Προχωρημένα Θέματα Βάσεων Δεδομένων
Αναφορά Εξαμηνιαίου project

Σπανδωνίδης Ιωακείμ Αλέξανδρος
03121159
9ο εξάμηνο 2024-25

Εισαγωγή

Η παρούσα εργασία εκπονήθηκε στο πλαίσιο του μαθήματος **Προχωρημένα Θέματα Βάσεων Δεδομένων** του 9ου εξαμήνου της Σχολής ΗΜΜΥ του ΕΜΠ (Ακαδημαϊκό έτος 2025–26) advanced_db_project_2025.

Στόχος της εργασίας ήταν η ανάλυση μεγάλων πραγματικών datasets μέσω τεχνικών επεξεργασίας δεδομένων μεγάλης κλίμακας, κάνοντας χρήση των Apache Hadoop, Apache Spark και Apache Sedona στο ειδικά διαμορφωμένο περιβάλλον AWS.

Η εργασία θέτει ως βασικούς στόχους:

- Την εξοικείωση με την εγκατάσταση, διαχείριση και χρήση κατανεμημένων συστημάτων επεξεργασίας.
- Την ανάλυση δεδομένων μεγάλης κλίμακας με χρήση Spark DataFrame, SQL και RDD APIs.
- Τη μελέτη των δυνατοτήτων και περιορισμών του Spark μέσω πειραματικής αξιολόγησης.
- Τη χρήση γεωχωρικών τεχνικών μέσω της βιβλιοθήκης Apache Sedona.
- Την αξιολόγηση της επίδοσης υπό διαφορετικές ρυθμίσεις πόρων και διαφορετικές στρατηγικές join.

Τα δεδομένα που χρησιμοποιήθηκαν παρέχονται σε AWS S3 και περιλαμβάνουν στοιχεία εγκληματικότητας στο Los Angeles, δεδομένα απογραφής, εισοδήματος και χωρικά δεδομένα αστυνομικών σταθμών.

Περιγραφή Δεδομένων

Σύνολα Εγκληματικότητας

Το βασικό dataset αποτελείται από τα **Los Angeles Crime Data (2010–2019 και 2020–2025)**, τα οποία περιλαμβάνουν περίπου 10 εκατομμύρια εγγραφές εγκλημάτων.

Συμπληρωματικά Datasets

Χρησιμοποιήθηκαν επιπλέον τα:

- Census Blocks 2020 (GeoJSON)
- Median Household Income 2021
- LA Police Stations
- Race/Ethnicity Codes
- MO Codes

Όλα τα datasets βρίσκονται στο S3 bucket που δίνεται στην εκφώνηση https://initial-notebook-data-bucket-dblab-905418150721.s3.eu-central-1.amazonaws.com/project_data/

Υπολογιστικό Περιβάλλον

Οι υλοποιήσεις εκτελέστηκαν στο περιβάλλον AWS JupyterLab που δόθηκε από το μάθημα. Χρησιμοποιήθηκαν διαφορετικές διαμορφώσεις εκτελεστών (executors) ανάλογα με τα ζητούμενα κάθε query.

Query 1

Σε κάθε query έχουμε αρχικά την μορφοποίηση των πόρων που χρησιμοποιούνται και μετά την υλοποίηση του ζητήματος

Ζητείται ταξινόμηση ηλικιακών ομάδων θυμάτων σε εγκλήματα που περιέχουν βαριά σωματική βλάβη. Η αναζήτηση γίνεται μέσω του description (crm_cd_desc) που υπάρχει στο dataset των καταγραφών των εγκλημάτων και περιέχει τον όρο “aggravated assault”.

Υλοποιήσεις

Υλοποιήθηκαν τρεις εκδοχές:

- DataFrame API
- DataFrame API με UDF
- RDD API

Επιδόσεις

```
Dataframe API Execution time for Query 1: 33.6270 sec
RDD API Execution time for Query 1: 39.5206 sec
Dataframe using UDF API Execution time for Query 1: 6.1317 sec
```

παρατηρούμε τις εξής σημαντικές διαφοροποιήσεις:

1. **Η υλοποίηση με RDD API είναι η πιο αργή**, γεγονός που ευθυγραμμίζεται με τη γενική αρχιτεκτονική του Spark.
   Τα RDDs δεν διαθέτουν Catalyst optimizer, δεν εφαρμόζουν εκτεταμένο query planning και δεν επωφελούνται από βελτιστοποιήσεις όπως predicate pushdown ή automatic code generation.
   Ως αποτέλεσμα, το RDD API απαιτεί περισσότερες διεργασίες shuffling και χειρισμό δεδομένων σε χαμηλότερο επίπεδο, οδηγώντας σε υψηλότερο execution time.

