Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

# Εξαμηνιαία Εργασία

*Εργασία για το μάθημα του $9^{ου}$ εξαμήνου:*
**"Προχωρημένα Θέματα Βάσεων Δεδομένων"**

Διδάσκων: **Δημήτριος Τσουμάκος**

Ονοματεπώνυμο: **Κελλάρη Μυρσίνη**

Αριθμός Μητρώου: **03119082**

Ονοματεπώνυμο: **Ξανθόπουλος Παναγιώτης**

Αριθμός Μητρώου: **03119084**

Ομάδα: **$6^{η}$**

Αθήνα 17 Ιανουαρίου 2024

# Ζητούμενο 1

# Εγκατάσταση και διαμόρφωση πλατφόρμας

Για την εγκατάσταση και διαμόρφωση της πλατφόρμας εκτέλεσης Apache Spark και του διαχειριστή πόρων του YARN, ακολουθήσαμε τα παρακάτω βήματα:

- Κατεβάζουμε το repo από το github:
  `git clone https://github.com/pxanthopoulos/adv-databases.git`
- Κατεβάζουμε τα σύνολα δεδομένων στον κατάλληλο φάκελο:
  ```
  cd adv-databases/data
  wget https://data.lacity.org/Public-Safety/Crime-Data-from-2010-to-2019/63jg-8b9z
  wget https://data.lacity.org/Public-Safety/Crime-Data-from-2020-to-Present/2nrs-mtv8
  cd ..
  ```
- Εκκινούμε τα services :
  ```
  $HADOOP_HOME/bin/hdfs namenode -format
  start-dfs.sh
  start-yarn.sh
  $SPARK_HOME/sbin/start-history-server.sh
  ```
- Ανεβάζουμε τα δεδομένα στο hdfs:
  ```
  hdfs dfs -mkdir /spark.eventLog
  hdfs dfs -mkdir /user/input
  hdfs dfs -copyFromLocal data/Crime_Data_from_2010_to_2019_insert date.csv /user/input/
  hdfs dfs -copyFromLocal data/Crime_Data_from_2020_to_Present_insert date.csv /user/inpu
  hdfs dfs -copyFromLocal data/income /user/input/
  hdfs dfs -copyFromLocal data/revgecoding.csv /user/input/
  hdfs dfs -copyFromLocal data/LAPD_Police_Stations.csv /user/input/
  ```
- Εκτελούμε τα scripts με `spark-submit`
  Για παράδειγμα `spark-submit ~/scripts/part2.py` και
  `spark-submit - -num-executors 4 ~/scripts/part3-df.py` όταν θέλουμε να προσδιορίσουμε τον αριθμό των executors.

Επιβεβαιώνουμε ότι οι web εφαρμογές των HDFS (port 9870), YARN (port 8088) και Spark History Server (port 18080) να είναι διαθέσιμες και προσβάσιμες

# Ζητούμενο 2

# Δημιουργία DataFrame

Διαβάζουμε τα 2 σύνολα δεδομένων, τα ενώνουμε και πραγματοποιούμε τις απαραίτητες μετατροπές. Το script που πραγματοποιεί το ζητούμενο αυτό είναι το part2.py. Παρακάτω είναι η έξοδος του script αυτού, δηλαδή ο συνολικός αριθμός γραμμών του συνόλου δεδομένων και ο τύπος κάθε στήλης.

```
Number of rows in dataset: 2993433

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)
```

Σχήμα 2.1: Έξοδος δεύτερου ζητούμενου

# Ζητούμενο 3

# Υλοποίηση Query 1

Υλοποιούμε το Query 1 χρησιμοποιώντας τα DataFrame και SQL APIs, με 4 Spark executors. Παρατηρούμε ότι η διαφορά στον χρόνο εκτέλεσης είναι πολύ μικρή (58s για το Dataframe API και 59s για το SQL API). Αυτό συμβαίνει καθώς τα 2 API που χρησιμοποιήσαμε (Dataframe και SQL) χρησιμοποιούν σχετικά παρόμοια optimizations. Η διαφορά που παρατηρούμε μπορεί να οφείλεται σε πολλές αιτίες όπως η διαφορετική χρήση του Catalyst Optimizer, ο οποίος είναι υπεύθυνος για την απλούστευση των εκφράσεων, στις διαφορετικές στρατηγικές παραλληλοποίησης, ένωσης και caching των ενδιάμεσων αποτελεσμάτων αλλά και στην διαφορετική χρήση του code generation από το Spark Tungsten. Παρακάτω είναι το αποτέλεσμα του συγκεκριμένου Query (Πίνακας 3.1), για την παραγωγή του οποίου χρησιμοποιήθηκαν τα script `part3-df.py` και `part3-sql.py`.

