ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ
ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Προχωρημένα Θέματα Βάσεων Δεδομένων

Αναφορά Εξαμηνιαίας Εργασίας

Γεώργιος-Αλέξιος Καπετανάκης, 03119062
Δημήτριος Γεωργούσης, 03119005

Όλα τα αρχεία που αναφέροντα παρακάτω μπορούν να βρεθούν στο GitHub repository μας: https://github.com/ntua-el19062/adv_db_project.

Ζητούμενο 1

Για τη διαμόρφωση του περιβάλλοντος εκτέλεσης των υπόλοιπων ερωτημάτων χρησιμοποιούμε τους διαθέσιμους πόρους μας στον ιστότοπο *okeanos-knossos*. Χρησιμοποιούμε δύο εικονικές μηχανές Ubuntu Server LTS διαθέτοντας 4 CPUs και 8 GiB RAM για την κάθε μια. Για την διαμόρφωση των εικονικών μηχανών χρησιμοποιούμε το script **setup_vm.sh**.

Συνοπτικά το script:

- Εγκαθιστά τα αναγκαία packages για χρήση Python, Java, Hadoop και Spark.
- Εγκαθιστά το OpenJDK Java 8.
- Εγκαθιστά το εργαλείο pyenv, καθώς και την έκδοση 3.8.18 της Python.
- Εγκαθιστά τα πακέτα geopy, pyspark της Python.
- Εγκαθιστά τα Hadoop και Spark.
- Διαμορφώνει κατάλληλα διάφορα αρχεία (~/.bashrc, /etc/hosts, κλπ) ώστε όλα τα παραπάνω να λειτουργούν σωστά.
- Αλλάζει το hostname της μηχανής και έπειτα την επανεκκινεί.

Αφού εκτελεστεί το παραπάνω script, χρησιμοποιώντας τις εντολές ssh-keygen και ssh-copy-id διαμορφώνουμε τις μηχανές κατάλληλα ώστε να μπορούμε να εκτελέσουμε passwordless ssh από κάθε μηχανή προς την άλλη και προς τον εαυτό της.

Έπειτα εκτελούμε το script **format_hdfs.sh**, το οποίο εκκινεί το HDFS και δημιουργεί όλα τα αρχεία που χρειάζονται για την σωστή εκτέλεση των επόμενων ερωτημάτων.

Μετά την εκκίνηση των HDFS, Yarn και Spark, οι web εφαρμογές τους είναι διαθέσιμες στις ακόλουθες διευθύνσεις (εφόσον δεν χρειαστεί να δημιουργήσουμε νέα IP):

- HDFS: http://83.212.80.118:9870/
- Yarn: http://83.212.80.118:8088/cluster
- Spark: http://83.212.80.118:18080/

Ζητούμενο 2

Κατεβάζουμε τα σύνολα δεδομένων χρησιμοποιώντας το script **download_datasets.sh** και τα αντιγράφουμε στο HDFS χρησιμοποιώντας το script **copy_datasets.sh**.
Στο αρχείο **schemas.py** δημιουργούμε schemas για τα datasets μας χρησιμοποιώντας συναρτήσεις της pyspark. Στο αρχείο **paths.py** δημιουργούμε μεταβλητές με τα μονοπάτια των datasets στο HDFS. Στο αρχείο **common.py** δημιουργούμε συναρτήσεις που χρησιμοποιούνται σε (σχεδόν) όλα τα queries.
Για την ενωποίηση των δύο βασικών συνόλων δεδομένων και την προσαρμογή των τύπων δεδομένων όπως αναφέρεται στην εκφώνηση του ζητουμένου χρησιμοποιούμε το script **preprocessing/df_api.py**, μέσω της εντολής:

- `spark-submit preprocessing/df_api.py`

Ζητούμενο 2 – Κώδικας

```
# common.py
def load_crime_data_from_2010_to_2019(spark):
    return (
        spark.read.option("header", True)
        .option("ignoreLeadingWhiteSpace", True)
        .option("ignoreTrailingWhiteSpace", True)
        .option("dateFormat", "MM/dd/yyyy hh:mm:ss a")
        .csv(
            path=paths.CRIME_DATA_FROM_2010_TO_2019_PATH,
            schema=schemas.CRIME_DATA_FROM_2010_TO_2019_SCHEMA,
        )
    )

# preprocessing/df_api.py
crime_data_pt_1 = load_crime_data_from_2010_to_2019(spark)
crime_data_pt_2 = load_crime_data_from_2020_to_present(spark)
crime_data = crime_data_pt_1.union(crime_data_pt_2).dropDuplicates()
```

Ζητούμενο 2 – Έξοδος

```
PREPROCESSING: Dataset 1 (2010 - 2020) has 2122469 rows

PREPROCESSING: Dataset 2 (2020 - Present) has 879106rows

PREPROCESSING: The merged dataset (2010 - Present) has 3001575rows (0
duplicate rows were dropped)

PREPROCESSING: The contents of the merged dataset (2010 - Present)
were successfully written to disk

PREPROCESSING: The merged dataset (2010 - Present) has the following
schema:
root
 |-- DR_NO: integer (nullable = true)
 |-- Date Rptd: date (nullable = true)
 |-- DATE OCC: date (nullable = true)
 |-- TIME OCC: string (nullable = true)
 |-- AREA: integer (nullable = true)
 |-- AREA NAME: string (nullable = true)
 |-- Rpt Dist No: integer (nullable = true)
 |-- Part 1-2: string (nullable = true)
 |-- Crm Cd: integer (nullable = true)
 |-- Crm Cd Desc: string (nullable = true)
 |-- Mocodes: string (nullable = true)
 |-- Vict Age: integer (nullable = true)
 |-- Vict Sex: string (nullable = true)
 |-- Vict Descent: string (nullable = true)
 |-- Premis Cd: integer (nullable = true)
 |-- Premis Desc: string (nullable = true)
 |-- Weapon Used Cd: integer (nullable = true)
 |-- Weapon Desc: string (nullable = true)
 |-- Status: string (nullable = true)
 |-- Status Desc: string (nullable = true)
 |-- Crm Cd 1: integer (nullable = true)
 |-- Crm Cd 2: integer (nullable = true)
 |-- Crm Cd 3: integer (nullable = true)
 |-- Crm Cd 4: integer (nullable = true)
 |-- LOCATION: string (nullable = true)
 |-- Cross Street: string (nullable = true)
 |-- LAT: double (nullable = true)
 |-- LON: double (nullable = true)
```

