

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ

ΠΡΟΧΩΡΗΜΕΝΑ ΘΕΜΑΤΑ ΒΑΣΕΩΝ ΔΕΔΟΜΕΝΩΝ
ΕΞΑΜΗΝΟ 9ο - ΑΚΑΔΗΜΑΪΚΟ ΕΤΟΣ 2025 - 2026

ΥΠΕΥΘΥΝΟΣ ΚΑΘΗΓΗΤΗΣ: Δ. ΤΣΟΥΜΑΚΟΣ

Εξαμηνιαία Εργασία

Ομάδα 21

Γιασίν Αχμέντ	A.M.: 03121161	Email: el21161@mail.ntua.gr
Κάκος Σωτήριος	A.M.: 03121110	Email: el21110@mail.ntua.gr

Contents

1	Φόρτωση Δεδομένων	2
2	Runtime Measurement	3
3	Query 1	4
3.1	Υλοποίηση	4
3.1.1	DataFrame API	4
3.1.2	DataFrame API με UDF	5
3.1.3	RDD API	5
3.2	Αποτελέσματα	6
3.3	Σύγκριση Απόδοσης	6
4	Query 2	7
4.1	Υλοποίηση	7
4.1.1	DataFrame API	7
4.1.2	SQL API	8
4.2	Αποτελέσματα	9
4.3	Σύγκριση Απόδοσης	9
5	Query 3	10
5.1	Υλοποίηση	10
5.1.1	DataFrame API	10
5.1.2	RDD API	11
5.2	Αποτελέσματα	12
5.3	Σύγκριση Απόδοσης και Στρατηγικές Join	13
6	Query 4	13
6.1	Υλοποίηση	14
6.2	Αποτελέσματα	15
6.3	Ανάλυση Απόδοσης και Κλιμάκωση	16
6.3.1	Κλιμάκωση Υπολογιστικών Πόρων	16
7	Query 5	17
7.1	Υλοποίηση	17
7.2	Αποτελέσματα	19
7.3	Ανάλυση Πλάνου Εκτέλεσης και Απόδοσης	20
7.3.1	Κλιμάκωση και Χρόνοι Εκτέλεσης	20

Github Repo: [Link στο Repository](#)

1 Φόρτωση Δεδομένων

Στην υλοποίηση κάθε query υπάρχουν στην αρχή συναρτήσεις που είναι υπεύθυνες για το φόρτωμα των datasets με το κατάλληλο τρόπο (για csv, json, txt etc.). Στο αρχείο `config.py` έχουμε θέσει τα URI όλων των datasets που βρίσκονται σε ένα aws s3 cloud bucket στο s3://initial-notebook-data-bucket-dblab-905418150721/project_data.

Τα δεδομένα εγκληματικότητας, τα οποία είναι διαχωρισμένα χρονικά σε δύο αρχεία CSV (περίοδος 2010-2019 και 2020-σήμερα), εισάγονται στο Spark με ενεργοποιημένη την αυτόματη αναγνώριση σχήματος (`inferSchema`). Τα δύο αυτά σύνολα ενοποιούνται σε ένα ενιαίο `DataFrame` μέσω της εντολής `unionByName`. Ωστόσο, κατά την επισκόπηση των δεδομένων παρατηρήθηκε η ύπαρξη διπλότυπων εγγραφών, γεγονός που καθιστούσε αναγκαίο τον καθαρισμό τους. Για τον λόγο αυτό, εφαρμόζεται η μέθοδος `dropDuplicates` με βάση το μοναδικό αναγνωριστικό αναφοράς `DR_NO`.

```
1 def _load_data(spark, data_paths):
2     # Load Crime Data
3     df_2010_2019 = spark.read.csv(data_paths["crime_data_2010_2019"],
4     header=True, inferSchema=True)
5     df_2020_present = spark.read.csv(data_paths["crime_data_2020_present"],
6     header=True, inferSchema=True)
7     crime_df = df_2010_2019.unionByName(df_2020_present)
8
9     # Drop duplicate DR_NO values (keep first occurrence)
10    crime_df = crime_df.dropDuplicates(["DR_NO"])
11
12    # Load Police Stations
13    stations_df = spark.read.csv(data_paths["police_stations"], header=
14    True, inferSchema=True)
15
16    return crime_df, stations_df
```

Listing 1: Φόρτωση και Καθαρισμός Δεδομένων Εγκληματικότητας

Παράλληλα, η φόρτωση κάποιων βοηθητικών αρχείων απαιτεί εξειδικευμένο χειρισμό λόγω της μη δομημένης φύσης τους. Συγκεκριμένα, οι κωδικοί Modus Operandi (MO Codes) εισάγονται ως απλό αρχείο κειμένου και στη συνέχεια δομούνται μέσω κανονικών εκφράσεων (Regular Expressions), διαχωρίζοντας τον τετραψήφιο κωδικό από την περιγραφή του.

```
1 def _load_mo_codes(spark, mo_codes_path):
2     mo_codes_df = spark.read.text(mo_codes_path).select(
3         F.regexp_extract(F.col("value"), r"^\d{4}", 1).alias("Code"),
4         F.regexp_extract(F.col("value"), r"^\d{4}\s+(.+)$", 1).alias("
5         Description")
6     ).filter(F.col("Code") != "")
7
8     return mo_codes_df
```

Listing 2: Φόρτωση και Επεξεργασία MO Codes

Τέλος, για τις ανάγκες της κοινωνικοοικονομικής ανάλυσης, τα δεδομένα εισοδήματος φορτώνονται ορίζοντας ρητά ως διαχωριστικό το ερωτηματικό (;), ενώ τα γεωχωρικά δεδομένα των απογραφικών τετραγώνων (Census Blocks) εισάγονται αρχικά ως ακατέργαστο κείμενο (raw JSON strings) και γίνεται η περαιτέρω επεξεργασία τους σε dataframe στην ίδια την υλοποίηση του query 5.

```
1 def _load_data(spark, data_paths):
2     # Load Income Data
3     income_df = spark.read.option("header", "true") \
4         .option("delimiter", ";") \
5         .csv(data_paths["median_income"])
6
7     # Load Census Blocks - Read raw JSON only
8     blocks_raw = spark.read \
9         .option("multiLine", "false") \
10        .text(data_paths["census_blocks"])
11
12    # Load Crime Data (2020-Present)
13    crime_df = spark.read.option("header", "true").csv(data_paths["
crime_data_2020_present"])
14
15    return income_df, blocks_raw, crime_df
```

Listing 3: Φόρτωση Δεδομένων Εισοδήματος και Census Blocks

2 Runtime Measurement

Ένας από τους βασικούς στόχους της εργασίας είναι η συγκριτική ανάλυση των διαφορετικών στρατηγικών εκτέλεσης των ερωτημάτων. Για τον σκοπό αυτό υλοποιήθηκε η συνάρτηση `run_and_time`, η οποία χρησιμοποιείται για την αξιόπιστη μέτρηση του χρόνου εκτέλεσης κάθε ερωτήματος.

Στο Apache Spark εφαρμόζεται η αρχή του **Lazy Evaluation**, σύμφωνα με την οποία οι μετασχηματισμοί (*transformations*) δεν εκτελούνται άμεσα, αλλά μόνο όταν πυροδοτηθεί κάποιο **Action**. Επομένως, η απλή δημιουργία ενός DataFrame δεν αρκεί για τη μέτρηση του πραγματικού χρόνου εκτέλεσης, καθώς το υπολογιστικό πλάνο δεν έχει ακόμη εκτελεστεί.