| year | month | crime_total | # | year | month | crime_total | # |
|---|---|---|---|---|---|---|---|
| 2010 | 1 | 19515 | 1 | 2017 | 10 | 20431 | 1 |
| 2010 | 3 | 18131 | 2 | 2017 | 7 | 20192 | 2 |
| 2010 | 7 | 17856 | 3 | 2017 | 1 | 19833 | 3 |
| 2011 | 1 | 18135 | 1 | 2018 | 5 | 19973 | 1 |
| 2011 | 7 | 17283 | 2 | 2018 | 7 | 19875 | 2 |
| 2011 | 10 | 17034 | 3 | 2018 | 8 | 19761 | 3 |
| 2012 | 1 | 17943 | 1 | 2019 | 7 | 19121 | 1 |
| 2012 | 8 | 17661 | 2 | 2019 | 8 | 18979 | 2 |
| 2012 | 5 | 17502 | 3 | 2019 | 3 | 18856 | 3 |
| 2013 | 8 | 17440 | 1 | 2020 | 1 | 18498 | 1 |
| 2013 | 1 | 16820 | 2 | 2020 | 2 | 17256 | 2 |
| 2013 | 7 | 16644 | 3 | 2020 | 5 | 17205 | 3 |
| 2014 | 7 | 13531 | 1 | 2021 | 12 | 25453 | 1 |
| 2014 | 10 | 13362 | 2 | 2021 | 10 | 24653 | 2 |
| 2014 | 8 | 13317 | 3 | 2021 | 11 | 24276 | 3 |
| 2015 | 10 | 19219 | 1 | 2022 | 5 | 20419 | 1 |
| 2015 | 8 | 19011 | 2 | 2022 | 10 | 20276 | 2 |
| 2015 | 7 | 18709 | 3 | 2022 | 6 | 20204 | 3 |
| 2016 | 10 | 19659 | 1 | 2023 | 8 | 19772 | 1 |
| 2016 | 8 | 19490 | 2 | 2023 | 7 | 19709 | 2 |
| 2016 | 7 | 19448 | 3 | 2023 | 1 | 19637 | 3 |

Πίνακας 3.1: Αποτελέσματα Query 1

# Ζητούμενο 4

# Υλοποίηση Query 2

Υλοποιούμε το Query 1 χρησιμοποιώντας τα DataFrame και RDD APIs, με 4 Spark executors. Οι χρόνοι εκτέλεσης μετρώνται σε 56 sec για το DataFrame και 78 sec για το RDD. Η διαφορά αυτή οφείλεται στο ότι η υλοποίηση του RDD είναι πιο low-level συγκριτικά με αυτή του DataFrame και δεν διαθέτει τις τεχνικές Query Optimization που αυτό διαθέτει. Συγκεκριμένα, το DataFrame διαθέτει code generation (Tungsten) και optimization σε πολλά επίπεδα (Catalyst) όπως predicate pushdown και constant folding. Επίσης, το Dataframe API χρησιμοποιεί optimized συναρτήσεις για πράγματα που στο RDD API η υλοποίηση επιβαρύνει τον προγραμματιστή με κίνδυνο να χρησιμοποιήσει μη αποδοτικές λύσεις. Παρακάτω είναι το αποτέλεσμα του Query 2 (Πίνακας 4.1), το οποίο παράχθηκε από τα scripts `part4-rdd.py` και `part4-df.py`.

| partofday | crime_total |
|---|---|
| night | 237605 |
| evening | 187306 |
| afternoon | 148180 |
| morning | 123846 |

Πίνακας 4.1: Αποτελέσματα Query 2

# Ζητούμενο 5

# Υλοποίηση Query 3

Υλοποιούμε το Query 3 χρησιμοποιώντας τα DataFrameAPI, με 2, 3 και 4 Spark executors. Οι χρόνοι εκτέλεσης είναι 79s, 77s και 72s αντίστοιχα. Παρατηρούμε πως ο χρόνος εκτέλεσης μειώνεται ελαφρώς όσο αυξάνουμε τους executors. Αυτό οφείλεται στην παραλληλοποίηση διαφόρων tasks από το spark και η ανάθεσή τους στους διάφορους executors ώστε να εκτελεστούν παράλληλα. Πιο συγκεκριμένα, μπορούν να παραλληλοποιηθούν tasks όπως το filtering, το aggregation και τα joins ώστε να έχουμε καλύτερο CPU utilization και μικρότερο χρόνο εκτέλεσης. Βέβαια, καθώς παρατηρούμε αρκετά μικρή βελτίωση, καταλαβαίνουμε πως τα περιθώρια παραλληλοποίησης του κώδικά μας είναι μικρά και το overhead των επιπλέον executors ακυρώνει μερικώς το κέρδος που δίνουν σε χρόνο εκτέλεσης. Παρακάτω παρουσιάζεται το αποτέλεσμα του Query 3 (Πίνακας 5.1), το οποίο παράχθηκε από το script `part5-df.py`.

| Victim Descent | # |
|---|---|
| Hispanic/Latin/Mexican | 1605 |
| Black | 1112 |
| White | 1052 |
| Other | 510 |
| Other Asian | 120 |
| Korean | 9 |
| American Indian/Alaskan Native | 3 |
| Japanese | 3 |
| Chinese | 2 |
| Filipino | 2 |

Πίνακας 5.1: Αποτελέσματα Query 3

# Ζητούμενο 6

# Υλοποίηση Query 4

Παρακάτω είναι τα 4 αποτελέσματα του query 4 τα οποία παράχθηκαν από τα scripts `part6-1a-df.py`, `part6-1b-df.py`, `part6-2a-df.py`, `part6-2b-df.py`. Το 1a αφορά εγκλήματα ανά έτος και τα τμήματα που τα ανέλαβαν, το 1b εγκλήματα ανά τμήμα και τα τμήματα που τα ανέλαβαν, το 2a εγκλήματα ανά έτος και τα πλησιέστερα τμήματα ενώ το 2b εγκλήματα ανά τμήμα και τα πλησιέστερα τμήματα.