Ζητούμενο 3

Υλοποιούμε το Query 1 με τα DataFrame και SQL APIs. Τα εκτελούμε με τις εντολές:

- `spark-submit --num-executors 4 query_1/df_api.py`
- `spark-submit --num-executors 4 query_1/sql_api.py`

Ο κώδικας μας λειτουργεί ως εξής:

- Δημιουργούμε τις στήλες 'Year', 'Month' από την στήλη 'DATE OCC' (η ημερομηνία στην οποία έγινε το έγκλημα).
- Ομαδοποιούμε τα εγκλήματα κατά έτος και μήνα και μετράμε πόσα εγκλήματα έγιναν σε κάθε συνδυασμό έτους-μήνα.
- Για κάθε έτος αναδιατάσσουμε την σειρά των μηνών του έτους ανάλογα με τον αριθμό εγκλημάτων σε φθίνουσα σειρά. Αριθμούμε τις σειρές από πάνω προς τα κάτω (ο μήνας με τα περισσότερα εγκλήματα παίρνει την τιμή 1, ο αμέσως επόμενος την τιμή 2, κλπ.).
- Κρατάμε για κάθε έτος μόνο τους μήνες με τιμές 1, 2, ή 3 και ταξινομούμε τα αποτελέσματα μας κατά αύξον έτος και φθίνοντα αριθμό εγκλημάτων.

Παρατηρούμε ότι η επίδοση των διαφορετικών APIs είναι σε γενικές γραμμές ίδια. Ο λόγος που συμβαίνει αυτό είναι το γεγονός ότι και τα δύο APIs χρησιμοποιούν το ίδιο execution engine και τον ίδιο optimizer (Catalyst optimizer). Χρησιμοποιώντας, μάλιστα, την μέθοδο explain παρατηρούμε ότι και τα δύο queries παράγουν το ίδιο execution plan.

Ζητούμενο 3 – Κώδικας

```python
# query_1/df_api.py
crime_data = (
    load_crime_data_from_2010_to_present(spark)
    .withColumn("Year", f.year("DATE OCC"))
    .withColumn("Month", f.month("DATE OCC"))
    .groupBy(f.col("Year"), f.col("Month"))
    .count()
    .withColumnRenamed("count", "Total Crimes")
    .withColumn(
        "#",
        f.row_number().over(
            Window.partitionBy(f.col("Year")).orderBy(
                f.col("Total Crimes").desc()
            )
        ),
    )
    .filter(f.col("#") <= 3)
    .orderBy(f.col("Year").asc(), f.col("Total Crimes").desc())
)
```

```python
# query_1/sql_api.py
crime_data = spark.sql(
    """
    SELECT *
    FROM (SELECT YEAR(`DATE OCC`) AS `Year`,
                 MONTH(`DATE OCC`) AS `Month`,
                 COUNT(*) AS `Total Crimes`,
                 ROW_NUMBER() OVER (PARTITION BY YEAR(`DATE OCC`)
                                    ORDER BY COUNT(*) DESC) AS `#`
          FROM `crime_data`
          GROUP BY `Year`, `Month`)
    WHERE `#` <= 3
    ORDER BY `Year` ASC, `Total Crimes` DESC
    """
)
```

Ζητούμενο 3 – Έξοδος

```
Q1: Query results:
+----+-----+------------+---+
|Year|Month|Total Crimes|#  |
+----+-----+------------+---+
|2010|1    |19517       |1  |
|2010|3    |18131       |2  |
|2010|7    |17856       |3  |
|2011|1    |18138       |1  |
|2011|7    |17283       |2  |
|2011|10   |17034       |3  |
|2012|1    |17946       |1  |
|2012|8    |17661       |2  |
|2012|5    |17502       |3  |
|2013|8    |17441       |1  |
|2013|1    |16822       |2  |
|2013|7    |16644       |3  |
|2014|10   |17329       |1  |
|2014|7    |17258       |2  |
|2014|12   |17198       |3  |
|2015|10   |19220       |1  |
|2015|8    |19011       |2  |
|2015|7    |18709       |3  |
|2016|10   |19659       |1  |
|2016|8    |19491       |2  |
|2016|7    |19448       |3  |
|2017|10   |20433       |1  |
|2017|7    |20193       |2  |
|2017|1    |19835       |3  |
|2018|5    |19974       |1  |
+----+-----+------------+---+
only showing top 25 rows
```

Ζητούμενο 4

Υλοποιούμε το Query 2 με τα DataFrame και RDD APIs. Τα εκτελούμε με τις εντολές:

- `spark-submit --num-executors 4 query_2/df_api.py`
- `spark-submit --num-executors 4 query_2/rdd_api.py`

Ο κώδικας μας λειτουργεί ως εξής:

- Φιλτράρουμε το dataset και κρατάμε μόνο εγκλήματα που έγιναν στο δρόμο ('Premis Desc' == 'STREET') και που έχουν τιμή στο πεδίο 'TIME OCC'.
- Έπειτα ελέγχουμε την τιμή του 'TIME OCC' και τα κατηγοριοποιούμε ανάλογα σε 'Morning', 'Afternoon', 'Evening', 'Night'. Ονομάζουμε τη νέα στήλη 'Phase of Day'.
- Ομαδοποιούμε τα εγκλήματα και μετράμε πόσα έγιναν ανα 'Phase of Day', ταξινομώντας τα κατά φθίνουσα σειρά.
- Στην υλοποίηση με το RDD API προσέχουμε τα εξής:
    - Κατά την ανάγνωση του dataset χρησιμοποιούμε έναν CSV reader από το `csv` module της python, ώστε να χειριστεί σωστά την εμφάνιση κόμα (,) εντός των τιμών κάποιων στηλών.
    - Κατά την κατηγοριοποίηση σε φάσεις της μέρας επιστρέφουμε tuples της μορφής (<phase>, 1). Αυτό το κάνουμε γιατί αμέσως μετά καλούμε την `reduceByKey()` για να αθροίσει τους άσσους. Έτσι υλοποιούμε την μέτρηση των εγκλημάτων κάθε φάσης.

