

# ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΒΑΣΕΩΝ ΓΝΩΣΕΩΝ ΚΑΙ ΔΕΔΟΜΕΝΩΝ



## Προχωρημένα Θέματα Βάσεων Δεδομένων

Ακ. Έτος 2023-2024, 9ο Εξάμηνο

Δημήτρης Μητρόπουλος, 03118608

Δημήτρης Χούπας, 03118116

# Table of contents

<i>Github Link</i> .....	2
<i>Setup</i> .....	2
<i>Apache Hdfs</i> .....	2
<i>Apache Hadoop Yarn</i> .....	3
<i>Apache Spark</i> .....	3
<i>Data Loading</i> .....	4
<i>Query 1</i> .....	6
<i>Κώδικας</i> .....	6
<i>Αποτελέσματα</i> .....	8
<i>Χρόνοι εκτέλεσης</i> .....	12
<i>Query 2</i> .....	13
<i>Κώδικας</i> .....	13
<i>Αποτελέσματα</i> .....	15
<i>Χρόνοι εκτέλεσης</i> .....	16
<i>Query 3</i> .....	17
<i>Κώδικας</i> .....	17
<i>Αποτελέσματα</i> .....	18
<i>Χρόνοι εκτέλεσης</i> .....	20
<i>Query 4</i> .....	22
<i>Κώδικας</i> .....	22
<i>Αποτελέσματα</i> .....	27
<i>Join hint and explain</i> .....	31
<i>Broadcast Hash Join</i> .....	31
<i>Shuffle Hash, Shuffle Sort Merge Join</i> .....	35
<i>Shuffle_Replicate_Nested_Loop_Join or Cartesian Join</i> .....	37
<i>Συμπεράσματα</i> .....	38

## *Github Link*

- [https://github.com/dimitrismit/ntua\\_advanced\\_databases](https://github.com/dimitrismit/ntua_advanced_databases)

## *Setup*

Στο παρόν project χρησιμοποιήθηκαν οι εξής τεχνολογίες:

1. OpenJDK 11.0.21
2. Apache Hadoop 3.3.6
3. Apache Spark 3.3.1
4. Python 3.10.12
5. Py4j 0.10.9.7
6. Pyspark 3.5.0

Πιο συγκεκριμένα για το project χρησιμοποιήθηκαν πόροι από τον Okeanos, ένα cloud service του GRNET. Από τους διαθέσιμους πόρους (60GB disk size, 16 GB RAM και 8 CPU cores) δημιουργήθηκαν δύο VMs με το καθένα να έχει τους μισούς από τους διαθέσιμους πόρους. Για την επικοινωνία των δύο μηχανημάτων και την εύρυθμη λειτουργία των HDFS/Spark δημιουργήθηκε ένα local network με το ένα μηχάνημα να παίρνει διεύθυνση 192.168.0.1 και το άλλο 192.168.0.2.

Το μηχάνημα με την IPv4 192.168.0.1, ονομάστηκε ο master node του συστήματος και του δόθηκε και μια Public IPv4 διεύθυνση για πρόσβαση στο Internet, ώστε να μπορούν σε αρχικό στάδιο να γίνουν οι εγκαταστάσεις των απαραίτητων τεχνολογιών και σε επόμενο στάδιο, να είναι προσβάσιμα τα Web UIs των Spark και HDFS/Hadoop.

## Apache Hdfs

Το HDFS είναι ένα σύστημα αποθήκευσης δεδομένων το οποίο χρησιμοποιεί μια distributed λογική, το οποίο παρέχει επεκτασιμότητα και αξιοπιστία για αποθήκευση και επεξεργασία μεγάλων δεδομένων. Μαζί με το Apache Spark, είναι ένας από τους πιο διαδεδομένους συνδυασμούς συστημάτων για την επεξεργασία και την αποθήκευση μεγάλου όγκου δεδομένων.