| year | average_distance | # |
|---|---|---|
| 2010 | 2.783223419785917 | 8212 |
| 2011 | 2.792627777807431 | 7232 |
| 2012 | 2.8357548501766847 | 6532 |
| 2013 | 2.826132495130459 | 5838 |
| 2014 | 2.7728985198619114 | 4526 |
| 2015 | 2.7058134792640014 | 6763 |
| 2016 | 2.71712670126355 | 8100 |
| 2017 | 2.72395610209098 | 7786 |
| 2018 | 2.7323065125352657 | 7413 |
| 2019 | 2.7394299042987393 | 7129 |
| 2020 | 2.6897964610757588 | 8487 |
| 2021 | 2.6918773930386815 | 12324 |
| 2022 | 2.6081129878209417 | 10025 |
| 2023 | 2.547571482691284 | 8896 |

Πίνακας 6.1: Αποτελέσματα Query 4-1a

| division | average_distance | # |
|---|---|---|
| 77TH STREET | 2.6972210253552595 | 16567 |
| SOUTHEAST | 2.10279027224706 | 12917 |
| NEWTON | 2.0146354540304747 | 9617 |
| SOUTHWEST | 2.6994199151872293 | 8641 |
| HOLLENBECK | 2.6491807909509015 | 6113 |
| HARBOR | 4.083699422322548 | 5444 |
| RAMPART | 1.5786026931349553 | 4998 |
| MISSION | 4.717500308836131 | 4463 |
| OLYMPIC | 1.8335794806033647 | 4336 |
| FOOTHILL | 3.8036985818505467 | 3943 |
| NORTHEAST | 3.9056313802873457 | 3848 |
| HOLLYWOOD | 1.4549117855759066 | 3560 |
| CENTRAL | 1.1352127088279258 | 3485 |
| WILSHIRE | 2.3208496377628425 | 3428 |
| NORTH HOLLYWOOD | 2.7196218940480428 | 3394 |
| WEST VALLEY | 3.525605236393506 | 2789 |
| VAN NUYS | 2.214839845763575 | 2649 |
| PACIFIC | 3.7329758718238124 | 2646 |
| DEVONSHIRE | 4.0191674326059 | 2602 |
| TOPANGA | 3.4792441423584566 | 2313 |
| WEST LOS ANGELES | 4.263889484803211 | 1510 |

Πίνακας 6.2: Αποτελέσματα Query 4-1b

| year | average_distance | # |
|---|---|---|
| 2010 | 2.4342351304104075 | 8212 |
| 2011 | 2.4610050761252147 | 7232 |
| 2012 | 2.5055255745226757 | 6532 |
| 2013 | 2.455543758235323 | 5838 |
| 2014 | 2.387929444580322 | 4526 |
| 2015 | 2.387261321545352 | 6763 |
| 2016 | 2.428195036520009 | 8100 |
| 2017 | 2.391618931052631 | 7786 |
| 2018 | 2.4082079744379796 | 7413 |
| 2019 | 2.4294088128732088 | 7129 |
| 2020 | 2.3836158389670907 | 8487 |
| 2021 | 2.3527163780992493 | 9745 |
| 2022 | 2.312096006091396 | 10025 |
| 2023 | 2.2659285234391233 | 8896 |

Πίνακας 6.3: Αποτελέσματα Query 4-2a

| division | average_distance | # |
|---|---|---|
| 77TH STREET | 1.7215559690336242 | 13489 |
| SOUTHEAST | 2.195598619983745 | 11816 |
| SOUTHWEST | 2.2793433427796046 | 11209 |
| NEWTON | 1.5693029411378485 | 7161 |
| WILSHIRE | 2.4446505641229015 | 6253 |
| HOLLENBECK | 2.6368430404057097 | 6174 |
| OLYMPIC | 1.6654702490469095 | 5415 |
| HOLLYWOOD | 2.008655911594125 | 5378 |
| HARBOR | 3.8982379424901263 | 5322 |
| RAMPART | 1.3978586840098526 | 4700 |
| FOOTHILL | 3.600357826588798 | 4693 |
| VAN NUYS | 2.9734701285369516 | 4673 |
| CENTRAL | 1.0176983016676135 | 3584 |
| NORTH HOLLYWOOD | 2.7455806390009436 | 3389 |
| NORTHEAST | 3.7555638736291987 | 3096 |
| MISSION | 3.8067591623819337 | 2853 |
| WEST VALLEY | 2.7933743009198384 | 2755 |
| PACIFIC | 3.70145546764868 | 2522 |
| TOPANGA | 3.0519644469818297 | 2435 |
| DEVONSHIRE | 2.9846027227001146 | 1332 |
| WEST LOS ANGELES | 2.7684715765746977 | 1014 |

Πίνακας 6.4: Αποτελέσματα Query 4-2b

# Ζητούμενο 7

# HINT & EXPLAIN

Το πλάνο εκτέλεσης των join έχει ληφθεί τόσο σε μορφή κειμένου όσο και γραφικά από το Spark History UI και βρίσκεται στον φάκελο join-plans του repo. Παρουσιάζουμε τους χρόνους εκτέλεσης για το Query 3 και 4α-4β, μετρώντας τον χρόνο εκτέλεσης της μεθόδου showString (λόγω του lazy execution), στους παρακάτω πίνακας (Πίνακες 7.1, 7.2 και 7.3).

| Μέθοδος | Χρόνος |
|---|---|
| BROADCAST | 14 s |
| MERGE | 14 s |
| SHUFFLE HASH | 13 s |
| SHUFFLE REPLICATE NL | 10 min |