Ο χρόνος εκτέλεσης με χρήση DataFrame API είναι περίπου 10-12 sec (μέσος όρος από πολλαπλές εκτελέσεις). Ο χρόνος εκτέλεσης με χρήση RDD API είναι περίπου 14-16 sec (επίσης μέσος όρος από πολλαπλές εκτελέσεις). Η διαφορά στους χρόνους εκτέλεσης προκύπτει από το γεγονός ότι τα queries με χρήση RDD API δεν γίνονται optimized, αντίθετα με τα queries με χρήση DataFrame API.

Ζητούμενο 4 – Κώδικας

```python
# query_2/df_api.py
crime_data = (
    load_crime_data_from_2010_to_present(spark)
    .filter(f.col("Premis Desc") == "STREET")
    .filter(f.col("TIME OCC").isNotNull())
    .withColumn(
        "Phase of Day",
        f.when(f.col("TIME OCC").cast(IntegerType())
            .between(500, 1159), "Morning")
        .when(f.col("TIME OCC").cast(IntegerType())
            .between(1200, 1659), "Afternoon")
        .when(f.col("TIME OCC").cast(IntegerType())
            .between(1700, 2059), "Evening")
        .otherwise("Night"),
    )
    .groupBy(f.col("Phase of Day"))
    .count()
    .withColumnRenamed("count", "Total Crimes")
    .orderBy(f.col("Total Crimes").desc())
)
```

```python
# query_2/rdd_api.py
cols = {"TIME OCC": 3, "Premis Desc": 15}
crime_data = (
    load_crime_data_from_2010_to_present_as_rdd(spark)
    .map(lambda row: next(
        csv.reader(io.StringIO(row), dialect="unix")))
    .filter(
        lambda row: (row[cols["Premis Desc"]].strip() == "STREET")
        and (row[cols["TIME OCC"]].strip() != "")
    )
    .map(lambda row: (
        "Morning"
        if 500 <= int(row[cols["TIME OCC"]]) <= 1159
        else ("Afternoon"
            if 1200 <= int(row[cols["TIME OCC"]]) <= 1659
            else ("Evening"
                if 1700 <= int(row[cols["TIME OCC"]]) <= 2059
                else "Night"
            )),
        1,
    ))
    .reduceByKey(operator.add)
    .sortBy(lambda tup: tup[1], ascending=False)
)
```

Ζητούμενο 4 – Έξοδος

```
Q2: Query results:
+------------+------------+
|Phase of Day|Total Crimes|
+------------+------------+
|Night       |237692      |
|Evening     |187172      |
|Afternoon   |147657      |
|Morning     |123371      |
+------------+------------+

Q2: Query results:
('Night', 237692)
('Evening', 187172)
('Afternoon', 147657)
('Morning', 123371)
```

Ζητούμενο 5

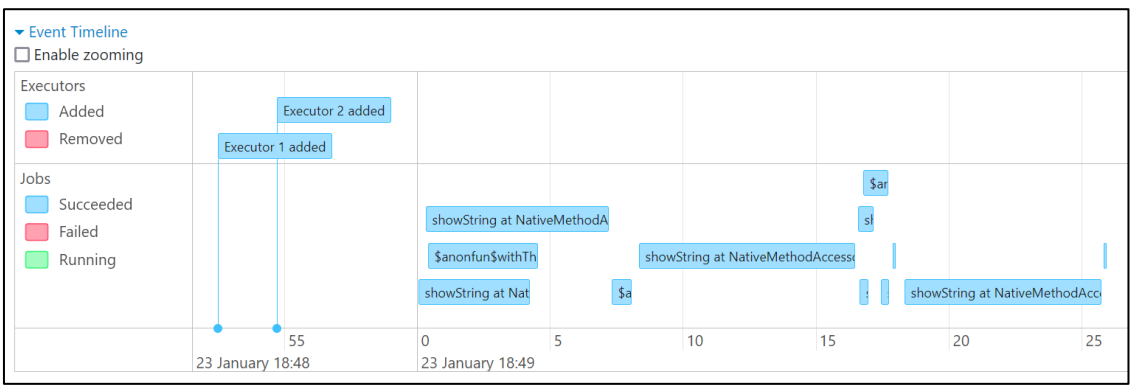
Υλοποιούμε το Query 3 χρησιμοποιώντας το DataFrame API. Το εκτελούμε με τις εντολές:

- `spark-submit --num-executors 2 query_3/df_api.py`
- `spark-submit --num-executors 3 query_3/df_api.py`
- `spark-submit --num-executors 4 query_3/df_api.py`

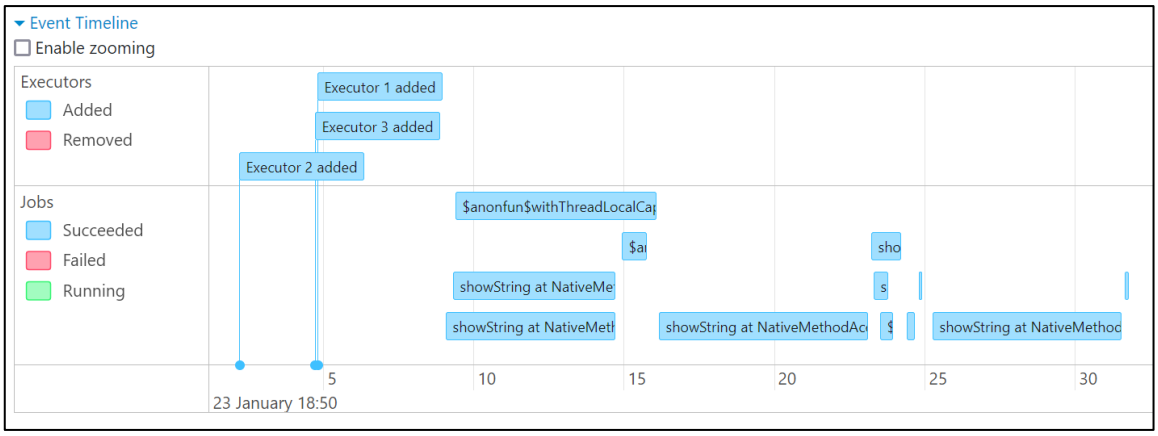
Ο κώδικας μας λειτουργεί ως εξής:

