Προχωρημένα Θέματα Βάσεων Δεδομένων
Εργαστηριακή Αναφορά

Ευθύμιος Καραγιάννης (03119434)

Χειμερίνο 2023-2024

Generate Dataframe

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf, col
from pyspark.sql.types import IntegerType, DoubleType, DateType
from datetime import date

APP_NAME = "Generate_DataFrame"
HDFS_DATA_DIR = "hdfs://okeanos-master:54310/data"

spark = SparkSession.builder.appName(APP_NAME).getOrCreate()


def convert_date_format(input_date: str):
    date_part, _, _ = input_date.split(" ")
    month, day, year = date_part.split("/")
    return date(int(year), int(month), int(day))


convert_date_udf = udf(convert_date_format, DateType())

csv_file_path1 = f"{HDFS_DATA_DIR}/Crime_Data_from_2010_to_2019.csv"
csv_file_path2 = f"{HDFS_DATA_DIR}/Crime_Data_from_2020_to_Present.csv"

df1 = spark.read.csv(csv_file_path1, header=True)
df2 = spark.read.csv(csv_file_path2, header=True)

df = df1.union(df2)


df = (
    df.withColumn("Date␣Rptd", convert_date_udf(col("Date␣Rptd")))
    .withColumn("DATE_OCC", convert_date_udf(col("DATE_OCC")))
    .withColumn("Vict␣Age", df["Vict␣Age"].cast(IntegerType()))
    .withColumn("LAT", df["LAT"].cast(DoubleType()))
    .withColumn("LON", df["LON"].cast(DoubleType()))
)

output_csv_path = f"{HDFS_DATA_DIR}/Full_Data"
df.write.option("header", True).mode("overwrite").csv(output_csv_path)
```

Query 1

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import month, year, row_number, desc
from pyspark.sql.window import Window
import time

APP_NAME = "DF_Query_1"
HDFS_DATA_DIR = "hdfs://okeanos-master:54310/data"
HDFS_OUTPUT_DIR = "hdfs://okeanos-master:54310/results"

spark = SparkSession.builder.appName(APP_NAME).getOrCreate()

csv_file_path = f"{HDFS_DATA_DIR}/Full_Data"

df = spark.read.csv(csv_file_path, header=True)

start_time = time.time()

window = Window.partitionBy("year").orderBy(desc("crime_total"))
results_df = (
    df.withColumn("year", year("Date␣Rptd"))
    .withColumn("month", month("Date␣Rptd"))
    .groupBy("year", "month")
    .count()
    .withColumnRenamed("count", "crime_total")
    .withColumn(
        "rank",
        row_number().over(window),
    )
    .filter("rank␣<=␣3")
```

```python
    .select("year", "month", "crime_total", "rank")
)

results_df.show(results_df.count(), truncate=False)
exec_time = time.time() - start_time
print(f"\n\nExec time: {exec_time} sec")

spark.stop()
```

```python
from pyspark.sql import SparkSession
from time import time

APP_NAME = "SQL_Query_1"
HDFS_DATA_DIR = "hdfs://okeanos-master:54310/data"

spark = SparkSession.builder.appName(APP_NAME).getOrCreate()

csv_file_path = f"{HDFS_DATA_DIR}/Full_Data"
df = spark.read.csv(csv_file_path, header=True)

start_time = time()

df.createOrReplaceTempView("tmp_view")
sql_query = """
    SELECT year, month, crime_total, rank
    FROM (
        SELECT
            year(`Date Rptd`) AS year,
            month(`Date Rptd`) AS month,
            COUNT(*) AS crime_total,
            ROW_NUMBER() OVER (PARTITION BY year(`Date Rptd`) ORDER BY
                COUNT(*) DESC) AS rank
        FROM
            tmp_view
        GROUP BY
            year(`Date Rptd`), month(`Date Rptd`)
    ) tmp
    WHERE rank <= 3
"""
results_df = spark.sql(sql_query)

results_df.show(results_df.count(), truncate=False)
exec_time = time() - start_time
print(f"\n\nExec time: {exec_time} sec\n\n")