Για την αντιμετώπιση αυτού του ζητήματος, η συνάρτηση πυροδοτεί την εκτέλεση του πλήρους πλάνου μέσω της εντολής:

```
result.write.format("noop").mode("overwrite").save()
```

Η χρήση της `write` αντί της `show` είναι σκόπιμη. Η `write` αποτελεί Action που αναγκάζει το Spark να εκτελέσει ολόκληρο το query plan, χωρίς όμως να επιβαρύνει τη μέτρηση με κόστος εγγραφής στο δίσκο, καθώς χρησιμοποιείται το format `"noop"`. Με αυτόν τον τρόπο αποφεύγεται ο θόρυβος που θα εισήγαγε η φυσική εγγραφή δεδομένων και η μέτρηση επικεντρώνεται αποκλειστικά στο υπολογιστικό κόστος των αλγορίθμων και των στρατηγικών Join.

Πριν την εγγραφή, το αποτέλεσμα αποθηκεύεται στη μνήμη με την εντολή `cache()`, ώστε να αποφευχθεί επανεκτέλεση του ερωτήματος σε μεταγενέστερα στάδια. Η εντολή `result.show()` καλείται εκτός του χρονομέτρου, αποκλειστικά για την παρουσίαση των αποτελεσμάτων, και ανακτά τα δεδομένα από τη μνήμη, χωρίς να επηρεάζει τον μετρούμενο χρόνο εκτέλεσης.

Τέλος, αξίζει να σημειωθεί ότι τα πειράματα εκτελέστηκαν σε κοινόχρηστο AWS cluster. Ως εκ τούτου, ενδέχεται να παρατηρούνται μικρές αποκλίσεις στους χρόνους εκτέλεσης λόγω μεταβαλλόμενου φόρτου από άλλους χρήστες του συστήματος.

```
1 import time
2
3 def run_and_time(fn, explain=False, join_strategy_name=None):
4     start = time.time()
5     result = fn()
6     # cache to avoid re-executing when calling show after write
7     result.cache()
8     result.write.format("noop").mode("overwrite").save()
9     end = time.time()
10
11     # used after timer to avoid including formatting in comparison
12     result.show(truncate=False)
13
14     if explain:
15         print("\n" + "=" * 80)
16         print(f"Query Execution Plan (Join Strategy Hinted: {
17 join_strategy_name or 'Optimizer Choice'})")
18         print("=" * 80)
19         result.explain(extended=True)
20         print("=" * 80 + "\n")
21
22     return end - start
```

Listing 4: Συνάρτηση μέτρησης χρόνου εκτέλεσης (run_and_time)

3 Query 1

Με το συγκεκριμένο ερώτημα επιχειρείται η ανάλυση των ηλικιακών ομάδων των θυμάτων σε περιστατικά εγκλημάτων που χαρακτηρίζονται ως **βαριά σωματική βλάβη (aggravated assault)**. Εστιάζουμε σε τέσσερις διακριτές ηλικιακές κατηγορίες:

- **Παιδιά (Children):** < 18 ετών
- **Νεαροί ενήλικες (Young adults):** 18 – 24 ετών
- **Ενήλικες (Adults):** 25 – 64 ετών
- **Ηλικιωμένοι (Seniors):** > 64 ετών

Στόχος του ερωτήματος είναι η ταξινόμηση των ηλικιακών αυτών ομάδων σε **φθίνουσα σειρά**, με βάση τη συχνότητα εμφάνισής τους στα συγκεκριμένα περιστατικά.

3.1 Υλοποίηση

3.1.1 DataFrame API

Στην υλοποίηση με το **DataFrame API**, αρχικά φιλτράρονται οι εγγραφές ώστε να διατηρηθούν μόνο εκείνες που αφορούν περιστατικά *aggravated assault* και για τις οποίες η ηλικία του θύματος είναι έγκυρη και μεγαλύτερη του μηδενός. Στη συνέχεια, δημιουργείται μία νέα στήλη που αντιστοιχίζει κάθε θύμα στην κατάλληλη ηλικιακή ομάδα βάσει των

προκαθορισμένων συνθηκών. Τέλος, τα δεδομένα ομαδοποιούνται ανά ηλικιακή κατηγορία, υπολογίζεται το πλήθος των περιστατικών για κάθε ομάδα και τα αποτελέσματα ταξινομούνται σε φθίνουσα σειρά.

```
1 def query1_df(crime_df):
2     filtered_df = crime_df.filter(
3         F.lower(F.col("Crm Cd Desc")).contains("aggravated assault") & (
4             F.col("Vict Age") > 0)
5     )
6     age_groups_df = filtered_df.withColumn(
7         "Age Group",
8         F.when(F.col("Vict Age") < 18, "Children")
9         .when(F.col("Vict Age").between(18, 24), "Young adults")
10        .when(F.col("Vict Age").between(25, 64), "Adults")
11        .when(F.col("Vict Age") > 64, "Seniors")
12        .otherwise("Unknown")
13    )
14    result_df = age_groups_df.groupBy("Age Group").count().orderBy(F.col("count").desc())
15    return result_df
```

Listing 5: Υλοποίηση Query 1 με DataFrame API

3.1.2 DataFrame API με UDF

Εναλλακτικά, η κατηγοριοποίηση των ηλικιών πραγματοποιήθηκε μέσω **User Defined Function (UDF)**. Σε αυτή την προσέγγιση, η λογική αντιστοίχισης της ηλικίας σε ηλικιακή ομάδα υλοποιείται σε ξεχωριστή συνάρτηση Python, η οποία εφαρμόζεται στο σύνολο δεδομένων πριν την ομαδοποίηση και την ταξινόμηση των αποτελεσμάτων.

```
1 def _categorize_age(age):
2     if age is None:
3         return None
4     if age < 18:
5         return "Children"
6     elif 18 <= age <= 24:
7         return "Young adults"
8     elif 25 <= age <= 64:
9         return "Adults"
10    elif age > 64:
11        return "Seniors"
12    return None
```

Listing 6: Συνάρτηση κατηγοριοποίησης ηλικίας (UDF)

3.1.3 RDD API

Όσον αφορά την υλοποίηση με το **RDD API**, η επεξεργασία βασίζεται σε συναρτήσεις τύπου *lambda* για το φιλτράρισμα των σχετικών εγκλημάτων και την αντιστοίχιση κάθε ηλικίας στην αντίστοιχη κατηγορία. Ακολούθως, με τη χρήση συνάθροισης (*reduceByKey*) υπολογίζεται το πλήθος εμφανίσεων κάθε ηλικιακής ομάδας, ενώ η τελική ταξινόμηση πραγματοποιείται σε φθίνουσα σειρά ως προς τη συχνότητα.

```
1 def query1_rdd(crime_df):
2     crime_rdd = crime_df.select("Crm Cd Desc", "Vict Age").rdd
```

```

3     filtered_rdd = crime_rdd.filter(
4         lambda row: row["CrimeType Desc"]
5         and "aggravated assault" in row["CrimeType Desc"].lower()
6         and row["Vict Age"] is not None
7         and isinstance(row["Vict Age"], int)
8         and row["Vict Age"] > 0
9     )
10    mapped = filtered_rdd.map(lambda row: (_categorize_age_rdd(row["Vict
11    Age"])), 1))
12    valid = mapped.filter(lambda x: x[0] is not None)
13    reduced = valid.reduceByKey(lambda a, b: a + b)
14    sorted_rdd = reduced.sortBy(lambda x: x[1], ascending=False)
15    result_df = sorted_rdd.toDF(["Age Group", "count"])
    return result_df

```

Listing 7: Υλοποίηση Query 1 με RDD API

3.2 Αποτελέσματα

Τα αποτελέσματα του συγκεκριμένου ερωτήματος, όπως αυτά προέκυψαν από την εκτέλεση στο Spark, παρουσιάζονται στον παρακάτω πίνακα:

Table 1: Πλήθος περιστατικών Aggravated Assault ανά ηλικιακή ομάδα

Age Group	Count
Adults	121,660
Young adults	33,758
Children	10,904
Seniors	6,011

3.3 Σύγκριση Απόδοσης

Στην υλοποίηση με το **DataFrame API** χρησιμοποιούνται αποκλειστικά εγγενείς συναρτήσεις του Spark, γεγονός που οδηγεί σε αισθητά καλύτερη απόδοση. Η συγκεκριμένη προσέγγιση εμφανίζει τον μικρότερο χρόνο εκτέλεσης (**10.85 δευτερόλεπτα**), καθώς επιτρέπει στον **Catalyst Optimizer** να αναλύσει πλήρως το λογικό πλάνο και να παραγάγει το βέλτιστο δυνατό πλάνο εκτέλεσης.