- Αρχικά φιλτράρουμε το crime dataset και κρατάμε μόνο τα εγκλήματα που έγιναν το 2015 και για τα οποία έχουμε πληροφορία για το θύμα τους.
- Εκτελούμε inner join του crime dataset με το revgeocoding ώστε να δημιουργηθεί η στήλη 'Zip Code'.
- Φορτώνουμε το income dataset και εκτελούμε inner join με τα (distinct) zip codes που προκύπτουν από το προηγούμενο join. Έτσι, κρατάμε δεδομένα εισοδήματος μόνο για τα zip codes για τα οποία έχουμε δεδομένα εγκλημάτων.
- Ταξινομούμε το income dataset κατά αύξον εισόδημα και κρατάμε τα πρώτα 3 rows. Κάνουμε το ίδιο αλλά κατά φθίνον εισόδημα και ενώνουμε τα αποτελέσματα. Έχουμε έτσι τα zip codes για τις 3 περιοχές με το υψηλότερο εισόδημα και τις 3 περιοχές με το χαμηλότερο εισόδημα.
- Τέλος, εκτελούμε inner join του crime dataset με τα 6 zip codes ώστε να κρατήσουμε μόνο εγκλήματα που εγίναν σε αυτές τις 6 περιοχές. Ομαδοποιούμε το αποτέλεσμα κατά καταγωγή θύματος ('Vict Descent') και μετράμε πόσα rows αντιστοιχούν σε κάθε καταγωγή. Αντικαθιστούμε τους κωδικούς καταγωγής με την αντίστοιχη καταγωγή (όπως περιγράφονται στην ιστοσελίδα του dataset) για να είναι πιο κατανοητό το αποτέλεσμα.

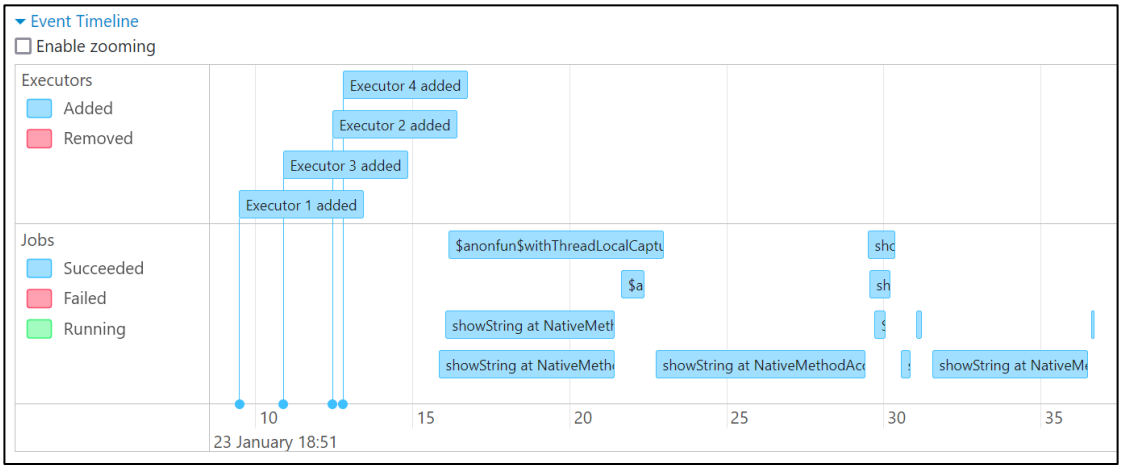
Παρατηρούμε ότι όσο αυξάνουμε τους executors, τόσο μειώνεται ο χρόνος εκτέλεσης (~27 sec για 2, ~25 sec για 3, ~23 sec για 4), αλλά η μείωση είναι σχετικά μικρή. Αυτό συμβαίνει λόγω του περιορισμένου παραλληλισμού που υπάρχει στο συγκεκριμένο query, καθώς και στις τρεις περιπτώσεις παρατηρώντας το job timeline βλέπουμε μικρό job-level parallelism (βλ. εικόνες 1, 2, 3). Φυσικά υπάρχει και παραλληλισμός εντός κάθε job (τα tasks του job), αλλά δεν έχει τόσο μεγάλη επιρροή στον χρόνο εκτέλεσης. Η μείωση του χρόνου με την αύξηση των executors οφείλεται κυρίως σε task-level parallelism, γι αυτό και είναι σχετικά μικρή.



1. Jobs timeline για 2 executors.



2. Jobs timeline για 3 executors.



3. Jobs timeline για 4 executors.

Ζητούμενο 5 – Κώδικας

```
rev_geocoding = (
    load_revgecoding(spark)
    .withColumnRenamed("ZIPcode", "Zip Code")
    .dropDuplicates(["LAT", "LON"]))

crime_data = (
    load_crime_data_from_2010_to_present(spark)
    .filter(f.year(f.col("DATE OCC")) == 2015)
    .filter(f.col("Vict Descent").isNotNull()))

crime_data = crime_data.join(rev_geocoding, ["LAT", "LON"])

la_income = (
    load_la_income_2015(spark)
    .withColumn(
        "Estimated Median Income",
        # income is given in '$56,000' format
        f.regexp_replace(
            f.col("Estimated Median Income"), "[\$\,]", "")
        .cast(IntegerType()
    ))
    .select(f.col("Zip Code"), f.col("Estimated Median Income")))

zip_codes_with_data = crime_data.select(f.col("Zip Code")).distinct()

la_income = la_income.join(
    zip_codes_with_data, "Zip Code"
).persist()

zip_codes = (
    la_income.orderBy(f.col("Estimated Median Income").desc())
    .limit(3)
    .union(
        la_income.orderBy(
            f.col("Estimated Median Income").asc()).limit(3))
    .select(f.col("Zip Code"))
    .distinct())

crime_data = (
    crime_data.join(zip_codes, "Zip Code")
    .groupBy(f.col("Vict Descent"))
    .count()
    .withColumnRenamed("count", "#")
    .orderBy(f.col("#").desc())
    # replace descent codes with descent names
    .replace(descent_dict, subset=["Vict Descent"])
)
```