Για την εγκατάσταση του HDFS ακολουθήθηκαν τα εξής βήματα:

1. Εγκατάσταση Java στα μηχανήματα
2. Δημιουργία ζεύγους δημόσιου/ιδιωτικού κλειδιού και αποθήκευσή τους στα κατάλληλα αρχεία στο κάθε μηχάνημα, ώστε το ssh από το ένα μηχάνημα στο άλλο να λειτουργεί χωρίς να χρειάζεται ο κωδικός, προαπαιτούμενο για την λειτουργία των Spark/HDFS.
3. Εγκατάσταση του Apache Hadoop στα μηχανήματα
4. Ρύθμιση των μεταβλητών περιβάλλοντος στο αρχείο .bashrc
5. Επεξεργασία των αρχείων hdfs-site.xml και core-site.xml
6. Format του namenode και start το HDFS service

## Apache Hadoop Yarn

Το Apache Hadoop Yarn λειτουργεί ως ο διαχειριστής των πόρων (RM) ενός cluster στις διάφορες εφαρμογές που εκτελούνται, όπως και ο προγραμματιστής των εργασιών που θα εκτελέσει το σύστημα (AM). Η ξεχωριστή εγκατάσταση του δεν είναι απαραίτητη, μιας και εγκαθίσταται μαζί με το Apache HDFS. Ωστόσο, όπως και στο HDFS, έτσι και εδώ αντίστοιχα, χρειάζεται να επεξεργαστούμε το αρχείο yarn-site.xml και έπειτα ξεκινάμε το αντίστοιχο service.

## Apache Spark

Το Apache Spark είναι ένα ευρέως διαδεδομένο open-source analytics engine, για ανάλυση και επεξεργασία Big Data. Παρέχει fault tolerant και data parallelism optimizations, το οποίο το κάνει ιδιαίτερα χρήσιμο για data analytics σε υπολογιστικά clusters. Αντίστοιχα με το HDFS, γίνεται η εγκατάσταση και στα δύο μηχανήματα και στη συνέχεια η διαμόρφωση κάποιων default παραμέτρων στο αρχείο spark-defaults.conf. Στη συνέχεια ξεκινάμε τον spark history server, αφού πρώτα δημιουργήσουμε στο HDFS ένα directory για να αποθηκεύονται τα Spark logs.

## Data Loading

Για το συγκεκριμένο project χρησιμοποιήθηκαν δύο dataset, το καθένα με δεδομένα για εγκλήματα στην πόλη του Los Angeles από διαφορετικές χρονικές περιόδους. Πιο συγκεκριμένα ένα για την περίοδο 2010-2019 και ένα για την περίοδο 2020-Present. Ωστόσο στα queries, οι απαντήσεις πρέπει να περιλαμβάνουν και τα δύο datasets. Οπότε δημιουργήθηκε dataset το οποίο ενώνει τα 2 επιμέρους datasets. Έπειτα, έγιναν διάφοροι έλεγχοι για δεδομένα που “μολύνουν” το dataset μας, όπως π.χ. για δεδομένα πριν το 2010.

Οι αρχικοί τύποι των δεδομένων, μετά από το φόρτωμα του συνόλου δεδομένων (και κρατώντας τα ονόματα των στηλών ίδια) είναι:

```
This is the original schema
root
|-- DR_NO: integer (nullable = true)
|-- Date Rptd: string (nullable = true)
|-- DATE OCC: string (nullable = true)
|-- TIME OCC: integer (nullable = true)
|-- AREA : integer (nullable = true)
|-- AREA NAME: string (nullable = true)
|-- Rpt Dist No: integer (nullable = true)
|-- Part 1-2: integer (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)
```

Μετά τις αλλαγές των στηλών ως εξής:

1. Date Rptd: date
2. DATE OCC: date
3. Vict Age: integer
4. LAT: double
5. LON: double

όπως αναφέρονται και στην εκφώνηση έχουμε το εξής schema στο dataframe:

```
This is the updated schema
root
|-- DR_NO: integer (nullable = true)
|-- Date Rptd: date (nullable = true)
|-- DATE OCC: date (nullable = true)
|-- TIME OCC: integer (nullable = true)
|-- AREA : integer (nullable = true)
|-- AREA NAME: string (nullable = true)
|-- Rpt Dist No: integer (nullable = true)
|-- Part 1-2: integer (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)
```

Αξίζει να σημειωθεί πως στο dataset για τα εγκλήματα από το 2010-2019 παρατηρήθηκαν δεδομένα μετά το 2019. Ωστόσο, μιας και μας νοιάζουν δεδομένα μέχρι σήμερα, αποφασίσαμε να τα κρατήσουμε. Όμως, μιας και υπήρχε dataset με δεδομένα μετά το 2019, έγινε και ένας έλεγχος για duplicates δεδομένων.

