolder\415016318_2332516870264725_6255201778689457217_n.jpeg1ο Μέρος - Dataframe

### 2ο Μέρος – Dataframe

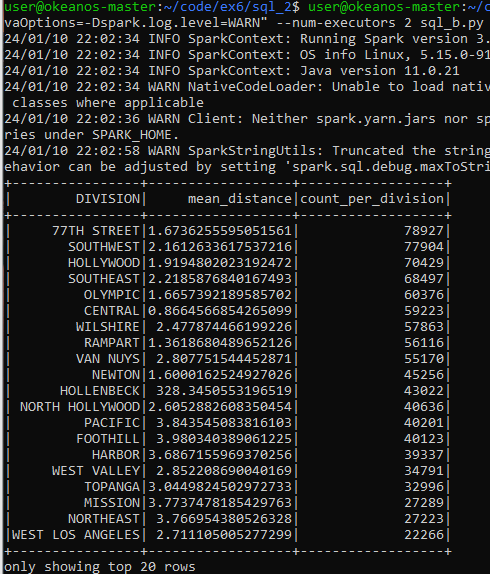
### 1ο Μέρος – SQL

### 2o Μέρος – SQL

### 

## Ερώτημα β 1ο μέρος - Dataframe 2ο μέρος - Dataframe

### 1ο Μέρος – SQL

2ο Μέρος – SQL

Σχόλιο:   
Παρατηρούμε πως τα επιμέρους αποτελέσματα σε sql και dataframe API είναι αντίστοιχα ίδια.   
Για το ερώτημα α, προφανώς το count\_per\_year για τον αριθμό των εγκλημάτων είναι ίδιο σε όλες τιε περιπτώσεις, ωστόσο παρατηρούμε πως η μέση απόσταση στις απαντήσεις για το 2ο μέρος είναι μικρότερες, πράγμα λογικό αφού το 2ο μέρος αφορά τη μέση απόσταση (σε km) των σημείων όπου έλαβαν χώρα τα εγκλήματα, από το πλησιέστερο σε αυτά αστυνομικό τμήμα και όχι από αυτό που ανέλαβε την έρευνα.  
Για το ερώτημα β, προφανώς τα count\_per\_year αλλάζουν από το πρώτο στο δεύτερο μέρος, αφού μας ενδιαφέρει ο αριθμός εγκλημάτων (με καταγραφή χρήσης οποιαδήποτε μορφής όπλων) που έλαβαν χώρα πλησιέστερα στο εκάστοτε τμήμα (2ο μέρος), και όχι ο αριθμός εγκλημάτων που ανέλαβαν (1ο τμήμα).

# ΖΗΤΟΥΜΕΝΟ 7

Υλοποιήθηκε με DATAFRAME API

Κατά τη βελτιστοποίηση των joins στο PySpark, τα hints broadcast, merge, shuffle\_hash και shuffle\_replicate\_nl μπορούν να χρησιμοποιηθούν για να επηρεάσουν τη στρατηγική εκτέλεσης. Βοηθούν το Spark να βελτιστοποιήσει το φυσικό πλάνο για τις λειτουργίες join.

### BROADCAST

Κατάλληλη όταν ένα από τα DataFrames είναι αρκετά μικρό για να χωρέσει στη μνήμη κάθε κόμβου εργασίας. Αποφεύγει την ανακατανομή των δεδομένων στο δίκτυο.

### MERGE

Χρήσιμη όταν και τα δύο DataFrames είναι μεγάλα και έχουν παρόμοιο μέγεθος. Εκτελεί μια ένωση συγχώνευσης χωρίς shuffle, η οποία μπορεί να είναι αποδοτική.

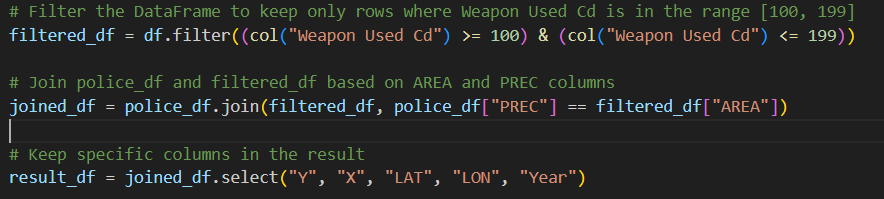
### SHUFFLE\_HASH

Επιβάλλει μια ένωση shuffle βάσει κατακερματισμού. Είναι επωφελής κατά την ένωση μεγάλων πινάκων με άνισα κατανεμημένα κλειδιά.

### SHUFFLE\_REPLICATE\_NL

Εκτελεί μια ένωση με shuffle , όπου το μικρότερο DataFrame μεταδίδεται σε κάθε partition του μεγαλύτερου DataFrame, αποφεύγοντας το ανακάτεμα των δεδομένων. Μειώνει το ανακάτεμα των δεδομένων σε σύγκριση με ένα κανονικό shuffle NL join.