Ζητούμενο 5 – Έξοδος

```
Q3: Query results:
+------------------------------+----+
|Vict Descent                  |#   |
+------------------------------+----+
|Hispanic/Latin/Mexican        |1583|
|Black                         |1108|
|White                         |1041|
|Other                         |499 |
|Other Asian                   |119 |
|Unknown                       |91  |
|Korean                        |9   |
|Japanese                      |3   |
|American Indian/Alaskan Native|3   |
|Chinese                       |2   |
|Filipino                      |1   |
+------------------------------+----+
```

Ζητούμενο 6

Υλοποιούμε το Query 4 χρησιμοποιώντας το DataFrame API. Το εκτελούμε με την εντολή:

- `spark-submit --num-executors 4 query_4/df_api.py`

Ο κώδικας μας λειτουργεί ως εξής:

- Για το ερώτημα 1: Φιλτράρουμε το crime dataset για να κρατήσουμε μόνο εγκλήματα για τα οποία γνωρίζουμε την τοποθεσία στην οποία έγιναν και τα οποία έγιναν με χρήση πυροβόλου όπλου. Δημιουργούμε επίσης τη στήλη 'Year' από την στήλη 'Date Rptd' (έτος κατά το οποίο γνωστοποιήθηκε το έγκλημα στην αστυνομία).
- Εκτελούμε inner join του crime dataset με το LAPD police stations dataset ώστε να αποκτήσουμε τις συντεταγμένες και το όνομα ('Division') του αστυνομικού τμήματος που ανέλαβε κάθε έγκλημα και δημιουργούμε τη στήλη 'Distance to Station' χρησιμοποιώντας τη συνάρτηση `geopy.distance.geodesic()`.
    - Για το υποερώτημα α: Ομαδοποιούμε τα εγκλήματα ανα έτος και υπολογίζουμε τον μέσο όρο των 'Distance to Station' για κάθε έτος, καθώς και τον αριθμό των εγκλημάτων ανα έτος. Τέλος, ταξινομούμε κατά αύξον έτος.
    - Για το υποερώτημα β: Ομαδοποιούμε τα εγκλήματα ανα 'Division' και υπολογίζουμε τον μέσο όρο των 'Distance to Station' καθώς και τον αριθμό των εγκλημάτων για κάθε 'Division'. Τέλος, ταξινομούμε κατά φθίνοντα αριθμο εγκλημάτων.
- Για το ερώτημα 2: Φιλτράρουμε το crime dataset για να κρατήσουμε μόνο εγκλήματα για τα οποία γνωρίζουμε την τοποθεσία στην οποία έγιναν και τα οποία έγιναν με χρήση πυροβόλου όπλου. Δημιουργούμε επίσης τη στήλη 'Year' από την στήλη 'Date Rptd' (έτος κατά το οποίο γνωστοποιήθηκε το έγκλημα στην αστυνομία).
- Εκτελούμε cross join του crime dataset με το LAPD police stations dataset ώστε να αποκτήσουμε κάθε συνδυασμό εγκλήματος – συντεταγμένων αστυνομικού τμήματος. Δημιουργούμε τη στήλη 'Distance to Station' χρησιμοποιώντας τη συνάρτηση `geopy.distance.geodesic()`.
- Έστω (1) το dataset που προκύπτει μετά από το προηγούμενο βήμα και (2) το dataset που προκύπτει από το (1) αφού ομαδοποιήσουμε τα εγκλήματα ανα 'DR_NO' (division recorn number) και υπολογίσουμε το ελάχιστο 'Distance to Station' για κάθε 'DR_NO'. Εκτελούμε inner join των (1), (2) ώστε στο (1) να μείνει για κάθε έγκλημα μόνο το row που περιέχει την ελάχιστη απόσταση από κάποιο αστυνομικό τμήμα ('Distance to Closest Station').
    - Για το υποερώτημα α: Ομαδοποιούμε τα εγκλήματα ανα έτος και υπολογίζουμε το μέσο όρο των 'Distance to Closest Station' για κάθε έτος, καθώς και τον αριθμό των εγκλημάτων ανα έτος. Τέλος, ταξινομούμε κατά αύξον έτος.
    - Για το υποερώτημα β: Ομαδοποιούμε τα εγκλήματα ανα 'Division' και υπολογίζουμε τον μέσο όρο των 'Distance to Closest Station' καθώς και τον αριθμό των εγκλημάτων ανα 'Division'. Τέλος, ταξινομούμε κατά φθίνοντα αριθμο εγκλημάτων.