Πίνακας 7.1: Χρόνος εκτέλεσης Query 3

| Μέθοδος | Χρόνος (sec) |
|---|---|
| BROADCAST | 14 |
| MERGE | 30 |
| SHUFFLE HASH | 30 |
| SHUFFLE REPLICATE NL | 14 |

Πίνακας 7.2: Χρόνος εκτέλεσης Query 4-1a

| Μέθοδος | Χρόνος (sec) |
|---|---|
| BROADCAST | 15 |
| MERGE | 30 |
| SHUFFLE HASH | 30 |
| SHUFFLE REPLICATE NL | 15 |

Πίνακας 7.3: Χρόνος εκτέλεσης Query 4-1b

Στο query 3 έχουμε αρχικά ένα join μεταξύ εγκλημάτων του 2015 (∼190K rows) και των zipcodes (∼336K rows) και στη συνέχεια ένα join μεταξύ εγκλημάτων (∼188K rows) και του income dataset (∼2,5K rows).

Παρατηρούμε ότι η μέθοδος Shuffle Replicate NL παίρνει πολύ περισσότερο χρόνο από τις υπόλοιπες. Η μέθοδος αυτή χρησιμοποιεί καρτεσιανό γινόμενο, δηλαδή ένα nested loop όπου συγκρίνει κάθε row του ενός dataframe με κάθε row του άλλου dataframe. Λόγω του υψηλού αριθμού των rows κυρίως στο πρώτο join), είναι αναμενόμενο το cartesian product να παίρνει πολύ χρόνο.

Αναφορικά με τις άλλες μεθόδους, μπορούμε να αναφέρουμε τα εξής. Η μέθοδος broadcast συμφέρει σε joins όπου έχουμε 1 μικρό και ένα μεγάλο dataframe και το μικρό μπορεί να χωρέσει στην κύρια μνήμη κάθε executor και να γίνει broadcast χωρίς μεγάλο κόστος. Χωρίζει το μεγάλο dataframe σε κομμάτια και τα αναθέτει στους executors. Το μικρό dataframe γίνεται broadcast σε όλους τους executors και ο καθένας αναλαμβάνει να το συνδέσει με το κομμάτι του μεγάλου που του ανατέθηκε, χρησιμοποιώντας hash join. Οπότε, η μέθοδος broadcast συμφέρει για το δεύτερο join, ενώ για το πρώτο λιγότερο.

Οι μέθοδοι Merge και Shuffle Hash συμφέρουν σε joins όπου τα dataframes είναι μεγάλα και κανένα δεν μπορεί να γίνει broadcast αποδοτικά. Αμφότερες αρχικά κάνουν shuffle τα δεδομένα, δηλαδή τα μοιράζουν βάσει του join key. Στη συνέχεια, η Merge ταξινομεί τα rows βάσει του join key και στη συνέχεια, χρησιμοποιώντας την ταξινόμηση, τα ενώνει. Η Shuffle Hash δεν ταξινομεί αλλά κάνει hash join. Οι μέθοδοι αυτές προσφέρουν καλύτερη απόδοση για το πρώτο join καθώς και τα 2 dataframes είναι αρκετά μεγάλα ενώ υπολείπονται της broadcast στο δεύτερο. Συνεπώς παρατηρούμε παρόμοια συνολική επίδοση μεταξύ των 3ων μεθόδων καθώς τα 2 join που έχουμε διαφορετικής πληθικότητας. Αναφορικά με την διαφορά μεταξύ Merge και Shuffle hash, μπορούμε να την αποδόσουμε στο κόστος ταξινόμησης το οποίο υπερισχύει του κέρδους στην ένωση που προσφέρει. Επίσης, η Suffle Hash μπορεί να διαχειριστεί καλύτερα σύνολά δεδομένων με άνιση κατανομή και πληθικότητα καθώς χρησιμοποιεί hash tables για την ένωση, τα οποία εξομαλύνουν εν μέρει τις διαφορές αυτές.

Στο query 4 έχουμε σε κάθε περίπτωση ένα join μεταξύ των εγκλημάτων και των αστυνομικών τμημάτων. Τα αστυνομικά τμήματα είναι μόνο 21 ενώ τα εγκλήματα, μετά το filtering που κάναμε, είναι περίπου 109K. Οπότε, σύμφωνα και με τα παραπάνω, μπορούμε να συμπεράνουμε ότι η μέθοδος broadcast θα είναι πολύ αποδοτική καθώς το dataframe με τα τμήματα είναι πολύ μικρό και μπορεί να μοιραστεί στους executors χωρίς μεγάλο overhead. Οι μέθοδοι Merge και Shuffle Hash δεν συμφέρουν καθώς το overhead του shuffling (δηλαδή του διαχωρισμού με βάση το join key) αλλά και το επιπλέον overhead του sorting στην μέθοδο Merge υπερκαλύπτουν όποιο κέρδος προκύπτει από αυτά. Τέλος, η Shuffle Replicate NL παρατηρούμε ότι έχει ίδια απόδοση με την broadcast. Η μέθοδος αυτή μοιράζει με παρόμοιο τρόπο τα dataframes αλλά, ανά κόμβο, δεν κάνει hash join αλλά καρτεσιανό γινόμενο. Καθώς όμως έχουμε πολύ μικρό αριθμό rows στο dataframe με τα τμήματα, η χρήση του hash join δεν προσφέρει μεγάλο κέρδος έναντι του καρτεσιανού γινομένου (δηλαδή του απλού nested loop).

