ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Δ.Π.Μ.Σ. ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΜΗΧΑΝΙΚΗ ΜΑΘΗΣΗ

Διαχείριση Δεδομένων Μεγάλης Κλίμακας

Εξαμηνιαία Εργασία

Λυδία Ιωάννα Κολίτση, Α.Μ. : 03400252
Αντώνιος Μπαροτσάκης, Α.Μ. : 03400260

19 Μαΐου 2025

Περιεχόμενα

Κατάλογος σχημάτων

Κατάλογος πινάκων

1 Εισαγωγή

Η παρούσα εργασία εκπονήθηκε στο πλαίσιο του μαθήματος «Διαχείριση Δεδομένων Μεγάλης Κλίμακας» του ΔΠΜΣ «Επιστήμη Δεδομένων και Μηχανική Μάθηση» του ΕΜΠ. Σκοπός της εργασίας είναι η ανάλυση συνόλων δεδομένων μεγάλης κλίμακας σχετικά με την εγκληματικότητα και δημογραφικά στοιχεία της κομητείας του Los Angeles, με χρήση των τεχνολογιών Apache Spark και Apache Hadoop σε ένα πλήρως κατανεμημένο περιβάλλον υποδομής βασισμένο σε Kubernetes και HDFS.

Η υλοποίηση συνδυάζει βασικές αρχές του προγραμματιστικού μοντέλου MapReduce, καθώς και τη χρήση των Resilient Distributed Dataset (RDD), DataFrame και SQL APIs του Spark, με στόχο την εκτέλεση σύνθετων ερωτημάτων. Παράλληλα, μελετάται η απόδοση των διαφορετικών προσεγγίσεων και αξιολόγηση της επεκτασιμότητας (scalability) των υλοποιήσεων. Εξετάστηκε η επίδραση παραμέτρων όπως η επιλογή του API (RDD vs DataFrame/SQL), η χρήση User-Defined Functions (UDFs), το format αποθήκευσης των δεδομένων (CSV vs Parquet) και η παραμετροποίηση των Spark jobs, συμπεριλαμβανομένου του αριθμού των executors, των διαθέσιμων πυρήνων (cores) και της εκχωρημένης μνήμης ανά executor. Επιπλέον, αναλύονται οι στρατηγικές συνένωσης (join strategies) που επιλέχθηκαν αυτόματα από τον Catalyst Optimizer, αξιολογώντας την ορθότητα και την αποδοτικότητά τους.

Στις επόμενες ενότητες της παρούσας αναφοράς, παρουσιάζεται αρχικά μια λεπτομερής περιγραφή των συνόλων δεδομένων που χρησιμοποιήθηκαν (Ενότητα 2) και του υπολογιστικού περιβάλλοντος υλοποίησης (Ενότητα 3). Ακολουθεί η ανάλυση της διαδικασίας προεπεξεργασίας των δεδομένων, με έμφαση στη μετατροπή τους σε μορφή Parquet (Ενότητα 4). Στην κύρια ενότητα της εργασίας (Ενότητα 5), αναλύεται η υλοποίηση κάθε ενός από τα τέσσερα ερωτήματα, παραθέτοντας τον κώδικα, τα αποτελέσματα, καθώς και την ανάλυση απόδοσης και τις στρατηγικές join όπου αυτό απαιτείται, συμπεριλαμβανομένων των πειραμάτων κλιμάκωσης. Τέλος, η εργασία ολοκληρώνεται με τη σύνοψη των βασικών συμπερασμάτων που προέκυψαν (Ενότητα 6).

Οι κώδικες που χρησιμοποιήθηκαν για την υλοποίηση της εν λόγω εργασίας βρίσκονται στο αποθετήριο github, στη διεύθυνση: **https://github.com/imlydianna/bigdata_project-ntua-2025**.

2 Περιγραφή Δεδομένων

Στο πλαίσιο της παρούσας εργασίας χρησιμοποιήθηκαν έξι σύνολα δεδομένων που σχετίζονται με την εγκληματικότητα και τα κοινωνικοοικονομικά χαρακτηριστικά του Los Angeles. Τα δεδομένα είναι διαθέσιμα στο κατανεμημένο σύστημα αρχείων HDFS του εργαστηρίου και προέρχονται από δημόσια αποθετήρια των αρχών της πόλης. Ακολουθεί συνοπτική περιγραφή κάθε συνόλου:

2.1 Los Angeles Crime Data (2010–2019 & 2020–2025)

Αποτελεί το κύριο dataset της ανάλυσης και περιλαμβάνει περίπου 3.120.000 καταγραφές εγκληματικών περιστατικών στο Los Angeles από το 2010 έως σήμερα. Τα δεδομένα παρέχονται σε δύο αρχεία .csv, ένα για την περίοδο 2010–2019 και ένα για την περίοδο 2020–2025, τα οποία θα ενοποιηθούν στο πλαίσιο της εργασίας. Τα χαρακτηριστικά του dataset παρουσιάζονται στον Πίνακα 1.

Όνομα Στήλης	**Περιγραφή**	**Data Type**
dr_no	Επίσημος αριθμός αρχείου (2ψήφιο έτος, ID περιοχής και 5 ψηφία)	Text
time_occ	Ώρα συμβάντος σε μορφή 24ωρου	Text
area	Κωδικός από 1 έως 21 που υποδηλώνει γεωγραφική περιοχή του LAPD	Text
area_name	Όνομα της γεωγραφικής περιοχής/τμήματος περιπολίας	Text
rpt_dist_no	Τετραψήφιος κωδικός της υπο-περιοχής αναφοράς εντός της περιοχής	Text
crm_cd	Κωδικός εγκλήματος που διαπράχθηκε (ίδιος με το Crime Code 1)	Text
crm_cd_desc	Περιγραφή του εγκλήματος σύμφωνα με τον αντίστοιχο κωδικό	Text
mocodes	Κωδικοί MO (modus operandi) για τη μέθοδο διάπραξης του εγκλήματος	Text
vict_age	Ηλικία του θύματος (δύο ψηφία)	Text
vict_sex	Φύλο του θύματος (F: Female, M: Male, X: Unknown)	Text
vict_descent	Εθνικότητα/Καταγωγή του θύματος (κωδικοποιημένη)	Text
premis_cd	Κωδικός είδους τοποθεσίας όπου συνέβη το περιστατικό	Text
premis_desc	Περιγραφή του είδους τοποθεσίας	Text
weapon_used_cd	Κωδικός του τύπου όπλου που χρησιμοποιήθηκε	Text
weapon_desc	Περιγραφή του όπλου (αν υπάρχει)	Text
status	Κωδικός κατάστασης υπόθεσης	Text
status_desc	Περιγραφή του status code	Text
crm_cd_1	Κύριος κωδικός εγκλήματος (πιο σοβαρό περιστατικό)	Text
crm_cd_2	Επιπλέον (δευτερεύων) κωδικός εγκλήματος	Text
crm_cd_3	Τρίτος πιθανός κωδικός εγκλήματος (λιγότερο σοβαρός)	Text
crm_cd_4	Τέταρτος πιθανός κωδικός εγκλήματος (λιγότερο σοβαρός)	Text
location	Διεύθυνση όπου συνέβη το περιστατικό	Text
cross_street	Κοντινή διασταύρωση στην αναφερόμενη διεύθυνση	Text
part_1_2	Κατηγορία εγκλήματος: Part I ή Part II σύμφωνα με το FBI UCR	Number
lat	Γεωγραφικό πλάτος του σημείου (Latitude)	Number
lon	Γεωγραφικό μήκος του σημείου (Longitude)	Number
date_rptd	Ημερομηνία καταγραφής του περιστατικού από την αστυνομία	Floating Timestamp
date_occ	Ημερομηνία που συνέβη το περιστατικό	Floating Timestamp

Πίνακας 1: Χαρακτηριστικά του συνόλου δεδομένων *Los Angeles Crime Data (2010–2025)* και η περιγραφή τους.

2.2 Census Populations by Zip Code (2010)

Παρουσιάζει δημογραφικά στοιχεία ανά ταχυδρομικό κώδικα (ZIP Code) στην κομητεία του Los Angeles, σύμφωνα με την απογραφή του 2010. Περιλαμβάνει 319 σειρές με τα χαρακτηριστικά που περιγράφονται στον Πίνακα 2.

Όνομα Στήλης	**Περιγραφή (στα ελληνικά)**	**Data Type**
`zip_code`	Ταχυδρομικός Κωδικός (Zip Code) της περιοχής απογραφής	Number
`total_population`	Συνολικός πληθυσμός κατοίκων στην περιοχή	Number
`median_age`	Διάμεση ηλικία κατοίκων στην περιοχή	Number
`total_males`	Συνολικός αριθμός ανδρών κατοίκων	Number
`total_females`	Συνολικός αριθμός γυναικών κατοίκων	Number
`total_households`	Συνολικός αριθμός νοικοκυριών στην περιοχή	Number
`average_household_size`	Μέσο μέγεθος νοικοκυριού σε αριθμό ατόμων	Number

Πίνακας 2: Χαρακτηριστικά του συνόλου δεδομένων *Census Populations by Zip Code (2010).*

2.3 Median Household Income by Zip Code (2015)

Παρέχει εκτιμήσεις για το μέσο εισόδημα ανά νοικοκυριό σε κάθε ταχυδρομικό κώδικα της κομητείας του Los Angeles, βάσει στοιχείων του 2015, που απεικονίζονται στον Πίνακα 3.

Όνομα Στήλης	**Περιγραφή (στα ελληνικά)**	**Data Type**
`zip_code`	Ταχυδρομικός Κωδικός της περιοχής	Integer
`community`	Όνομα περιοχής ή γειτονιάς	String
`estimated_median_income`	Εκτιμώμενο ετήσιο εισόδημα ανά νοικοκυριό σε δολλάρια - με σύμβολο $ και διαχωριστικό χιλιάδων (,)	Currency (String)

Πίνακας 3: Χαρακτηριστικά του συνόλου δεδομένων *Median Household Income by Zip Code (2015).*

2.4 LA Police Stations

Περιέχει γεωγραφικά δεδομένα για τα 21 αστυνομικά τμήματα του Los Angeles σε μορφή .csv και περιλαμβάνει τα χαρακτηριστικά που παρουσιάζονται στον Πίνακα 4.

Όνομα Στήλης	**Περιγραφή (στα ελληνικά)**	**Data Type**
`division`	Όνομα του αστυνομικού τμήματος	String
`location`	Περιγραφή της γεωγραφικής τοποθεσίας	String
`prec`	Αναγνωριστικός κωδικός του τμήματος (Precinct ID)	Integer
`geometry`	Συντεταγμένες γεωγραφικής θέσης (γεωμετρία σημείου WGS84 - πλάτος/μήκος)	Geometry(Point)

Πίνακας 4: Χαρακτηριστικά του συνόλου δεδομένων *LA Police Stations.*

2.5 MO Codes in Numerical Order

Το αρχείο MO_codes.txt περιέχει κωδικοποιημένες περιγραφές δραστηριότητας ή μεθόδου δράσης των δραστών (Modus Operandi). Χρησιμοποιείται για την κατηγοριοποίηση εγκλημάτων (π.χ. με εμπλοκή όπλων) μέσω αντιστοίχισης με το πεδίο `Mocodes` του συνόλου δεδομένων Crime Data. Όσον αφορά τη δομή του αρχείου, κάθε γραμμή ξεκινά με έναν ακέραιο κωδικό MO και ακολουθεί η περιγραφή.

3 Περιβάλλον Υλοποίησης

Για την υλοποίηση της παρούσας εργασίας αξιοποιήθηκε η κατανεμημένη υποδομή του εργαστηρίου CSLab, η οποία βασίζεται σε Apache Spark και Hadoop Distributed File System (HDFS), ενορχηστρωμένα από Kubernetes.

Ειδικότερα, η σύνδεση στην απομακρυσμένη υποδομή Kubernetes του εργαστηρίου πραγματοποιήθηκε μέσω OpenVPN, χρησιμοποιώντας το κατάλληλο αρχείο ρυθμίσεων .ovpn που παρείχε το εργαστήριο. Το αρχείο ρυθμίσεων config του Kubernetes, που επίσης παραχωρήθηκε, τοποθετήθηκε στον κατάλογο /.kube/config του τοπικού μας συστήματος, συγκεκριμένα σε περιβάλλον WSL (Windows Subsystem for Linux) Ubuntu, το οποίο αποτέλεσε το βασικό περιβάλλον ανάπτυξης. Για την οπτική παρακολούθηση και διαχείριση των πόρων του Kubernetes (όπως pods, services, deployments) σε πραγματικό χρόνο, χρησιμοποιήθηκε επικουρικά το εργαλείο k9s.