```

```
spark.stop()
```

Εκτελούμε τα παραπάνω queries στον master ή στον worker node με τις παρακάτω εντολές:
spark-submit –num-executors 4 ./src/query_1.py
spark-submit –num-executors 4 ./src/sql_query_1.py
Παρακάτω παρουσιάζονται τα αποτελέσματα των παραπάνω queries καθώς και χρόνοι εκτέλεσης των SQL και DataFrame API. Όσον αφορά τους χρόνους εκτέλεσης, και τα δύο APIs εμφανίζουν σχεδόν τα ίδια αποτελέσματα, αφού και τα δύο χρησιμοποιούν το Spark SQL engine και τον Catalyst optimizer. Οι παρακάτω χρόνοι αποτελούν τον μέσο χρόνο 5 εκτελέσεων.

Table 1: Query 1 results

Year	**Month**	**Crime Total**	**Rank**
2010	3	12828	1
2010	7	12085	2
2010	4	12070	3
2011	8	20643	1
2011	5	20643	2
2011	10	20533	3
2012	10	30963	1
2012	8	30646	2
2012	5	29842	3
2013	1	11429	1
2013	3	8229	2
2013	8	8056	3
2014	1	6297	1
2014	6	5712	2
2014	5	5694	3
2015	7	10106	1
2015	5	10102	2
2015	3	10065	3
2016	10	16405	1
2016	12	15796	2
2016	8	15761	3
2017	3	27409	1
2017	8	27216	2
2017	7	26858	3
2018	1	8357	1
2018	2	6167	2
2018	8	6135	3
2019	7	19127	1
2019	8	18874	2
2019	5	18538	3

2020	1	7130	1
2020	2	5818	2
2020	7	5210	3
2021	7	28426	1
2021	10	27955	2
2021	8	27690	3
2022	8	32162	1
2022	7	31372	2
2022	10	30936	3
2023	8	26308	1
2023	10	26249	2
2023	7	25974	3
2024	1	1	1

Table 2: Query 1 times

API	**Time (sec)**
SQL	28.55
DataFrame	32.04

Query 2

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, desc
from time import time

APP_NAME = "DF_Query_2"
HDFS_DATA_DIR = "hdfs://okeanos-master:54310/data"

spark = SparkSession.builder.appName(APP_NAME).getOrCreate()

csv_file_path = f"{HDFS_DATA_DIR}/Full_Data"

df = spark.read.csv(csv_file_path, header=True)

start_time = time()

results_df = (
    df.filter(col("Premis_Desc") == "STREET")
    .withColumn(
        "day_split",
        when((col("TIME_OCC") >= "0500") & (col("TIME_OCC") < "1200"),
            "morning")
        .when((col("TIME_OCC") >= "1200") & (col("TIME_OCC") <
            "1700"), "afternoon")
        .when((col("TIME_OCC") >= "1700") & (col("TIME_OCC") <
            "2100"), "evening")
        .otherwise("night"),
    )
    .groupBy("day_split")
    .count()
    .withColumnRenamed("count", "crime_total")
    .orderBy(desc("crime_total"))
)

results_df.show(results_df.count(), truncate=False)
exec_time = time() - start_time
print(f"\n\nExec time: {exec_time} sec\n\n")

spark.stop()
```

```
from pyspark import SparkContext
from pyspark.sql import SparkSession
from time import time

APP_NAME = "RDD_Query_2"
HDFS_DATA_DIR = "hdfs://okeanos-master:54310/data"

sc = SparkContext(appName=APP_NAME)
spark = SparkSession(sc)

csv_file_path = f"{HDFS_DATA_DIR}/Full_Data"
init_rdd = spark.read.csv(csv_file_path, header=True).rdd


def map_time_to_day_split(row):
    time_occ = row[3]
    if "0500" <= time_occ < "1200":
        return "morning", 1
    elif "1200" <= time_occ < "1700":
        return "afternoon", 1
    elif "1700" <= time_occ < "2100":
        return "evening", 1
    else:
        return "night", 1


start_time = time()