# Παράρτημα Α

# Scripts

Κώδικας 1: Part 2

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_timestamp

spark = SparkSession.builder.appName("part2").getOrCreate()

first_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2010_to_2019_20231224.csv",
        header=True, inferSchema=True)
second_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2020_to_Present_20231224.csv",
        header=True, inferSchema=True)

dataset_df = first_dataset.union(second_dataset)

dataset_df = dataset_df.withColumn("Date Rptd", to_timestamp(
    "Date Rptd", "MM/dd/yyyy hh:mm:ss a").cast("date"))
dataset_df = dataset_df.withColumn("DATE OCC", to_timestamp(
    "DATE OCC", "MM/dd/yyyy hh:mm:ss a").cast("date"))

print("Number of rows in dataset:", dataset_df.count())
dataset_df.printSchema()

spark.stop()
```

Κώδικας 2: Part 3 - DF

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql import Window

spark = SparkSession.builder.appName("part3-df").getOrCreate()

first_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2010_to_2019_20231224.csv",
        header=True, inferSchema=True)
second_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2020_to_Present_20231224.csv",
```

```
        header=True, inferSchema=True)

dataset_df = first_dataset.union(second_dataset)

dataset_df = dataset_df.withColumn("Date Rptd", F.to_timestamp(
    "Date Rptd", "MM/dd/yyyy hh:mm:ss a").cast("date"))
dataset_df = dataset_df.withColumn("DATE OCC", F.to_timestamp(
    "DATE OCC", "MM/dd/yyyy hh:mm:ss a").cast("date"))

dataset_filt_df = (
    dataset_df
    .select(["DATE OCC"])
)

df_top_months = (
    dataset_filt_df
    .withColumn("year", F.year("DATE OCC"))
    .withColumn("month", F.month("DATE OCC"))
    .groupBy("year", "month")
    .agg(F.count("*").alias("crime_total"))
    .withColumn("rank", F.row_number().over(Window.partitionBy("year").orderBy(F.desc("crime_total"))))
    .filter("rank <= 3")
    .orderBy("year", F.desc("crime_total"))
    .withColumnRenamed("rank", "#")
)

df_top_months.show(42)

spark.stop()
```

Κώδικας 3: Part 3 - SQL

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F

spark = SparkSession.builder.appName("part3-sql").getOrCreate()

first_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2010_to_2019_20231224.csv",
        header=True, inferSchema=True)
second_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2020_to_Present_20231224.csv",
        header=True, inferSchema=True)

dataset_df = first_dataset.union(second_dataset)
dataset_df = dataset_df.withColumn("Date Rptd", F.to_timestamp(
    "Date Rptd", "MM/dd/yyyy hh:mm:ss a").cast("date"))
dataset_df = dataset_df.withColumn("DATE OCC", F.to_timestamp(
    "DATE OCC", "MM/dd/yyyy hh:mm:ss a").cast("date"))
dataset_df.createOrReplaceTempView("crimes")

filter_query = """
    SELECT
        `DATE OCC`
    FROM
        crimes
```

```
"""

dataset_filt_df = spark.sql(filter_query)
dataset_filt_df.createOrReplaceTempView("crimes")

sql_query = """
    WITH ranked_crimes AS (
        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 rank
        FROM crimes
        GROUP BY YEAR(`DATE OCC`), MONTH(`DATE OCC`)
    )
    SELECT
        year,
        month,
        crime_total,
        rank
    FROM
        ranked_crimes
    WHERE
        rank <= 3
    ORDER BY
        year, rank ASC
"""

df_top_months = spark.sql(sql_query)

df_top_months.show(42)

spark.stop()
```

Κώδικας 4: Part 4 - DF

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.types import StringType

spark = SparkSession.builder.appName("part4-df").getOrCreate()

first_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2010_to_2019_20231224.csv",
    header=True, inferSchema=True)
second_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2020_to_Present_20231224.csv",
    header=True, inferSchema=True)

dataset_df = first_dataset.union(second_dataset)

dataset_df = dataset_df.withColumn("Date Rptd", F.to_timestamp(
    "Date Rptd", "MM/dd/yyyy hh:mm:ss a").cast("date"))
dataset_df = dataset_df.withColumn("DATE OCC", F.to_timestamp(
    "DATE OCC", "MM/dd/yyyy hh:mm:ss a").cast("date"))
```

```
dataset_filt_df = (
    dataset_df
    .select(["TIME OCC", "Premis Cd"])
)


def calc_part(time):
    if time >= 500 and time <= 1159:
        return "morning"
    if time >= 1200 and time <= 1659:
        return "afternoon"
    if time >= 1700 and time <= 2059:
        return "evening"
    return "night"


calc_part_udf = F.udf(calc_part, StringType())

df_timeofday = (
    dataset_filt_df
    .withColumn("partofday", calc_part_udf(F.col("TIME OCC")))
    .filter(F.col("Premis Cd") == 101)
    .groupBy("partofday")
    .agg(F.count("*").alias("crime_total"))
    .orderBy(F.desc("crime_total"))
)

df_timeofday.show(4)

spark.stop()
```