Μετά από το φιλτράρισμα των δεδομένων, άλλαξε η στήλη “Vict Desc”, που περιγράφει με ένα γράμμα την καταγωγή του θύματος. Το κάθε γράμμα αντιστοιχήθηκε με την λέξη (π.χ. A-Asian), μια αλλαγή που χρειάζεται για το query 3 και δεν επηρεάζει κάπως τα υπόλοιπα queries.

Μετά και τις αλλαγές οι γραμμές του συνόλου δεδομένων είναι 2993433.

## Query 1

Σημείωση: Για την λήψη όσο πιο έγκυρων μετρήσεων γίνεται, το query εκτελέστηκε 5 φορές και για τους timers πάρθηκε ο μέσος όρος των τιμών. Επίσης το formatting των αποτελεσμάτων έχει αλλάξει ώστε είναι πιο ευπαρουσίαστες στην αναφορά. Για κάθε implementation θα αφήνεται link του github με τα αποτελέσματα σαν footnote. Τέλος, ο κώδικας που ακολουθεί δεν είναι ο πλήρης κώδικας, καθώς κομμάτια όπως ο ορισμός των dataframes, ή το ξεκίνημα ενός spark session δεν περιλαμβάνονται για εξοικονόμηση χώρου. Για τον πλήρη κώδικα υπάρχει στην αρχή του εγγράφου το link στο github repository.

### Κώδικας

-----Spark Dataframe-----

```
#extract year and month from the timestamp
crime_df = crime_df.withColumn("Year",
year("Date")).withColumn("Month", month("Date"))

#group by year and month, then count the crimes, and order by
count in descending order
result = (crime_df.groupBy("Year", "Month")
            .agg(count("*").alias("CrimeCount"))
            .orderBy("Year", "CrimeCount", ascending=[True, False]))

#rank the months within each year based on the crime count and
keep the top three for each year
result = result.withColumn("Rank",
dense_rank().over(Window.partitionBy("Year").orderBy(desc("CrimeCo
unt"))))
result = result.filter(F.col("Rank") <= 3)
```

```
result = spark.sql("""
    SELECT
        Year,
        Month,
        CrimeCount,
        Rank
    FROM (
        SELECT
            YEAR(`DATE OCC`) as Year,
            MONTH(`DATE OCC`) as Month,
            COUNT(*) as CrimeCount,
            DENSE_RANK() OVER (PARTITION BY YEAR(`DATE OCC`) ORDER
BY COUNT(*) DESC) as Rank
        FROM crime_table
        GROUP BY Year, Month
    ) ranked
    WHERE Rank <= 3
    ORDER BY Year ASC, Rank
""")
```



## Αποτελέσματα

-----Spark Dataframe<sup>1</sup>-----

Year	Month	CrimeCount	Rank	Year	Month	CrimeCount	Rank
2010	1	19515	1	2013	8	17440	1
2010	3	18131	2	2013	1	16820	2
2010	7	17856	3	2013	7	16644	3
2011	1	18135	1	2014	7	13531	1
2011	7	17283	2	2014	10	13362	2
2011	10	17034	3	2014	8	13317	3
2012	1	17943	1	2015	10	19219	1
2012	8	17661	2	2015	8	19011	2
2012	5	17502	3	2015	7	18709	3

---

<sup>1</sup>[https://github.com/dimitrismit/ntua\\_advanced\\_databases/blob/main/results/comma\\_separated/query1/query1\\_df\\_results.csv](https://github.com/dimitrismit/ntua_advanced_databases/blob/main/results/comma_separated/query1/query1_df_results.csv)

Year	Month	CrimeCount	Rank	Year	Month	CrimeCount	Rank
2016	10	19659	1	2019	7	19121	1
2016	8	19490	2	2019	8	18979	2
2016	7	19448	3	2019	3	18856	3
2017	10	20431	1	2020	1	18498	1
2017	7	20192	2	2020	2	17256	2
2017	1	19833	3	2020	5	17205	3
2018	5	19973	1	2021	12	25453	1
2018	7	19875	2	2021	10	24653	2
2018	8	19761	3	2021	11	24276	3