res_rdd = (
    init_rdd.filter(lambda row: row[15] == "STREET")
    .map(map_time_to_day_split)
    .reduceByKey(lambda x, y: x + y)
    .sortBy(lambda x: x[1], ascending=False)
)

exec_time = time() - start_time
print(f"\n\nExec time: {exec_time} sec\n\n")

results = res_rdd.collect()
for row in results:
    print(row[0], row[1])

sc.stop()
```

Υποβάλλουμε τις παραπάνω εργασίες στο spark με τις παρακάτω εντολές:
spark-submit –num-executors 4 ./src/query_2.py
spark-submit –num-executors 4 ./src/rdd_query_2.py
Στην συνέχεια παρουσιάζονται τα αποτελέσματα και οι χρόνοι εκτέλεσης των DataFrame και RDD API. Στην περίπτωση αυτή βλέπουμε τα δύο APIs να παρουσιάζουν αρκετά διαφορετικούς χρόνους εκτέλεσης, με το RDD να είναι σημαντικά πιο αργό. Αυτό συμβαίνει διότι τα RDDs αποτελούν περισσότερο μία περιγραφή της λύσης και δεν μπορούν να γίνουν optimized από το Spark, καθώς δεν είναι γνωστός ο τύπος των δεδομένων και οι πράξεις μεταξύ τους. Επιπλέον απελούν αντικείμενα στην μνήμη του JVM, με αποτελέσματα να εξαρτώνται σε μεγάλο βαθμό από αυτό και να προστίθεται επιπλέον overhead όταν ο όγκος των δεδομένων μεγαλώνει (Garbage Collection, Object Serialization).

Table 3: Query 2 results

Day Split	**Crime Total**
Night	223171
Evening	174170
Afternoon	139129
Morning	117035

Table 4: Query 2 times

API	Time (sec)
RDD	91.02
DataFrame	27.00

Query 3

```python
from pyspark.sql import SparkSession
from pyspark.sql.types import StringType
from pyspark.sql.functions import col, year, desc, udf, split,
    regexp_replace
from time import time

APP_NAME = "DF_Query_3"
HDFS_DATA_DIR = "hdfs://okeanos-master:54310/data"

spark = SparkSession.builder.appName(APP_NAME).getOrCreate()

basic_csv_path = f"{HDFS_DATA_DIR}/Full_Data"
revgecoding_csv_path = f"{HDFS_DATA_DIR}/revgecoding.csv"
income_csv_path = f"{HDFS_DATA_DIR}/LA_income_2015.csv"

basic_df = spark.read.csv(basic_csv_path, header=True)
revgecoding_df = spark.read.csv(revgecoding_csv_path, header=True)
income_df = spark.read.csv(income_csv_path, header=True)

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

mapping_udf = udf(lambda x: victim_descent_mapping.get(x, x),
   StringType())
results = []

start_time = time()

inner_join_df = (
    basic_df.filter((year(col("Date Rptd")) == 2015) & (col("Vict 
       Descent") != ""))
    .join(
        revgecoding_df,
        (basic_df["LAT"] == revgecoding_df["LAT"])
        & (basic_df["LON"] == revgecoding_df["LON"]),
    )
    .select("Vict Descent", "ZIPcode")
)

income_df = (
    income_df.withColumn(
        "Estimated Median Income", regexp_replace("Estimated Median 
           Income", "\\$", "")
    )
    .withColumn(
        "Estimated Median Income", regexp_replace("Estimated Median 
           Income", ",", "")
    )
    .withColumn(
        "Estimated Median Income", col("Estimated Median 
           Income").cast("integer")
    )
)

for asc in [True, False]:
    sorted_income_ZIP_codes_df = (
        income_df.sort("Estimated Median Income", ascending=asc)
        .limit(3)
        .select("Zip Code")
        .collect()
    )

    data = [row["Zip Code"] for row in sorted_income_ZIP_codes_df]

    results_df = (
        inner_join_df.filter(split(col("ZIPcode"),
            "-").getItem(0).isin(data))
        .withColumnRenamed("Vict Descent", "Victim Descent")
        .groupBy("Victim Descent")
        .count()
        .withColumnRenamed("count", "crime_total")
        .orderBy(desc("crime_total"))
        .withColumn("Victim Descent", mapping_udf(col("Victim 
            Descent")))
    )