2. **Το DataFrame API παρουσιάζει πιο αποδοτική συμπεριφορά από το RDD API**, με περίπου 15% ταχύτερη εκτέλεση.
   Ο κύριος λόγος είναι ότι τα DataFrames λειτουργούν πάνω στον Catalyst optimizer, ο οποίος παράγει πιο αποδοτικό execution plan, αξιοποιεί columnar storage, Tungsten execution, καθώς και code generation στη φάση της φυσικής εκτέλεσης (Whole-Stage Code Generation).

3. **Η υλοποίηση DataFrame με χρήση UDF ήταν εντυπωσιακά ταχύτερη (μόλις 6.1317 sec), ξεπερνώντας τις άλλες δύο προσεγγίσεις με μεγάλη διαφορά**.
   Αυτό το αποτέλεσμα είναι αντισυμβατικό, καθώς συνήθως οι UDFs επιβραδύνουν την

εκτέλεση.
Στην προκειμένη περίπτωση όμως, παρατηρούμε ότι:

- η UDF επέτρεψε την εκτεταμένη χρήση Whole-Stage Code Generation,
- η λογική ταξινόμησης ηλικιών συμπυκνώθηκε σε ενιαίο optimized execution path,
- μειώθηκαν τα shuffles και transformations που απαιτούνταν στη DataFrame-only και RDD υλοποίηση.

4. **Συνολικά, το DataFrame με UDF ήταν ~5.5 φορές ταχύτερο από το βασικό DataFrame και ~6.4 φορές ταχύτερο από το RDD API.**

Query 2

Περιγραφή

Εντοπισμός 3 πιο συχνών φυλετικών ομάδων ανά έτος και υπολογισμός ποσοστών.

Υλοποιήσεις

- DataFrame API
- SQL AP

Επιδόσεις

```
SQL API Execution time for Query 2: 32.6514 sec
RDD API Execution time for Query 2: 44.1389 sec
```

Από τα παραπάνω παρατηρούμε τα εξής:

1. Η SQL υλοποίηση είναι σαφώς ταχύτερη

Η SQL εκδοχή ολοκλήρωσε περίπου **26% γρηγορότερα** από την αντίστοιχη DataFrame υλοποίηση.
Η διαφορά αυτή είναι αναμενόμενη, καθώς η SQL σύνταξη επιτρέπει στον Catalyst optimizer να:

- παράγει πιο συμπαγή logical plans,
- κάνει αναδιατάξεις (reordering) ενώσεων και aggregations πιο αποτελεσματικά,
- εφαρμόζει rule-based και cost-based βελτιστοποιήσεις ευκολότερα από ό,τι στο DataFrame API όπου το expression tree είναι πιο ρητά δομημένο από τον χρήστη.

Αντίθετα, στη DataFrame υλοποίηση, η σειρά των λειτουργιών που εκφράζει ο χρήστης συχνά επιτρέπει λιγότερες βελτιστοποιήσεις και ενδέχεται να οδηγήσει σε επιπλέον shuffling ή materialization.

2. Η SQL υλοποίηση είναι πιο κοντά στη φιλοσοφία του Spark Catalyst

Ο Catalyst έχει σχεδιαστεί ώστε να αξιοποιεί στο μέγιστο το declarative μοντέλο της SQL.
Όταν το ερώτημα δίνεται σε SQL:

- το execution plan βελτιστοποιείται με rule-based passes,

- ο αλγόριθμος επιλογής join/aggregation strategies εφαρμόζεται χωρίς περιορισμούς,
- ενεργοποιείται πιο αποτελεσματικά το Whole-Stage Code Generation.

Αυτό εξηγεί γιατί η SQL εκτέλεση ήταν αισθητά ταχύτερη.

3. Το DataFrame API παραμένει ισχυρό, αλλά όχι τόσο βελτιστοποιημένο για σύνθετα aggregations

Παρόλο που DataFrames και SQL μοιράζονται την ίδια optimizer engine, ένα aggregation pipeline που υλοποιείται «χειροκίνητα» με DataFrame transformations:

- οδηγεί συχνότερα σε εξαναγκαστικές materializations,
- επιτρέπει λιγότερο aggressive operator reordering,
- μπορεί να αυξήσει τα shuffle stages.