Η ανάπτυξη των PySpark scripts πραγματοποιήθηκε τοπικά, εντός του περιβάλλοντος WSL. Τα αρχεία Python (.py) που περιείχαν τον κώδικα των queries, αφού δημιουργούνταν και αποθηκεύονταν σε έναν τοπικό φάκελο στο WSL, μεταφορτώνονταν στο HDFS στον προσωπικό μας χώρο, μέσω εντολών μορφής:

```
hdfs dfs -put -f <local_script_path.py> /user/<username>/code/
```

Η υποβολή εργασιών Spark (Spark jobs) έγινε μέσω της εντολής spark-submit, με ορισμό του master στο Kubernetes API endpoint του εργαστηρίου (k8s://https://termi7.cslab.ece.ntua.gr:6443). Κάθε εφαρμογή εκτελέστηκε σε cluster mode, εντός του προσωπικού μας namespace στο Kubernetes (<username>-priv), με τα Spark containers (driver και executors) να βασίζονται στην επίσημη εικόνα apache/spark. Το εκάστοτε Python script της εφαρμογής καθοριζόταν στο spark-submit μέσω του HDFS path του, για παράδειγμα, hdfs:///user/<username> /code/Query.py.

```
spark-submit \
  --master k8s://https://termi7.cslab.ece.ntua.gr:6443 \
  --deploy-mode cluster \
  --name Query \
  --conf spark.kubernetes.namespace=<username>-priv \
  --conf spark.kubernetes.container.image=apache/spark \
  --conf spark.eventLog.enabled=true \
  --conf spark.eventLog.dir=hdfs://hdfs-namenode:9000/user/<username>/logs \
  hdfs://hdfs-namenode:9000/user/<username>/project_scripts/Query.py
```

Η βασική διαχείριση των δεδομένων έγινε μέσω HDFS, όπου είχαν ήδη αποθηκευτεί όλα τα απαιτούμενα datasets (.csv, .txt). Κατά την έναρξη της υλοποίησης, όλα τα αρχεία .csv και .txt μετατράπηκαν σε μορφή .parquet και αποθηκεύτηκαν στον κατάλογο /user/<username>/data/parquet/. Αυτή η μετατροπή πραγματοποιήθηκε με χρήση PySpark, ώστε να επιταχυνθεί η προσπέλαση των δεδομένων και να αξιοποιηθούν τα πλεονεκτήματα του columnar storage format που προσφέρει το Parquet, ειδικά για αναλυτικά queries. Έτσι, η περαιτέρω επεξεργασία των δεδομένων σε όλα τα queries έγινε κυρίως πάνω σε Parquet αρχεία.

Για την παρακολούθηση της εκτέλεσης των εργασιών Spark και την αξιολόγηση της αποδοτικότητάς τους, αξιοποιήθηκε ένας τοπικός Spark History Server. Αυτός ο server στήθηκε μέσω Docker Compose στον προσωπικό μας υπολογιστή (εντός WSL). Ο History Server ρυθμίστηκε να διαβάζει τα Spark event logs απευθείας από την HDFS (από τον κατάλογο /user/<username>/logs όπου γράφονταν αυτόματα). Παρείχε ένα web UI μέσω browser, μέσω του οποίου μπορούσαμε να εξετάζουμε αναλυτικά την κατανομή των tasks σε executors, το execution DAG, τα stages, τις shuffle operations και, κυρίως, τις στρατηγικές συνενώσεων (join strategies) που επέλεγε ο Spark Catalyst Optimizer για κάθε query.

4 Προεπεξεργασία Δεδομένων

Η προεπεξεργασία των δεδομένων είχε ως στόχο τη μετατροπή των αρχικών συνόλων δεδομένων από τις μορφές .csv και .txt σε μια βελτιστοποιημένη, columnar μορφή αποθήκευσης, το Apache Parquet, στο HDFS. Αξίζει να σημειωθεί ότι η columnar φύση του Parquet επιτρέπει σημαντικά ταχύτερη ανάγνωση και επεξεργασία δεδομένων στα επόμενα στάδια της ανάλυσης, ειδικά για τα αναλυτικά queries που εκτελούνται από το Apache Spark. Η όλη διαδικασία υλοποιήθηκε μέσω ενός Spark script με όνομα prepare_data.py, αξιοποιώντας κατά κύριο λόγο το Spark DataFrame API.

Αρχικά, στο script prepare_data.py, πραγματοποιείται η αρχικοποίηση της SparkSession, η οποία αποτελεί το σημείο εισόδου για κάθε λειτουργικότητα του Spark. Ορίζεται ένα όνομα για την εφαρμογή και ρυθμίζεται το επίπεδο καταγραφής μηνυμάτων (log level) σε "WARN" για καθαρότερη έξοδο κατά την εκτέλεση.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import split, trim, col

def main():
    username = "username"

    spark = SparkSession.builder \
        .appName(f"Data Preparation for {username}") \
        .getOrCreate()
    sc = spark.sparkContext
    sc.setLogLevel("WARN")

    print(f"Ξεκίνησε η προετοιμασία δεδομένων για τον χρήστη: {username}")
```

Στη συνέχεια, ορίζονται οι διαδρομές εισόδου για τα αρχικά αρχεία δεδομένων, τα οποία βρίσκονται στον κοινόχρηστο κατάλογο /user/root/data του HDFS, καθώς και η βασική διαδρομή εξόδου για τα επεξεργασμένα αρχεία Parquet, η οποία είναι εξατομικευμένη στον προσωπικό φάκελο του χρήστη στο HDFS.

```
    base_input_path = "hdfs://hdfs-namenode:9000/user/root/data"
    crime_2010_input_path = f"{base_input_path}/LA_Crime_Data_2010_2019.csv"
    # ... ομοίως για τα υπόλοιπα αρχεία εισόδου ...

    output_base_path = f"hdfs://hdfs-namenode:9000/user/{username}/data/parquet"
```

Για τα πέντε από τα έξι σύνολα δεδομένων (LA_Crime_Data_2010_2019.csv, LA_Crime_Data_2020_2025.csv, LA_Police_Stations.csv, LA_income_2015.csv, 2010_Census_Populations_by_Zip_Code.csv), τα οποία ήταν σε μορφή CSV, η διαδικασία μετατροπής σε Parquet ήταν σχετικά άμεση. Το Spark script χρησιμοποίησε τη μέθοδο spark.read.csv() για την ανάγνωση κάθε αρχείου. Η παράμετρος header=True υποδεικνύει στο Spark ότι η πρώτη γραμμή του CSV περιέχει τα ονόματα των στηλών, ενώ η inferSchema=True δίνει εντολή στο Spark να προσπαθήσει να ανιχνεύσει αυτόματα τον τύπο δεδομένων κάθε στήλης. Αν και η επιλογή inferSchema=True είναι βολική, καθώς το Spark διαβάζει ένα δείγμα των δεδομένων για να καθορίσει τους τύπους των στηλών, αξίζει να σημειωθεί ότι αυτή η διαδικασία μπορεί να εισάγει ένα μικρό επιπλέον κόστος χρόνου κατά την ανάγνωση, ειδικά σε πολύ μεγάλα σύνολα δεδομένων. Ωστόσο, για τις ανάγκες της παρούσας εργασίας και δεδομένων των συγκεκριμένων μεγεθών, η χρήση του inferSchema=True κρίθηκε επαρκής για την ευκολία που προσφέρει.

Μετά την επιτυχή ανάγνωση και δημιουργία του DataFrame, κάθε σύνολο δεδομένων αποθηκεύτηκε στο HDFS σε μορφή Parquet στην καθορισμένη διαδρομή εξόδου, φροντίζοντας να γίνεται αντικατάσταση τυχόν προηγούμενων αρχείων στην ίδια θέση - mode("overwrite").

```
# prepare_data.py παράδειγμα επεξεργασίας CSV
    try:
        print(f"Επεξεργασία: {crime_2010_input_path}")
        crime_2010_df = spark.read.csv(crime_2010_input_path, header=True, inferSchema=True)
```

```
        crime_2010_output = f"{output_base_path}/crime_2010_2019"
        crime_2010_df.write.mode("overwrite").parquet(crime_2010_output)
        print(f"Ok: Το {crime_2010_output} γράφτηκε σε Parquet.")
    except Exception as e:
        print(f"Σφάλμα στην επεξεργασία Crime Data 2010-2019: {e}")
```

Ειδικός χειρισμός απαιτήθηκε για το αρχείο MO_codes.txt. Αυτό το αρχείο ήταν απλό αρχείο κειμένου, όπου κάθε γραμμή περιείχε έναν αριθμητικό κωδικό (MO Code) ακολουθούμενο από την περιγραφή του (Description), διαχωρισμένα από τον πρώτο κενό χαρακτήρα. Για την επεξεργασία του, αρχικά διαβάστηκε ως ένα DataFrame με μία μόνο στήλη "value" τύπου String, που περιείχε κάθε γραμμή του αρχείου.

Στη συνέχεια, εφαρμόστηκαν μετασχηματισμοί για τον διαχωρισμό του κωδικού από την περιγραφή. Η συνάρτηση split(col("value"), " ", 2) χρησιμοποιήθηκε για να διαχωρίσει κάθε γραμμή στον πρώτο κενό χαρακτήρα (το 2 ως όριο εξασφαλίζει ότι μόνο ο πρώτος κενός χαρακτήρας χρησιμοποιείται ως διαχωριστής, διατηρώντας τα υπόλοιπα κενά στην περιγραφή). Έπειτα, έγινε εξαγωγή του κωδικού και της περιγραφής, καθαρισμός από περιττά κενά με τη trim(), μετατροπή του κωδικού σε ακέραιο (cast("int")), και φιλτράρισμα για την αφαίρεση κενών ή μη έγκυρων γραμμών.

```
# prepare_data.py επεξεργασία( MO_codes.txt)
    mo_codes_input_path = f"{base_input_path}/MO_codes.txt"
    try:
        print(f"Επεξεργασία: {mo_codes_input_path}")
        mo_codes_raw_df = spark.read.text(mo_codes_input_path)

        # Σπάσιμο κάθε γραμμής στον πρώτο κενό χαρακτήρα
        # getItem(0) -> Κωδικός, getItem(1) -> Περιγραφή
        mo_codes_df = mo_codes_raw_df.withColumn(
            "parts", split(col("value"), " ", 2)
        ).filter(col("value").isNotNull() & (col("value") != "") & (col("parts").getItem(0).
    isNotNull())).select(
            trim(col("parts").getItem(0)).cast("int").alias("MO_Code"),
            trim(col("parts").getItem(1)).alias("Description")
        ).filter(col("MO_Code").isNotNull())


        mo_codes_output = f"{output_base_path}/mo_codes"
        mo_codes_df.write.mode("overwrite").parquet(mo_codes_output)
        print(f"Ok: Το {mo_codes_output} γράφτηκε σε Parquet.")
    except Exception as e:
        print(f"Σφάλμα στην επεξεργασία MO Codes: {e}")
```

Με την ολοκλήρωση αυτού του σταδίου, όλα τα απαραίτητα σύνολα δεδομένων μετατράπηκαν επιτυχώς και αποθηκεύτηκαν σε μορφή Parquet στον προσωπικό μας φάκελο στο HDFS: *hdfs://hdfs-namenode:9000/user/<username>/data/parquet/*. Τα δεδομένα είναι πλέον έτοιμα για χρήση στα queries που θα περιγραφούν στις επόμενες ενότητες.

5 Υλοποίηση Ερωτημάτων (Q1–Q4)

5.1 Query 1

Στο Query 1 ζητείται η ταξινόμηση, σε φθίνουσα ηλικιακή σειρά, των θυμάτων σε εγκλήματα που περιλαμβάνουν οποιαδήποτε μορφή “βαριάς σωματικής βλάβης”. Η λογική της υλοποίησης περιλαμβάνει αρχικά την φόρτωση των απαραίτητων δεδομένων (είτε απευθείας από τα αρχικά αρχεία CSV για την RDD υλοποίηση, είτε από τα προ-επεξεργασμένα αρχεία Parquet για τις DataFrame υλοποιήσεις) και την ένωσή τους, καθώς και την κανονικοποίηση των ονομάτων στηλών προς αποφυγήν σφαλμάτων. Εν συνεχεία γίνεται φιλτράρισμα μόνο των περιστατικών που περιλαμβάνουν "aggravated assault" στην περιγραφή (`Crm Cd Desc`), και έχουν καθορισμένη ηλικία θύματος (`Vict Age`) κατηγοριοποίηση των ηλικιών των θυμάτων στις τέσσερις δοθείσες ομάδες, ομαδοποίηση και καταμέτρηση των εγγραφών ανά ηλικιακή ομάδα και ταξινόμηση των αποτελεσμάτων. Το Query 1 υλοποιήθηκε με τρεις διαφορετικούς τρόπους, αξιοποιώντας τα RDD API, το DataFrame API χωρίς User-Defined Function (UDF), και το DataFrame API με χρήση UDF.

5.1.1 Υλοποίηση με RDD API

Η προσέγγιση με RDDs περιλαμβάνει την απευθείας ανάγνωση των αρχικών αρχείων .csv από το HDFS χρησιμοποιώντας sc.textFile(). Τα δύο RDDs ενοποιούνται με union() και γίνεται φιλτράρισμα του header. Η πρώτη γραμμή (header) του ενοποιημένου RDD αφαιρείται, και το υπόλοιπο RDD μετασχηματίζεται έτσι ώστε κάθε γραμμή (που είναι ένα string) να αναλυθεί (parsed) σε μια λίστα από strings, χρησιμοποιώντας τη βιβλιοθήκη csv της Python.

```
csv_2010 = "hdfs://hdfs-namenode:9000/user/root/data/LA_Crime_Data_2010_2019.csv"
csv_2020 = "hdfs://hdfs-namenode:9000/user/root/data/LA_Crime_Data_2020_2025.csv"
output_path = f"hdfs://hdfs-namenode:9000/user/{username}/results/RddQ1_csv_{job_id}"

rdd_2010 = sc.textFile(csv_2010)
rdd_2020 = sc.textFile(csv_2020)

rdd = rdd_2010.union(rdd_2020)

header = rdd.first()
columns = header.split(",")
print("\nHeader columns:")
for i, col in enumerate(columns):
    print(f"{i}: {col}")

data = rdd.filter(lambda line: line != header)

def parse_csv(line):
    return next(csv.reader(StringIO(line)))

rows = data.map(parse_csv)
```

Στη συνέχεια, εφαρμόζεται φιλτράρισμα. Μας ενδιαφέρουν οι στήλες `Crm Cd Desc` (στήλη με δείκτη 9, μετά το parsing) και `Vict Age` (με δείκτη 11). Κρατάμε μόνο τις γραμμές όπου αυτές οι στήλες δεν είναι κενές, η γραμμή έχει επαρκή αριθμό στηλών, και η περιγραφή του εγκλήματος (αφού έχει μετατραπεί σε πεζά) περιέχει τη συμβολοσειρά "aggravated assault".