Κώδικας 5: Part 4 - RDD

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql import Row

spark = SparkSession.builder.appName("part4-rdd").getOrCreate()

first_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2010_to_2019_20231224.csv",
    header=True, inferSchema=True)
second_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2020_to_Present_20231224.csv",
    header=True, inferSchema=True)

dataset_df = first_dataset.union(second_dataset)

dataset_df = dataset_df.withColumn("Date Rptd", F.to_timestamp(
    "Date Rptd", "MM/dd/yyyy hh:mm:ss a").cast("date"))
dataset_df = dataset_df.withColumn("DATE OCC", F.to_timestamp(
    "DATE OCC", "MM/dd/yyyy hh:mm:ss a").cast("date"))

dataset_rdd = dataset_df.rdd
```

```
def calc_part(row):
    time = row["TIME OCC"]
    if time >= 500 and time <= 1159:
        part_of_day = "morning"
    elif time >= 1200 and time <= 1659:
        part_of_day = "afternoon"
    elif time >= 1700 and time <= 2059:
        part_of_day = "evening"
    else:
        part_of_day = "night"
    return Row(partofday=part_of_day, Premis_Cd=row["Premis Cd"])


rdd_partofday = dataset_rdd.map(calc_part)
rdd_partofday = rdd_partofday.filter(lambda x: x["Premis_Cd"] == 101)
rdd_partofday = rdd_partofday.map(lambda x: (
    x["partofday"], 1)).reduceByKey(lambda x, y: x + y)
rdd_partofday = rdd_partofday.sortBy(lambda x: x[1], ascending=False)

print(rdd_partofday.take(4))
```

Κώδικας 6: Part 5

```
from pyspark.sql import SparkSession
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.types import StringType, IntegerType

spark = SparkSession.builder.appName("part5-df").getOrCreate()

first_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2010_to_2019_20231224.csv",
    header=True, inferSchema=True)
second_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2020_to_Present_20231224.csv",
    header=True, inferSchema=True)

dataset_df = first_dataset.union(second_dataset)

dataset_df = dataset_df.withColumn("Date Rptd", F.to_timestamp(
    "Date Rptd", "MM/dd/yyyy hh:mm:ss a").cast("date"))
dataset_df = dataset_df.withColumn("DATE OCC", F.to_timestamp(
    "DATE OCC", "MM/dd/yyyy hh:mm:ss a").cast("date"))

income2015_df = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/income/LA_income_2015.csv", header=True, inferSchema=True)
revgeocoding_df = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/revgecoding.csv", header=True, inferSchema=True)

dataset_filt_df = (
    dataset_df
    .select(["DATE OCC", "Vict Descent", "LAT", "LON"])
)


def descent_description(descent):
```

```
    if descent == "A":
        return "Other Asian"
    if descent == "B":
        return "Black"
    if descent == "C":
        return "Chinese"
    if descent == "D":
        return "Cambodian"
    if descent == "F":
        return "Filipino"
    if descent == "G":
        return "Guamanian"
    if descent == "H":
        return "Hispanic/Latin/Mexican"
    if descent == "I":
        return "American Indian/Alaskan Native"
    if descent == "J":
        return "Japanese"
    if descent == "K":
        return "Korean"
    if descent == "L":
        return "Laotian"
    if descent == "O":
        return "Other"
    if descent == "P":
        return "Pacific Islander"
    if descent == "S":
        return "Samoan"
    if descent == "U":
        return "Hawaiian"
    if descent == "V":
        return "Vietnamese"
    if descent == "W":
        return "White"
    if descent == "Z":
        return "Asian Indian"


descript_udf = F.udf(descent_description, StringType())


def get_income(string_income):
    return int(string_income[1:].replace(",", ""))


get_income_udf = F.udf(get_income, IntegerType())

df_descent_zip = (
    dataset_df
    .withColumn("year", F.year("DATE OCC"))
    .filter("year == 2015")
    .filter((F.col("Vict Descent") != "X") & (F.col("Vict Descent").isNotNull()))
    .join(revgeocoding_df, (revgeocoding_df["LAT"] == dataset_df["LAT"]) & (revgeocoding_df["LON"] == datas
    .withColumn("Zip code", F.col("ZIPcode").substr(1, 5).cast("int"))
    .select(["Vict Descent", "Zip code"])
)

df_descent_income = (
    df_descent_zip
```

```
    .join(income2015_df, income2015_df["Zip Code"] == df_descent_zip["Zip code"])
    .select(["Vict Descent", "Estimated Median Income"])
    .withColumn("Victim Descent", descript_udf(F.col("Vict Descent")))
)

df_descent_income = (
    df_descent_income
    .select(["Victim Descent", "Estimated Median Income"])
    .withColumn("Estimated Median Income", get_income_udf(F.col("Estimated Median Income")))
)

income_values = (
    df_descent_income
    .groupBy("Estimated Median Income")
    .agg(F.count("*").alias("#"))
    .orderBy(F.col("Estimated Median Income").desc())
    .collect()
)

all_income_descent = (
    df_descent_income
    .filter((F.col("Estimated Median Income") == income_values[0][0])
            | (F.col("Estimated Median Income") == income_values[1][0])
            | (F.col("Estimated Median Income") == income_values[2][0])
            | (F.col("Estimated Median Income") == income_values[-1][0])
            | (F.col("Estimated Median Income") == income_values[-2][0])
            | (F.col("Estimated Median Income") == income_values[-3][0]))
    .groupBy("Victim Descent")
    .agg(F.count("*").alias("#"))
    .orderBy(F.col("#").desc())
)

all_income_descent.show()