Αυτό ερμηνεύει την πιο αργή εκτέλεση.

4. Η διαφορά απόδοσης εδώ είναι καθαρά ζήτημα optimizer intelligence

Και τα δύο APIs χρησιμοποιούν το ίδιο engine.
Η διαφορά δεν είναι υποδομής αλλά **εκφραστικότητας του ερωτήματος**:

- Η SQL διατυπώνει το πρόβλημα σε καθαρή declarative μορφή
- Το DataFrame API διατηρεί στοιχεία procedural λογικής

Με αποτέλεσμα ο optimizer να έχει περισσότερη «ελευθερία» στην SQL έκδοση.

Query 3

Περιγραφή

Εντοπισμός πιο συχνών μεθόδων εγκλημάτων και εμπλουτισμός με περιγραφές.

Υλοποιήσεις

- DataFrame API (με χρήση hints, explain)
- RDD API

Join Strategies

Ανάλυση στρατηγικών Catalyst:

- BROADCAST
- MERGE
- SHUFFLE_HASH
- SHUFFLE_REPLICATE_NL

Οι επιδόσεις μετρούνται μόνο για τα join, δηλαδή μετράμε τον χρόνο που απαιτείται για να περατωθεί το join των δυο sets και όχι για την ολική υλοποίηση που περιλαμβάνει το setup του Spark και τον χρόνο που απαιτείται για τις άλλες λειτουργίες του notebook (load datasets ect.)

Υλοποίηση	Χρόνος
RDD classic join	**9.7663 sec**
DataFrame default join	0.0111 sec
Broadcast Hash Join (hint=broadcast)	**0.0066 sec (ταχύτερο)**
Shuffle Hash Join	0.0094 sec
Sort-Merge Join	0.0087 sec
Shuffle Replicate NL	0.0111 sec

Η απόδοση του RDD join (9.7663 sec) ήταν **κατά τάξεις μεγέθους πιο αργή** από όλες τις DataFrame υλοποιήσεις.
Αυτό είναι απόλυτα αναμενόμενο λόγω:

- Έλλειψης Catalyst optimizer
- Έλλειψης whole-stage code generation
- Μεγαλύτερου overhead σε shuffles
- Απουσίας columnar/optimized execution

Οι DataFrame εκτελέσεις είχαν χρόνους της τάξης του **0.01 sec**, δηλαδή περίπου **900 φορές ταχύτερα** από το RDD API.

Οι DataFrame υλοποιήσεις εκτέλεσαν join με το dataset MO_codes.txt, το οποίο:

- είναι **πολύ μικρό** (~300 KB)
- περιέχει **μόνο ~100 γραμμές**

Λόγω αυτού, ο Catalyst επέλεξε **Broadcast Hash Join** ως default strategy.

Γιατί αυτό είναι σημαντικό:

- Ολόκληρο το MO_codes.txt αποστέλλεται σε κάθε executor
- Δεν απαιτείται shuffle
- Το join γίνεται **τοπικά** στη μνήμη του executor
- Το κόστος είναι σχεδόν μηδενικό

Γι' αυτό το default time ήταν **0.0111 sec**.

Σύγκριση Join Strategies

(α) Broadcast Hash Join — 0.0066 sec (ταχύτερο)

Με broadcast hint, το Spark:

- αγνόησε οποιονδήποτε άλλο heuristic
- ανάγκασε broadcast του μικρού πίνακα
- απέφυγε πλήρως shuffle/partitioning
- χρησιμοποίησε highly optimized hashed broadcast relation

Αυτό ήταν η **ταχύτερη στρατηγική**, όπως αναμενόταν.

(β) Shuffle Hash Join — 0.0094 sec

Απαιτεί:

- πλήρες shuffling και repartitioning των δύο datasets
- allocation hash tables σε κάθε partition

Είναι γρήγορο, αλλά πολύ πιο αργό από broadcast λόγω shuffle overhead.