```
# Indices: Crm Cd Desc -> 9, Vict Age -> 11
filtered = rows.filter(lambda x: len(x) > 11 and x[9] != "" and x[11] != "") \
               .filter(lambda x: "aggravated assault" in x[9].lower())
```

Για την κατηγοριοποίηση των ηλικιών ορίζεται η συνάρτηση Python `age_group(age)`, η οποία λαμβάνει την ηλικία ως string, την μετατρέπει σε ακέραιο, και επιστρέφει την αντίστοιχη ηλικιακή ομάδα. Αυτή

η συνάρτηση εφαρμόζεται σε κάθε στοιχείο του RDD μέσω ενός map transformation, το οποίο δημιουργεί ένα νέο RDD από ζεύγη (key-value) της μορφής (ηλικιακή_ομάδα, 1). Οι εγγραφές όπου η ηλικιακή ομάδα είναι "Unknown" (π.χ., λόγω αποτυχίας μετατροπής της ηλικίας σε ακέραιο) φιλτράρονται. Τέλος, εφαρμόζεται reduceByKey για την άθροιση των τιμών (counts) για κάθε ηλικιακή ομάδα, και το τελικό RDD ταξινομείται με sortBy με βάση το πλήθος, σε φθίνουσα σειρά.

```
def age_group(age):
    try:
        age = int(age)
        if age < 18:
            return "Children"
        elif age <= 24:
            return "Young Adults"
        elif age <= 64:
            return "Adults"
        else:
            return "Elderly"
    except:
        return "Unknown"

result = (
    filtered.map(lambda x: (age_group(x[11]), 1))
            .filter(lambda x: x[0] != "Unknown")
            .reduceByKey(lambda a, b: a + b)
            .sortBy(lambda x: x[1], ascending=False)
)
```

Τα αποτελέσματα συλλέγονται με collect() και εμφανίζονται (Σχήμα 1). Τέλος, μετατρέπονται σε μορφή κατάλληλη για αποθήκευση ως text file.

```
print("\Κατανομήn ηλικιακών ομάδων θυμάτων (Aggravated Assault):")
for group, count in result.collect():
    print(f"{group}: {count}")

result.map(lambda x: f"{x[0]},{x[1]}") \
      .coalesce(1) \
      .saveAsTextFile(output_path)
```

AgeGroup	Count
Adults	122727
Young Adults	34036
Children	16124
Elderly	6082

Σχήμα 1: Κατανομή ηλικιακών ομάδων θυμάτων σε aggravated assault.

Ένα βασικό χαρακτηριστικό της λειτουργίας του Spark είναι η "lazy evaluation" (νωχελική αποτίμηση), που σημαίνει ότι οι μετασχηματισμοί που ορίζουν ένα RDD, όπως το result rdd στην προκειμένη περίπτωση, δεν εκτελούνται άμεσα αλλά καταγράφονται ως ένα πλάνο εκτέλεσης. Αυτοί οι υπολογισμοί ενεργοποιούνται μόνο όταν συναντάται μια "action" (ενέργεια). Στον παραπάνω κώδικα αυτό το RDD υπολογίζεται δύο φορές. Η πρώτη εκτέλεση πυροδοτείται από την action collect(), η οποία χρησιμοποιείται για την εμφάνιση των

αποτελεσμάτων στην κονσόλα. Στη συνέχεια, εκτελείται μια δεύτερη action, η saveAsTextFile(output_path), για την αποθήκευση των ίδιων αποτελεσμάτων στο HDFS. Επειδή το result δεν έχει αποθηκευτεί στη μνήμη (μέσω caching), το Spark αναγκάζεται να επαναϋπολογίσει ολόκληρη την αλυσίδα των μετασχηματισμών που οδηγούν στο result για να εκτελέσει και αυτή τη δεύτερη action. Συνεπώς, η "συνταγή" για τη δημιουργία του result εκτελείται δύο φορές. Αυτή η προσέγγιση, της εμφάνισης των αποτελεσμάτων μέσω collect() και της ταυτόχρονης αποθήκευσής τους, ακολουθήθηκε σε όλες τις υλοποιήσεις των queries για λόγους πληρότητας και άμεσης επισκόπισης. Έτσι, οι χρόνοι εκτέλεσης είναι συγκρίσιμοι μεταξύ τους, παρόλο που ενδέχεται να περιλαμβάνουν αυτόν τον διπλό υπολογισμό.

5.1.2 Υλοποίηση με DataFrame API (χωρίς UDF)

Η προσέγγιση με DataFrames περιλαμβάνει την ανάγνωση των αρχείων Parquet ως DataFrames, των οποίων τα ονόματα στηλών κανονικοποιούνται με τη βοήθεια της συνάρτησης trim_colnames, και την ενοποίησή τους σε ένα ενιαίο DataFrame, df. Γίνεται φιλτράρισμα χρησιμοποιώντας τη μέθοδο filter() του DataFrame API και τις ενσωματωμένες συναρτήσεις col(), isNotNull() και rlike() για case-insensitive αναζήτηση της φράσης "aggravated assault".

```
path_2010 = f"hdfs://hdfs-namenode:9000/user/{username}/data/parquet/crime_2010_2019"
path_2020 = f"hdfs://hdfs-namenode:9000/user/{username}/data/parquet/crime_2020_present"
output_dir = f"hdfs://hdfs-namenode:9000/user/{username}/results/DfQ1_{job_id}"

def trim_colnames(df):
    for col_name in df.columns:
        df = df.withColumnRenamed(col_name, col_name.strip())
    return df

df = df.filter(
    (col("Crm Cd Desc").isNotNull()) &
    (col("Vict Age").isNotNull()) &
    (col("Crm Cd Desc").rlike("(?i)aggravated assault"))
)
```

Η κατηγοριοποίηση των ηλικιών γίνεται με τη δημιουργία μιας νέας στήλης "AgeGroup" χρησιμοποιώντας τη μέθοδο withColumn() και μια αλυσίδα από .when() και .otherwise() συνθήκες.

```
df = df.withColumn(
    "AgeGroup",
    when(col("Vict Age") < 18, "Children")
    .when(col("Vict Age") <= 24, "Young Adults")
    .when(col("Vict Age") <= 64, "Adults")
    .otherwise("Elderly")
    # No need for .otherwise("Unknown"), groupBy will ignore any nulls in AgeGroup
)
```

Τέλος, η ομαδοποίηση γίνεται με groupBy("AgeGroup"), ακολουθούμενη από count() για τον υπολογισμό του πλήθους, και orderBy(col("count").desc()) για την ταξινόμηση.

```
result = df.groupBy("AgeGroup").count().orderBy(col("count").desc())

print("\n Κατανομή ηλικιακών ομάδων θυμάτων σε aggravated assault:")
result.show()

result.coalesce(1).write.mode("overwrite").option("header", True).csv(output_dir)

print("\nPhysical Plan:")
result.explain(True)
```

5.1.3 Υλοποίηση με DataFrame API + UDF

Αυτή η προσέγγιση είναι παρόμοια με την προηγούμενη DataFrame υλοποίηση, αλλά η κατηγοριοποίηση των ηλικιών ενσωματώνεται σε μια User-Defined Function (UDF). Η φόρτωση, η κανονικοποίηση ονομάτων στηλών και το αρχικό φιλτράρισμα είναι ίδια με την DfQ1.py.

Ορίζεται μια συνάρτηση Python map_age_to_group(age) παρόμοια με αυτή της RDD υλοποίησης. Αυτή η συνάρτηση Python "τυλίγεται" σε μια Spark UDF χρησιμοποιώντας udf(map_age_to_group, StringType()). Το StringType() καθορίζει τον τύπο δεδομένων που επιστρέφει η UDF. Η UDF αυτή εφαρμόζεται για τη δημιουργία της στήλης "AgeGroup".

```
def map_age_to_group(age):
    try:
        age = int(age)
        if age < 18:
            return "Children"
        elif age <= 24:
            return "Young Adults"
        elif age <= 64:
            return "Adults"
        else:
            return "Elderly"
    except:
        return "Unknown"

age_group_udf = udf(map_age_to_group, StringType())

df = df.withColumn("AgeGroup", age_group_udf(col("Vict Age")))
```

Πριν την τελική ομαδοποίηση, εφαρμόζεται ένα επιπλέον φίλτρο για την αφαίρεση των γραμμών που η UDF επέστρεψε "Unknown". Η υπόλοιπη διαδικασία (groupBy, count, orderBy) είναι ίδια με την υλοποίηση χωρίς UDF.

```
result = df.filter(col("AgeGroup") != "Unknown") \
           .groupBy("AgeGroup") \
           .count() \
           .orderBy(col("count").desc())
```

```
result = df.filter(col("AgeGroup") != "Unknown") \
           .groupBy("AgeGroup") \
           .count() \
           .orderBy(col("count").desc())
```

5.1.4 Σχόλια και Ανάλυση Απόδοσης

Οι τρεις υλοποιήσεις του Query 1 προσφέρουν διαφορετικούς τρόπους προσέγγισης του προβλήματος, με διαφορετικές επιδόσεις, όπως φαίνεται στον Πίνακα 5. Η ανάλυση αυτή βασίζεται σε μετρικές που συλλέχθηκαν από το Spark History Server, εστιάζοντας κυρίως στη διάρκεια εκτέλεσης του κύριου job, την αξιοποίηση του Catalyst Optimizer, τον όγκο των δεδομένων shuffle, και τον αριθμό των tasks.

Η πρώτη υλοποίηση πραγματοποιήθηκε αποκλειστικά με RDD API και περιλαμβάνει την απευθείας ανάγνωση των αρχικών αρχείων .csv και την εφαρμογή μετασχηματισμών σε χαμηλότερο επίπεδο αφαίρεσης. Αυτή η προσέγγιση απαιτεί χειροκίνητο parsing των γραμμών του .csv και, δεν επωφελείται από τις βελτιστοποιήσεις του Catalyst optimizer, ούτε από την αποδοτικότητα της columnar μορφής Parquet. Αυτό αντικατοπτρίζεται στον σημαντικά μεγαλύτερο χρόνο εκτέλεσης της κύριας εργασίας (Job Duration: 15s) σε σύγκριση με τις DataFrame προσεγγίσεις. Παρατηρούμε επίσης ελαφρώς περισσότερα tasks (12) στο κύριο stage. Είναι ενδιαφέρον ότι ο όγκος των δεδομένων shuffle read/write (1.1 KiB) είναι ο χαμηλότερος από τις τρεις υλοποιήσεις,

πιθανόν λόγω της φύσης των RDD transformations που εφαρμόστηκαν σε σχέση με την εσωτερική διαχείριση του groupBy στα DataFrames.

Η δεύτερη υλοποίηση βασίστηκε αποκλειστικά στο DataFrame API (χωρίς UDF), διαβάζοντας τα προ-επεξεργασμένα δεδομένα Parquet. Ένα σημαντικό πλεονέκτημα εδώ είναι ότι το Parquet είναι ένα format που αποθηκεύει το schema μαζί με τα δεδομένα. Όταν το Spark διαβάζει αρχεία Parquet, μπορεί να αξιοποιήσει άμεσα αυτό το ενσωματωμένο schema, αποφεύγοντας την ανάγκη για inferSchema (που απαιτείται για την απευθείας ανάγνωση από CSVs και προσθέτει χρόνο). Αυτό καθιστά την ανάγνωση των δεδομένων εξαιρετικά αποδοτική. Αυτή η αποδοτική ανάγνωση και η σαφής γνώση του schema από την αρχή, παρέχουν στον Catalyst Optimizer του Spark μια σταθερή βάση για να εφαρμόσει εκτενείς βελτιστοποιήσεις στο query plan. Το αποτέλεσμα είναι η ταχύτερη εκτέλεση με διαφορά (Job Duration: 4s). Η χρήση των ενσωματωμένων συναρτήσεων (when, rlike, groupBy, count) επιτρέπει στο Spark να κατανοήσει πλήρως τη λογική και να δημιουργήσει ένα πολύ αποδοτικό πλάνο. Το shuffle (2.9 KiB) είναι ελαφρώς αυξημένο σε σχέση με την RDD προσέγγιση, αλλά ο συνολικός χρόνος εκτέλεσης είναι σημαντικά μικρότερος.

Η τρίτη εκδοχή αξιοποιεί DataFrame API συνδυαστικά με User Defined Function (UDF) για την ηλικιακή ομαδοποίηση, διαβάζοντας επίσης δεδομένα Parquet. Ενώ προσφέρει ευελιξία επιτρέποντας τη χρήση κώδικα Python απευθείας στο pipeline, η UDF λειτουργεί ως "black box" για τον Catalyst Optimizer. Αυτό οδηγεί σε αύξηση του χρόνου εκτέλεσης (Job Duration: 6s) σε σύγκριση με την προσέγγιση που χρησιμοποιεί αμιγώς DataFrame (4s), αν και παραμένει σημαντικά ταχύτερη από την RDD υλοποίηση (15s). Οι υπόλοιπες μετρικές, όπως το shuffle (2.9 KiB) κι ο αριθμός των tasks (10), είναι παρόμοια με την παραπάνω DataFrame υλοποίηση, υποδεικνύοντας ότι το κύριο overhead προέρχεται από την εκτέλεση της μη-βελτιστοποιημένης UDF και την επικοινωνία μεταξύ JVM και Python interpreter.

Συνολικά, τα αποτελέσματα επιβεβαιώνουν ότι η χρήση του DataFrame API χωρίς UDFs, σε συνδυασμό με την αποθήκευση των δεδομένων σε μορφή Parquet, προσφέρει την καλύτερη απόδοση για αυτό το ερώτημα. Αυτή η προσέγγιση επιτρέπει την πλήρη αξιοποίηση των μηχανισμών βελτιστοποίησης του Spark (Catalyst Optimizer) και την αποδοτική ανάγνωση δεδομένων. Η χρήση UDFs, αν και προσφέρει ευελιξία για την υλοποίηση πολύπλοκης λογικής σε Python, εισάγει ένα αισθητό overhead στην απόδοση. Τέλος, η προσέγγιση με RDD API και απευθείας ανάγνωση από αρχεία .csv είναι η λιγότερο αποδοτική, κυρίως λόγω του κόστους του parsing και της περιορισμένης δυνατότητας βελτιστοποίησης από τον Catalyst.

Πίνακας 5: Σύγκριση απόδοσης των υλοποιήσεων RDD, DataFrame και DataFrame με UDF.

Υλοποίηση	Total Uptime	Job Duration (κύριο)	Catalyst Optimization	Shuffle Read/Write
RDD Q1	39s	15s	✗	1.1 KiB
DF Q1	38s	4s	✓	2.9 KiB
DF + UDF Q1	40s	6s	✗στο βήμα UDF	2.9 KiB

5.2 Query 2

Στο Query 2 ζητείται ο εντοπισμός, για κάθε έτος, των τριών αστυνομικών περιφερειών με το υψηλότερο ποσοστό περατωμένων υποθέσεων, όπου δηλαδή το πεδίο `Status Desc` είναι διαφορετικό από "UNK" ή "Invest Cont". Η στρατηγική της υλοποίησης περιλαμβάνει τη φόρτωση και ένωση των Crime Data από τις δύο χρονικές περιόδους (2010–2019 και 2020–2025), το φιλτράρισμα εγγραφών με έγκυρες ημερομηνίες (`DATE OCC`), ονόματα υποδιευθύνσεων (`AREA NAME`), και γεωγραφικές συντεταγμένες (`LAT` $\neq 0$ και `LON` $\neq 0$) και την εξαγωγή του έτους από την ημερομηνία σε νέο πεδίο `Year`. Κατόπιν, ορίζεται η μεταβλητή "Closed" που παίρνει την τιμή 1 αν `Status Desc` $\notin$ {"UNK", "Invest Cont"} και 0 διαφορετικά. Επιπλέον, γίνεται ομαδοποίηση ανά (`Year, AREA NAME`) για τη συνολική καταμέτρηση των υποθέσεων, τον υπολογισμό του πλήθους "κλειστών"

υποθέσεων, και την εξαγωγή του ποσοστού $Rate = \frac{Closed}{Total}$. Τέλος, γίνεται κατάταξη των περιφερειών ανά έτος, με βάση το ποσοστό Rate και επιλογή των top-3 ανά έτος, με χρήση dense_rank. Το Query 2 υλοποιήθηκε με χρήση των RDD API, DataFrame API και Spark SQL.

5.2.1 Υλοποίηση με RDD API

Για την υλοποίηση του Query 2 με χρήση του RDD API, ακολουθήθηκε μια προσέγγιση όπου αρχικά τα δεδομένα φορτώνονται από τα αρχεία Parquet ως DataFrames. Συγκεκριμένα, τα δύο σύνολα δεδομένων εγκλημάτων διαβάστηκαν με την spark.read.parquet(), τα ονόματα των στηλών τους κανονικοποιήθηκαν (αφαιρώντας τυχόν κενά), και στη συνέχεια τα δύο DataFrames ενώθηκαν σε ένα ενιαίο DataFrame, το df_all.Από αυτό το ενιαίο DataFrame (df_all), επιλέχθηκαν οι απαραίτητες στήλες για το query (`DATE OCC`, `AREA NAME`, `Status Desc`, `LAT`, `LON`) και το αποτέλεσμα αυτής της επιλογής μετατράπηκε σε ένα RDD μέσω της μεθόδου .rdd . Πάνω σε αυτό το RDD εφαρμόστηκε η υπόλοιπη λογική του query, αξιοποιώντας τις RDD transformations. Η διαδικασία περιλάμβανε αρχικό φιλτράρισμα για την αφαίρεση εγγραφών με τιμές null στις κρίσιμες στήλες, καθώς και την απομάκρυνση των περιστατικών που αντιστοιχούσαν στο "Null Island"."