Year	Month	CrimeCount	Rank	Year	Month	CrimeCount	Rank
2022	5	20419	1	2023	8	19772	1
2022	10	20276	2	2023	7	19709	2
2022	6	20204	3	2023	1	19637	3

## -----Spark SQL<sup>2</sup>-----

Year	Month	CrimeCount	Rank	Year	Month	CrimeCount	Rank
2010	1	19515	1	2013	8	17440	1
2010	3	18131	2	2013	1	16820	2
2010	7	17856	3	2013	7	16644	3
2011	1	18135	1	2014	7	13531	1
2011	7	17283	2	2014	10	13362	2
2011	10	17034	3	2014	8	13317	3
2012	1	17943	1	2015	10	19219	1
2012	8	17661	2	2015	8	19011	2
2012	5	17502	3	2015	7	18709	3

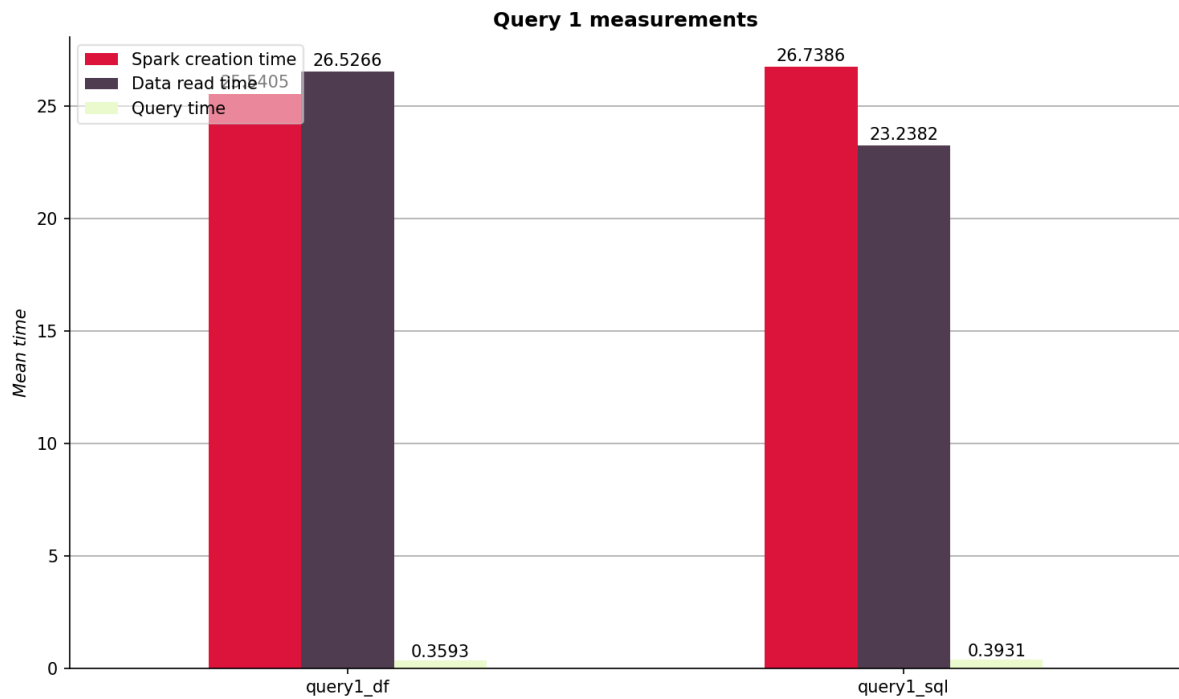
---

<sup>2</sup>[https://github.com/dimitrismit/ntua\\_advanced\\_databases/blob/main/results/comma\\_separated/query1/query1\\_sql\\_results.csv](https://github.com/dimitrismit/ntua_advanced_databases/blob/main/results/comma_separated/query1/query1_sql_results.csv)

Year	Month	CrimeCount	Rank	Year	Month	CrimeCount	Rank
2016	10	19659	1	2019	7	19121	1
2016	8	19490	2	2019	8	18979	2
2016	7	19448	3	2019	3	18856	3
2017	10	20431	1	2020	1	18498	1
2017	7	20192	2	2020	2	17256	2
2017	1	19833	3	2020	5	17205	3
2018	5	19973	1	2021	12	25453	1
2018	7	19875	2	2021	10	24653	2
2018	8	19761	3	2021	11	24276	3