Ζητούμενο 6 – Κώδικας

```
# declare a user-defined function
geodesic_udf = f.udf(
    f=lambda lat1, lon1, lat2, lon2: geopy.distance.geodesic(
        (lat1, lon1), (lat2, lon2)
    ).km,
    returnType=DoubleType())

lapd_police_stations = (
    load_lapd_police_stations(spark)
    .withColumnRenamed("X", "Station LON")
    .withColumnRenamed("Y", "Station LAT")
    .withColumnRenamed("DIVISION", "Division")
    .select(
        f.col("Station LON"), f.col("Station LAT"),
        f.col("Division"), f.col("PREC")))

crime_data_1 = (
    load_crime_data_from_2010_to_present(spark)
    .filter(f.col("LAT") != 0.0)
    .filter(f.col("LON") != 0.0)
    .filter(f.col("Weapon Used Cd").between(100, 199))
    .withColumn("Year", f.year(f.col("Date Rptd")))
    .withColumnRenamed("AREA", "PREC"))

crime_data_1 = (
    crime_data_1.join(lapd_police_stations, "PREC")
    .withColumn(
        "Distance to Station",
        geodesic_udf(
            f.col("LAT"), f.col("LON"),
            f.col("Station LAT"), f.col("Station LON")))
    .select(
        f.col("Year"),
        f.col("Division"),
        f.col("Distance to Station"))
    .persist())

crime_data_1a = (
    crime_data_1.groupBy(f.col("Year"))
    .agg(
        f.avg(f.col("Distance to Station"))
        .alias("Average Distance to Station"),
        f.count(f.expr("*")).alias("#"))
    .orderBy(f.col("Year").asc()))
```

```
crime_data_1b = (
    crime_data_1.groupBy(f.col("Division"))
    .agg(
        f.avg(f.col("Distance to Station"))
        .alias("Average Distance to Station"),
        f.count(f.expr("*")).alias("#"))
    .orderBy(f.col("#").desc()))

crime_data_2 = (
    load_crime_data_from_2010_to_present(spark)
    .filter(f.col("LAT") != 0.0)
    .filter(f.col("LON") != 0.0)
    .filter(f.col("Weapon Used Cd").between(100, 199))
    .withColumn("Year", f.year(f.col("Date Rptd")))
    .withColumnRenamed("AREA", "PREC"))

crime_data_2 = (
    crime_data_2.crossJoin(lapd_police_stations)
    .withColumn("Distance to Station",
        geodesic_udf(
            f.col("LAT"), .col("LON"),
            f.col("Station LAT"), f.col("Station LON")))

crime_data_2_tmp = crime_data_2.persist()

min_distances = crime_data_2.groupBy(f.col("DR_NO")).agg(
    f.min(f.col("Distance to Station")).alias("Distance to Station"))

crime_data_2 = (
    crime_data_2.join(
        min_distances,
        ["DR_NO", "Distance to Station"])
    .withColumnRenamed(
        "Distance to Station",
        "Distance to Closest Station")
    .select(
        f.col("Year"),
        f.col("Division"),
        f.col("Distance to Closest Station"),)
    .persist())

crime_data_2a = (
    crime_data_2.groupBy(f.col("Year"))
    .agg(
        f.avg(f.col("Distance to Closest Station"))
        .alias("Average Distance to Closest Station"),
        f.count(f.expr("*")).alias("#"))
    .orderBy(f.col("Year").asc()))
```

```
crime_data_2b = (
    crime_data_2.groupBy(f.col("Division"))
    .agg(
        f.avg(f.col("Distance to Closest Station"))
        .alias("Average Distance to Closest Station"),
        f.count(f.expr("*")).alias("#"))
    .orderBy(f.col("#").desc()))
```

Ζητούμενο 6 – Έξοδος

```
Q4: Query results (1a, broadcast):
+----+--------------------------+-----+
|Year|Average Distance to Station|#    |
+----+--------------------------+-----+
|2010|2.783696047269014         |8161 |
|2011|2.790426955613486         |7225 |
|2012|2.8343652755969226        |6521 |
|2013|2.830277105758436         |5851 |
|2014|2.733088559711193         |5929 |
|2015|2.703595992355386         |6766 |
|2016|2.717656509409694         |8094 |
|2017|2.7212269533164672        |7783 |
|2018|2.735764748851274         |7415 |
|2019|2.7408624585165544        |7136 |
|2020|2.687914973224239         |8493 |
|2021|2.6375931082053317        |9748 |
|2022|2.610219041558848         |10031|
|2023|2.54960880529589          |9482 |
|2024|2.4430265681116463        |354  |
+----+--------------------------+-----+

Q4: Query results (1b, broadcast):
+----------------+--------------------------+-----+
|Division        |Average Distance to Station|#    |
+----------------+--------------------------+-----+
|77TH STREET     |2.6927547958919194        |16700|
|SOUTHEAST       |2.105002896406666         |12634|
|NEWTON          |2.016270068670076         |9681 |
|SOUTHWEST       |2.700615280714631         |8709 |
|HOLLENBECK      |2.65092382544742          |6137 |
|HARBOR          |4.080449607048249         |5497 |
|RAMPART         |1.5766882736582195        |5051 |
|MISSION         |4.705282439681721         |4400 |
|OLYMPIC         |1.8227182453541246        |4337 |
|NORTHEAST       |3.905963751966555         |3875 |
|FOOTHILL        |3.800913612906989         |3699 |
|HOLLYWOOD       |1.4577596183463502        |3586 |
|CENTRAL         |1.138532636288764         |3547 |
|WILSHIRE        |2.3168914786079724        |3458 |
|NORTH HOLLYWOOD |2.7179195104737284        |3372 |
|WEST VALLEY     |3.521784264453557         |2815 |
|VAN NUYS        |2.2168453015319813        |2664 |
|PACIFIC         |3.7369631284284877        |2662 |
|DEVONSHIRE      |4.010477513822523         |2410 |
|TOPANGA         |3.485203169857805         |2236 |
|WEST LOS ANGELES|4.255995450544175         |1519 |
+----------------+--------------------------+-----+
```

```
Q4: Query results (2a, broadcast):
+----+-----------------------------------+-----+
|Year|Average Distance to Closest Station|#    |
+----+-----------------------------------+-----+
|2010|2.4349477369509684                 |8161 |
|2011|2.458267744224069                  |7225 |
|2012|2.5046546495176374                 |6521 |
|2013|2.4592839268144955                 |5851 |
|2014|2.3809074428274775                 |5929 |
|2015|2.386665105327522                  |6766 |
|2016|2.42585116309508                   |8094 |
|2017|2.3902494493339974                 |7783 |
|2018|2.411711276721766                  |7415 |
|2019|2.4303510725294513                 |7136 |
|2020|2.3817739655085055                 |8493 |
|2021|2.351856827983618                  |9748 |
|2022|2.3140475376797567                 |10031|
|2023|2.268370137452765                  |9482 |
|2024|2.1491827819887894                 |354  |
+----+-----------------------------------+-----+

Q4: Query results (2b, broadcast):
+----------------+-----------------------------------+-----+
|Division        |Average Distance to Closest Station|#    |
+----------------+-----------------------------------+-----+
|77TH STREET     |1.7190788618963713                 |13562|
|SOUTHEAST       |2.2030893184569993                 |11578|
|SOUTHWEST       |2.277649926893795                  |11273|
|NEWTON          |1.56940614427311                   |7197 |
|WILSHIRE        |2.4438709710669313                 |6303 |
|HOLLENBECK      |2.63640139045426                   |6199 |
|OLYMPIC         |1.659564311851239                  |5448 |
|HOLLYWOOD       |2.0076579991121846                 |5401 |
|HARBOR          |3.8941929826940496                 |5373 |
|RAMPART         |1.3982366199537841                 |4762 |
|VAN NUYS        |2.962885260188082                  |4660 |
|FOOTHILL        |3.597132157470032                  |4467 |
|CENTRAL         |1.0212104014528278                 |3639 |
|NORTH HOLLYWOOD |2.7301871915462854                 |3347 |
|NORTHEAST       |3.7579984418749395                 |3120 |
|MISSION         |3.8027845458999865                 |2795 |
|WEST VALLEY     |2.783805681041158                  |2769 |
|PACIFIC         |3.7058362403270797                 |2538 |
|TOPANGA         |3.0433842463875433                 |2298 |
|DEVONSHIRE      |2.9852680008189942                 |1238 |
|WEST LOS ANGELES|2.7660855706718697                 |1022 |
+----------------+-----------------------------------+-----+
```