```
rdd = df_all.select("DATE OCC", "AREA NAME", "Status Desc", "LAT", "LON").rdd \
    .filter(lambda row: row["DATE OCC"] and row["AREA NAME"] and row["Status Desc"]) \
    .filter(lambda row: not (row["LAT"] == 0.0 and row["LON"] == 0.0))  # Omit Null Island
```

Χρησιμοποιούνται οι βοηθητικές συναρτήσεις Python extract_year(date_str) για την εξαγωγή του έτους από τη συμβολοσειρά ημερομηνίας με string splitting και is_closed(status_desc), η οποία επιστρέφει True αν η υπόθεση είναι κλειστή και False διαφορετικά. Το αρχικό RDD μετασχηματίζεται με map. Κάθε γραμμή αντιστοιχίζεται σε ένα ζεύγος key-value: το κλειδί είναι ένα tuple (year, area_name) και η τιμή είναι ένα άλλο ζεύγος (1, closed_flag), όπου closed_flag είναι 1 αν η υπόθεση είναι κλειστή και 0 αλλιώς. Οι εγγραφές όπου το έτος δεν μπόρεσε να εξαχθεί φιλτράρονται. Στη συνέχεια, εφαρμόζεται reduceByKey για να αθροιστούν οι μετρητές (συνολικές υποθέσεις και κλειστές υποθέσεις) για κάθε μοναδικό ζεύγος (year, area_name).

```
def extract_year(date_str):
    try:
        return int(date_str.split("/")[2][:4])
    except:
        return None

def is_closed(status_desc):
    return status_desc not in ("UNK", "Invest Cont")

# Map: ((year, precinct), (1, closed_flag))
year_precinct_flags = rdd.map(lambda row: (
    (extract_year(row["DATE OCC"]), row["AREA NAME"]),
    (1, 1 if is_closed(row["Status Desc"]) else 0)
)).filter(lambda x: x[0][0] is not None)

agg = year_precinct_flags.reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))
```

Το aggregated rdd, agg, περιέχει τώρα στοιχεία της μορφής ((year, area_name), (total_cases, closed_cases)). Ένα επόμενο map transformation υπολογίζει το ποσοστό κλειστών υποθέσεων και αναδιατάσσει τα δεδομένα ώστε το έτος να γίνει το κλειδί. Το groupByKey() ομαδοποιεί όλες τις πληροφορίες των τμημάτων (όνομα τμήματος και ποσοστό) ανά έτος.

```
year_to_precincts = agg.map(lambda x: (
    x[0][0],  # key: year
    (x[0][1], x[1][1] / x[1][0])  # value: (precinct, closed_rate)
)).groupByKey() # (year, iterable_of_precinct_rates)
```

Τέλος, ορίζεται μια συνάρτηση Python top3_with_rank, η οποία δέχεται ως είσοδο ένα ζεύγος (year, iterable _of_precinct_rates). Μέσα στη συνάρτηση, τα τμήματα ταξινομούνται βάσει του ποσοστού τους (σε φθίνουσα σειρά), επιλέγονται τα τρία κορυφαία, και προστίθεται ο αριθμός κατάταξης (rank). Αυτή η συνάρτηση εφαρμόζεται σε κάθε ομάδα του year_to_precinct_data RDD χρησιμοποιώντας flatMap για να "ξεδιπλωθούν" τα αποτελέσματα σε μια δομή. Το τελικό RDD ταξινομείται κατά έτος και κατάταξη.

```
def top3_with_rank(group):
    year, precincts = group
    sorted_top = sorted(precincts, key=lambda x: -x[1])[:3]
    return [(year, p[0], p[1], i+1) for i, p in enumerate(sorted_top)]

result = year_to_precincts.flatMap(top3_with_rank).sortBy(lambda x: (x[0], x[3]))
```

2010	Rampart	32.85	#1
2010	Olympic	31.52	#2
2010	Harbor	29.36	#3
2011	Olympic	35.03	#1
2011	Rampart	32.50	#2
2011	Harbor	28.52	#3
2012	Olympic	34.33	#1
2012	Rampart	32.46	#2
2012	Harbor	29.53	#3
2013	Olympic	33.58	#1
2013	Rampart	32.12	#2
2013	Harbor	29.71	#3
2014	Southeast	66.67	#1
2014	Topanga	50.00	#2
2014	Mission	33.33	#3
2015	Van Nuys	32.27	#1
2015	Mission	30.47	#2
2015	West Valley	30.34	#3
2016	Van Nuys	31.95	#1
2016	West Valley	30.96	#2
2016	Foothill	29.92	#3
2017	Van Nuys	32.06	#1
2017	Mission	31.07	#2
2017	Foothill	30.47	#3
2018	Foothill	30.73	#1
2018	Mission	30.72	#2
2018	Van Nuys	28.92	#3
2019	Mission	30.71	#1
2019	West Valley	30.59	#2
2019	N Hollywood	29.24	#3
2020	West Valley	30.79	#1
2020	Mission	30.33	#2
2020	Harbor	29.79	#3
2021	Mission	30.55	#1
2021	West Valley	29.11	#2
2021	Foothill	28.22	#3
2022	West Valley	26.66	#1
2022	Harbor	26.39	#2
2022	Topanga	26.31	#3
2023	Topanga	26.95	#1
2023	Foothill	26.90	#2
2023	Mission	25.99	#3
2024	N Hollywood	20.15	#1
2024	Foothill	18.06	#2
2024	77th Street	17.73	#3
2025	Central	100.00	#1
2025	Topanga	20.00	#2
2025	Foothill	14.29	#3

Σχήμα 2: Top-3 precincts ανά έτος.

5.2.2 Υλοποίηση με DataFrame API

Στην προσέγγιση με DataFrame API, μετά τη φόρτωση, ένωση, κανονικοποίηση ονομάτων στηλών και το αρχικό φιλτράρισμα (συμπεριλαμβανομένου του "Null Island"), προστίθενται οι νέες στήλες `Year` και `Closed`.

```
df = df.filter(
    (col("DATE OCC").isNotNull()) &
    (col("AREA NAME").isNotNull()) &
    (col("Status Desc").isNotNull()) &
    ~((col("LAT") == 0.0) & (col("LON") == 0.0))  # Omit Null Island
)

```

```
df = df.withColumn("Year", year(to_timestamp(col("DATE OCC"), "MM/dd/yyyy hh:mm:ss a"))) \
                        .withColumn("Closed", when(~col("Status Desc").isin("UNK", "Invest
    Cont"), 1).otherwise(0))
```

Το DataFrame ομαδοποιείται (groupBy) κατά `Year` και `AREA NAME`. Χρησιμοποιώντας agg(), υπολογίζεται το συνολικό πλήθος εγγραφών (count("*")) και το άθροισμα της `Closed_Flag` (που δίνει τον αριθμό των κλειστών υποθέσεων). Στη συνέχεια, υπολογίζεται το ποσοστό closed_case_rate.

```
agg = df.groupBy("Year", "AREA NAME") \
                .agg(
                    count("*").alias("Total"),
                    _sum("Closed").alias("Closed")
                ) \
                .withColumn("closed_case_rate", (col("Closed") / col("Total")) * 100)
```

Για την κατάταξη των αστυνομικών τμημάτων εντός κάθε έτους, αξιοποιήθηκαν οι Window Functions. Αρχικά, ορίστηκε ένα παράθυρο (WindowSpec) το οποίο διαμερίζει (partitionBy) τα δεδομένα ανά Year και τα ταξινομεί (orderBy) εντός κάθε κομματιού της διαμέρισης σε φθίνουσα σειρά, με βάση το closed_case_rate. Στη συνέχεια, η συνάρτηση dense_rank().over(windowSpec) εφαρμόστηκε για να υπολογίσει και να προσθέσει μια νέα στήλη Rank που αντικατοπτρίζει την κατάταξη κάθε τμήματος εντός του έτους του. Τέλος, φιλτράρονται οι εγγραφές διατηρώντας μόνο αυτές όπου το Rank είναι μικρότερο ή ίσο του 3 (δηλαδή, τα top-3 τμήματα), και τα τελικά αποτελέσματα ταξινομούνται κατά Year και Rank σε αύξουσα σειρά, πριν τυπωθούν και αποθηκευτούν."

```
window = Window.partitionBy("Year").orderBy(col("closed_case_rate").desc())

result = agg.withColumn("Rank", dense_rank().over(window)) \
            .filter(col("Rank") <= 3) \
            .orderBy("Year", "Rank")

print("\Κατανομήn Top-3 precincts ανά έτος με βάση το ποσοστό κλεισμένων υποθέσεων:")
rows = result.select("Year", "AREA NAME", "closed_case_rate", "Rank").collect()
for row in rows:
    year = row["Year"]
    precinct = row["AREA NAME"]
    rate = f"{row['closed_case_rate']:.2f}"
    rank = f"#{row['Rank']}"
    print(f"{year:<6} {precinct:<20} {rate:<20} {rank}")


# Αποθήκευση στο HDFS
result.select("Year", "AREA NAME", "closed_case_rate", "Rank") \
         .coalesce(1) \
         .write.mode("overwrite") \
         .option("header", True) \
         .csv(output_path)

# Εμφάνιση φυσικού πλάνου
#print("\nPhysical Plan:")
#result.explain(True)
```

5.2.3 Υλοποίηση με SQL API

Αυτή η προσέγγιση μεταφράζει τη λογική του DataFrame API σε ένα SQL query. Τα αρχικά βήματα (φόρτωση, ένωση, κανονικοποίηση, φιλτράρισμα και προσθήκη των στηλών `Year` και `Closed`) είναι πανομοιότυπα με την υλοποίηση DfQ2.py μέχρι τη δημιουργία του df. Το τελευταίο καταχωρείται ως μια προσωρινή όψη (Temporary View) με το όνομα "crimes" ώστε να μπορεί να χρησιμοποιηθεί σε SQL queries.