Αντίθετα, στην υλοποίηση με **DataFrame και UDF**, η συνάρτηση κατηγοριοποίησης ηλικίας αποτελεί «μαύρο κουτί» για τον optimizer. Ως αποτέλεσμα, ο Catalyst δεν μπορεί να εφαρμόσει τις ίδιες βελτιστοποιήσεις, γεγονός που οδηγεί σε αυξημένο χρόνο εκτέλεσης (16.99 δευτερόλεπτα).

Η υλοποίηση με το **RDD API** παρουσιάζει τη χαμηλότερη επίδοση, με χρόνο εκτέλεσης 28.77 δευτερόλεπτα. Το αποτέλεσμα αυτό είναι αναμενόμενο στο περιβάλλον του PySpark, καθώς τα RDDs δεν επωφελούνται από τις βελτιστοποιήσεις του Catalyst Optimizer ούτε από τη γνώση του σχήματος των δεδομένων. Επιπλέον, η φόρτωση και επεξεργασία δεδομένων από αρχεία CSV σε επίπεδο RDD είναι λιγότερο αποδοτική.

Συνολικά, το DataFrame API υπερέχει σε απόδοση, καθώς αξιοποιεί τόσο το σχήμα των δεδομένων όσο και τις εσωτερικές βελτιστοποιήσεις του Spark.

Table 2: Χρόνοι Εκτέλεσης Query 1 ανά API

API	Execution Time (sec)
DataFrame	10.85
DataFrame + UDF	16.99
RDD	28.77

4 Query 2

Στο συγκεκριμένο ερώτημα αναλύεται η δημογραφική κατανομή των θυμάτων εγκληματικότητας στο Los Angeles με βάση το φυλετικό τους προφίλ. Συγκεκριμένα, ζητείται ο εντοπισμός, για κάθε έτος, των τριών φυλετικών ομάδων με τον μεγαλύτερο αριθμό θυμάτων, ταξινομημένων σε φθίνουσα σειρά. Παράλληλα, υπολογίζεται και παρουσιάζεται το ποσοστό συμμετοχής κάθε φυλετικής ομάδας επί του συνολικού αριθμού θυμάτων του αντίστοιχου έτους.

4.1 Υλοποίηση

4.1.1 DataFrame API

Στην υλοποίηση με το **DataFrame API**, αρχικά εξάγεται το έτος από το πεδίο της ημερομηνίας καταγραφής του εγκλήματος. Ακολουθεί η ένωση των δεδομένων εγκληματικότητας με τον πίνακα αντιστοίχισης φυλετικών κωδικών, ώστε να χρησιμοποιηθεί η πλήρης περιγραφή κάθε φυλετικής ομάδας. Στη συνέχεια, τα δεδομένα ομαδοποιούνται ανά έτος και φυλετική κατηγορία και υπολογίζεται το πλήθος των θυμάτων για κάθε συνδυασμό.

```

1 def query2_df(crime_df, re_codes_df):
2     # 1. Extract Year from DATE OCC
3     # Format based on description: "yyyy MMM dd hh:mm:ss a"
4     # We transform string to timestamp, then extract year
5     crime_with_year = crime_df.withColumn(
6         "year",
7         F.year(F.to_timestamp(F.col("DATE OCC"), "yyyy MMM dd hh:mm:ss a")
8     ))
9
10    # 2. Join with Race Codes to get Full Description
11    joined_df = crime_with_year.join(
12        re_codes_df,
13        crime_with_year["Vict Descent"] == re_codes_df["Vict Descent"],
14        "inner"
15    ).select(
16        F.col("year"),
17        F.col("Vict Descent Full").alias("Victim_Descent")
18    )
19
20    # 3. Group by Year and Descent to get counts
21    grouped_counts = joined_df.groupBy("year", "Victim_Descent").count()
```

Listing 8: Προετοιμασία και Ομαδοποίηση (DataFrame API)

Για τον υπολογισμό των ποσοστών και της κατάταξης, εφαρμόζονται αναλυτικές συναρτήσεις παραθύρου με διαμέριση ανά έτος. Μέσω αυτών υπολογίζεται αφενός ο συνολικός

αριθμός θυμάτων ανά έτος, ο οποίος χρησιμοποιείται ως παρονομαστής για τον υπολογισμό του ποσοστού, και αφετέρου η κατάταξη των φυλετικών ομάδων βάσει του πλήθους των θυμάτων. Τέλος, διατηρούνται μόνο οι τρεις πρώτες ομάδες κάθε έτους και τα αποτελέσματα παρουσιάζονται σε φθίνουσα σειρά.

```

1  # 4. Window functions for Total per Year and Ranking
2  window_year = Window.partitionBy("year")
3  window_rank = Window.partitionBy("year").orderBy(F.col("count").desc
4  ())
5
6  final_df = grouped_counts.withColumn(
7      "total_victims_year",
8      F.sum("count").over(window_year)
9  ).withColumn(
10     "percentage",
11     F.format_number((F.col("count") / F.col("total_victims_year")) *
12     100, 1)
13 ).withColumn(
14     "rank",
15     F.rank().over(window_rank)
16 )
17
18 # 5. Filter top 3 and format output
19 result = final_df.filter(F.col("rank") <= 3)\
20     .select(
21         F.col("year"),
22         F.col("Victim_Descent").alias("Victim Descent"),
23         F.col("count").alias("#"),
24         F.col("percentage").alias("%")
25     )\
26     .orderBy(F.col("year").desc(), F.col("#").desc())
27
28 return result

```

Listing 9: Υπολογισμός Ποσοστών και Κατάταξη (DataFrame API)

4.1.2 SQL API

Η προσέγγιση με το **SQL API** ακολουθεί την ίδια ακριβώς λογική επεξεργασίας. Τα δεδομένα εγγράφονται ως προσωρινά views και η ανάλυση υλοποιείται μέσω διαδοχικών λογικών βημάτων: εξαγωγή έτους, ένωση πινάκων, ομαδοποίηση για την καταμέτρηση των θυμάτων και χρήση αναλυτικών συναρτήσεων παραθύρου για τον υπολογισμό του ετήσιου συνόλου και της κατάταξης. Το τελικό αποτέλεσμα προκύπτει με φιλτράρισμα βάσει της κατάταξης και υπολογισμό του ποσοστού συμμετοχής κάθε ομάδας στο σύνολο του έτους.

```

1  def query2_sql(spark, crime_df, re_codes_df):
2      # Register Temporary Views
3      crime_df.createOrReplaceTempView("crime_data")
4      re_codes_df.createOrReplaceTempView("re_codes")
5
6      sql_query = """
7      WITH CrimeWithYear AS (
8          SELECT
9              year(to_timestamp('DATE OCC', 'yyyy MMM dd hh:mm:ss a')) as
10             year,
11             'Vict Descent' as v_code