    results.append(results_df)


for df in results:
    df.show(df.count(), truncate=False)
    print()
exec_time = time() - start_time
print(f'\n\nExec time: {exec_time} sec\n\n')

spark.stop()
```

Για το ερώτημα αυτό, τα queries για τις περιοχές με το υψηλότερο και χαμηλότερο οικεγενειακό εισόδημα εκτελέστηκαν χωριστά, καθώς με αυτό το τρόπο τα αποτελέσματα παρουσιάζουν περισσότερο ενδιαφέρον. Αυτήν την φορά υποβάλλουμε την εργασία, για διαφορετικό πλήθος spark-executors, με την παρακάτω εντολή:
spark-submit –num-executors n ./src/query_3.py όπου **n = 2, 3, 4**.
Αυξάνοντας τον αριθμό των spark-executors, αυξάνουμε τον αριθμό των tasks της υποβολής που μπορούν να εκτελεστούν παράλληλα. Επιπλέον, κάθε executor δεσμεύει πόρους μνήμης και CPU τους κόμβου, ώστε να αποθηκεύει τοπικά τα δεδομένα και συνεπώς να εκτελεί τις πράξεις γρηγορότερα. Επομένως, αύξηση του πλήθους τους μπορεί να οδηγήσει σε καλύτερες επιδόσεις. Ωστόσο υπάρχουν περιπτώσεις όπου τα δεδομένα δεν είναι κατανεμημένα ομοιόμορφα στους κόμβους και η αύξηση της παραλληλίας δεν οφελεί, ή τα tasks είναι πολύ μικρά και η δρομολόγηση τους σε executors προσθέτει σημαντικό overhead σε σχέση με την επεξεργασία των δεδομένων, με αποτελέσματα την αύξηση του χρόνου εκτέλεσης.

Table 5: Query 3 results (Table A)

Victim Descent	Crime Total
Hispanic/Latin/Mexican	144
White	116
Black	101
Unknown	39
Other	32
Other Asian	6
Chinese	1

Table 6: Query 3 results (Table B)

Victim Descent	Crime Total
Other	2
Other Asian	1
White	1
Hispanic/Latin/Mexican	1

Table 7: Query 3 times

Spark-executors	Time (sec)
2	48.46
3	43.95
4	53.98

Query 4

```
from pyspark.sql import SparkSession, Window
from pyspark.sql.types import FloatType
from pyspark.sql.functions import (
    col,
    length,
    udf,
    year,
    avg,
    count,
    desc,
    min as spark_min,
)
from geopy.distance import geodesic

APP_NAME = "DF_Query_4"
HDFS_DATA_DIR = "hdfs://okeanos-master:54310/data"

spark = SparkSession.builder.appName(APP_NAME).getOrCreate()

basic_csv_path = f"{HDFS_DATA_DIR}/Full_Data"
departmens_csv_path = f"{HDFS_DATA_DIR}/LAPD_Police_Stations.csv"

basic_df = spark.read.csv(basic_csv_path, header=True)
departments_df = spark.read.csv(departmens_csv_path, header=True)


def compute_distance(lat1, lon1, lat2, lon2):
    crime_location = (lat1, lon1)
    department_location = (lat2, lon2)
    return geodesic(crime_location, department_location).kilometers


compute_distance_udf = udf(compute_distance, FloatType())

results_df_list = []

filtered_df = basic_df.filter(
    col("Weapon_Used_Cd").startswith("1")
    & (length(col("Weapon_Used_Cd")) == 3)
    & ((col("LAT") > 0) & (col("LON") < 0))
)

filtered_df = filtered_df.withColumn("AREA",
   filtered_df["AREA"].cast("integer"))
departments_df = departments_df.withColumn(
    "PREC", departments_df["PREC"].cast("integer")
)

inner_join_df = filtered_df.join(departments_df, col("AREA") ==
   col("PREC")).withColumn(
    "distance",
    compute_distance_udf(
        filtered_df["LAT"],
        filtered_df["LON"],
        departments_df["Y"],
        departments_df["X"],
    ),
)

