National Technical University of Athens • Electrical and Computer Engineering
• Advanced Topics in Database Systems • Semester Assignment 2023-2024

LA Crime Data Analytics

Team ID : 19
Ioannis Palaios - el18184
Petros Avgerinos - el15074
Github Repository Link : /la-crime-data-analytics

I. Introduction

The goal of this assignment is to get familiar with the Apache Spark unified analytics engine by executing and benchmarking specific queries and comparing different Spark APIs and join methods in order to compare their performance and further understand distributed query processing in the Spark engine. We were assigned to perform 4 types of different queries on a data set containing Los Angeles Crime Data and measure their performance using the Spark Dataframe/SQL API on all of them and the low-level RDD API on the second query. Lastly, we benchmarked and compared the 3rd and 4th queries under the 4 different Spark join strategy types.

II. Our setup

Our cluster consists of two nodes with 4 CPU cores (with 2.3GHz clock speed), 8GB of RAM and 30GB of disk. We utilize the Hadoop Distributed File System (HDFS) as an infrastructure for a distributed file system across the two nodes of our cluster. We also use Apache Hadoop's YARN software for resource management and job/task scheduling across the specified number of executors.

The Spark application (Driver) process is running onto the one node named 'okeanos-master' in addition to one or more executors. Onto the other node, named 'okeanos-worker', only executor processes can be created. All of the queries will be benchmarked using 4 executors of which 2 of them are located on the 'okeanos-master' node and the other 2 on the 'okeanos-worker' node. Also, in the case we benchmark some queries with 3 executors, 2 will be placed on 'okeanos-worker' node and the other one on the 'okeanos-master' node. In the case of 2 executors, one executor will be placed in each of the two nodes.

III. The Dataset

We were assigned to analyze a data set consisting of Los Angeles Crime Data from the years 2010 to 2019 and 2020 to 2023 from separate .csv files. We merge them into one single .csv file which is of size 800 MB. We load this merged csv file onto the HDFS and it gets stored onto the datanodes (or onto exactly one datanode). Also, for some queries we make use of three other secondary datasets, namely :

- 'LA Police Stations' : which contains the location of the 21 police stations of Los Angeles
- 'Median Household Income by Zip Code' : which maps the Median Household income to the areas Zip Code

- 'Reverse Geocoding' : which maps the coordinates of each crime in the data set onto Zip codes in the LA area.

Because of the analytical nature of all the queries we were assigned to benchmark, it makes sense to transform the data (especially the big datasets such as the LA Crime Data file) from the row-oriented file format that CSV is, to a column-oriented file format such as Parquet. This is because the queries we are asked to perform are aggregation heavy and thus, reading the data row by row would not be the most efficient way. As such, we expect a performance benefit when performing the queries on the Parquet file format in comparison to the Csv file format.

IV. First Query

In the first query we essentially group by each row for each year and month and then aggregate the rows for each year and month and count their occurrences (named '#'). Lastly, we sort by both year and the count column and then select the 3 rows of each year with the highest count and add a row number column for each year. We benchmarked this query using Apache Spark's Dataframe API and SQL API. The source code of this query using both of these APIs is shown in Listing 1 and 2 below.

```
def query_1_SQL_API():
    start_time = time.time()
    crime_data.createOrReplaceTempView("crime_data")

    query = """
        SELECT * 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
                crime_data
            GROUP BY
                year(‘Date Rptd‘), month(‘Date Rptd‘)
        ) ranked_data
        WHERE rank <= 3
        ORDER BY year, rank
    """

    result_df = spark.sql(query)
    result_df.show()

    end_time = time.time()

    return end_time - start_time
```

Listing 1: The source code of the First Query using PySpark and the SQL API.

```
def query_1_Dataframe_API():
    start_time = time.time()
    crime_counts = crime_data.withColumn("year", F.year("Date Rptd")) \
                            .withColumn("month", F.month("Date Rptd")) \
                            .groupBy("year", "month") \
                            .agg(F.count("*").alias("crime_total"))

    window_spec = Window.partitionBy("year").orderBy(F.desc("crime_total"))

    ranked_crime = crime_counts.withColumn("rank", F.row_number().over(window_spec))

    result_df = ranked_crime.filter("rank <= 3").orderBy("year", "rank")

    result_df.show()
    end_time = time.time()

    return end_time - start_time
```

Listing 2: The source code of the First Query using PySpark and the Dataframe API.