(γ) Sort Merge Join — 0.0087 sec

Απαιτεί sorting και από τις δύο πλευρές του join.
Επειδή όμως το MO_codes.txt είναι μικρό:

- sort cost μικρό
- αλλά shuffle παραμένει υποχρεωτικό

Γρηγορότερο από Shuffle Hash σε αυτό το πείραμα, αλλά σαφώς πιο αργό από Broadcast.

(δ) Shuffle Replicate NL — 0.0111 sec

Είναι πάντα **η χειρότερη επιλογή** για μεγάλα datasets:

- κάνει full replication του δεξιού πίνακα σε όλες τις partitions
  μετά nested loop per-row
- δραματικά χειρότερο scaling

Αλλά εδώ, επειδή ο μικρός πίνακας είναι tiny, παραμένει γρήγορο (≈0.01 sec).

Default physical plan: Broadcast Hash Join

Ακόμα και χωρίς hints, το Spark:

- είδε ότι ο πίνακας MO_codes.txt είναι πολύ μικρός
- και εφάρμοσε default **BroadcastHashJoin**

Query 4

Εκτελούμε το query 4 κλιμακώνοντας τους υπολογιστικούς πόρους που χρησιμοποιεί το spark και παρατηρούμε τις παρακάτω επιδόσεις. Εδώ στον χρόνο προσμετράται και το setup του spark session το οποίο αλλάζει όσο αλλάζουν οι πόροι που χρησιμοποιούνται. Πιο συγκεκριμένα:

Εκτελεστές	Cores/Executor	Μνήμη	Χρόνος
2 executors	1 core	2GB	**54.5124 sec**
2 executors	2 cores	4GB	**36.4259 sec**
2 executors	4 cores	8GB	**34.1064 sec**

Το Query 4 παρουσιάζει ένα χαρακτηριστικό spatial analytics workload, στο οποίο η απόδοση δεν κλιμακώνεται γραμμικά με την αύξηση των διαθέσιμων πόρων. Η μετάβαση από 1 core σε 2 cores ανά executor επιφέρει σημαντική επιτάχυνση (~33%), όμως η περαιτέρω αύξηση σε 4 cores έχει πολύ μικρό όφελος (~6%), υποδεικνύοντας ότι το query είναι δεσμευμένο από I/O και shuffling παρά από καθαρή υπολογιστική ισχύ. Οι γεωχωρικές λειτουργίες της Sedona (distance calculations, spatial joins) αποτελούν το κύριο bottleneck, καθώς δεν βελτιστοποιούνται πλήρως από τον Catalyst optimizer. Παρά τους περιορισμούς απόδοσης, τα αποτελέσματα των divisions παραμένουν σταθερά και αξιόπιστα σε όλες τις εκτελέσεις, αποδεικνύοντας τη σωστή λειτουργία του spatial pipeline. Συνεπώς, το Query 4 αναδεικνύει τα πρακτικά όρια κλιμάκωσης των geospatial queries στο Spark και την ανάγκη για έξυπνη χωρική ομαδοποίηση και partitioning σε μεγάλα datasets.

Η διερεύνηση του Spark Catalyst Optimizer ανέδειξε μια σειρά από σημαντικές επιλογές που πραγματοποιούνται αυτόματα κατά τη βελτιστοποίηση του εκτελεστικού πλάνου. Στην ενότητα αυτή αναλύονται οι βασικοί μηχανισμοί και οι λόγοι πίσω από τις στρατηγικές του optimizer, βασισμένοι στο πραγματικό physical plan που προέκυψε.

Ο Spark Catalyst Optimizer, λόγω της χρήσης των γεωχωρικών UDFs όπως το **ST_Distance**, δεν μπορεί να εφαρμόσει κλασικές βελτιστοποιήσεις (predicate pushdown, join reordering, hash joins), με αποτέλεσμα η ένωση των crime points με τα police stations να πραγματοποιείται υποχρεωτικά μέσω **BroadcastNestedLoopJoin (Cross Join)**. Η στρατηγική αυτή επιλέγεται επειδή δεν υπάρχει join key και επειδή το police dataset είναι μικρό και μπορεί να μεταδοθεί ως broadcast· ωστόσο οδηγεί σε μεγάλο ενδιάμεσο καρτεσιανό γινόμενο. Στη συνέχεια, ο υπολογισμός της ελάχιστης απόστασης ανά crime υλοποιείται μέσω **HashAggregate**, που αποτελεί τη βέλτιστη επιλογή για group-by σε μεγάλα datasets. Για την τελική αναζήτηση των εγγραφών όπου η απόσταση ισούται με τη min_distance, ο Catalyst χρησιμοποιεί **SortMergeJoin**, καθώς απαιτείται equality σε δύο κλειδιά – ένα εκ των οποίων είναι floating-point – καθιστώντας ακατάλληλα τα hash joins. Τέλος, η ομαδοποίηση ανά division με