## QUERY 4 - Μέρος 1

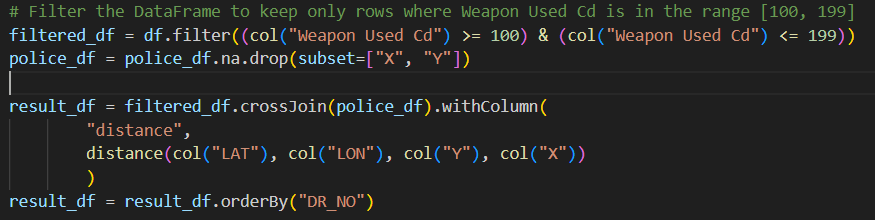
  
  
Χρησιμοποιήσαμε τις 4 περιπτώσεις join στον ίδιο κώδικα. Αναλύοντας το physical plan που λάβαμε από το Spark History UI, καθώς και τα χαρακτηριστικά των δυο dataframe, καταλήγουμε ότι:  
  
Broadcast: Αποτελεί την πιο ταιριαστή επιλογή αφού το ένα από τα DataFrames είναι αρκετά μικρό για να χωρέσει στη μνήμη κάθε εκτελεστή Spark. Έτσι η μέθοδος αυτή μειώνει την ανακατανομή των δεδομένων και την επιβάρυνση του δικτύου και είναι γενικά αποτελεσματική για την ένωση ενός μικρού DataFrame με ένα μεγαλύτερο, όπως στην περίπτωσή μας.  
  
Merge: Αφορά κυρίως μεγάλα dataset, όπου δεν μπορεί να γίνει broadcast, άρα δεν ενδείκνυται στην περίπτωσή μας, αφού απαιτεί ταξινόμηση των DataFrames, η οποία συνεπάγεται πρόσθετη επεξεργασία.

Shuffle Hash Join: Περιλαμβάνει ανακάτεμα δεδομένων, το οποίο μπορεί να είναι απαιτητικό σε πόρους και αφορά κι αυτό μεγαλύτερα dataframes.

Shuffle Replicate Nested Loop: Κατάλληλο όταν ένα από τα DataFrames είναι αρκετά μικρό για να χωρέσει στη μνήμη κάθε εκτελεστή Spark και το άλλο DataFrame είναι πολύ μεγάλο για να μεταδοθεί. Ωστόσο λόγω nested loop δεν είναι όσο αποδοτικό είναι το broadcast.

## QUERY 4 - Μέρος 2

Εδώ έχουμε την εκτέλεση μιας λειτουργίας crossJoin:

  
  
Χρησιμοποιήσαμε τις 4 περιπτώσεις join σε διαφορετικούς κώδικες. Αναλύοντας το physical plan που λάβαμε από το Spark History UI, καθώς και τα χαρακτηριστικά των δυο dataframe, καταλήγουμε ότι:  
  
Το μικρότερο DataFrame (police\_df) μεταδίδεται σε όλους τους κόμβους για να αποφευχθεί η ανακατανομή μεγάλου όγκου δεδομένων. Το broadcasting ενός μικρού DataFrame είναι γενικά πιο αποτελεσματικό από το shuffle ενός μεγάλου DataFrame και μπορεί να βελτιώσει σημαντικά την απόδοση.   
Στην περίπτωσή μας, το πλαίσιο δεδομένων police\_df είναι σχετικά μικρό (21 γραμμές) και η εκτέλεση sort-merge join σε ένα μικρό dataframe είναι περιττή επιβάρυνση.

Το shuffle\_hush μπορεί να οδηγήσει σε περιττή επιβάρυνση επιδόσεων.

Το shuffle\_replicate\_nl είναι κατάλληλο για μικρά DataFrames, αλλά το broadcast είναι γενικά πιο αποδοτικό στην περίπτωσηαυτή, καθώς μεταδίδει ρητά το μικρότερο DataFrame σε όλους τους κόμβους χωρίς να το αναπαράγει σε κάθε executor.

## Query 3

Στο query αυτό έχουμε 2 joins:

* joined\_df = reverse\_df.join(new\_df.select("Zip Code"), reverse\_df["ZIPcode"] == new\_df["Zip Code"], "inner")

Συνδέει πληροφορίες από το reverse\_df (που περιέχουν τις πληροφορίες από το reverse geocoding) με πρόσθετες πληροφορίες από το new\_df (που είναι ένα union των 3 πρώτων και τελευταίων σειρών του ταξινομημένου income\_df). Το αποτέλεσμα, joined\_df, συνδυάζει πληροφορίες τις πληροφορίες των δυο dataframe για συγκεκριμένους κωδικούς ZIP.  
  
BROADCAST Join:

Είναι κατάλληλη όταν ένα από τα DataFrames είναι αρκετά μικρό για να χωρέσει στη μνήμη κάθε κόμβου εργασίας. Στην περίπτωσή σας, το new\_df έχει μόνο 6 γραμμές, γεγονός που το καθιστά αρκετά μικρό για να μεταδοθεί. Η μετάδοση του new\_df θα διανείμει αποτελεσματικά τα αντίγραφά του σε όλους τους worker nodes, αποφεύγοντας την ανάγκη για shuffle.  
  
-> joined\_df = reverse\_df.join(broadcast(new\_df.select("Zip Code")), reverse\_df["ZIPcode"] == new\_df["Zip Code"], "inner")

Οι άλλες επιλογές hint, όπως αναλύσαμε και για το query 4, δε θα είναι τόσο αποδοτικές, λόγω της μεγάλης διαφοράς μεγέθους των δυο dataframes.

* filtered\_df = df.join(unique\_lat\_lon\_pairs, ["LAT", "LON"], "inner")

Εδώ έχουμε ένα join μεταξύ του μεγάλου df (2,924,482 rows) και του μικρότερου unique\_lat\_lon\_pairs (474 rows). Για τους ίδιους λόγους που αναφέραμε και παραπάνω θα προτιμήσουμε το broadcast hint.