window_spec = Window.partitionBy("DR_NO").orderBy("distance")

cross_join_df = (
    filtered_df.crossJoin(
        departments_df.withColumnRenamed("LOCATION", "department location")
    )
    .withColumn(
        "distance",
        compute_distance_udf(
            filtered_df["LAT"],
            filtered_df["LON"],
            departments_df["Y"],
            departments_df["X"],
        ),
    )
    .withColumn("min_distance", spark_min("distance").over(window_spec))
    .filter(col("distance") == col("min_distance"))
    .drop("min_distance")
)

results_df_list = []

for join_df in [inner_join_df, cross_join_df]:
    results_df = (
        join_df.withColumn("year", year("Date Rptd"))
        .groupBy("year")
        .agg(
            avg("distance").alias("average distance"), count("*").alias("total crimes")
        )
        .select("year", "average distance", "total crimes")
        .orderBy("year")
    )
    results_df_list.append(results_df)

    results_df = (
        join_df.withColumnRenamed("DIVISION", "division")
        .groupBy("division")
        .agg(
            avg("distance").alias("average_distance"),
               count("*").alias("total_crimes")
        )
        .select("division", "average_distance", "total_crimes")
        .orderBy(desc("total_crimes"))
    )
    results_df_list.append(results_df)

for df in results_df_list:
    df.show(df.count(), truncate=False)
    print()

spark.stop()
```

Για να υποβάλλουμε την εργασία 4 στο spark χρείαζεται αρχικά να δημιουργήσουμε ένα virtual environment και να εγκαταστήσουμε τη βιβλιοθήκη geopy. Στη συνέχεια παράγουμε το αρχείο requirements.txt, το οποίο θα χρησημοποιηθεί για την δημιουγία του dependencies.zip.
python3 -m venv venv
source venv/bin/activate
pip3 install geopy
pip3 freeze > requirements.txt
pip3 install -t dependencies -r requirements.txt
zip dependencies.zip dependencies/*
Τέλος, υποβάλλουμε την εργασία στο spark με την εντολή:
spark-submit –py-files dependencies.zip ./src/query_4.py

Table 8: Query 4 (1a)

Year	Average Distance	Total Crimes
2010	2.667	5842
2011	2.835	8186
2012	2.834	11956
2013	2.747	2322
2014	2.681	2354
2015	2.655	3505
2016	2.681	6496
2017	2.762	9883
2018	2.577	2282
2019	2.741	7100
2020	2.489	2287
2021	2.652	14415
2022	2.627	15722
2023	2.624	11767

Table 9: Query 4 (1b)

Division	Average Distance	Total Crimes
77TH STREET	2.657	16372
SOUTHEAST	2.103	10100
NEWTON	2.018	9469
SOUTHWEST	2.694	8867
HOLLENBECK	2.674	6382
HARBOR	4.065	5709
RAMPART	1.577	4994
NORTHEAST	3.860	4157
HOLLYWOOD	1.438	3946
OLYMPIC	1.830	3640
WILSHIRE	2.374	3639
MISSION	4.705	3592
CENTRAL	1.135	3470
WEST VALLEY	3.501	3270
FOOTHILL	3.824	3141
NORTH HOLLYWOOD	2.723	2968
VAN NUYS	2.214	2719
DEVONSHIRE	3.968	2209
PACIFIC	3.727	2193
TOPANGA	3.454	1666
WEST LOS ANGELES	4.232	1614

Table 10: Query 4 (2a)

Year	Average Distance	Total Crimes
2010	2.279	5842
2011	2.510	8186
2012	2.498	11956
2013	2.429	2322
2014	2.162	2354
2015	2.460	3505
2016	2.368	6496
2017	2.420	9883
2018	2.358	2282
2019	2.431	7100
2020	2.266	2287
2021	2.361	14415
2022	2.295	15722

2023	2.345	11764

Table 11: Query 4 (2b)

Division	Average Distance	Total Crimes
77TH STREET	1.688	12961
SOUTHWEST	2.282	11175
SOUTHEAST	2.237	9479
NEWTON	1.575	7044
WILSHIRE	2.456	6464
HOLLENBECK	2.665	6452
HOLLYWOOD	1.966	5793
HARBOR	3.871	5574
OLYMPIC	1.664	4814
RAMPART	1.409	4741
VAN NUYS	2.935	4580
FOOTHILL	3.596	3767
CENTRAL	1.011	3594
NORTHEAST	3.699	3337
NORTH HOLLYWOOD	2.763	2983
WEST VALLEY	2.726	2963
MISSION	3.819	2257
PACIFIC	3.706	2078
TOPANGA	3.031	1829
DEVONSHIRE	3.002	1164
WEST LOS ANGELES	2.686	1065

Join Strategies

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StringType
from pyspark.sql.functions import col, year, desc, udf, split,
    regexp_replace
from time import time
import sys

APP_NAME = "Join_Query_3"
HDFS_DATA_DIR = "hdfs://okeanos-master:54310/data"

spark = SparkSession.builder.appName(APP_NAME).getOrCreate()