In Figure 1, we observe very similar execution times between the two APIs. This happens because both of those two APIs use the same underlying Catalyst optimizer and thus, code written for the same query from the two different APIs is likely going to be translated to the same physical plan. This claim is easily verified when we use the *explain()* method onto the final dataframes created, as we get almost identical physical plans (except one final 'Project' from the SQL API).

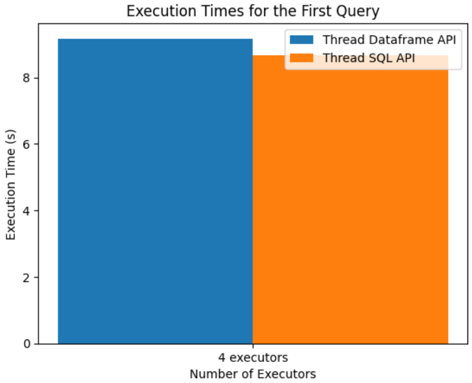

Figure 1: Bar plot of execution times of the second query comparing the Dataframe API with RDD API

V. Second Query

In the second query, the objective is to find how many crimes happen on the street on the 4 parts of the day : Morning, Noon, Afternoon and Night and then appear them sorted by descending order. As we do not need to take into account crimes that did not happen on the Street, we first filter them out in order to reduce the dataset (even if we performed it later the optimizer would make the filter to happen first). We then have to categorize each crime that happened on the Street by the part of the day that it took part.

- On the Dataframe API, we add a column that has a different String value for each part of the day. Then we perform a grouping on this new column, count the rows that are aggregated and then order by this count (aggregated) column.

- On the RDD API, we create a key-value pair with the key equal to the String value of the part of the day by mapping each row to a specific function that returns this string value and with value equal to 1. Then, we use the *reduceByKey()* transformation in order to merge the values of each key while using an associate reduce function similar to add. This process is similar to a group by and count aggregation as we merge the key-value pairs with the same key and count the rows aggregated by adding the values of each pair (that are equal to one). We then sort the pairs by their value (in descending order) and print them.

The source code of both of these APIs is shown on Listings 3 and 4 below.

```
def query_2_Dataframe_API():
    start_time = time.time()
    filtered_df = crime_data.filter(crime_data[’Premis_Desc’] == ’STREET’)

    time_group_df = filtered_df.withColumn("time_group",
                                       # TIME OCC is in 24 hour military time integer values
                                      F.when((F.col(’TIME OCC’).between(500, 1159)), ’Morning’)
                                      .when((F.col(’TIME OCC’).between(1200, 1659)), ’Noon’)
                                      .when((F.col(’TIME OCC’).between(1700, 2059)), ’Afternoon’)
                                      .otherwise(’Night’))

    result_df = time_group_df.groupBy("time_group").agg(F.count("*").alias("count"))

    result_df = result_df.orderBy(col("count").desc())
    result_df.show()
    end_time = time.time()
    # call explain() method in order
    # to see the query’s physical plan
    # and improve the RDD query
    result_df.explain()
    return end_time - start_time
```

Listing 3: The source code of the Second Query using PySpark and the Dataframe API

```
def time_segs(row):
    if 500 <= int(row[’TIME OCC’]) <= 1159:
        return ’Morning’
    elif 1200 <= int(row[’TIME OCC’]) <= 1659:
        return ’Noon’
    elif 1700 <= int(row[’TIME OCC’]) <= 2059:
        return ’Afternoon’
    else:
        return ’Night’

def query_2_rdd():
    start_time = time.time()
    crime_data_rdd = crime_data.rdd.filter(lambda x: x[’Premis_Desc’] == ’STREET’) \
                                .map(lambda x: (time_segs(x),1)) \
                                .reduceByKey(lambda k1,k2: k1+k2) \
                                .sortBy(lambda x: x[1], ascending = False)

    result = crime_data_rdd.collect()
    for time_of_day, count in result:
        print(f"{time_of_day}: {count}")

    end_time = time.time()
    return end_time - start_time
```

Listing 4: The source code of the Second Query using PySpark and the RDD API.