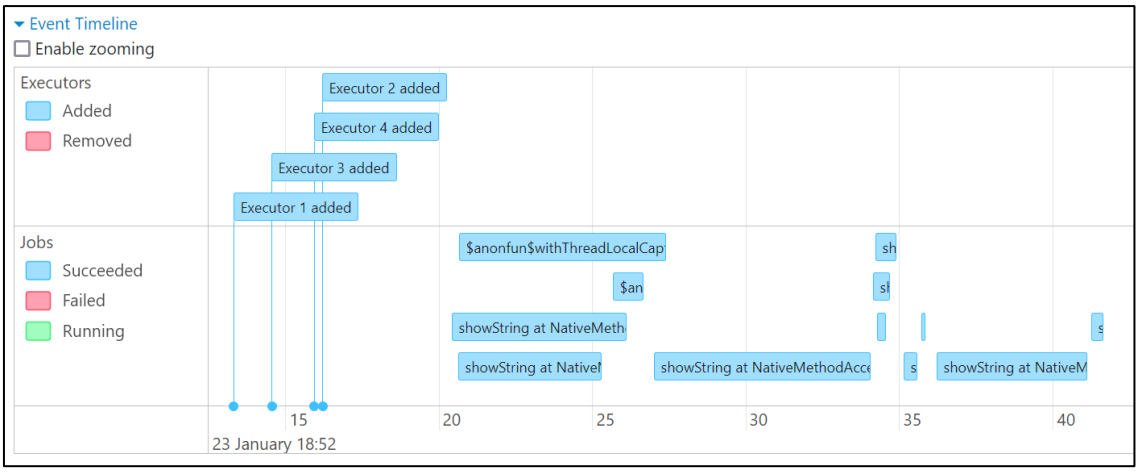
Ζητούμενο 7

Εκτελούμε τα Queries 3 και 4 με τις παρακάτω εντολές:

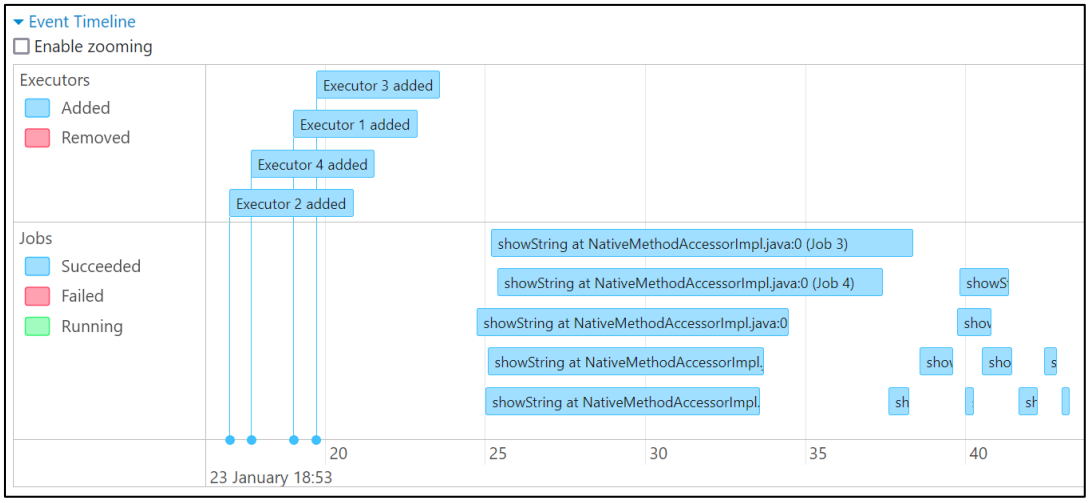
- `spark-submit --num-executors 4 query_3/df_api.py \ --explain --join-strategy broadcast`
- `spark-submit --num-executors 4 query_3/df_api.py \ --explain --join-strategy merge`
- `spark-submit --num-executors 4 query_3/df_api.py \ --explain --join-strategy shuffle_hash`
- `spark-submit --num-executors 4 query_3/df_api.py \ --explain --join-strategy shuffle_replicate_nl`
- `spark-submit --num-executors 4 query_4/df_api.py \ --explain --join-strategy broadcast`
- `spark-submit --num-executors 4 query_4/df_api.py \ --explain --join-strategy merge`
- `spark-submit --num-executors 4 query_4/df_api.py \ --explain --join-strategy shuffle_hash`
- `spark-submit --num-executors 4 query_4/df_api.py \ --explain --join-strategy shuffle_replicate_nl`

Για το Query 3:

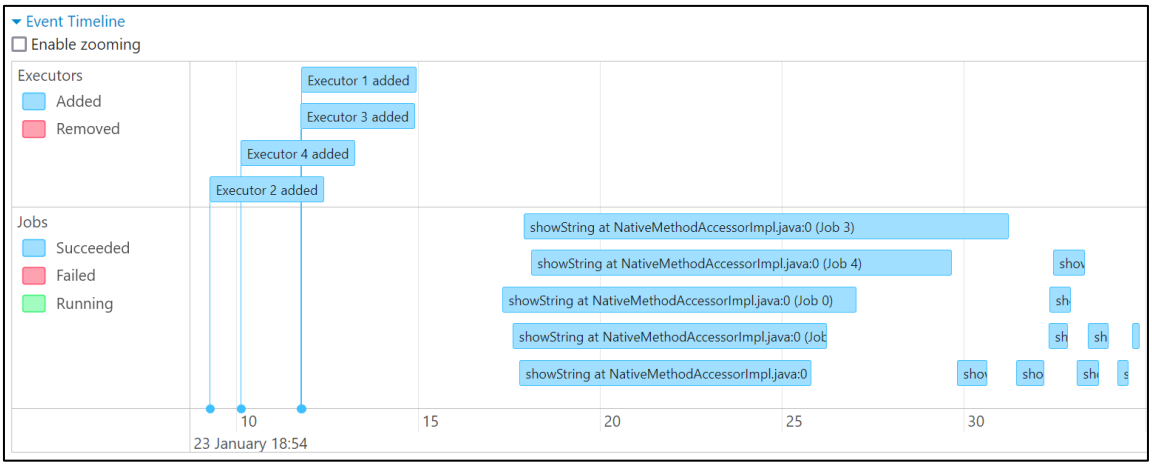
- Στο Query 3 εκτελούνται 3 joins. Σε κάθε ένα από αυτά το ένα από τα δύο DataFrames που χρησιμοποιείται έχει μέγεθος μικρότερο του 1 MiB. Γι' αυτό το λόγο περιμένουμε το **broadcast** join να έχει την καλύτερη απόδοση.
- Στην πράξη, όμως, παρατηρούμε ότι οι στρατηγικές **merge**, **shuffle_hash** έχουν καλύτερη απόδοση (κατά περίπου 2 και 3 sec αντίστοιχα). Αυτό συμβαίνει γιατί όταν επιλέγουμε **merge** ή **shuffle_hash** το τελικό execution plan αξιοποιεί περισσότερο τους διαθέσιμους executors και παραλληλοποιείται (job-level) καλύτερα η εκτέλεση (βλ. εικόνες 4, 5, 6). Με στρατηγική **shuffle_hash** η εκτέλεση είναι λίγο ταχύτερη από ότι με **merge**, πιθανώς λόγω της αποφυγής του sorting πριν από κάθε join, αλλά η διαφορά είναι μικρή.
- Επιλέγοντας τη στρατηγική **shuffle_replicate_nl**, η εκτέλεση του query δεν τελείωνει ποτέ καθώς προκύπτει exception από το JVM: «GC Overhead Limit Exceeded Error». Η συγκεκριμένη στρατηγική πραγματοποιεί cross join αφού πρώτα μεταφερθούν όλα τα δεδομένα σε όλους τους executors (κάθε executor αποκτά στη μνήμη του όλα τα partitions). Αυτό απαιτεί περισσότερη μνήμη από όση έχουμε διαθέσιμη.



4. Job timeline του Query 3 με broadcast join strategy (4 executors).



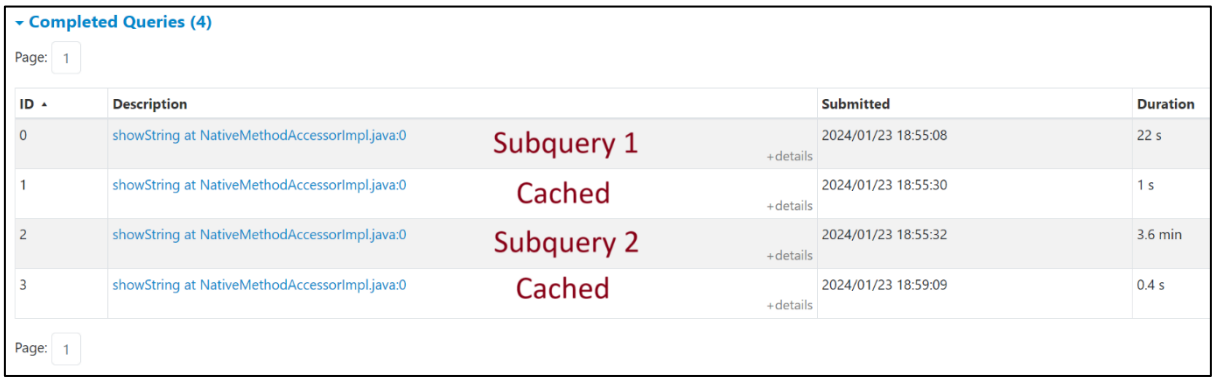
5. Job timeline του Query 3 με merge join strategy (4 executors).



6. Job timeline του Query 3 με shuffle_hash join strategy (3 executors).

Για το Query 4:

- Στο πρώτο subquery του Query 4 εκτελείται ένα join, το ένα από τα δύο DataFrames του οποίου είναι σχετικά μικρό σε μέγεθος, άρα περιμένουμε να ωφεληθεί από τη στρατηγική **broadcast**. Αυτό ακριβώς παρατηρούμε να γίνεται και στην πράξη, καθώς η περίπτωση **broadcast** εκτελείται σχεδόν 50% ταχύτερα από τις περιπτώσεις **merge**, **shuffle_hash**. Κοιτώντας το execution plan όλων των περιπτώσεων επιβεβαιώνουμε πως η μόνη διαφορά ανάμεσα στα τρια plans είναι το join.
- Στο δεύτερο subquery εκτελούνται δύο joins, ένα εκ των οποίων είναι cross join. Το cross join εκτελείται πάντα με **broadcast** (στο output του query λαμβάνουμε και αντίστοιχο warning όταν δίνουμε hints για **merge**, **shuffle_hash**). Το τελευταίο join αναμένουμε να εκτελεστεί γρηγορότερα με τα **merge** ή **shuffle_hash**, αλλά ο χρόνος που απαιτεί σε σχέση με το cross join είναι πολύ μικρός, οπότε η διαφορά χρόνου των τριών στρατηγικών είναι πολύ μικρή (ως ποσοστό του συνολικού χρόνου εκτέλεσης). Κοιτώντας το query plan, πάλι η μόνη διαφορά ανάμεσα στα τρια plans είναι το δεύτερο join.
- Η μέθοδος **shuffle_hash_nl** πάλι δεν τερμάτισε, για τον ίδιο λόγο με το Query 3.



7. Τα subqueries του Query 4 (broadcast).

8. Τα subqueries του Query 4 (shuffle_hash).
Παρατηρούμε ότι το subquery 1 εκτελείται πολύ γρηγορότερα σε αυτή την περίπτωση.

Όλα τα execution plans (κείμενο και γραφικά) υπάρχουν στο repository μας, καθώς ήταν πολύ μεγάλα (σε διαστάσεις) για να χωρέσουν εδώ και να είναι ευανάγνωστα.