Year	Month	CrimeCount	Rank	Year	Month	CrimeCount	Rank
2022	5	20419	1	2023	8	19772	1
2022	10	20276	2	2023	7	19709	2
2022	6	20204	3	2023	1	19637	3

## Χρόνοι εκτέλεσης



Παρατηρούμε πως οι χρόνοι εκτέλεσης των queries είναι αρκετά παρόμοιοι, με τις διαφορές να βρίσκονται σε εκατοστά του δευτερολέπτου. Αυτή η συμπεριφορά είναι αναμενόμενη μιας και οι δύο υλοποιήσεις χρησιμοποιούν τον Catalyst optimizer και τον Tungsten execution engine. Πιο συγκεκριμένα, το πρώτο χρησιμοποιεί διάφορα optimizations της γλώσσας προγραμματισμού Scala, για ένα πιο αποτελεσματικό physical execution plan. Ο Tungsten execution engine, είναι ένα πιο low-level σύστημα του Spark το οποίο κάνει optimize την χρήση μνήμης και CPU του Spark κατά την εκτέλεση εφαρμογών.

## Query 2

Σημείωση: Για την λήψη όσο πιο έγκυρων μετρήσεων γίνεται, το query εκτελέστηκε 5 φορές και για τους timers πάρθηκε ο μέσος όρος των τιμών. Ο κώδικας που ακολουθεί δεν είναι ο πλήρης κώδικας, καθώς κομμάτια όπως ο ορισμός των dataframes, ή το ξεκίνημα ενός spark session δεν περιλαμβάνονται για εξοικονόμηση χώρου. Για τον πλήρη κώδικα υπάρχει στην αρχή του εγγράφου το link στο github repository.

### Κώδικας

-----Spark Dataframe-----

```
crime_df = crime_df.filter(col("Premis Desc") == "STREET")

crime_df = crime_df.withColumn("day_part",
                                when((crime_df['TIME OCC'] >= 500)
    & (crime_df['TIME OCC'] <1200), 'Morning')
                                .when((crime_df['TIME OCC'] >= 1200)
    & (crime_df['TIME OCC'] <1700), 'Afternoon')
                                .when((crime_df['TIME OCC'] >= 1700)
    & (crime_df['TIME OCC'] <2100), 'Evening')
                                .otherwise('Night'))

# Group by day_part and count the crimes
result =
crime_df.groupBy("day_part").agg(count("*").alias("crime_count"))
result = result.select('day_part',
    'crime_count').orderBy('crime_count', ascending=False)
```

## -----Spark RDD-----

```
# Assuming header is the first line of the file containing column
names
header = crime_rdd.first()

#filtered out the rows where row[15] (Premis Desc) is not STREET
filtered_rdd = crime_rdd.filter(lambda row: row[15] == "STREET")

#create a new column 'day_part' based on row[3] (Time OCC)
def map_to_day_part(row):
    time_occ = row[3] # Assuming 'TIME OCC' is at index 3
    if 500 <= int(time_occ) < 1200:
        return ('Morning', 1)
    elif 1200 <= int(time_occ) < 1700:
        return ('Afternoon', 1)
    elif 1700 <= int(time_occ) < 2100:
        return ('Evening', 1)
    else:
        return ('Night', 1)

#apply the mapping function and reduce by key to count crimes in
each 'day_part'
result_rdd = (filtered_rdd
               .map(map_to_day_part)
               .reduceByKey(lambda x, y: x + y)
               .sortBy(lambda x: x[1], ascending=False))
```