In order to explain the performance gap of these two APIs it is essential to understand their major differences :

- In Apache Spark, RDD is a fundamental data structure and a Low-Level API providing more control of the data and lower level optimizations compared to the Dataframes API. On the contrary, the Dataframes API is a higher level API built on top of RDDs and essentially is a higher level abstraction with various optimizations and better programmability.
- While RDDs provide more control over the data, the optimization is left to the programmer and thus is more prone to perform worse. On the contrary, the Dataframes API is optimized for performance and makes use of the Catalyst optimizer which parses the written query and creates an effective physical plan (the set of operations executed).

We benchmark those two queries and observe a huge performance difference in favor of the Dataframe API in Figure 2. This is largely due to the high level optimizations that are performed by the Catalyst optimizer. The Catalyst Optimizer takes the code written for this query and converts it into a smart and efficient execution plan comprised from operations that are going to be applied onto our data. This optimization only happens on the Dataframe API as we mentioned above and is something that the lower-level RDD API is in lack of. We could likely improve the RDD's performance by trying to mimic the physical plan generated by the Dataframes query (and the Catalyst Optimizer) but this is outside the scope of this assignment.

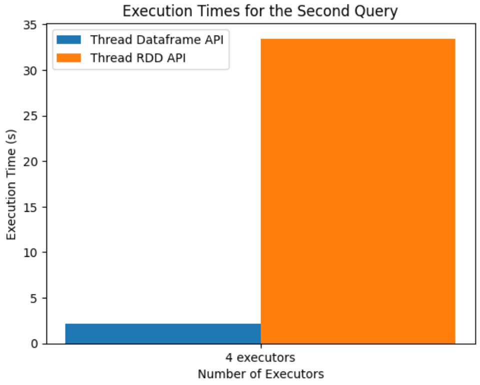

Figure 2: Bar plot of execution times of the second query comparing the Dataframe API with RDD API

VI. Third Query

In this query, we are asked to find the victim descent of crimes that took place on the year 2015 and in Los Angeles' top 3 highest and lowest ZIP Codes in terms of median household income. Because each crime does not have a column with the ZIP Code but the coordinates it took place in, we have to join the crime dataset with the given reverse geocoding dataset that maps each pair of coordinates existing on the dataset to a specific ZIP Code. Also, with trivial grouping and aggregation we extract the top 3 highest and lowest median household income ZIP Codes. We make these Zip Codes a list, and filter out any rows in the joined crime dataset who don't have one of those ZIP codes in the List or did not happen in 2015.

The source code of this query is shown in Listing 5.

```
def query_3(method = 'CONTINUE'):
    start_time = time.time()
    crime_data_join_revge = crime_data.join(revge, ['LAT', 'LON'], 'inner') \
        .withColumnRenamed('ZIPcode', 'Zip Code') \
        .withColumn("Zip Code", col("Zip Code").cast("int")) \
        .filter((col('Vict Descent') != 'X') & (col('Vict Sex') != 'X'))

    crime_data_join_income = crime_data_join_revge.join(income, 'Zip Code', 'inner')
     \
                                .withColumn('Estimated Median Income',
                                            regexp_replace(col('Estimated Median
    Income'), '[$,]', '')) \
                                .withColumn('Estimated Median Income',
                                            col('Estimated Median Income') \
                                .cast('double'))

    max_income_zip_codes = crime_data_join_income.groupBy('Zip Code') \
                            .agg({'Estimated Median Income': 'max'}) \
                            .withColumnRenamed('max(Estimated Median Income)', '
    MaxIncome') \
                            .orderBy(col('MaxIncome').desc()) \
                            .limit(3)

    min_income_zip_codes = crime_data_join_income.groupBy('Zip Code') \
                            .agg({'Estimated Median Income': 'min'}) \
                            .withColumnRenamed('min(Estimated Median Income)', '
    MinIncome') \
                            .orderBy(col('MinIncome')) \
                            .limit(3)

    zip_codes = min_income_zip_codes.union(max_income_zip_codes)

    zip_codes_list = [row['Zip Code'] for row in zip_codes.collect()]

    result = crime_data_join_income \
                .filter(col('Zip Code').isin(zip_codes_list)) \
                .filter(year(col('Date Rptd')) == 2015) \
                .groupBy('Vict Descent') \
                .count() \
                .withColumnRenamed('count', '#') \
                .orderBy(col('#').desc())

    result.show()
    end_time = time.time()
    result.explain()
    print(f'Method : {method} | Time {end_time - start_time}')
    return end_time - start_time
```