```
df.createOrReplaceTempView("crime")
```

Εν συνεχεία, γράφεται ένα SQL query που υπολογίζει το σύνολο, τις κλειστές υποθέσεις και το ποσοστό ανά Year και AREA NAME (που μετονομάζεται σε Precinct) ως agg και εφαρμόζει τη window function DENSE_RANK() πάνω στα αποτελέσματα του agg για να κατατάξει τα τμήματα ανά έτος ως ranked. Το τελικό SELECT επιλέγει τις επιθυμητές στήλες, φιλτράρει για Rank <= 3, και ταξινομεί. Το query εκτελείται με spark.sql(sql_query).

```
query = """
WITH base AS (
    SELECT
        Year,
        `AREA NAME` AS precinct,
        COUNT(*) AS Total,
        SUM(CASE WHEN `Status Desc` NOT IN ('UNK', 'Invest Cont') THEN 1 ELSE 0 END) AS Closed
    FROM crime
    GROUP BY Year, `AREA NAME`
),
rates AS (
    SELECT
        Year,
        precinct,
        ROUND(100.0 * Closed / Total, 10) AS closed_case_rate
    FROM base
),
ranked AS (
    SELECT *,
           DENSE_RANK() OVER (PARTITION BY Year ORDER BY closed_case_rate DESC) AS rank
    FROM rates
)
SELECT Year, precinct, closed_case_rate, rank
FROM ranked
WHERE rank <= 3
ORDER BY Year ASC, rank ASC
"""

result = spark.sql(query)
```

5.2.4 Σχόλια και Ανάλυση Απόδοσης

Οι τρεις υλοποιήσεις του Query 2 (RDD, DataFrame, και SQL API) παράγουν ισοδύναμα αποτελέσματα, δηλαδή τα τρία κορυφαία αστυνομικά τμήματα ανά έτος με βάση το ποσοστό περατωμένων υποθέσεων (Σχήμα 2). Η ανάλυση των επιδόσεών τους, όπως φαίνεται στον Πίνακα 6, αποκαλύπτει σημαντικές διαφορές που οφείλονται κυρίως στον τρόπο με τον οποίο κάθε API αλληλεπιδρά με τον Catalyst Optimizer του Spark και διαχειρίζεται τα δεδομένα.

Η πρώτη υλοποίηση με RDD API πραγματοποιεί την επεξεργασία μέσω map-reduce λογικής, με πιο χειροκίνητη ή προγραμματιστική προσέγγιση στο grouping και sorting. Παρότι προσφέρει καλό έλεγχο της ροής δεδομένων, στερείται βελτιστοποιήσεων από τον Catalyst Optimizer και παρουσιάζει σημαντικό overhead: το κύριο job έχει διάρκεια 15s, ενώ το Total Uptime είναι 50s. Αντίθετα, οι υλοποιήσεις DataFrame και SQL επιτυγχάνουν παρόμοιο, ταχύτερο χρόνο εκτέλεσης του κύριου job 7s, αξιοποιώντας πλήρως τον Catalyst Optimizer. Παρόλο που παρουσιάζουν μεγαλύτερο όγκο shuffle (DF: 98.8/93.7 KiB, SQL: 99.9/95.0 KiB), οι συνολικές βελτιστοποιήσεις οδηγούν σε ανώτερη απόδοση. Η SQL υλοποίηση εμφανίζει ελαφρώς καλύτερο συνολικό χρόνο λειτουργίας (43s έναντι 47s του DataFrame και 50s του RDD), καθιστώντας τις DataFrame/SQL προσεγγίσεις τις βέλτιστες επιλογές για αυτό το query.

Πίνακας 6: Σύγκριση απόδοσης των υλοποιήσεων RDD, DataFrame και DataFrame με UDF.

Υλοποίηση	Total Uptime	Job Duration (κύριο)	Catalyst Optimization	Shuffle Read/Write
RDD Q2	50s	15s	✗	65.8 KiB / 34.5 KiB
DF Q2	47s	7s	✓	98.8 KiB / 93.7 KiB
SQL Q2	43s	7s	✓	99.9 KiB / 95.0 KiB

5.3 Query 3

Στο Query 3 ζητείται ο υπολογισμός του μέσου εισοδήματος ανά άτομο (per capita) σε κάθε ταχυδρομικό κώδικα (ZIP code) του Los Angeles, συνδυάζοντας δύο πηγές πληροφορίας, το Median household income (εισόδημα ανά νοικοκυριό) και το Average household size (μέσο πλήθος ατόμων ανά νοικοκυριό). Η λογική υπολογισμού είναι η εξής:

$$\text{Income per Person} = \frac{\text{Estimated median household income}}{\text{Average household size}}$$

Η υπολογιστική ροή περιλαμβάνει την ανάγνωση των datasets la_income_2015 για τα εισοδήματα (string με σύμβολα $ και ,) και census_2010_zipcode για το μέγεθος νοικοκυριών, τον καθαρισμό των πεδίων (αφαίρεση $ και , από τα εισοδήματα και μετατροπή σε αριθμητική μορφή καθώς και μετατροπή Average Household Size σε FloatType και φιλτράρισμα μη ρεαλιστικών τιμών (π.χ. >15)), την εσωτερική συνένωση (join) των δύο datasets βάσει Zip Code, αφού γίνουν κατάλληλες μετατροπές τύπων και padding και τέλος τον υπολογισμό του εισοδήματος ανά άτομο. Το Query 3 υλοποιήθηκε με χρήση RDD API και DataFrame API (με είσοδο Parquet και CSV).

5.3.1 Υλοποίηση με RDD API

Στην προσέγγιση με RDDs τα Parquet αρχεία διαβάζονται αρχικά ως DataFrames και στη συνέχεια μετατρέπονται σε RDDs. Χρησιμοποιούνται βοηθητικές συναρτήσεις Python, όπως η clean_income για τον καθαρισμό της συμβολοσειράς του εισοδήματος και η try_keys για την ευέλικτη πρόσβαση σε στήλες με πιθανά διαφορετικά ονόματα.

```
def try_keys(row, options):
    for key in options:
        if key in row.asDict():
            return row[key]
    return None

def clean_income(val):
    if val is None:
        return None
    try:
        val_str = str(val).strip().replace('$', '').replace(',', '')
        val_float = float(val_str)
        if val_float <= 0:
            return None
        return val_float
    except:
        return None
```

Η επεξεργασία των RDDs income_rdd και hhsize_rdd γίνεται μέσω map transformations που εξάγουν τον Zip Code και το καθαρισμένο εισόδημα ή το μέσο μέγεθος νοικοκυριού αντίστοιχα, και εφαρμόζονται φίλτρα για την αφαίρεση εγγραφών με μη έγκυρες τιμές.

```
hhsize_rdd = df_pop.rdd.map(lambda row: (
    str(try_keys(row, ["Zip Code", "zip_code"])).zfill(5),
    try_keys(row, ["Average Household Size", "average_household_size"])
)).filter(lambda x: x[0] is not None and x[1] is not None) \
  .map(lambda x: (x[0], float(x[1]))) \
  .filter(lambda x: 0 < x[1] <= 15)

income_rdd = df_income.rdd.map(lambda row: (
    str(try_keys(row, ["Zip Code", "ZipCode"])).zfill(5),
    clean_income(try_keys(row, ["Estimated Median Income", "Income"]))
)).filter(lambda x: x[0] is not None and x[1] is not None)
```

Τα δύο RDDs, τα οποία έχουν πλέον τη μορφή (zip_code, value), συνενώνονται με join(). Το αποτέλεσμα είναι ένα RDD της μορφής (zip_code, (income, household_size)). Ένα τελικό map υπολογίζει το εισόδημα ανά άτομο, και το αποτέλεσμα ταξινομείται κατά ταχυδρομικό κώδικα.

```
joined = income_rdd.join(hhsize_rdd)

result = joined.map(lambda x: (
    x[0],
    round(x[1][0] / x[1][1], 2)
)).sortBy(lambda x: x[0])
```

zip	income_per_person
90001	7701.59
90002	6975.46
90003	7299.76
90004	14876.19
90005	12456.8
90006	10070.61
90007	7434.67
90008	15692.7
90010	24484.49
90011	6477.73

Σχήμα 3: Ενδεικτικά αποτελέσματα με .take(10).

5.3.2 Υλοποίηση με DataFrame API (Parquet)

Η προσέγγιση με DataFrames είναι πιο συνοπτική. Στην πρώτη υλοποίηση τα δεδομένα φορτώνονται απευθείας από τα αρχεία Parquet και γίνεται προεπεξεργασία και καθαρισμός. Για το εισόδημα χρησιμοποιείται regexp_replace για την αφαίρεση $ και , και cast(FloatType()) για τη μετατροπή, οι Τ.Κ. μετατρέπονται σε string και εφαρμόζονται φίλτρα. Για τον πληθυσμό, η στήλη Average Household Size μετατρέπεται σε FloatType, οι Τ.Κ. μετατρέπονται σε string και εφαρμόζονται φίλτρα.

```
df_pop = spark.read.parquet(pop_path) \
    .select(
        col("Zip Code").cast("string").alias("zip"),
        col("Average Household Size").cast("float").alias("household_size")
    ).filter("household_size > 0")

df_income = spark.read.parquet(income_path) \
    .select(
```

```
        regexp_replace(regexp_replace(col("Estimated Median Income"), "\\$", ""), ",", "")
        .cast(FloatType()).alias("income"),
        col("Zip Code").cast("string").alias("zip")
    ).filter("income > 0")
```

Τελικά γίνεται συνένωση (Join), υπολογισμός και ταξινόμηση.

```
df_joined = df_income.join(df_pop, on="zip", how="inner")
df_joined.explain(mode="formatted")

df_result = df_joined.withColumn(
    "income_per_person", spark_round(col("income") / col("household_size"), 2)
).select("zip", "income_per_person").orderBy("zip")
```

5.3.3 Υλοποίηση με DataFrame API (CSV)

Αυτή η υλοποίηση διαβάζει απευθείας τα αρχικά CSV αρχεία. Η επιλογή ("inferSchema", True) είναι σημαντική εδώ, καθώς τα CSV δεν έχουν εγγενές schema.

```
pop_csv_path = "hdfs://hdfs-namenode:9000/user/root/data/2010_Census_Populations_by_Zip_Code.
    csv"
income_csv_path = "hdfs://hdfs-namenode:9000/user/root/data/LA_income_2015.csv"

df_pop = spark.read.option("header", True).option("inferSchema", True).csv(pop_csv_path)
df_income = spark.read.option("header", True).option("inferSchema", True).csv(income_csv_path)
```

Ο καθαρισμός της στήλης εισοδήματος, το padding των Zip Codes, η συνένωση, ο υπολογισμός του μέσου εισοδήματος ανά άτομο και η ταξινόμηση ακολουθούν παρόμοια λογική με την Parquet DataFrame υλοποίηση, προσαρμοσμένη στα αρχικά ονόματα στηλών των CSV αρχείων.

```
df_income_clean = df_income.withColumn(
    "Cleaned_Income",
    regexp_replace(col("Estimated Median Income"), "[$,]", "").cast("float")
)

joined = df_pop.join(df_income_clean, on="Zip Code", how="inner")
joined.explain(mode="formatted")

result = joined.withColumn(
    "avg_income_per_person",
    round(col("Cleaned_Income") / col("Average Household Size"), 2)
).select("Zip Code", "avg_income_per_person").orderBy("Zip Code")