```

```

11         FROM crime_data
12     ),
13     JoinedData AS (
14         SELECT
15             c.year,
16             r.'Vict Descent Full' as victim_desc
17         FROM CrimeWithYear c
18         JOIN re_codes r ON c.v_code = r.'Vict Descent'
19     ),
20     GroupedCounts AS (
21         SELECT
22             year,
23             victim_desc,
24             COUNT(*) as victims_count
25         FROM JoinedData
26         GROUP BY year, victim_desc
27     ),
28     CalculatedMetrics AS (
29         SELECT
30             year,
31             victim_desc,
32             victims_count,
33             SUM(victims_count) OVER (PARTITION BY year) as total_year,
34             RANK() OVER (PARTITION BY year ORDER BY victims_count DESC)
35         as rnk
36         FROM GroupedCounts
37     )
38     SELECT
39         year,
40         victim_desc as 'Victim Descent',
41         victims_count as '#',
42         format_number((victims_count / total_year) * 100, 1) as '%'
43     FROM CalculatedMetrics
44     WHERE rnk <= 3
45     ORDER BY year DESC, victims_count DESC
46     """
47     return spark.sql(sql_query)

```

Listing 10: Υλοποίηση Query 2 με SQL API

4.2 Αποτελέσματα

Τα αποτελέσματα του συγκεκριμένου ερωτήματος, όπως αυτά προέκυψαν από την εκτέλεση στο Spark, παρουσιάζονται στον παρακάτω πίνακα:

4.3 Σύγκριση Απόδοσης

Συγκρίνοντας τους χρόνους εκτέλεσης των δύο υλοποιήσεων, παρατηρείται ότι το DataFrame API και το SQL API παρουσιάζουν ουσιαστικά ταυτόσημη απόδοση. Το αποτέλεσμα αυτό είναι αναμενόμενο, καθώς και οι δύο προσεγγίσεις μεταφράζονται από τον Spark στο ίδιο φυσικό πλάνο εκτέλεσης. Ο Catalyst Optimizer παράγει ισοδύναμες βελτιστοποιημένες εντολές χαμηλού επιπέδου, με αποτέλεσμα ο υπολογιστικός φόρτος και η συνολική απόδοση να είναι πανομοιότυπα.

Table 3: Top-3 Φυλετικές Ομάδες Θυμάτων ανά Έτος (Δείγμα)

Year	Victim Descent	#	%
2025	Hispanic/Latin/Mexican	34	40.5
2025	Unknown	24	28.6
2025	White	13	15.5
2024	Hispanic/Latin/Mexican	28,576	29.1
2024	White	22,958	23.3
2024	Unknown	19,984	20.3
2023	Hispanic/Latin/Mexican	69,401	34.6
2023	White	44,615	22.2
2023	Black	30,504	15.2
2022	Hispanic/Latin/Mexican	73,111	35.6
2022	White	46,695	22.8
2022	Black	34,634	16.9
...

Table 4: Χρόνοι Εκτέλεσης Query 2 ανά API

API	Execution Time (sec)
DataFrame	20.9
SQL	21.3

5 Query 3

Στο τρίτο ερώτημα ζητείται η κατάταξη των μεθόδων διάπραξης εγκλημάτων (MO Codes) με βάση τη συχνότητα εμφάνισής τους, σε φθίνουσα σειρά. Για κάθε μέθοδο παρουσιάζεται τόσο ο κωδικός όσο και η περιγραφή της, η οποία προκύπτει μέσω αντιστοίχισης με το βοηθητικό σύνολο δεδομένων των MO Codes.

5.1 Υλοποίηση

5.1.1 DataFrame API

Στην υλοποίηση με το **DataFrame API**, αρχικά γίνεται κανονικοποίηση των δεδομένων, καθώς κάθε εγγραφή μπορεί να περιέχει πολλαπλούς κωδικούς μεθόδων διάπραξης. Οι κωδικοί αυτοί απομονώνονται και μετατρέπονται σε ξεχωριστές εγγραφές, ώστε να είναι δυνατή η ορθή καταμέτρησή τους. Στη συνέχεια, πραγματοποιείται ομαδοποίηση και υπολογισμός της συχνότητας εμφάνισης κάθε κωδικού.

Το κρίσιμο στάδιο της επεξεργασίας είναι η ένωση των αποτελεσμάτων με το σύνολο δεδομένων που περιέχει τις περιγραφές των κωδικών. Σε αυτό το σημείο, χρησιμοποιούνται τόσο η προεπιλεγμένη στρατηγική του Catalyst Optimizer όσο και εναλλακτικές στρατηγικές ένωσης μέσω υποδείξεων (hints), προκειμένου να μελετηθεί η επίδρασή τους στην απόδοση. Τέλος, τα αποτελέσματα ταξινομούνται με βάση τη συχνότητα εμφάνισης.

```

1 def query3_df(crime_df, mo_codes_df, join_strategy=None):
2     # Explode Mocodes column (contains space-separated codes)
3     crime_with_codes = crime_df.select(

```

```

4      F.explode(F.split(F.col("Mocodes"), " ")).alias("Code")
5    ).filter(F.col("Code") != "")
6
7    # Count occurrences of each code
8    code_counts = crime_with_codes.groupBy("Code").count()
9
10   # Join with MO codes descriptions
11   if join_strategy:
12       # Use hint to suggest join strategy
13       result_df = code_counts.join(
14           mo_codes_df.hint(join_strategy),
15           "Code",
16           "left"
17       )
18   else:
19       # Let Spark optimizer decide
20       result_df = code_counts.join(mo_codes_df, "Code", "left")
21
22   # Select and order results
23   result_df = result_df.select(
24       "Code",
25       F.coalesce(F.col("Description"), F.lit("Unknown")).alias("
Description"),
26       F.col("count").alias("Frequency")
27   ).orderBy(F.col("Frequency").desc())
28
29   return result_df

```

Listing 11: Υλοποίηση Query 3 με DataFrame API

5.1.2 RDD API

Στην υλοποίηση με το **RDD API**, η επεξεργασία πραγματοποιείται σε χαμηλότερο επίπεδο αφαίρεσης. Αρχικά, οι κωδικοί μεθόδων διάπραξης εξάγονται και «επιπεδοποιούνται» από τις εγγραφές μέσω κατάλληλου μετασχηματισμού, λειτουργώντας αντίστοιχα με τη λογική του *explode* στο DataFrame API. Ακολουθεί η κλασική προσέγγιση Map-Reduce, όπου κάθε κωδικός αντιστοιχίζεται σε μία εμφάνιση και στη συνέχεια αθροίζεται το πλήθος εμφανίσεων ανά κωδικό.

Για την απόκτηση της περιγραφής κάθε μεθόδου, το σύνολο δεδομένων των MO Codes μετατρέπεται επίσης σε ζεύγη κλειδιού-τιμής και εφαρμόζεται αριστερή ένωση. Η τελική κατάταξη πραγματοποιείται με βάση τη συχνότητα εμφάνισης.

```

1 def query3_rdd(crime_df, mo_codes_df):
2     # Extract Mocodes column as RDD
3     mocodes_rdd = crime_df.select("Mocodes").rdd
4
5     # Explode space-separated codes and filter empty strings
6     codes_rdd = mocodes_rdd.flatMap(
7         lambda row: row["Mocodes"].split(" ") if row["Mocodes"] else []
8     ).filter(lambda code: code.strip() != "")
9
10    # Count occurrences: map to (code, 1) and reduce
11    code_counts_rdd = codes_rdd.map(lambda code: (code.strip(), 1)).
    reduceByKey(lambda a, b: a + b)
12