Listing 5: The source code of the Third Query using PySpark and the Dataframe API

We are asked to benchmark the query for 2,3 and 4 executors. In order to understand, the reason they are expected to have a performance difference, we first need to understand what executors are in Apache Spark. Executors are the processes in charge of running the individual tasks of a given Spark job in each worker node. Of course, more than 1 executor can exist in a given worker node but there is a threshold in each node, on how many executors can prove to be efficient. As separate JVM processes, even executors running on the same worker node cannot share the same memory (as by definition have a different memory address space) and need Inter-Process Communication. Another thing that we should take note is the number of cores per executor which is heavily application/workload specific but we will not configure it (so it will take 1 - its default value).

Another thing of critical importance, is the number of cores we assign for each executor. Increasing the number of cores per executor (instead of increasing the number of executors) can minimize the cost of communication and increase task completion speed. So, at first glance, it is reasonable to assume that the best approach to number of executors and number of cores per executor will be to create 'Fat executors'. Fat executors are essentially the approach of creating executors with a large number of cores (almost all the cores of a worker's node) and as a result having one executor per node. This minimizes communication between them. But these executors will likely have large memory pools and as a result JVM garbage collector delays would slow down our jobs unreasonably. This is especially true for nodes with a lot of cores (8+). The consensus in most Spark tuning guides is that 5 cores per executor is the optimum number of cores in terms of parallel processing.

So, as we are asked to benchmark the Third query for 2,3 and 4 executors, it makes sense to configure the cores per executor that we have in each case. As our nodes, have 4 cores, in our master node we have to have 2 cores free for the Operating System, our Cluster Manager (YARN) and the Driver process. We will also follow this approach on our worker node. The 3 cases we will benchmark are :

- 2 executors, each having 2 cores and 2 GB of memory each. - 16.93s
- 3 executors, each having 1 core and 1 GB of memory each (placing 2 executors on the worker node). - 17.47s
- 4 executors, each having 1 core and 1 GB of memory each. - 20.98s

We will keep the cores per driver equal to 1 and driver memory equal to 1GB for all the cases.

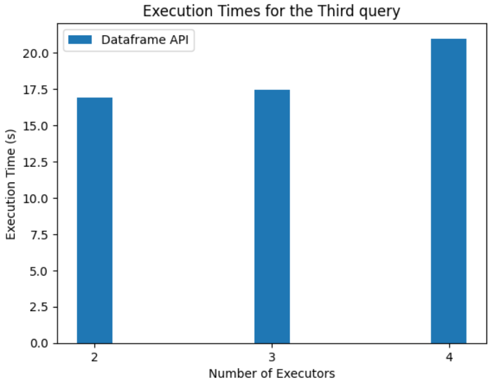

Figure 3: Bar plot of execution times of the Third query comparing them for different numbers of executors

In Figure 3, we observe that we achieve comparable performance in the cases with 2 and 3 executors while in the case of 4 executors performance visibly worsens 20%. This is likely due to the communication overhead between these executors in our query that is join-heavy, performing a Broadcast Exchange, a Broadcast Hash Join, a Hash Aggregation etc (as we seen from the printed physical plan). Having less executors reduces this overhead and decreases execution time. We expected to notice a bigger difference in performance between the 2 and 3 executors due to having less total cores and memory. This was likely avoided due to our query not being computationally intensive and a smart task allocation by the Spark Engine.

VII. Fourth Query

For the 4th query we were assigned to create 4 subqueries of two different types. The first type is to find the number of crimes per year performed with firearms and their average distance from the police station that investigated the crime/the closest police station to the crime location. The second type is to find the number of crimes per police station that investigated them/were closer to that station, with any type of gun and their average distance from that station.

But, the 4th query can be categorized for whether we are searching for the police station that investigated the crime or for the closest police station of the crime. As our crime data set does not contain information about the LAPD stations, we have to join this dataframe with the dataframe containing the LAPD station location information. Then we have to calculate the distance of the police station (investigated or were closest) using the haversine function which finds the distance of two points on a sphere surface. The source code of the haversine function is shown on the Listing below.

```python
def haversine(lon1, lat1, lon2, lat2):
    R = 6371
    dLat = math.radians(lat2 - lat1)
    dLon = math.radians(lon2 - lon1)
    a = math.sin(dLat / 2) * math.sin(dLat / 2) + \
        math.cos(math.radians(lat1)) * math.cos(math.radians(lat2)) * math.sin(dLon
    / 2) * math.sin(dLon / 2)
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    distance = R * c
    return distance

haversine_udf = udf(haversine, DoubleType())
```