result.show(10)
```

5.3.4 Σχόλια και Ανάλυση Απόδοσης

Κατά τον πειραματισμό με την εισαγωγή δεδομένων για το Query 3 χρησιμοποιώντας DataFrame API, παρατηρήθηκαν διαφορές στην εκτέλεση ανάλογα με το format των αρχείων εισόδου (CSV έναντι Parquet), όπως φαίνεται στον Πίνακα 7. Παρόλο που ο χρόνος εκτέλεσης του κύριου job (Job Duration) ήταν παρόμοιος (4s) και για τις δύο περιπτώσεις, η χρήση Parquet είναι προτιμότερη. Το Parquet, ως columnar format με ενσωματωμένο schema, επιτρέπει ταχύτερη ανάγνωση μέσω projection pushdown και αποφεύγει το overhead του inferSchema που απαιτείται για τα CSV. Επιπλέον, υποστηρίζει αποτελεσματική συμπίεση και predicate pushdown, βελτιστοποιήσεις που γίνονται πιο εμφανείς σε μεγαλύτερους όγκους δεδομένων.

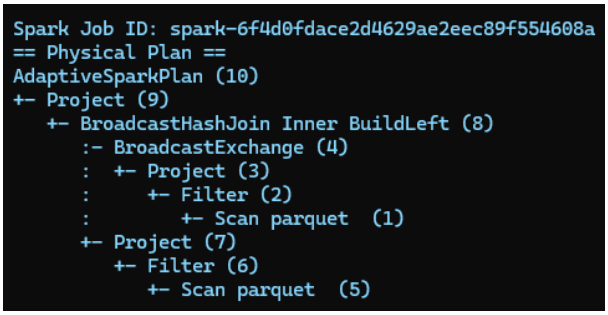
Στην προκειμένη περίπτωση, ο ελαφρώς χαμηλότερος συνολικός χρόνος λειτουργίας (Total Uptime) της Parquet υλοποίησης (40s έναντι 41s της CSV) επιτεύχθηκε παρά τον οριακά μεγαλύτερο (αν και συνολικά πολύ

μικρό) όγκο shuffle (16.4 KiB έναντι 13.9 KiB της CSV). Αυτό υποδηλώνει ότι, ενώ ο κύριος υπολογισμός ήταν εξίσου γρήγορος, τα οφέλη από την αποδοτικότερη αρχική φόρτωση και προετοιμασία των δεδομένων Parquet (λόγω projection/predicate pushdown και αποφυγής inferSchema) υπερκάλυψαν τυχόν αμελητέες διαφορές στο shuffle. Συμπερασματικά, παρότι οι διαφορές μπορεί να μην είναι δραματικές σε μικρά datasets, η επιλογή Parquet προσφέρει σημαντικά πλεονεκτήματα απόδοσης και scalability για επεξεργασία δεδομένων με Spark.

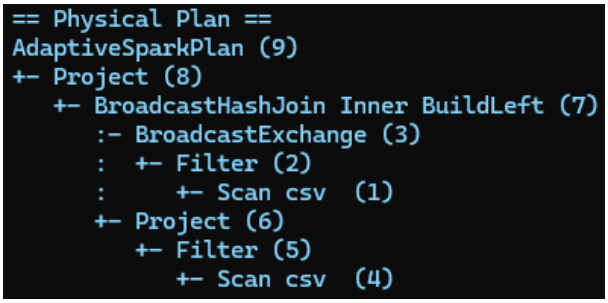
Πίνακας 7: Σύγκριση απόδοσης μεταξύ RDD, DF (Parquet) και DF (CSV) για το Query 3.

Υλοποίηση	Total Uptime	Job Duration (κύριο)	Catalyst Optimization	Shuffle Read/Write
RDD Q3	41s	5s	✗	6.2 KiB
DF Q3 (Parquet)	40s	4s	✓	16.4 KiB / 16.4 KiB
DF Q3 (CSV)	41s	4s	✓	13.9 KiB / 13.9 KiB

5.3.5 Ανάλυση Επιλογής Join Στρατηγικής από τον Catalyst Optimizer



Σχήμα 4: Physical Plan της υλοποίησης DataFrame Parquet.



Σχήμα 5: Physical Plan της υλοποίησης DataFrame CSV.

Κατά την εκτέλεση του Query 3, το οποίο περιλαμβάνει ένωση των datasets για πληθυσμό και εισόδημα με κοινό κλειδί τον ταχυδρομικό κώδικα, ο Catalyst Optimizer του Spark επιλέγει ως στρατηγική join τον μηχανισμό Broadcast Hash Join και στις δύο περιπτώσεις υλοποίησης με DataFrame API—είτε με αρχεία Parquet είτε με CSV. Η επιλογή αυτή είναι εμφανής από τα παραγόμενα physical plans (screenshots 4 και 5), όπου εντοπίζεται το στάδιο BroadcastExchange πριν το BroadcastHashJoin.

Η επιλογή του Broadcast Hash Join είναι απολύτως λογική και αναμενόμενη, δεδομένων των χαρακτηριστικών του join: πρόκειται για μια απλή εσωτερική ένωση (Inner Join) σε ένα κοινό πεδίο (zip), με τον ένα πίνακα να είναι σαφώς μικρότερος (πιθανότατα το dataset εισοδήματος, το οποίο περιλαμβάνει λιγότερες εγγραφές σε σχέση με τους πληθυσμούς). Ο Broadcast μηχανισμός μεταδίδει τον μικρότερο πίνακα σε όλους τους executors, αποφεύγοντας την ανάγκη για shuffle των δεδομένων, κάτι που μειώνει σημαντικά το δίκτυο και το

I/O overhead. Ο Spark Catalyst μπορεί να λάβει αυτή την απόφαση αυτόματα όταν το μέγεθος του build-side πίνακα είναι κάτω από το spark.sql.autoBroadcastJoinThreshold, το οποίο εξ ορισμού είναι περίπου 10MB.

Ειδικά στην περίπτωση της υλοποίησης με αρχεία Parquet, η στρατηγική Broadcast είναι ακόμα πιο αποδοτική, καθώς το push-down filtering που υποστηρίζει η columnar μορφή περιορίζει το μέγεθος των δεδομένων πριν την ένωση. Το γεγονός ότι χρησιμοποιείται Broadcast Hash Join επιβεβαιώνει ότι τα φιλτραρισμένα δεδομένα ήταν επαρκώς μικρά, και ο Optimizer κατάφερε να εντοπίσει το πιο ελαφρύ σχέδιο εκτέλεσης.

Αντίστοιχα, και στη δεύτερη περίπτωση με αρχεία CSV, παρότι η επεξεργασία είναι πιο δαπανηρή λόγω έλλειψης columnar storage και περιορισμένων optimizations κατά το filtering, η στρατηγική που επέλεξε ο Optimizer παρέμεινε Broadcast Join. Αυτό δηλώνει ότι είτε ο αριθμός των εγγραφών ήταν μικρός, είτε το filtering ήταν επαρκώς περιοριστικό, ώστε να δικαιολογήσει broadcast.

Συμπερασματικά, η επιλογή του Catalyst να εφαρμόσει Broadcast Hash Join και στις δύο περιπτώσεις είναι θεωρητικά τεκμηριωμένη και βέλτιστη. Αντίθετα, η χρήση Merge Join ή Shuffle Hash Join θα ήταν αναγκαία μόνο αν και οι δύο πίνακες ήταν σημαντικά μεγάλοι, απαιτώντας ταξινόμηση και ανακατανομή των δεδομένων στο cluster. Στην παρούσα περίπτωση, η στρατηγική που επιλέχθηκε οδηγεί σε πολύ γρήγορη ένωση με χαμηλό latency και χωρίς shuffle overhead.

5.4 Query 4

Στο Query 4 ζητείται να υπολογιστεί, ανά αστυνομικό τμήμα, ο αριθμός εγκλημάτων με εμπλοκή όπλων (πυροβόλων ή όχι) που έλαβαν χώρα πλησιέστερα σε αυτό, καθώς και η μέση απόστασή του από τις τοποθεσίες όπου σημειώθηκαν τα συγκεκριμένα περιστατικά, με τα αποτελέσματα ταξινομημένα κατά αριθμό περιστατικών, με φθίνουσα σειρά. Για το σκοπό αυτό φορτώνονται τα δεδομένα εγκλημάτων, MO codes και αστυνομικών τμημάτων. Εντοπίζονται οι MO codes που σχετίζονται με όπλα και τα εγκλήματα φιλτράρονται ώστε να περιλαμβάνουν μόνο αυτά με τους σχετικούς MO codes και έγκυρες συντεταγμένες. Γίνεται cross join των εγκλημάτων με τα τμήματα για να υπολογιστεί η απόσταση Haversine μεταξύ όλων των ζευγών. Για κάθε έγκλημα, εντοπίζεται το πλησιέστερο τμήμα. Τέλος, τα αποτελέσματα ομαδοποιούνται ανά τμήμα για να υπολογιστεί ο αριθμός εγκλημάτων και η μέση απόσταση. Το Query 4 υλοποιείται αποκλειστικά με DataFrame API, για βέλτιστη απόδοση και δυνατότητα παρακολούθησης scaling συμπεριφοράς με διαφορετικά cluster configurations.

5.4.1 Υλοποίηση με DataFrame API

Τα δεδομένα εγκληματικότητας από τις περιόδους 2010-2019 και 2020-σήμερα, οι κωδικοί Modus Operandi (MO codes), και οι πληροφορίες των αστυνομικών τμημάτων φορτώνονται από αρχεία Parquet. Εφαρμόζεται η βοηθητική συνάρτηση trim_colnames για την αφαίρεση τυχόν κενών χαρακτήρων από τα ονόματα των στηλών και τα δύο DataFrames εγκληματικότητας ενοποιούνται με unionByName για να δημιουργηθεί ένα ενιαίο DataFrame crime_df.

```
crime_2010_path = f"hdfs://hdfs-namenode:9000/user/{username}/data/parquet/crime_2010_2019"
crime_2020_path = f"hdfs://hdfs-namenode:9000/user/{username}/data/parquet/crime_2020_present"
mo_codes_path = f"hdfs://hdfs-namenode:9000/user/{username}/data/parquet/mo_codes"
stations_path = f"hdfs://hdfs-namenode:9000/user/{username}/data/parquet/la_police_stations"
output_path = f"hdfs://hdfs-namenode:9000/user/{username}/results/DfQ4_haversine_{job_id}"

# Καθαρισμός ονομάτων στηλών
def trim_colnames(df):
    new_columns = [c.strip() for c in df.columns]
    if new_columns == df.columns:
        return df
    return df.select([col(original_name).alias(new_name)
                      for original_name, new_name in zip(df.columns, new_columns)])
```

```

# Ανάγνωση και προεπεξεργασία
df_2010 = trim_colnames(spark.read.parquet(crime_2010_path))
df_2020 = trim_colnames(spark.read.parquet(crime_2020_path))
crime_df_initial = df_2010.unionByName(df_2020)

mo_df = trim_colnames(spark.read.parquet(mo_codes_path))
stations_df = trim_colnames(spark.read.parquet(stations_path))
```

Από το mo_df, φιλτράρονται οι εγγραφές όπου η στήλη Description (μετατραμμένη σε πεζά) περιέχει τις λέξεις "gun" ή "weapon" κι επιλέγονται μόνο οι αντίστοιχοι κωδικοί. Στο crime_df_initial, φιλτράρονται αρχικά οι εγγραφές όπου η στήλη Mocodes δεν είναι null. Στη συνέχεια, η στήλη Mocodes (η οποία περιέχει πολλαπλούς κωδικούς διαχωρισμένους με κενό) διασπάται (split) και "εκρήγνυται" (explode) ώστε κάθε MO code να αντιστοιχεί σε μια νέα γραμμή. Το προκύπτον DataFrame (crime_df_exploded_mocodes) συνενώνεται (inner join) με το weapon_mo DataFrame (που περιέχει τους κωδικούς όπλων) με βάση τον MO_Code. Εφαρμόζεται φιλτράρισμα για την αφαίρεση εγγραφών με μη έγκυρες γεωγραφικές συντεταγμένες (LAT, LON isNotNull και όχι 0.0 για την αφαίρεση του "Null Island"). Επιλέγονται οι στήλες DR_NO (μοναδικό αναγνωριστικό εγκλήματος), LAT, LON, και εφαρμόζεται distinct() για να διασφαλιστεί ότι κάθε έγκλημα (DR_NO) εξετάζεται μόνο μία φορά, ακόμα κι αν έχει πολλαπλούς MO codes που σχετίζονται με όπλα. Αυτό το βήμα δημιουργεί το crime_weapon_df.

```
# Εξαγωγή weapon-related MO codes
weapon_mo = mo_df.filter(
    lower(col("Description")).contains("gun") |
    lower(col("Description")).contains("weapon")
).select(col("MO_Code").cast("int").alias("code"))

# Ανάλυση MOcodes και φιλτράρισμα περιστατικών με όπλα
crime_df_exploded_mocodes = crime_df_initial.filter(col("Mocodes").isNotNull()) \
                                    .withColumn("Mocodes_split", split(col("Mocodes"), " ")) \
                                    .withColumn("MO_Code_str", explode(col("Mocodes_split"))) \
                                    .withColumn("MO_Code", col("MO_Code_str").cast("int"))

# Join με weapon_mo για να πάρουμε μόνο τα weapon related crimes
crime_weapon_df = crime_df_exploded_mocodes.join(weapon_mo, crime_df_exploded_mocodes["MO_Code
    "] == weapon_mo["code"], "inner") \
                                    .filter(col("LAT").isNotNull() & col("LON").isNotNull())
    \
                                    .filter(~((col("LAT") == 0.0) & (col("LON") == 0.0))) \
                                    .select("DR_NO", "LAT", "LON") \
                                    .distinct() # Για να έχουμε ένα DR_NO ανά τοποθεσία αν
    ένα έγκλημα έχει πολλαπλούς weapon mocodes
```

Από το crime_weapon_df, επιλέγονται οι στήλες DR_NO, LAT (ως crime_lat), και LON (ως crime_lon), μετατρέποντας τις συντεταγμένες σε float και αποθηκεύονται ως crime_locations. Από το stations_df, επιλέγονται οι στήλες DIVISION (ως division, με αφαίρεση κενών), Y (ως station_lat), και X (ως station_lon), επίσης ως float, και αποθηκεύονται ως stations. Πραγματοποιείται ένα Cartesian Join (crossJoin) μεταξύ του crime_locations και του stations. Αυτό είναι απαραίτητο καθώς πρέπει να υπολογιστεί η απόσταση κάθε εγκλήματος από όλα τα αστυνομικά τμήματα, και δεν υπάρχει φυσικό κλειδί συνένωσης μεταξύ των δύο DataFrames για αυτόν τον σκοπό. Το αποτέλεσμα είναι το crime_cross.