```

```
basic_csv_path = f"{HDFS_DATA_DIR}/Full_Data"
revgecoding_csv_path = f"{HDFS_DATA_DIR}/revgecoding.csv"
income_csv_path = f"{HDFS_DATA_DIR}/LA_income_2015.csv"

basic_df = spark.read.csv(basic_csv_path, header=True)
revgecoding_df = spark.read.csv(revgecoding_csv_path, header=True)
income_df = spark.read.csv(income_csv_path, header=True)

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

mapping_udf = udf(lambda x: victim_descent_mapping.get(x, x),
    StringType())
results = []

start_time = time()

revgecoding_df = revgecoding_df.hint(sys.argv[1])

inner_join_df = (
    basic_df.filter((year(col("Date_Rptd")) == 2015) & (col("Vict 
        Descent") != ""))
    .join(
        revgecoding_df,
        (basic_df["LAT"] == revgecoding_df["LAT"])
        & (basic_df["LON"] == revgecoding_df["LON"]),
    )
    .select("Vict Descent", "ZIPcode")
)

income_df = (
    income_df.withColumn(
        "Estimated Median Income", regexp_replace("Estimated Median Income", "\\$", "")
    )
    .withColumn(
        "Estimated Median Income", regexp_replace("Estimated Median Income", ",", "")
    )
    .withColumn(
        "Estimated Median Income", col("Estimated Median Income").cast("integer")
    )
)

for asc in [True, False]:
    sorted_income_ZIP_codes_df = (
        income_df.sort("Estimated Median Income", ascending=asc)
        .limit(3)
        .select("Zip Code")
        .collect()
    )

    data = [row["Zip Code"] for row in sorted_income_ZIP_codes_df]

    results_df = (
        inner_join_df.filter(split(col("ZIPcode"),
            "-").getItem(0).isin(data))
        .withColumnRenamed("Vict Descent", "Victim Descent")
        .groupBy("Victim Descent")
        .count()
        .withColumnRenamed("count", "crime_total")
        .orderBy(desc("crime_total"))
        .withColumn("Victim Descent", mapping_udf(col("Victim Descent")))
    )

    results.append(results_df)


for df in results:
```

```python
    df.explain()
    df.show(df.count(), truncate=False)

exec_time = time() - start_time
print(f'\n\nExec time: {exec_time} sec\n\n')

spark.stop()
```

```python
from pyspark.sql import SparkSession, Window
from pyspark.sql.types import FloatType
from pyspark.sql.functions import (
    col,
    length,
    udf,
    year,
    avg,
    count,
    desc,
)
from geopy.distance import geodesic
import sys, time

APP_NAME = "Inner_Join_Query_4"
HDFS_DATA_DIR = "hdfs://okeanos-master:54310/data"

spark = SparkSession.builder.appName(APP_NAME).getOrCreate()

basic_csv_path = f"{HDFS_DATA_DIR}/Full_Data"
departmens_csv_path = f"{HDFS_DATA_DIR}/LAPD_Police_Stations.csv"

basic_df = spark.read.csv(basic_csv_path, header=True)
departments_df = spark.read.csv(departmens_csv_path, header=True)


def compute_distance(lat1, lon1, lat2, lon2):
    crime_location = (lat1, lon1)
    department_location = (lat2, lon2)
    return geodesic(crime_location, department_location).kilometers


compute_distance_udf = udf(compute_distance, FloatType())

results_df_list = []

start_time = time.time()
```

```

filtered_df = basic_df.filter(
    col("Weapon Used Cd").startswith("1")
    & (length(col("Weapon Used Cd")) == 3)
    & ((col("LAT") > 0) & (col("LON") < 0))
)

filtered_df = filtered_df.withColumn("AREA", filtered_df["AREA"].cast("integer"))
departments_df = departments_df.withColumn(
    "PREC", departments_df["PREC"].cast("integer")
)

departments_df = departments_df.hint(sys.argv[1])

join_df = filtered_df.join(departments_df, col("AREA") == col("PREC")).withColumn(
    "distance",
    compute_distance_udf(
        filtered_df["LAT"],
        filtered_df["LON"],
        departments_df["Y"],
        departments_df["X"],
    ),
)

results_df_list = []

results_df = (
    join_df.withColumn("year", year("Date Rptd"))
    .groupBy("year")
    .agg(
        avg("distance").alias("average distance"), count("*").alias("total crimes")
    )
    .select("year", "average distance", "total crimes")
    .orderBy("year")
)
results_df_list.append(results_df)

results_df = (
    join_df.withColumnRenamed("DIVISION", "division")
    .groupBy("division")
    .agg(
        avg("distance").alias("average distance"),
                count("*").alias("total crimes")
    )
    .select("division", "average distance", "total crimes")
    .orderBy(desc("total crimes"))
)
results_df_list.append(results_df)

for df in results_df_list:
    df.explain()
    df.show(df.count(), truncate=False)

exec_time = time.time() - start_time
print(f'\n\nExec time: {exec_time} sec\n\n')

spark.stop()
```