Listing 6: The source code of Haversine function

We then filter out the crimes that did not happen with firearms/gun and then group by the year or the police station and then trivially find their average distance (aggregating the rows with count() and sum()). The source codes for the first type of query for police stations that investigated the crime (and not the closest to the crime) and the second type of query for the crimes that were closer to each police station, are shown below :

```python
def query_4_1a():
    start_time = time.time()
    lapd_stations_new = lapd_stations.withColumnRenamed(’PREC’,’AREA’)

    crime_data_join_stations = crime_data.withColumnRenamed(’AREA ’, ’AREA’) \
                                    .join(lapd_stations_new, ’AREA’, ’inner’)

    crime_data_join_stations = crime_data_join_stations.withColumn(’distance’,
                                    haversine_udf(col("LON"), col("LAT"), col("X"),
    col("Y")))

    crime_data_join_stations = crime_data_join_stations.withColumn(’Weapon Used Cd’,
     col(’Weapon Used Cd’).cast(’int’)) \
                                    .filter(col(’Weapon Used Cd’) < 200) \
                                    .withColumn(’year’, F.year(’Date Rptd’))

    result = crime_data_join_stations.groupBy(’year’) \
        .agg((F.sum('distance') / F.count('*')).alias('average_distance'),
            F.count('*').alias('#')) \
       .orderBy(F.col('year'))

    result.show()
    end_time = time.time()
    print(f'Method : {method} | Time {end_time - start_time}')
    return end_time - start_time
```

Listing 7: The source code of the 4th query of the first type for the for police stations that investigated the crime.

```
# 2b)
def query_4_2b(method = 'CONTINUE'):
    start_time = time.time()

    crime_data_filtered = crime_data.withColumn('Weapon Used Cd', col('Weapon Used Cd').cast('int')) \
                                   .filter(F.col('Weapon Used Cd').isNotNull()) \
                                   .withColumn('year', F.year('Date Rptd'))


    combined_data = crime_data_filtered.crossJoin(lapd_stations)

    combined_data = combined_data.withColumn("closest_distance", haversine_udf(col("LON"), col("LAT"), col("X"), col("Y")))

    windowSpec = Window.partitionBy("DR_NO").orderBy("closest_distance")
    closest_stations = combined_data.withColumn("rank", rank().over(windowSpec)).filter(col("rank") == 1)

    final_data = closest_stations.select(col("DR_NO"), col("DIVISION").alias("closest_station"), col("closest_distance"))

    joined_result = crime_data_filtered.join(final_data, "DR_NO")
   # crime_data_join_stations = result.withColumn('Weapon Used Cd', col('Weapon Used Cd').cast('int')) \
   #                                 .filter(F.col('Weapon Used Cd').isNotNull()) \
   #                                   .withColumn('year', F.year('Date Rptd'))

    result = joined_result.groupBy('AREA NAME') \
        .agg(
            (F.sum('closest_distance') / F.count('*')).alias('average_distance'),
            F.count('*').alias('#')
        ) \
        .orderBy(F.col('#').desc()) \
        .withColumnRenamed('AREA NAME', 'division')

    result.show()
    end_time = time.time()
    print(f'Method : {method} | Time {end_time - start_time}')
    return end_time - start_time
```

Listing 8: The source code of the 4th query of the second type for the crimes that happened closer to each police station.

One important thing to note about the second type of the Fourth query is that instead of performing a Join with an equality condition, we have to perform a Cross join (a join with no equality condition) because of having to compare each row with all of the LAPD Police Station rows which are 21 in total.

VIII. Join Strategy Types

Lastly, we were asked to benchmark the Third and Fourth Queries for different types of joins in the Spark engine. In order compare them efficiently and understand their performance differences, we have to first analyze how these Join strategies work internally in the Spark engine.