```

```

13 # Convert MO codes to RDD for join: (code, description)
14 mo_codes_rdd = mo_codes_df.rdd.map(lambda row: (row["Code"], row["
Description"]))
15
16 # Left outer join to get descriptions
17 joined_rdd = code_counts_rdd.leftOuterJoin(mo_codes_rdd)
18
19 # Format results: (code, description, count)
20 result_rdd = joined_rdd.map(
21     lambda x: (x[0], x[1][1] if x[1][1] else "Unknown", x[1][0])
22 )
23
24 # Sort by frequency (descending)
25 sorted_rdd = result_rdd.sortBy(lambda x: x[2], ascending=False)
26
27 # Convert to DataFrame
28 result_df = sorted_rdd.toDF(["Code", "Description", "Frequency"])
29
30 return result_df

```

Listing 12: Υλοποίηση Query 3 με RDD API

5.2 Αποτελέσματα

Τα αποτελέσματα του συγκεκριμένου ερωτήματος είναι όσες και οι εγγραφές στο αρχείο κωδικών εγκλημάτων (Mocodes), δηλαδή 615. Παρακάτω παρουσιάζονται τα 20 συχνότερα:

Table 5: Top-20 Μέθοδοι Διάπραξης Εγκλημάτων (MO Codes)

Code	Description	Frequency
0344	Removes vict property	985,859
1822	Stranger	542,705
0416	Hit-Hit w/ weapon	397,621
0329	Vandalized	371,445
0913	Victim knew Suspect	275,387
2000	Domestic violence	250,771
1300	Vehicle involved	216,485
0400	Force used	210,091
1402	Evidence Booked (any crime)	174,636
1609	Smashed	129,202
1309	Susp uses vehicle	121,141
0325	Took merchandise	117,672
1202	Victim was aged (60 & over) or blind/disabled	117,435
1814	Susp is/was current/former boyfriend/girlfriend	115,714
0444	Pushed	114,483
1501	Other MO (see rpt)	113,416
1307	Breaks window	111,576
0334	Brandishes weapon	104,090
2004	Suspect is homeless/transient	92,721
0432	Intimidation	81,686

5.3 Σύγκριση Απόδοσης και Στρατηγικές Join

Όπως και στα προηγούμενα ερωτήματα, η υλοποίηση με το **DataFrame API** αποδεικνύεται σημαντικά πιο αποδοτική, με χρόνο εκτέλεσης **13.08 δευτερόλεπτα**, έναντι 30.9 δευτερολέπτων της υλοποίησης με το RDD API. Η διαφορά αυτή οφείλεται στην αξιοποίηση του Catalyst Optimizer και στη γνώση του σχήματος των δεδομένων, πλεονεκτήματα τα οποία δεν είναι διαθέσιμα στο RDD API.

Όλες οι εκτελέσεις πραγματοποιήθηκαν σε περιβάλλον με 4 executors, καθένas με 1 core και 2GB μνήμης.

Κατά την εκτέλεση του ερωτήματος με το DataFrame API χωρίς επιβολή συγκεκριμένης στρατηγικής ένωσης, η ανάλυση του πλάνου εκτέλεσης μέσω της `explain()` δείχνει ότι ο Catalyst Optimizer επιλέγει αυτόματα τη στρατηγική **Broadcast Hash Join**. Η επιλογή αυτή είναι απολύτως αναμενόμενη, καθώς το σύνολο δεδομένων των περιγραφών των MO Codes είναι εξαιρετικά μικρό (615 εγγραφές). Σε αυτή την περίπτωση, είναι αποδοτικότερο να διανεμηθεί ο μικρός πίνακας σε όλους τους executors, αποφεύγοντας την ακριβή ανακατανομή (shuffle) των δεδομένων.

Όταν προτείνεται η χρήση **Sort Merge Join**, το Spark αναγκάζεται να ανακατανείμει και να ταξινομήσει και τα δύο σύνολα δεδομένων με βάση το κλειδί ένωσης, διαδικασία που εισάγει σημαντικό επιπλέον κόστος. Η στρατηγική αυτή θα ήταν καταλληλότερη μόνο στην περίπτωση όπου και οι δύο σχέσεις είχαν μεγάλο όγκο δεδομένων, γεγονός που δεν ισχύει εδώ, με αποτέλεσμα αυξημένο χρόνο εκτέλεσης (16.38 δευτερόλεπτα).

Η χρήση **Shuffle Hash Join** οδηγεί επίσης σε ανακατανομή των δεδομένων, αλλά χωρίς την ανάγκη ταξινόμησης. Αν και εμφανίζεται ελαφρώς πιο αποδοτική από το Sort Merge Join (14.38 δευτερόλεπτα), εξακολουθεί να υπολείπεται της στρατηγικής broadcast λόγω του κόστους του shuffle.

Τέλος, η στρατηγική **Shuffle Replicated Nested Loop Join** αποτελεί τη λιγότερο αποδοτική επιλογή, καθώς συγκρίνει κάθε εγγραφή του μικρού πίνακα με κάθε εγγραφή του μεγάλου πίνακα σε κάθε partition. Για τον λόγο αυτό, ο Catalyst Optimizer αποφεύγει την εφαρμογή της και επιλέγει εκ νέου το Broadcast Hash Join.

Table 6: Χρόνοι Εκτέλεσης Query 3 ανά Στρατηγική Join (DataFrame API)

Join Strategy	Execution Time (sec)
Broadcast Hash Join	13.08
Merge Sort Join	16.38
Shuffle Hash Join	14.38
SHUFFLE REPLICATE NL	-

6 Query 4

Στο παρόν ερώτημα ζητείται, για κάθε αστυνομικό τμήμα, να υπολογιστεί ο αριθμός των εγκλημάτων που σημειώθηκαν πλησιέστερα σε αυτό σε σχέση με οποιοδήποτε άλλο τμήμα, καθώς και η μέση απόσταση των αντίστοιχων περιστατικών. Τα αποτελέσματα παρουσιάζονται ταξινομημένα κατά φθίνουσα σειρά πλήθους περιστατικών.

Αρχικά, τα δεδομένα εγκλημάτων καθαρίζονται από λανθασμένες ή ελλιπείς γεωγραφικές συντεταγμένες, με απομάκρυνση εγγραφών που αντιστοιχούν στο λεγόμενο *Null Island* (0,0). Στη συνέχεια, τόσο οι τοποθεσίες των εγκλημάτων όσο και των αστυνομικών τμημάτων μετατρέπονται από απλά αριθμητικά πεδία σε γεωμετρικά αντικείμενα τύπου σημείου (*ST_Point*), ώστε να είναι δυνατός ο υπολογισμός γεωδαιτικών αποστάσεων.

6.1 Υλοποίηση

Για τον εντοπισμό του πλησιέστερου αστυνομικού τμήματος ανά έγκλημα, εκτελείται καρτεσιανό γινόμενο (*cross join*) μεταξύ του συνόλου των εγκλημάτων και του συνόλου των τμημάτων. Για κάθε παραγόμενο ζεύγος υπολογίζεται η απόσταση πάνω στη σφαιρική επιφάνεια της Γης, σε μέτρα, μέσω της συνάρτησης *ST_DistanceSphere*.