Υποβάλλουμε τις εργασίες στο spark με τις παρακάτω εντολές:
spark-submit ./src/joins/query_3.py hint
spark-submit –py-files dependencies.zip ./src/joins/query_4.py hint
όπου **hint = broadcast, merge, shuffle_hash, shuffle_replicate_nl**.
Στις παραπάνω εργασίες μετρήθηκαν οι χρόνοι εκτέλεσης των ερωτημάτων 3 και 4 για διαφορετικές στρατηγικές join. Οι στρατηγικές που δοκιμάστηκαν είναι οι broadcast, merge, shuffle_hash και shuffle_replicate_nl. Παρατηρούμε ότι για το Query 3, οι στρατηγικές merge και shuffle_hash φαίνονται να παρουσιάζουν καλύτερα αποτελέσματα, ενώ στη περίπτωση του ερωτήματος 4, τα broadcast και shuffle_replicate_nl είναι αυτά με τους χαμηλότερους χρόνους. Αυτό οφείλεται στο γεγόνος ότι στην δεύτερη περίπτωση, το departments dataset είναι αρκετά μικρό, επομένως το broadcast ή το replication του μικρότερου dataset σε κάθε executor έχει καλύτερη επίδοση από τα άλλα δύο joins, τα οποία εφαρμόζουν hashing ή sorting και shuffling στα δεδομένα. Αντίθετα, για την πρώτη περίπτωση, όπου το revgecoding dataset είναι σχετικά μεγαλύτερο, οι άλλες δύο στρατηγικές εμφανίζουν καλύτερα αποτελέσματα.

Table 12: Query 3 join times

Join	**Time (sec)**
Broadcast	59.06
Merge	47.88
Shuffle Hash	45.28
Shuffle Replicate NL	58.00

Table 13: Query 4 join times

Join	**Time (sec)**
Broadcast	61.87
Merge	83.36
Shuffle Hash	80.96
Shuffle Replicate NL	58.07

`https://github.com/ekaragiannis/ntua_advanced_db`