```
# Επιλογή συντεταγμένων περιστατικών
crime_locations = crime_weapon_df.select(
    col("DR_NO"),
    col("LAT").cast("float").alias("crime_lat"),
    col("LON").cast("float").alias("crime_lon")
```

```
)

# Συντεταγμένες LAPD σταθμών σε( μοίρες)
stations = stations_df.select(
    trim(col("DIVISION")).alias("division"),
    col("Y").cast("float").alias("station_lat"),
    col("X").cast("float").alias("station_lon")
)

# Cartesian Join μεταξύ περιστατικών και τμημάτων
crime_cross = crime_locations.crossJoin(stations)
```

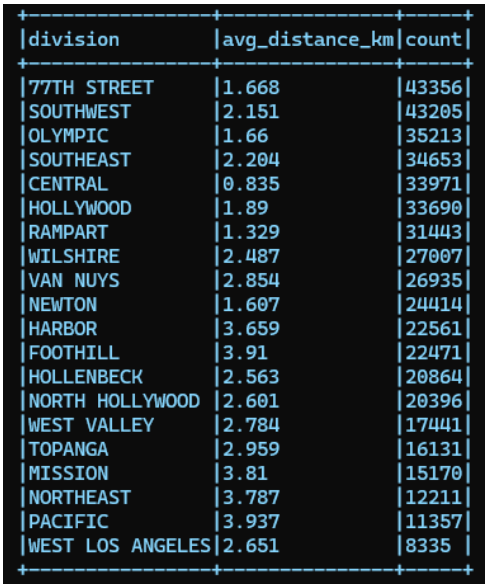
Η απόσταση μεταξύ κάθε ζεύγους (έγκλημα, αστυνομικό τμήμα) υπολογίζεται χρησιμοποιώντας τον τύπο Haversine, ο οποίος λαμβάνει υπόψη την καμπυλότητα της Γης και παρέχει ακριβέστερες αποστάσεις για γεωγραφικές συντεταγμένες. Η ακτίνα της Γης (R) ορίζεται σε 6371 χιλιόμετρα. Οι συντεταγμένες γεωγραφικού πλάτους και μήκους μετατρέπονται από μοίρες σε ακτίνια (radians). Εφαρμόζεται ο τύπος Haversine για τον υπολογισμό της απόστασης σε χιλιόμετρα, η οποία αποθηκεύεται σε μια νέα στήλη distance_km στο DataFrame crime_distances.

```
# Υπολογισμός απόστασης Haversine
R = 6371.0  # Ακτίνα της Γης σε χιλιόμετρα

# Μετατροπή μοιρών σε ακτίνια
crime_lat_rad = radians(col("crime_lat"))
crime_lon_rad = radians(col("crime_lon"))
station_lat_rad = radians(col("station_lat"))
station_lon_rad = radians(col("station_lon"))

# Υπολογισμός Δφ και Δλ
dlat = station_lat_rad - crime_lat_rad
dlon = station_lon_rad - crime_lon_rad

# Haversine formula
a = sin(dlat / 2)**2 + cos(crime_lat_rad) * cos(station_lat_rad) * sin(dlon / 2)**2
c = 2 * atan2(sqrt(a), sqrt(1 - a))
distance_km_col = R * c

crime_distances = crime_cross.withColumn("distance_km", distance_km_col)
```

Χρησιμοποιείται μια Window Function για να βρεθεί το πλησιέστερο αστυνομικό τμήμα για κάθε έγκλημα. Ορίζεται ένα WindowSpec που κάνει partition (διαμερισμό) των δεδομένων με βάση το DR_NO (δηλαδή, για κάθε έγκλημα ξεχωριστά) και τα ταξινομεί εντός κάθε partition με βάση την υπολογισμένη distance_km (σε αύξουσα σειρά). Η συνάρτηση rank().over(window_spec) προσθέτει μια στήλη κατάταξης rank_val. Φιλτράρονται οι γραμμές όπου rank_val == 1, κρατώντας έτσι μόνο την εγγραφή που αντιστοιχεί στο πλησιέστερο τμήμα για κάθε έγκλημα. Το αποτέλεσμα είναι το DataFrame nearest.

```
# Εύρεση πλησιέστερου τμήματος για κάθε περιστατικό με( βάση το DR_NO)

window_spec = Window.partitionBy("DR_NO").orderBy("distance_km") # Ταξινόμηση με τη νέα στήλη
    απόστασης
nearest = crime_distances.withColumn("rank_val", rank().over(window_spec)).filter(col("
    rank_val") == 1)
```

Το DataFrame nearest ομαδοποιείται (groupBy) κατά division (όνομα αστυνομικού τμήματος). Για κάθε τμήμα, υπολογίζονται η μέση απόσταση (avg("distance_km")), στρογγυλοποιημένη σε τρία δεκαδικά ψηφία (spark_round), ως avg_distance_km και ο συνολικός αριθμός των σχετιζόμενων με όπλα εγκλημάτων που είναι πλησιέστερα σε αυτό το τμήμα (count("*")), ως count. Τα αποτελέσματα ταξινομούνται (orderBy) με βάση τη



division	avg_distance_km	count
77TH STREET	1.668	43356
SOUTHWEST	2.151	43205
OLYMPIC	1.66	35213
SOUTHEAST	2.204	34653
CENTRAL	0.835	33971
HOLLYWOOD	1.89	33690
RAMPART	1.329	31443
WILSHIRE	2.487	27007
VAN NUYS	2.854	26935
NEWTON	1.607	24414
HARBOR	3.659	22561
FOOTHILL	3.91	22471
HOLLENBECK	2.563	20864
NORTH HOLLYWOOD	2.601	20396
WEST VALLEY	2.784	17441
TOPANGA	2.959	16131
MISSION	3.81	15170
NORTHEAST	3.787	12211
PACIFIC	3.937	11357
WEST LOS ANGELES	2.651	8335

Σχήμα 6: Ενδεικτικά αποτελέσματα με απόσταση σε km. Παρουσιάζονται οι πρώτες 20 σειρές.

στήλη count σε φθίνουσα σειρά. Εμφανίζεται το φυσικό πλάνο εκτέλεσης (result.explain(mode="formatted")) και τα τελικά αποτελέσματα.

```
# Ομαδοποίηση ανά division
result = nearest.groupBy("division").agg(
    spark_round(avg("distance_km"), 3).alias("avg_distance_km"), # Η μέση απόσταση είναι ήδη σε km
    count("*").alias("count")
).orderBy(col("count").desc())

# Εμφάνιση πλάνου εκτέλεσης Catalyst
print("\nCatalyst Physical Plan\n")
result.explain(mode="formatted")

# Αποθήκευση στο HDFS
result.coalesce(1).write.mode("overwrite").option("header", "true").csv(output_path)

# Εμφάνιση αποτελεσμάτων
print(" Top LAPD divisions με εγκλήματα που σχετίζονται με όπλα απόσταση( Haversine σε km):")
result.select("division", "avg_distance_km", "count").show(truncate=False)
```

5.4.2 Πειράματα Κλιμάκωσης (Scaling) για το Query 4

Το Query 4, λόγω του υπολογιστικά απαιτητικού crossJoin για τον υπολογισμό των αποστάσεων Haversine μεταξύ όλων των σχετιζόμενων με όπλα εγκλημάτων και όλων των αστυνομικών τμημάτων, αποτελεί ιδανικό υποψήφιο για τη διερεύνηση της συμπεριφοράς κλιμάκωσης (scaling). Πραγματοποιήθηκαν πειράματα οριζόντιας και κάθετης κλιμάκωσης, μεταβάλλοντας τους διαθέσιμους πόρους (αριθμός executors, πυρήνες ανά executor, μνήμη ανά executor), και αναλύθηκαν οι επιδόσεις μέσω των μετρικών του Spark History Server.

Πειράματα Οριζόντιας Κλιμάκωσης (Horizontal Scaling) – Σύνολο 8 Cores, 16GB RAM

Σε αυτό το σενάριο, ο συνολικός αριθμός πυρήνων (8 cores) και η συνολική μνήμη (16GB RAM) παρέμειναν σταθεροί, ενώ μεταβλήθηκε ο αριθμός των executors και, κατά συνέπεια, οι πόροι που αντιστοιχούσαν σε καθέναν.

- **Διαμόρφωση 2 executors × 4 cores/8GB memory**:

  Η εκτέλεση του Query 4 με διαμόρφωση 2 executors των 4 πυρήνων και 8 GB μνήμης πέτυχε η βέλτιστη απόδοση με Total Uptime 42s. Συνολικά ολοκληρώθηκαν 18 jobs (6 για το Query 1 και 8 για το Query 0), με σύνολο 74 tasks τα οποία κατανεμήθηκαν σχεδόν ομοιόμορφα στους δύο executors (34 και 40 tasks αντίστοιχα). Η συνολική διάρκεια εκτέλεσης των δύο queries ήταν 13 s και 7 s αντίστοιχα, με το Query 0 (υπολογισμός απόστασης – Haversine) να περιλαμβάνει το πιο απαιτητικό κομμάτι.

```
spark-submit \
  --master k8s://https://termi7.cslab.ece.ntua.gr:6443 \
  --deploy-mode cluster \
  --name query4_haversine_2x4 \
  --conf spark.kubernetes.namespace=username-priv \
  --conf spark.executor.instances=2 \
  --conf spark.executor.cores=4 \
  --conf spark.executor.memory=8g \
  --conf spark.driver.memory=4g \
  --conf spark.kubernetes.container.image=apache/spark \
  hdfs://hdfs-namenode:9000/user/username/project_code/DfQ4_Haversine.py
```

- **Διαμόρφωση 4 executors × 2 cores/4GB memory**:

  Διπλασιάζοντας τους executors και μειώνοντας στο ήμισυ τους πόρους ανά executor, το Total Uptime αυξήθηκε στα 53s. Η εκτέλεση του Query 0 (κύριο υπολογιστικό φορτίο) είχε διάρκεια 15 s, ενώ το Query 1 εκτελέστηκε σε 6 s, με σύνολο 18 jobs και 74 tasks. Τα tasks κατανεμήθηκαν ομοιόμορφα στους 4 executors: 18, 17, 19 και 20 tasks αντίστοιχα. Η επιβράδυνση στην εκτέλεση πιθανώς οφείλεται στην αυξημένη επικοινωνία μεταξύ των περισσότερων executors.

```
spark-submit \
  --master k8s://https://termi7.cslab.ece.ntua.gr:6443 \
  --deploy-mode cluster \
  --name query4_haversine_4x2 \
  --conf spark.kubernetes.namespace=username-priv \
  --conf spark.executor.instances=4 \
  --conf spark.executor.cores=2 \
  --conf spark.executor.memory=4g \
  --conf spark.driver.memory=4g \
  --conf spark.kubernetes.container.image=apache/spark \
  hdfs://hdfs-namenode:9000/user/username/project_code/DfQ4_Haversine.py
```

- **Διαμόρφωση 8 executors × 1 core/2GB memory**:

  Με τη μέγιστη οριζόντια κλιμάκωση, με 8 executors με 1 core και 2 GB μνήμη ανά executor η απόδοση υποβαθμίστηκε σημαντικά, με Total Uptime να ανέρχεται στα 1.2 min. Η περιορισμένη υπολογιστική ισχύς και μνήμη ανά executor επηρέασε αισθητά το χρόνο εκτέλεσης, με διάρκεια 22 s για το Query 0 και 8 s για το Query 1, τις υψηλότερες μεταξύ των configurations. Πιθανή αιτία αποτελεί η ανισομερής κατανομή των tasks και οι περιορισμένοι πόροι ανά executor, που δημιούργησαν bottlenecks.

```
spark-submit \
  --master k8s://https://termi7.cslab.ece.ntua.gr:6443 \
  --deploy-mode cluster \
  --name query4_haversine_8x1 \
  --conf spark.kubernetes.namespace=username-priv \
  --conf spark.executor.instances=8 \
  --conf spark.executor.cores=1 \
  --conf spark.executor.memory=2g \
  --conf spark.driver.memory=4g \
  --conf spark.kubernetes.container.image=apache/spark \
```

```
    hdfs://hdfs-namenode:9000/user/username/project_code/DfQ4_Haversine.py
```

Τελικά, φαίνεται ότι η οριζόντια κλιμάκωση έχει νόημα μέχρι ένα όριο. Η μεγάλη αύξηση του αριθμού των executors με ταυτόχρονη μείωση των πόρων ανά executor φαίνεται να οδηγεί σε υποβάθμιση της απόδοσης λόγω του overhead επικοινωνίας και διαχείρισης. Εν προκειμένω, η βέλτιστη απόδοση επετεύχθη με τη διαμόρφωση με 2 executors, καθένας με 4 πυρήνες και 8GB μνήμης.

Διαμόρφωση	Executors	Cores/Exec.	Memory/Exec.	Job Duration	Total Uptime	GC Time
2×4 cores / 8 GB	2	4	8 GB	13 s	42 s	2.2 min
4×2 cores / 4 GB	4	2	4 GB	15 s	53 s	2.4 min
8×1 core / 2 GB	8	1	2 GB	22 s	1.2 min	2.8 min

Πίνακας 8: Συγκριτικός Πίνακας Scaling Experiments (Query 4) – Οριζόντια Κλιμάκωση

Πειράματα Κάθετης Κλιμάκωσης (Vertical Scaling) – Διαφορετικοί Συνολικοί Πόροι

Σε αυτό το σενάριο, μεταβλήθηκαν οι συνολικοί διαθέσιμοι πόροι, διατηρώντας σταθερό τον αριθμό των executors (2 executors) αλλά αλλάζοντας την ισχύ (πυρήνες και μνήμη) καθενός.

- **Διαμόρφωση 2 executors × 4 cores/8GB memory (Σύνολο: 8 cores, 16GB RAM) :**

  Όπως αναφέρθηκε παραπάνω, αυτή η διαμόρφωση αποτέλεσε τη βέλτιστη, με Total Uptime 42s.

- **Διαμόρφωση 2 executors × 2 cores/4GB memory (Σύνολο: 4 cores, 8GB RAM) :**

  Μειώνοντας τους συνολικούς πόρους στο ήμισυ, το Total Uptime αυξήθηκε στα 48s, και ο χρόνος εκτέλεσης του κύριου query αυξήθηκε στα 16s. Η απόδοση παρέμεινε σταθερή, καθιστώντας την μια καλή επιλογή αν οι πόροι είναι περιορισμένοι.