Στη συνέχεια, εφαρμόζεται *Window Function* με ομαδοποίηση ανά μοναδικό αναγνωριστικό εγκλήματος και ταξινόμηση βάσει απόστασης, ώστε να επιλεγεί αποκλειστικά το πλησιέστερο τμήμα για κάθε περιστατικό. Τέλος, τα δεδομένα ομαδοποιούνται ανά αστυνομικό τμήμα για τον υπολογισμό του πλήθους των περιστατικών και της μέσης απόστασης (μετατρεπόμενης σε χιλιόμετρα), και ταξινομούνται σε φθίνουσα σειρά.

```
1 def query4_df(crime_df, stations_df):
2     """
3     Calculates the number of crimes closest to each police station and
4     the average distance.
5     """
6     # 1. Prepare Crime Data (Geometry & Filter)
7     # Filter Null Island (0,0) and ensure valid coordinates
8     crime_geo = crime_df.filter(
9         (F.col("LAT") != 0) & (F.col("LON") != 0) &
10        F.col("LAT").isNotNull() & F.col("LON").isNotNull()
11    ).select(
12        "DR_NO", # Unique Crime ID
13        F.col("LAT"),
14        F.col("LON")
15    )
16
17    # Convert to Sedona Geometry (Point)
18    # Note: ST_Point takes (Longitude, Latitude)
19    crime_geo = crime_geo.withColumn(
20        "crime_point",
21        ST_Point(F.col("LON").cast("decimal(24,20)"), F.col("LAT").cast(
22            "decimal(24,20)"))
23    )
24
25    # (Assumed similar processing for stations_df to create stations_geo
26    # with station_point)
27    # ...
28
29    # 3. Find Closest Station (cartesian/cross join)
30    joined_df = crime_geo.crossJoin(stations_geo)
31
32    # Calculate Distance (in meters) using ST_DistanceSphere
33    # ST_Distance would return degrees (Euclidean), Sphere returns
34    # meters on earth surface
35    with_distance = joined_df.withColumn(
```

```

33     "distance_meters",
34     ST_DistanceSphere("crime_point", "station_point")
35 )
36
37 # Window to find the nearest station for each crime
38 w = Window.partitionBy("DR_NO").orderBy(F.col("distance_meters").asc
39 ())
40
41 closest_station = with_distance.withColumn("rnk", F.rank().over(w))
42 \
43     .filter(F.col("rnk") == 1) \
44     .select("DIVISION", "distance_meters")
45
46 # 4. Aggregation
47 # Group by station, count incidents, average distance
48 # Convert meters to kilometers for readability (matching spec
49 example magnitude ~2.0)
50 result_df = closest_station.groupBy("DIVISION").agg(
51     F.count("*").alias("count"),
52     F.avg("distance_meters").alias("avg_dist_meters")
53 )
54
55 # 5. Formatting
56 final_df = result_df.select(
57     F.col("DIVISION").alias("division"),
58     F.format_number(F.col("avg_dist_meters") / 1000, 3).alias("
59 average_distance"), # Convert to km
60     F.col("count").alias("#")
61 ).orderBy(F.col("#").desc())
62
63 return final_df

```

Listing 13: Υλοποίηση Query 4 με DataFrame API και Sedona

6.2 Αποτελέσματα

Τα αποτελέσματα του ερωτήματος, όπως προέκυψαν από την εκτέλεση στο Spark, παρουσιάζονται στον παρακάτω πίνακα:

Table 7: Πλήθος Εγκλημάτων και Μέση Απόσταση ανά Αστυνομικό Τμήμα

Division	Avg Distance (km)	#
HOLLYWOOD	2.074	224,124
VAN NUYS	2.939	208,129
SOUTHWEST	2.191	189,119
WILSHIRE	2.593	186,383
77TH STREET	1.717	170,620
NORTH HOLLYWOOD	2.642	168,096
OLYMPIC	1.729	162,805
PACIFIC	3.853	162,027
CENTRAL	0.993	154,689
RAMPART	1.534	153,204
SOUTHEAST	2.444	143,803

Division	Avg Distance (km)	#
WEST VALLEY	3.022	136,622
FOOTHILL	4.260	132,482
TOPANGA	3.297	131,054
HARBOR	3.702	127,071
HOLLENBECK	2.677	116,235
WEST LOS ANGELES	2.790	115,969
NEWTON	1.635	111,392
NORTHEAST	3.623	108,243
MISSION	3.676	97,926
DEVONSHIRE	2.825	77,180

6.3 Ανάλυση Απόδοσης και Κλιμάκωση

Η ανάλυση του πλάνου εκτέλεσης μέσω της `explain()` δείχνει ότι ο Catalyst Optimizer επιλέγει τη στρατηγική **Broadcast Nested Loop Join**. Η επιλογή αυτή είναι αναμενόμενη και ορθή, καθώς το ερώτημα απαιτεί υποχρεωτικά καρτεσιανό γινόμενο για τον υπολογισμό όλων των αποστάσεων μεταξύ εγκλημάτων και αστυνομικών τμημάτων. Επιπλέον, το σύνολο δεδομένων των αστυνομικών τμημάτων είναι εξαιρετικά μικρό (21 εγγραφές), γεγονός που επιτρέπει την αναπαραγωγή του (broadcast) σε όλους τους executors, αποφεύγοντας το ιδιαίτερα δαπανηρό shuffle ενός πλήρους cross join.

6.3.1 Κλιμάκωση Υπολογιστικών Πόρων

Η υλοποίηση εκτελέστηκε με 2 executors και τις ακόλουθες διαμορφώσεις πόρων:

- 1 core, 2 GB μνήμη
- 2 cores, 4 GB μνήμη
- 4 cores, 8 GB μνήμη

Τα πειραματικά αποτελέσματα δείχνουν ότι ο χρόνος εκτέλεσης παραμένει σχεδόν σταθερός (περίπου **79 δευτερόλεπτα**), με μόνο οριακή μείωση όσο αυξάνονται οι διαθέσιμοι πόροι. Το φαινόμενο αυτό αντιστοιχεί σε **υπο-γραμμική επιτάχυνση (sublinear speedup)** και εξηγείται από τα ακόλουθα χαρακτηριστικά της εκτέλεσης:

1. **Πολλαπλά Shuffles και Δικτυακή Επιβάρυνση:** Το cross join δημιουργεί τεράστιο ενδιάμεσο σύνολο δεδομένων (~ 2.5 εκατ. εγκλήματα × 21 σταθμοί ≈ 52.5 εκατ. εγγραφές), το οποίο οδηγεί σε εκτεταμένη δικτυακή επικοινωνία. Η Window Function προκαλεί εκτεταμένο shuffle για την κατάταξη, ακολουθούμενη από δεύτερο shuffle κατά τη συνάντρωση. Η επαναλαμβανόμενη μεταφορά δεδομένων δημιουργεί έντονο **I/O bottleneck**.
2. **Task Overhead:** Η προεπιλεγμένη παραμετροποίηση `spark.sql.shuffle.partitions=200` οδηγεί στη δημιουργία υπερβολικά μεγάλου αριθμού partitions για το μέγεθος των δεδομένων. Το κόστος διαχείρισης των tasks (scheduling, serialization) καθίσταται συγκρίσιμο με το καθαρό υπολογιστικό έργο. Η μείωση των partitions (π.χ. σε 20-50) οδήγησε σε σημαντικά μικρότερους χρόνους.

Συμπερασματικά, το Query 4 αποτελεί ένα υπολογιστικά βαρύ ερώτημα, στο οποίο η απόδοση κυριαρχείται από το κόστος των shuffles και του task scheduling, και όχι από τη διαθέσιμη υπολογιστική ισχύ, με αποτέλεσμα η αύξηση των πόρων να μην μεταφράζεται σε γραμμική μείωση του χρόνου εκτέλεσης.

7 Query 5

Στόχος του ερωτήματος είναι να διερευνήσουμε αν υπάρχει συσχέτιση μεταξύ του οικονομικού επιπέδου μιας περιοχής και της εγκληματικότητας.

Η διαδικασία ξεκινά με τη φόρτωση των οικονομικών στοιχείων από αρχεία και τον καθαρισμό της στήλης του μέσου εισοδήματος. Αφαιρούνται χαρακτήρες μορφοποίησης όπως το σύμβολο του δολαρίου και τα κόμματα, οι τιμές μετατρέπονται σε αριθμητική μορφή και φιλτράρονται τυχόν κενές εγγραφές, ώστε να δημιουργηθεί ένα καθαρό DataFrame που αντιστοιχίζει Ταχυδρομικούς Κώδικες με εισοδήματα.