spark.stop()
```

Κώδικας 7: Part 6 - 1a

```
import distance
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.types import FloatType

spark = SparkSession.builder.appName("part6-1a-df").getOrCreate()

spark.sparkContext.addPyFile("/home/user/scripts/constants.py")
spark.sparkContext.addPyFile("/home/user/scripts/geodesiccapability.py")
spark.sparkContext.addPyFile("/home/user/scripts/geomath.py")
spark.sparkContext.addPyFile("/home/user/scripts/geodesic.py")
spark.sparkContext.addPyFile("/home/user/scripts/units.py")
spark.sparkContext.addPyFile("/home/user/scripts/util.py")
spark.sparkContext.addPyFile("/home/user/scripts/format.py")
spark.sparkContext.addPyFile("/home/user/scripts/point.py")
spark.sparkContext.addPyFile("/home/user/scripts/distance.py")
```

```
first_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2010_to_2019_20231224.csv",
    header=True, inferSchema=True)
second_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2020_to_Present_20231224.csv",
    header=True, inferSchema=True)

dataset_df = first_dataset.union(second_dataset)

dataset_df = dataset_df.withColumn("Date Rptd", F.to_timestamp(
    "Date Rptd", "MM/dd/yyyy hh:mm:ss a").cast("date"))
dataset_df = dataset_df.withColumn("DATE OCC", F.to_timestamp(
    "DATE OCC", "MM/dd/yyyy hh:mm:ss a").cast("date"))

policedept_df = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/LAPD_Police_Stations.csv", header=True, inferSchema=True)

dataset_filt_df = (
    dataset_df
    .withColumn("year", F.year("DATE OCC"))
    .filter(F.col("LAT") != 0.0)
    .filter(F.col("Weapon Used Cd").between(100, 199))
    .select(["AREA ", "LAT", "LON", "year"])
)

policedept_filt_df = (
    policedept_df
    .select(["X", "Y", "PREC"])
)

joined_df = (
    dataset_filt_df
    .join(policedept_filt_df, dataset_filt_df["AREA "] == policedept_filt_df["PREC"], "inner")
)


def dist(x1, y1, x2, y2):
    return distance.geodesic((x1, y1), (x2, y2)).km


dist_udf = F.udf(dist, FloatType())

final_df = (
    joined_df
    .withColumn("Distance", dist_udf(F.col("Y"), F.col("X"), F.col("LAT"), F.col("LON")))
    .groupBy("year")
    .agg(F.avg("Distance").alias("average_distance"),
         F.count("year").alias("#")
         )
    .orderBy(F.col("year"))
)

final_df.show(final_df.count())

spark.stop()
```

Κώδικας 8: Part 6 - 1b

```
import distance
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.types import FloatType

spark = SparkSession.builder.appName("part6-1b-df").getOrCreate()

spark.sparkContext.addPyFile("/home/user/scripts/constants.py")
spark.sparkContext.addPyFile("/home/user/scripts/geodesiccapability.py")
spark.sparkContext.addPyFile("/home/user/scripts/geomath.py")
spark.sparkContext.addPyFile("/home/user/scripts/geodesic.py")
spark.sparkContext.addPyFile("/home/user/scripts/units.py")
spark.sparkContext.addPyFile("/home/user/scripts/util.py")
spark.sparkContext.addPyFile("/home/user/scripts/format.py")
spark.sparkContext.addPyFile("/home/user/scripts/point.py")
spark.sparkContext.addPyFile("/home/user/scripts/distance.py")


first_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2010_to_2019_20231224.csv",
    header=True, inferSchema=True)
second_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2020_to_Present_20231224.csv",
    header=True, inferSchema=True)

dataset_df = first_dataset.union(second_dataset)

dataset_df = dataset_df.withColumn("Date Rptd", F.to_timestamp(
    "Date Rptd", "MM/dd/yyyy hh:mm:ss a").cast("date"))
dataset_df = dataset_df.withColumn("DATE OCC", F.to_timestamp(
    "DATE OCC", "MM/dd/yyyy hh:mm:ss a").cast("date"))

policedept_df = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/LAPD_Police_Stations.csv", header=True, inferSchema=True)

dataset_filt_df = (
    dataset_df
    .filter(F.col("LAT") != 0.0)
    .filter(F.col("Weapon Used Cd").between(100, 199))
    .select(["AREA ", "LAT", "LON"])
)

policedept_filt_df = (
    policedept_df
    .select(["DIVISION", "X", "Y", "PREC"])
    .withColumnRenamed("DIVISION", "division")
)

joined_df = (
    dataset_filt_df
    .join(policedept_filt_df, dataset_filt_df["AREA "] == policedept_filt_df["PREC"], "inner")
)


def dist(x1, y1, x2, y2):
    return distance.geodesic((x1, y1), (x2, y2)).km
```