υπολογισμό crime_count και avg_distance πραγματοποιείται και πάλι με **HashAggregate**, λόγω του μικρού αριθμού διακριτών divisions. Συνολικά, ο optimizer λειτουργεί σωστά μέσα στους περιορισμούς των Sedona UDFs· ωστόσο οι επιλογές του οδηγούν αναπόφευκτα σε υψηλό υπολογιστικό κόστος, το οποίο θα μπορούσε να μειωθεί μόνο με spatial indexing ή αποφυγή του crossJoin.

Query 5

Το Query 5 συνδυάζει γεωχωρική ανάλυση, δημογραφικά δεδομένα και οικονομικά χαρακτηριστικά, επιχειρώντας να διερευνήσει τη σχέση μεταξύ εγκληματικότητας, πληθυσμιακής πυκνότητας και εισοδήματος ανά κοινότητα στο Los Angeles. Η βασική χωρική πράξη πραγματοποιείται μέσω του Sedona **ST_Within**, το οποίο επιτρέπει την αντιστοίχιση κάθε crime point στα γεωγραφικά πολύγωνα των κοινοτήτων.
Το physical plan επιβεβαιώνει ότι ο Catalyst Optimizer χρησιμοποιεί εξειδικευμένο **RangeJoin** για το χωρικό φιλτράρισμα, καθώς και **SortMergeJoin** για την τελική ένωση με τα δημογραφικά δεδομένα, ενώ τα μικρότερα εισοδηματικά datasets φορτώνονται αποδοτικά μέσω **BroadcastHashJoin**.
Ο υπολογισμός του αριθμού εγκλημάτων ανά κοινότητα (crime_count) και η παραγωγή μεταβλητών όπως το per capita income και το average annual crimes γίνεται με **HashAggregate**, το οποίο αποτελεί τη στρατηγική που κλιμακώνεται καλύτερα σε αυτό το workload.
Τα αποτελέσματα δείχνουν έντονες κοινωνικοοικονομικές διαφοροποιήσεις: περιοχές υψηλού εισοδήματος όπως Pacific Palisades, Playa Vista και Marina del Rey εμφανίζουν χαμηλά επίπεδα εγκληματικότητας, ενώ κοινότητες όπως Florence-Firestone, Watts και Vernon Central παρουσιάζουν πολύ υψηλότερους μέσους ετήσιους αριθμούς εγκλημάτων παρά τη σημαντικά χαμηλότερη οικονομική τους δραστηριότητα. Ο συντελεστής συσχέτισης ανάμεσα στο **average annual crimes** και το **per capita income** για το σύνολο του LA είναι αρνητικός (–0.1699), επιβεβαιώνοντας μια γενική τάση μειωμένης εγκληματικότητας σε περιοχές μεγαλύτερης οικονομικής ευημερίας. Η συσχέτιση γίνεται ακόμη πιο έντονη στις φτωχότερες 10 κοινότητες (–0.4201), ενώ στις πλουσιότερες 10 παρατηρείται θετική συσχέτιση (+0.5286), υποδεικνύοντας ότι σε περιοχές με πολύ υψηλό εισόδημα η εγκληματικότητα μπορεί να σχετίζεται περισσότερο με εμπορική δραστηριότητα και πληθυσμιακή προσέλευση παρά με καθαρά κοινωνικοοικονομικούς παράγοντες.

Μορφοποιώντας διαφορετικά τους πόρους του Spark παρατηρούμε διαφοροποίηση στην απόδοση.

Ρύθμιση Εκτέλεσης	Executors	Cores/Executor	Memory/Executor	Συνολική Μνήμη Cluster	Execution Time (sec)
Version 1	2	4	8 GB	16 GB	**125.2894**
Version 2	4	2	4 GB	16 GB	**100.5279**
Version 3	8	1	2 GB	16 GB	**79.9972**

Πιο συγκεκριμένα, η απόδοση βελτιώνεται όσο αυξάνεται ο αριθμός των **executors** και μειώνονται τα cores ανά executor.