Στη συνέχεια, επεξεργαζόμαστε τα δεδομένα των απογραφικών τετραγώνων (Census Blocks) που βρίσκονται σε ακατέργαστη μορφή GeoJSON. Μέσω της συνάρτησης `from_json`, εξάγονται τα δημογραφικά χαρακτηριστικά (Κοινότητα, Ταχυδρομικός Κώδικας, Πληθυσμός), ενώ παράλληλα η γεωμετρία κάθε περιοχής μετατρέπεται από κείμενο σε γεωμετρικό αντικείμενο (`ST_GeomFromGeoJSON`) για να μπορεί να χρησιμοποιηθεί σε χωρικές πράξεις.

7.1 Υλοποίηση

```
1 def query5_df(income_df, blocks_raw, crime_df):
2     """
3     Implementation of Query 5 using DataFrame API and Sedona Spatial
4     Join.
5     """
6     # 1. Process Income Data
7     income_df = income_df.withColumn(
8         "Income",
9         F.regexp_replace(F.col("Estimated Median Income"), "[\\$,]", "").
10        cast(DoubleType())
11    ).filter(
12        F.col("Income").isNotNull()
13    ).select(
14        F.col("Zip Code").alias("ZipCode_Income"),
15        F.col("Income")
16    )
17    # 2. Process Census Blocks (GeoJSON)
18    # Define schema for properties
19    properties_schema = StructType([
20        StructField("COMM", StringType(), True),
21        StructField("ZCTA20", StringType(), True),
22        StructField("POP20", LongType(), True)
23    ])
```

Listing 14: Προετοιμασία Δεδομένων Εισοδήματος και Census Blocks

Τα δεδομένα της απογραφής ενώνονται με τα οικονομικά δεδομένα βάσει του Ταχυδρομικού Κώδικα. Ακολουθεί ομαδοποίηση ανά Κοινότητα (COMM), όπου υπολογίζεται ο συνολικός πληθυσμός και το μέσο εισόδημα της περιοχής. Για το μέσο εισόδημα χρησιμοποιείται σταθμισμένος μέσος όρος (Weighted Average) με βάση τον πληθυσμό κάθε τετραγώνου.

Παράλληλα, τα δεδομένα εγκληματικότητας φορτώνονται και υποβάλλονται σε επεξεργασία για τη δημιουργία γεωμετρικών σημείων. Οι στήλες γεωγραφικού μήκους και πλάτους (LON, LAT) μετατρέπονται σε αριθμούς και χρησιμοποιούνται από τη συνάρτηση ST_Point της βιβλιοθήκης Sedona για να δημιουργηθεί μια στήλη γεωμετρίας για κάθε περιστατικό, αφαιρώντας εγγραφές με ελλιπείς συντεταγμένες.

```

1  # 3. Calculate Community Income & Population
2  blocks_income_joined = blocks_filtered.join(
3      income_df,
4      blocks_filtered.ZCTA20 == income_df.ZipCode_Income,
5      "inner"
6  )
7
8  community_stats = blocks_income_joined.groupBy("COMM").agg(
9      F.sum("POP20").alias("Total_Population"),
10     (F.sum(F.col("POP20") * F.col("Income")) / F.sum("POP20")).alias
11     ("Avg_Income")
12     ).filter(
13         (F.col("Total_Population") > 0) &
14         (F.col("Avg_Income").isNotNull())
15     )
16
17 # 4. Process Crime Data
18 crime_filtered = crime_df.withColumn(
19     "Year", F.year(F.to_timestamp(F.col("DATE OCC"), "yyyy MMM dd hh
20 :mm:ss a"))
21 ).withColumn(
22     "LAT_clean", F.col("LAT").cast(DoubleType())
23 ).withColumn(
24     "LON_clean", F.col("LON").cast(DoubleType())
25 ).select(
26     F.expr("ST_Point(LON_clean, LAT_clean)").alias("crime_geometry")
27 ).filter(
28     F.col("crime_geometry").isNotNull()
29 )

```

Listing 15: Στατιστικά Κοινοτήτων και Γεωμετρία Εγκλημάτων

Έπειτα τα σημεία των εγκλημάτων ενώνονται με τα πολύγωνα των απογραφικών τετραγώνων βάσει της συνθήκης ST_Contains. Αυτή η διαδικασία ταυτοποιεί σε ποια κοινότητα ανήκει κάθε έγκλημα και επιτρέπει την ομαδοποίηση και καταμέτρηση του συνολικού αριθμού εγκλημάτων ανά κοινότητα.

Τέλος, τα συγκεντρωτικά στοιχεία εγκληματικότητας συνενώνονται με τα δημογραφικά και οικονομικά στατιστικά κάθε κοινότητας. Υπολογίζεται ο δείκτης εγκληματικότητας ανά άτομο και στη συνέχεια εφαρμόζεται η στατιστική συνάρτηση corr για να εξαχθεί ο συντελεστής συσχέτισης Pearson μεταξύ εισοδήματος και εγκληματικότητας, τόσο για το σύνολο των δεδομένων όσο και ξεχωριστά για τις 10 πλουσιότερες και 10 φτωχότερες περιοχές.

```

1  # 5. Spatial Join: Crimes -> Communities
2  blocks_geom_only = blocks_filtered.select("COMM", "block_geometry")

```

```

3
4 # Use left join to see what's not matching
5 crime_with_comm = crime_filtered.join(
6     blocks_geom_only,
7     F.expr("ST_Contains(block_geometry, crime_geometry)"),
8     "inner"
9 )
10
11 community_crimes = crime_with_comm.groupBy("COMM").count().
withColumnRenamed("count", "Total_Crimes")
12
13 # 6. Final Calculation & Correlation
14 final_df = community_stats.join(community_crimes, "COMM", "inner")
15
16 final_df = final_df.withColumn(
17     "Crimes_Per_Person",
18     (F.col("Total_Crimes") / 2.0) / F.col("Total_Population")
19 )

```

Listing 16: Spatial Join και Υπολογισμός Συσχέτισης

7.2 Αποτελέσματα

Τα αποτελέσματα του συγκεκριμένου ερωτήματος όπως αυτά εκτυπώνονται από το Spark είναι τα εξής:

QUERY 5 RESULTS: Income vs Crime Rate Correlation

```

=====
Correlation (All Communities): -0.3001
Correlation (Top 10 Income):    0.0555
Correlation (Bottom 10 Income):-0.4116
=====

```

Table 8: Εισόδημα και Εγκληματικότητα ανά Κοινότητα (Δείγμα)

COMM	Total Pop	Avg Income	Total Crimes	Crimes/Person
Miracle Mile	17,275	81,996	4,495	0.130
Harvard Heights	15,806	50,448	4,118	0.130
Mar Vista	39,576	94,651	5,519	0.069
West Hills	39,066	101,955	5,341	0.068
Glassell Park	28,146	82,100	3,791	0.067
Hollywood	65,469	62,834	32,035	0.244
Lennox	20,323	50,052	31	0.001
Hancock Park	16,342	66,132	4,157	0.127
Silverlake	41,129	77,172	8,901	0.108
Reseda	74,448	68,456	12,581	0.084
Park La Brea	12,287	87,868	4,084	0.166
Rosewood/Gardena	1,556	69,645	48	0.015
Del Rey	29,863	103,190	4,499	0.075
Playa Del Rey	2,958	110,884	778	0.131
Baldwin Hills	30,096	50,747	8,269	0.137
Shadow Hills	4,649	91,989	782	0.084

COMM	Total Pop	Avg Income	Total Crimes	Crimes/Person
Sunland	20,768	93,736	2,767	0.066
Elysian Valley	8,856	73,090	1,732	0.097
Toluca Terrace	1,358	61,761	71	0.026
Eagle Rock	36,289	91,132	4,565	0.062
Elysian Park	4,709	57,377	1,756	0.186