```
dist_udf = F.udf(dist, FloatType())

final_df = (
    joined_df
    .withColumn("Distance", dist_udf(F.col("Y"), F.col("X"), F.col("LAT"), F.col("LON")))
    .groupBy("division")
    .agg(F.avg("Distance").alias("average_distance"),
         F.count("division").alias("#")
         )
    .orderBy(F.col("#").desc())
)

final_df.show(final_df.count())

spark.stop()
```

Κώδικας 9: Part 6 - 2a

```
import distance
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.types import FloatType

spark = SparkSession.builder.appName("part6-2a-df").getOrCreate()

spark.sparkContext.addPyFile("/home/user/scripts/constants.py")
spark.sparkContext.addPyFile("/home/user/scripts/geodesiccapability.py")
spark.sparkContext.addPyFile("/home/user/scripts/geomath.py")
spark.sparkContext.addPyFile("/home/user/scripts/geodesic.py")
spark.sparkContext.addPyFile("/home/user/scripts/units.py")
spark.sparkContext.addPyFile("/home/user/scripts/util.py")
spark.sparkContext.addPyFile("/home/user/scripts/format.py")
spark.sparkContext.addPyFile("/home/user/scripts/point.py")
spark.sparkContext.addPyFile("/home/user/scripts/distance.py")


first_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2010_to_2019_20231224.csv",
    header=True, inferSchema=True)
second_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2020_to_Present_20231224.csv",
    header=True, inferSchema=True)

dataset_df = first_dataset.union(second_dataset)

dataset_df = dataset_df.withColumn("Date Rptd", F.to_timestamp(
    "Date Rptd", "MM/dd/yyyy hh:mm:ss a").cast("date"))
dataset_df = dataset_df.withColumn("DATE OCC", F.to_timestamp(
    "DATE OCC", "MM/dd/yyyy hh:mm:ss a").cast("date"))

policedept_df = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/LAPD_Police_Stations.csv", header=True, inferSchema=True)

dataset_filt_df = (
    dataset_df
    .withColumn("year", F.year("DATE OCC"))
```

```
    .filter(F.col("LAT") != 0.0)
    .filter(F.col("Weapon Used Cd").between(100, 199))
    .select(["DR_NO", "LAT", "LON", "year"])
)

policedept_filt_df = (
    policedept_df
    .select(["X", "Y"])
)

joined_df = (
    dataset_filt_df
    .crossJoin(policedept_filt_df)
)


def dist(x1, y1, x2, y2):
    return distance.geodesic((x1, y1), (x2, y2)).km


dist_udf = F.udf(dist, FloatType())

mindist_df = (
    joined_df
    .withColumn("Distance", dist_udf(F.col("Y"), F.col("X"), F.col("LAT"), F.col("LON")))
    .groupBy("DR_NO", "year")
    .agg(F.min("Distance").alias("min_dist"))
)

final_df = (
    mindist_df
    .groupBy("year")
    .agg(F.avg("min_dist").alias("average_distance"),
         F.count("year").alias("#")
         )
    .orderBy(F.col("year"))
)

final_df.show(final_df.count())

spark.stop()
```

Κώδικας 10: Part 6 - 2b

```
import distance
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.types import FloatType
from pyspark.sql.window import Window

spark = SparkSession.builder.appName("part6-2b-df").getOrCreate()

spark.sparkContext.addPyFile("/home/user/scripts/constants.py")
spark.sparkContext.addPyFile("/home/user/scripts/geodesiccapability.py")
spark.sparkContext.addPyFile("/home/user/scripts/geomath.py")
spark.sparkContext.addPyFile("/home/user/scripts/geodesic.py")
```

```
spark.sparkContext.addPyFile("/home/user/scripts/units.py")
spark.sparkContext.addPyFile("/home/user/scripts/util.py")
spark.sparkContext.addPyFile("/home/user/scripts/format.py")
spark.sparkContext.addPyFile("/home/user/scripts/point.py")
spark.sparkContext.addPyFile("/home/user/scripts/distance.py")


first_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2010_to_2019_20231224.csv",
    header=True, inferSchema=True)
second_dataset = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/Crime_Data_from_2020_to_Present_20231224.csv",
    header=True, inferSchema=True)

dataset_df = first_dataset.union(second_dataset)

dataset_df = dataset_df.withColumn("Date Rptd", F.to_timestamp(
    "Date Rptd", "MM/dd/yyyy hh:mm:ss a").cast("date"))
dataset_df = dataset_df.withColumn("DATE OCC", F.to_timestamp(
    "DATE OCC", "MM/dd/yyyy hh:mm:ss a").cast("date"))

policedept_df = spark.read.csv(
    "hdfs://okeanos-master:54310/user/input/LAPD_Police_Stations.csv", header=True, inferSchema=True)

dataset_filt_df = (
    dataset_df
    .filter(F.col("LAT") != 0.0)
    .filter(F.col("Weapon Used Cd").between(100, 199))
    .select(["DR_NO", "LAT", "LON"])
)

policedept_filt_df = (
    policedept_df
    .select(["DIVISION", "X", "Y"])
    .withColumnRenamed("DIVISION", "division")
)


def dist(x1, y1, x2, y2):
    return distance.geodesic((x1, y1), (x2, y2)).km


dist_udf = F.udf(dist, FloatType())

joined_df = (
    dataset_filt_df
    .crossJoin(policedept_filt_df)
    .withColumn("Distance", dist_udf(F.col("Y"), F.col("X"), F.col("LAT"), F.col("LON")))
)

window_spec = Window.partitionBy('DR_NO').orderBy('Distance')
mindist_df = (
    joined_df
    .withColumn('min_distance', F.min('Distance').over(window_spec))
)

mindist_df = (
    mindist_df
    .filter(mindist_df['Distance'] == mindist_df['min_distance'])
```

```
    .select(["division", "Distance"])
)

final_df = (
    mindist_df
    .groupBy("division")
    .agg(F.avg("Distance").alias("average_distance"),
         F.count("division").alias("#")
         )
    .orderBy(F.col("#").desc())
)

final_df.show(21)

spark.stop()
```