  ```
  spark-submit \
    --master k8s://https://termi7.cslab.ece.ntua.gr:6443 \
    --deploy-mode cluster \
    --name query4_haversine_2x2 \
    --conf spark.kubernetes.namespace=username-priv \
    --conf spark.executor.instances=2 \
    --conf spark.executor.cores=2 \
    --conf spark.executor.memory=4g \
    --conf spark.driver.memory=4g \
    --conf spark.kubernetes.container.image=apache/spark \
    hdfs://hdfs-namenode:9000/user/username/project_code/DfQ4_Haversine.py
  ```

- **Διαμόρφωση 2 executors × 1 core/2GB memory (Σύνολο: 2 cores, 4GB RAM) :**

  Με περαιτέρω μείωση των συνολικών πόρων, η απόδοση υποβαθμίστηκε σημαντικά, με Total Uptime 55s και χρόνο εκτέλεσης 19s για το κύριο query. Η περιορισμένη παραλληλία (μόνο 2 cores συνολικά) και η οριακά επαρκής μνήμη ανά executor (2GB) οδήγησαν σε αυξημένους χρόνους και πίεση στη μνήμη (υψηλό Garbage Collection time).

  ```
  spark-submit \
    --master k8s://https://termi7.cslab.ece.ntua.gr:6443 \
    --deploy-mode cluster \
    --name query4_haversine_2x1 \
    --conf spark.kubernetes.namespace=username-priv \
    --conf spark.executor.instances=2 \
    --conf spark.executor.cores=1 \
  ```

```
    --conf spark.executor.memory=2g \
    --conf spark.driver.memory=4g \
    --conf spark.kubernetes.container.image=apache/spark \
    hdfs://hdfs-namenode:9000/user/username/project_code/DfQ4_Haversine.py
```

Η κάθετη κλιμάκωση έδειξε σαφή συσχέτιση μεταξύ των διαθέσιμων πόρων ανά executor και της απόδοσης. Η αύξηση των πυρήνων και της μνήμης ανά executor βελτίωσε τους χρόνους εκτέλεσης, ενώ η μείωσή τους κάτω από ένα όριο οδήγησε σε σημαντική επιβράδυνση.

Πίνακας 9: Συγκριτικός Πίνακας Scaling Experiments (Query 4) – Κάθετη Κλιμάκωση

Διαμόρφωση	Executors	Cores/Exec.	Memory/Exec.	Job Duration	Total Uptime	GC Time
2×1 core / 2 GB	2	1	2 GB	19 s	55 s	1.8 min
2×2 cores / 4 GB	2	2	4 GB	16 s	53 s	2.0 min
2×4 cores / 8 GB	2	4	8 GB	13 s	42 s	2.0 min

Συνολικά, για το συγκεκριμένο, υπολογιστικά απαιτητικό Query 4, η διαμόρφωση 2 executors × 4 cores / 8GB memory αναδείχθηκε ως η βέλτιστη, όπως φαίνεται και στον συγκεντρωτικό Πίνακα 10. Προσέφερε την καλύτερη ισορροπία μεταξύ παραλληλίας και επαρκών πόρων ανά executor για την αποδοτική διαχείριση των ενδιάμεσων δεδομένων και την ταχεία εκτέλεση.

Πίνακας 10: Συγκριτικός πίνακας scaling πειραμάτων (Query 4 - DataFrame API)

Διαμόρφωση	Executors	Cores/Exec.	Memory/Exec.	Job Duration	Total Uptime	GC Time
2×4 cores / 8 GB	2	4	8 GB	13 s	42 s	2.2 min
4×2 cores / 4 GB	4	2	4 GB	15 s	53 s	2.4 min
8×1 core / 2 GB	8	1	2 GB	22 s	1.2 min	2.8 min
2×1 core / 2 GB	2	1	2 GB	19 s	55 s	1.8 min
2×2 cores / 4 GB	2	2	4 GB	16 s	53 s	2.0 min
2×4 cores / 8 GB	2	4	8 GB	13 s	42 s	2.0 min

5.4.3 Ανάλυση Στρατηγικής Join για το Query 4

Το Query 4 περιλαμβάνει δύο βασικές λειτουργίες συνένωσης (join), κάθε μία με διαφορετικά χαρακτηριστικά και απαιτήσεις ως προς τη στρατηγική εκτέλεσης. Η επιλογή της κατάλληλης στρατηγικής από τον Catalyst Optimizer ήταν καθοριστική για την αποδοτική υλοποίηση του ερωτήματος και παρουσιάζεται στο Σχήμα 7.

- **Join 1: Εγκλήματα με MO Codes**

  Η πρώτη συνένωση αφορά την αντιστοίχιση των περιστατικών εγκλήματος με συγκεκριμένους MO Codes που σχετίζονται με όπλα. Συγκεκριμένα, πραγματοποιείται join μεταξύ του πίνακα crime_df_exploded_mocodes (εγκλήματα με αποσπασμένους MO codes) και του μικρού πίνακα weapon_mo, ο οποίος περιέχει μόνο τους κωδικούς που σχετίζονται με χρήση όπλων.

  Η στρατηγική που επέλεξε ο Catalyst Optimizer είναι Broadcast Hash Join (BHJ), η οποία είναι και η αναμενόμενη με βάση τη θεωρία. Ειδικότερα, ο πίνακας weapon_mo είναι εξαιρετικά μικρός, γεγονός που επιτρέπει τη μετάδοσή του (broadcast) σε όλους τους executors. Έτσι, κάθε executor κατασκευάζει τοπικά ένα hash table και εκτελεί το join χωρίς να απαιτείται shuffle του μεγάλου πίνακα εγκλημάτων. Πράγματι, οι φάσεις BroadcastExchange για το συγκεκριμένο join είχαν χαμηλό χρόνο εκτέλεσης και ελάχιστο μέγεθος δεδομένων (920 B), επιβεβαιώνοντας την αποδοτικότητα της στρατηγικής. Σε όλα τα configurations, η φάση αυτή εκτελέστηκε σχεδόν ακαριαία.

- **Join 2: Τοποθεσίες Εγκλημάτων με Αστυνομικά Τμήματα**

  Η δεύτερη συνένωση έχει ως στόχο τον υπολογισμό της πλησιέστερης αστυνομικής υπηρεσίας για κάθε έγκλημα. Πρόκειται για καρτεσιανό γινόμενο μεταξύ των τοποθεσιών εγκλημάτων και του πίνακα stations (με τα 21 αστυνομικά τμήματα), καθώς δεν υπάρχει κοινό join key και πρέπει να υπολογιστούν όλες οι δυνατές αποστάσεις.

  Η στρατηγική που επέλεξε ο Catalyst Optimizer είναι Broadcast Nested Loop Join (BNLJ). Ο πίνακας stations είναι πολύ μικρός, άρα η χρήση BNLJ με broadcast είναι η βέλτιστη επιλογή για αυτό το cross join. Το μικρό DataFrame μεταδίδεται σε όλους τους executors και κάθε task εκτελεί το join τοπικά.

```
== Physical Plan ==
AdaptiveSparkPlan (...)
+- == Final Plan ==
   Execute InsertIntoHadoopFsRelationCommand (...)
   +- WriteFiles (...)
      +- Coalesce (...)
         +- * Sort (...)
            +- AQEShuffleRead (...)
               +- ShuffleQueryStage (...)
                  +- Exchange (...)
                     +- * HashAggregate (...)
                        ...
                           +- Window (...)
                              ...
                                 +- * Sort (...)
                                    ...
                                       +- * Project (...)
                                          +- * BroadcastNestedLoopJoin Cross BuildRight (31)
                                             :- * HashAggregate (...)
                                             :  ...
                                             :     +- * Project (...)
                                             :        +- * BroadcastHashJoin Inner BuildRight (19)
                                             :           :- ... Scan parquet (crime data) ...
                                             :           +- ... Scan parquet (mo_codes) ...
                                             +- BroadcastQueryStage (...)
                                                ...
                                                   +- Scan parquet (stations_data)
```

Σχήμα 7: Απόσπασμα Physical Plan για το Query 4 (Joins Highlighted). Τα αποσιωπητικά αφορούν λεπτομέρειες που παραλείφθηκαν για οικονομία χώρου.

6 Συμπεράσματα

Η παρούσα εργασία επικεντρώθηκε στην ανάλυση μεγάλων συνόλων δεδομένων εγκληματικότητας και δημογραφικών στοιχείων της κομητείας του Los Angeles, αξιοποιώντας το οικοσύστημα Apache Spark και Hadoop σε ένα κατανεμημένο περιβάλλον Kubernetes. Μέσα από την υλοποίηση τεσσάρων σύνθετων ερωτημάτων, διερευνήθηκε η εκφραστικότητα και η απόδοση των διαφορετικών APIs του Spark (RDD, DataFrame, SQL), η σημασία της προεπεξεργασίας των δεδομένων, οι επιπτώσεις της κλιμάκωσης των πόρων και οι στρατηγικές βελτιστοποίησης του Catalyst Optimizer.

Ένα από τα βασικά συμπεράσματα αφορά τη σημασία της επιλογής API. Οι υλοποιήσεις με το DataFrame API, ιδίως χωρίς UDFs, αποδείχθηκαν σημαντικά πιο αποδοτικές από τις RDD-based λύσεις. Η απόδοση αυτή οφείλεται κυρίως στη χρήση του Catalyst Optimizer, που παρέχει βελτιστοποιημένα execution plans, και στην εκμετάλλευση του columnar storage μέσω του Parquet format, που επιτάχυνε τη φόρτωση και το filtering δεδομένων.

Η προεπεξεργασία των δεδομένων και η επιλογή του κατάλληλου format αποθήκευσης αναδείχθηκαν ως κρίσιμοι παράγοντες. Η μετατροπή των αρχικών αρχείων CSV και TXT σε Apache Parquet, ένα columnar storage format, προσέφερε σημαντικά πλεονεκτήματα. Όπως φάνηκε στην ανάλυση του Query 3 (μέσο εισόδημα ανά άτομο), η ανάγνωση από Parquet ήταν αποδοτικότερη, επιτρέποντας projection και predicate pushdown και αποφεύγοντας το overhead του inferSchema που απαιτείται για τα CSV.

Επιπλέον, μέσω του Query 4, μελετήθηκε εμπειρικά η συμπεριφορά του Spark σε διαφορετικές ρυθμίσεις υποδομής (scaling experiments). Στην οριζόντια κλιμάκωση, διατηρώντας σταθερούς τους συνολικούς πόρους (8 cores, 16GB RAM), η διαμόρφωση με 2 executors × 4 cores/8GB αναδείχθηκε ως η βέλτιστη, επιτυγχάνοντας τον ταχύτερο χρόνο εκτέλεσης (42s). Η αύξηση του αριθμού των executors με ταυτόχρονη μείωση των πόρων ανά executor (π.χ., 4x2c/4GB, 8x1c/2GB) οδήγησε σε σταδιακή υποβάθμιση της απόδοσης, κυρίως λόγω του αυξημένου overhead επικοινωνίας, της διαχείρισης των tasks και, στην περίπτωση των 8 executors, της ανισομερούς κατανομής φορτίου. Αυτό υπογραμμίζει ότι η αλόγιστη αύξηση του παραλληλισμού χωρίς επαρκείς πόρους ανά εκτελεστική μονάδα μπορεί να είναι αντιπαραγωγική. Στην κάθετη κλιμάκωση, μεταβάλλοντας τους συνολικούς πόρους αλλά διατηρώντας 2 executors, παρατηρήθηκε σαφής συσχέτιση μεταξύ των διαθέσιμων πόρων ανά executor και της απόδοσης. Η αύξηση των πυρήνων και της μνήμης ανά executor (έως 4c/8GB) βελτίωσε σημαντικά τους χρόνους εκτέλεσης, ενώ η μείωσή τους οδήγησε σε αναμενόμενη επιβράδυνση. Η διαμόρφωση 2 executors × 4 cores/8GB αναδείχθηκε και εδώ ως η πιο αποδοτική, παρέχοντας την καλύτερη ισορροπία μεταξύ παραλληλίας και ικανότητας διαχείρισης ενδιάμεσων δεδομένων.

Τέλος, η ανάλυση των στρατηγικών συνένωσης (join strategies) που επέλεξε ο Catalyst Optimizer για τα Query 3 και Query 4 επιβεβαίωσε την ευφυΐα του συστήματος. Για joins όπου ο ένας πίνακας ήταν σημαντικά μικρότερος (π.χ., dataset εισοδήματος στο Q3, πίνακες MO codes και αστυνομικών τμημάτων στο Q4), ο Optimizer επέλεξε σταθερά Broadcast Hash Join ή Broadcast Nested Loop Join (για το cross join του Q4). Αυτή η επιλογή απέτρεψε δαπανηρά shuffles, μεταδίδοντας τους μικρούς πίνακες σε όλους τους executors και επιτρέποντας την τοπική εκτέλεση του join, κάτι που ήταν καθοριστικό για την επίτευξη καλών επιδόσεων, ειδικά στις διαμορφώσεις με επαρκείς πόρους.

Συνοψίζοντας, η εργασία αυτή επέτρεψε την πρακτική εφαρμογή και κατανόηση των θεμελιωδών αρχών της επεξεργασίας δεδομένων μεγάλης κλίμακας με Apache Spark. Αναδείχθηκε η σημασία της επιλογής του κατάλληλου API, της μορφής αποθήκευσης των δεδομένων, της ορθολογικής παραμετροποίησης των πόρων του cluster, και της κατανόησης των μηχανισμών βελτιστοποίησης του Spark, για την επίτευξη αποδοτικών και επεκτάσιμων λύσεων σε πραγματικά προβλήματα ανάλυσης δεδομένων.