7.3 Ανάλυση Πλάνου Εκτέλεσης και Απόδοσης

Μελετώντας το Physical Plan που προέκυψε από την εκτέλεση, εντοπίζουμε τρεις διαφορετικές στρατηγικές Join, μία για κάθε στάδιο της επεξεργασίας:

- **Broadcast Hash Join:** Εντοπίζεται στην ένωση των δεδομένων απογραφής με τα δεδομένα εισοδήματος. Ήταν αναμενόμενο διότι ο πίνακας των εισοδημάτων είναι πολύ μικρός σε μέγεθος. Αντί να ανακαταναείμει (shuffle) τα μεγάλα γεωμετρικά δεδομένα της απογραφής, έστειλε ένα αντίγραφο του μικρού πίνακα εισοδημάτων σε όλους τους executors.
- **Range Join (Spatial Join):** Εντοπίζεται στην ένωση των σημείων εγκλημάτων με τα πολύγωνα των περιοχών. Δεν πρόκειται για μια τυπική spark join στρατηγική, αλλά μια εξειδικευμένη στρατηγική της βιβλιοθήκης Apache Sedona. Το Spark αναγνωρίζει τη γεωχωρική φύση του ερωτήματος και χρησιμοποιεί το Range Join, το οποίο πιθανότατα αξιοποιεί εσωτερικά ευρετήρια για να ελέγξει γρήγορα ποιο σημείο ανήκει σε ποιο πολύγωνο, αποφεύγοντας το εξαιρετικά αργό καρτεσιανό γινόμενο.
- **Broadcast Hash Join:** Εντοπίζεται στην τελική ένωση των συγκεντρωτικών στατιστικών στοιχείων (Total_Population, Avg_Income) βάσει του κλειδιού COMM (Community) με τα συγκεντρωτικά στοιχεία εγκληματικότητας (Total_Crimes) επίσης βάσει του κλειδιού COMM. Ο αριθμός των διαφορετικών τιμών της στήλης COMM είναι μερικές εκατοντάδες, δηλαδή οι dataframes που κάνουμε join είναι μικροί. Για αυτό ο Optimizer επέλεξε αυτή τη στρατηγική διότι είναι πολύ πιο αποδοτικό να κάνει Broadcast έναν από τους 2 dataframes σε όλους τους executors που θα το αποθηκεύσουν στις μνήμες τους, αντί να γίνεται shuffle και να έχουμε network operations όπως θα γινόταν σε άλλες στρατηγικές όπως sort merge join και shuffle hash join.

7.3.1 Κλιμάκωση και Χρόνοι Εκτέλεσης

Ακολουθούν οι χρόνοι εκτέλεσης που καταγράφηκαν για τους συνδυασμούς πόρων:

Table 9: Χρόνοι Εκτέλεσης Query 5 ανά Configuration

Configuration	Execution Time (sec)
2 Executors × 4 Cores (8GB)	211.25
4 Executors × 2 Cores (4GB)	222.45
8 Executors × 1 Core (2GB)	228.37

Όπως φαίνεται από τα πειραματικά αποτελέσματα, παρατηρείται **σταδιακή αύξηση του χρόνου εκτέλεσης** καθώς αυξάνεται ο αριθμός των executors και ταυτόχρονα μειώνεται

ο αριθμός των cores και η διαθέσιμη μνήμη ανά executor, παρότι οι συνολικοί πόροι παραμένουν σταθεροί (8 cores, 16GB μνήμη). Συγκεκριμένα, η ταχύτερη εκτέλεση επιτυγχάνεται με 2 executors \times 4 cores, ενώ η πιο αργή με 8 executors \times 1 core.

Το φαινόμενο αυτό εξηγείται από τον τρόπο με τον οποίο το Spark εκτελεί ερωτήματα με πολλαπλά joins, χωρικές πράξεις και συγκεντρωτικές λειτουργίες. Πιο συγκεκριμένα:

1. Κυριαρχία Shuffles και Wide Transformations

Το ερώτημα περιλαμβάνει πολλές πράξεις που προκαλούν wide transformations, δηλαδή απαιτούν ανακατανομή δεδομένων μεταξύ executors:

- Join των Census Blocks με τα οικονομικά δεδομένα βάσει Ταχυδρομικού Κώδικα.
- Ομαδοποίηση (groupBy) ανά Κοινότητα για τον υπολογισμό πληθυσμού και μέσου εισοδήματος.
- Χωρικό join μεταξύ σημείων εγκλημάτων και πολυγώνων απογραφικών τετραγώνων μέσω της συνθήκης ST_Contains.
- Ομαδοποίηση εγκλημάτων ανά κοινότητα.
- Τελικό join δημογραφικών και εγκληματολογικών δεδομένων.
- Η εξαγωγή των 10 πλουσιότερων και 10 φτωχότερων περιοχών απαιτεί ταξινόμηση.

Κάθε μία από αυτές τις πράξεις συνεπάγεται shuffle, δηλαδή μεταφορά δεδομένων μέσω δικτύου, σειριοποίηση και αποσειριοποίηση. Το κόστος αυτών των shuffles δεν μειώνεται γραμμικά με την αύξηση των executors· αντίθετα, αυξάνεται όταν τα δεδομένα διαμοιράζονται σε περισσότερους executors με μικρότερο τοπικό χώρο μνήμης.

2. Επίδραση του Spatial Join (ST_Contains)

Η χωρική σύζευξη μεταξύ εγκλημάτων και απογραφικών τετραγώνων αποτελεί το βαρύτερο υπολογιστικά στάδιο του ερωτήματος. Η πράξη ST_Contains δεν είναι απλό equi-join, αλλά παράγει μεγάλο ενδιαμέσο σύνολο δεδομένων. Με περισσότερους executors των 1 core και 2GB μνήμης, κάθε executor διαθέτει περιορισμένο buffer μνήμης, με αποτέλεσμα αυξημένες εγγραφές σε δίσκο (spill) κατά τη διάρκεια του shuffle. Αυτό αυξάνει σημαντικά τον χρόνο εκτέλεσης.

3. Overhead Διαχείρισης Executors και Tasks

Η αύξηση του αριθμού των executors οδηγεί σε:

- Περισσότερες συνδέσεις δικτύου κατά τη διάρκεια των shuffles.
- Μεγαλύτερο κόστος task scheduling και συντονισμού από τον Spark driver.
- Μικρότερα tasks, των οποίων το διαχειριστικό κόστος γίνεται συγκρίσιμο με το καθαρό υπολογιστικό έργο.

Ιδιαίτερα στο configuration με 1 core ανά executor, η έλλειψη εσωτερικού παραλληλισμού οδηγεί σε υποαξιοποίηση των πόρων, ενώ το συνολικό overhead αυξάνεται.

Συμπέρασμα

Συμπερασματικά, η αύξηση του χρόνου εκτέλεσης με περισσότερους executors και λιγότερους πόρους ανά executor δεν αποτελεί ανωμαλία, αλλά αναμενόμενη συμπεριφορά για ένα δεδομένο ερώτημα, το οποίο:

- Κυριαρχείται από shuffle-heavy joins και χωρικές πράξεις.
- Επωφελείται από μεγαλύτερη μνήμη ανά executor.
- Δεν είναι καθαρά CPU-bound.

Ως αποτέλεσμα, η διάταξη με λιγότερους executors και περισσότερους πόρους ανά executor (**2 Executors × 4 Cores × 8GB**) προσφέρει την καλύτερη απόδοση, ενώ η υπερκατανομή executors οδηγεί σε αυξημένο overhead, disk spill και καθυστέρηση.