- Broadcast Hash Join (BHJ) : One of the two tables of the join gets sent on all of the nodes of the cluster in order to perform the join, each node having one of the two tables locally.
- Shuffle Sort Merge Join (SMJ): Sort all the data accordingly to the Join Key and then performs merge operation based on the sorted Join Key. It uses the shuffle data exchange mechanism (all-to-all executors) and can be used only for the equality join condition. Can prove to be useful for joining two large tables on an equality condition.
- Shuffle Hash Join : Uses the Hash Join algorithm (hashes the smaller table) and the shuffling data exchange mechanism.
- Shuffle and Replicate Nested Loop Join (Cartesian Product Join): Obviously uses an all to all communication between the executors but replicates the data partition of a table from an executor to all other executors. Then, each record in the data partition uses the Join key to fully iterate through the replicated data partition.

The factors influencing the performance of these joins can be the join condition and the size of each table(and it's partitions) while the number of executors and each executors memory size also playing a significant part. It is useful to note that in most Spark applications, the main bottleneck is the shuffling process (or shuffle as we call it above). This process happens mostly in joins, in order to ensure that data with the same join key end up in the same partition (in joins where the join condition is equality).

In the Third query join, we join the crime data dataframe with the reverse geocoding dataframe (which contains the mapping of each individual pair of coordinates to a specific ZIP codes). The Reverse geocoding data in the Parquet file format are around 0.17MB. The condition of the join is the equality of two columns *LAT* and *LONG*. As some crimes do not contain location, the total rows of the joined dataframe are less, from 3 million rows in the initial dataframe to 2.4M in the joined dataframe.

As shown in Figure 4 below, the performance of the Third query varies between the different Join Strategies but not by a big margin.

As we expected, the Join Strategy that performs the best is the Broadcast Hash Join (or Broadcast Join as we mentioned above) because the Reverse Geocoding dataframe is small enough to fit in memory of each executor in co-existence with each partition of the crime data dataframe and thus it makes sense to distribute it across the executors/nodes. Also, the broadcasted table's small size also minimizes the communication overhead between the nodes/executors (even though broadcasting is a network-intensive operation).

In the Shuffle Sort Merge Join, we observe lower performance which is likely due to the sorting that takes place and the shuffling that takes place in the Exchange operation. Also, in the case of the Shuffle Hash Join we observe lower performance than the BHJ but similar to the SMJ which is also likely to the shuffling overhead. We expect the performance of these two join strategies to increase if the size of the Reverse Geocoding data set was larger.

If we benchmark the Third query for the Shuffle and Replicate Nested Loop Join it does not even finish after 5 seconds. This is due to the inefficient implementation of this join for equality joins.

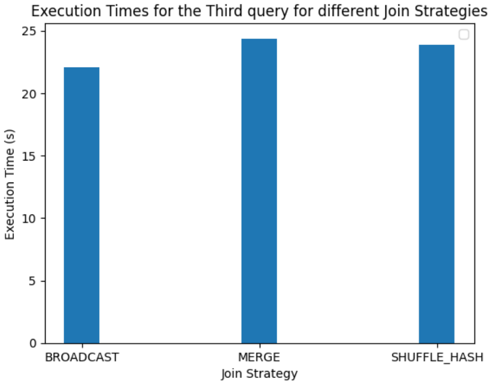

Figure 4: Bar plot of execution times of the Third query comparing it for different Join Strategies

All the above (and below) operations/transformations are verified from the physical plans of each query that we got from from the output of the *.explain()* method and the DAG visualization of each job obtained from the Spark UI.

In the Fourth query join, we observe that the dataframe containing the LAPD Station information is the small one and has only 21 rows. Thus, it easily fits onto each executor's memory and can co-exist there with the each crime data dataframe partition. In the subqueries for the police station that investigated each crime (of both types) it makes sense to broadcast this dataframe (to each node) in order to efficiently perform the join as it fits in each executor's memory. The bar plot for both types of the 4th query but only for the police stations that investigated each crime is shown in Figure 5 below. We observe very similar times between these different joins. This is likely due to the very small of the one dataframe and the equality nature of the join (on one column). This makes the sorting easier (as we only have 21 different values on the join column) and the shuffling process significantly less costly (for 3 joins that use this data exchange operation).

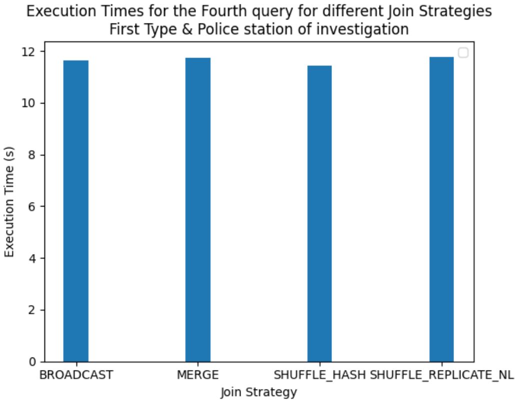

Figure 5: Bar plot of execution times of the Fourth query comparing it for the different Join Strategies

Obviously, some of the above Join Strategies only make sense to be used for joins that have some type of condition like equality, inequality, bigger than etc. These joins also do not support joins with no join equality key as it does not make sense to use them and compare them for joins that have no equality conditions, namely Cross Joins. So, for the subqueries for the *closest police station* where we perform a Cross Join we will benchmark them for the Broadcast Hash Join and the Shuffle Replicate Nested Loop Join as Shuffle Hash Join and Shuffle Sort Merge Join do not support joins with no key for equality. Both types of the 4th query use the cross join when searching for the *closest police station* so it makes sense to benchmark one of them (both have similar variance between the two join strategies). As we observe in Figure 6, these two strategies have very similar execution times due that these two operate similarly in this case (but apply different operations/transformations on the data as we see from the physical plan).

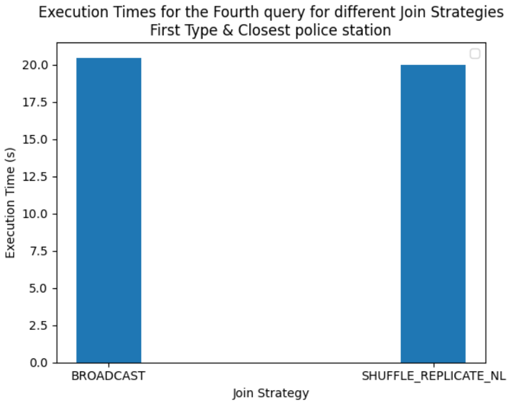

Figure 6: Bar plot of execution times of the Fourth query comparing it for the 3 different Join Strategies

Throughout all of the above benchmarked Join Strategies, we obtained their physical plan of the resulting each dataframe's execution in order to validate that the Join strategy we provided as input to the hint method actually got in the plan. This is also visible from the UI and it's execution DAG visualization.

IX. Further Optimizations

In most queries we executed above (or jobs in Spark terminology), we commonly observed that in each stage the max task time is larger than the median (observing this through the Spark UI), and this a sign that we have uneven data distribution among our partitions (data skew). We confirm this observation via the HDFS UI, where we see that the two nodes have used storage space : 300MB on the master and 100 MB on the worker so the data distribution is unequal and the executors on the master likely have higher workload than those on the worker and as a result those are the execution bottleneck. We can fix this issue by rebalancing the HDFS across it's datanodes but we run to various technical problems while attempting this.

Another thing, we could try is different Spark partition sizes and observe how these different sizes change performance but this is beyond this assignment. Also, we could also use the *cache()* and *persist()* in order to store data in memory (or sometimes in disk) and see how this could affect performance (most certainly increase it).

Lastly, we could study and change different Spark and YARN configurations like the shuffle configurations, the resource allocations, the memory management, task parallelism and more.

X. CONCLUSION

In this assignment, we efficiently wrote the queries for 4 different types of queries analyzing the Los Angeles crime dataset. We benchmarked (some of) them for different number of executors, different API's (namely the Dataframe API, the SQL API and the RDD API) and different Spark Join Strategies. In order to comment on the benchmarks, we became familiar with the internals of the Apache Spark unified analytics engine and observed the performance bottlenecks and optimizations that are possible by configuring some parts of the code or Spark's configurations.

References

[1] Apache Spark version 3.5 Documentation
https://spark.apache.org/docs/latest/.

[2] Spark: The Definitive Guide
https://www.oreilly.com/library/view/spark-the-definitive

[3] Learning Spark, 2nd Edition
https://www.oreilly.com/library/view/learning-spark-2nd.

[4] High Performance Spark, 2nd Edition https://www.oreilly.com/library/view/high-performance-spark/

[5] Apache Spark Join Strategies
https://blog.clairvoyantsoft.com/apache-spark-join-strategies-e4ebc7624b06

[6] A Systematic approach, tips, techniques, and best practices to enhance Apache Spark job performance
https://vijayaphanindra.medium.com/