## Αποτελέσματα

### -----Spark Dataframe-----

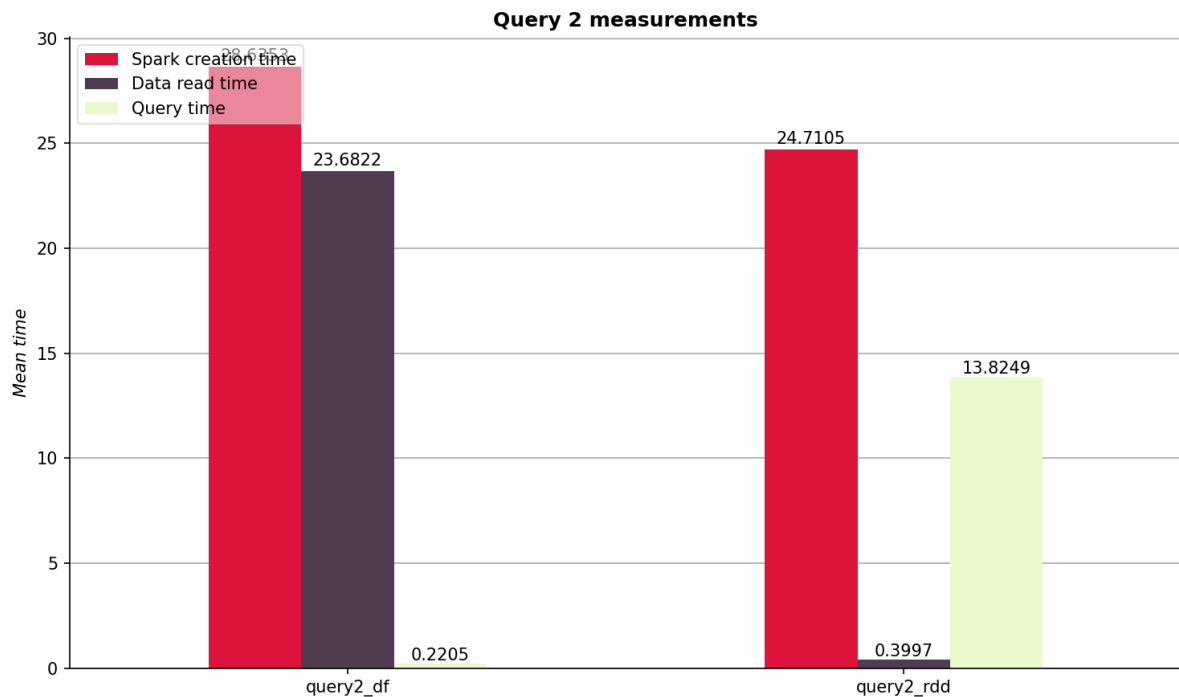
day_part	crime_count
Night	237605
Evening	187306
Afternoon	148180
Morning	123846

### -----Spark RDD-----

_1	_2
Night	237605
Evening	187306
Afternoon	148180
Morning	Morning



## Χρόνοι εκτέλεσης



Εδώ, σε αντίθεση με το query 1, όπου ανάμεσα στα 2 implementations υπήρχαν αμελητέες διαφορές στην απόδοση, εδώ είναι ξεκάθαρη η διαφορά, τόσο στο data read time, όσο και στο query processing time. Το RDD αντιπροσωπεύουν ένα lower-level of abstraction για τα δεδομένα, το οποίο είναι ιδανικό για διαχείριση unstructured δεδομένων. Ωστόσο, δεν χρησιμοποιεί τον Catalyst optimization, όπως το Dataframe, το οποίο οδηγεί και σε κατα πολύ αυξημένο χρόνο εκτέλεσης του query.

Όσον αφορά τη διαφορά στο data read time, αυτό οφείλεται στο `inferschema = True` πεδίο στο διάβασμα του csv αρχείου του, το οποίο είναι ένα πολύ ακριβό operation, operation που δεν χρησιμοποιείται στο RDD.

### Query 3

Σημείωση: Για την λήψη όσο πιο έγκυρων μετρήσεων γίνεται, το query εκτελέστηκε 5 φορές και για τους timers πάρθηκε ο μέσος όρος των τιμών. Τέλος, ο κώδικας που ακολουθεί δεν είναι ο πλήρης κώδικας, καθώς κομμάτια όπως ο ορισμός των dataframes, ή το ξεκίνημα ενός spark session δεν περιλαμβάνονται για εξοικονόμηση χώρου. Για τον πλήρη κώδικα υπάρχει στην αρχή του εγγράφου το link στο github repository.

#### Κώδικας

```
# Filter out rows where "Vict Descent" is not empty in the
crime_data csv
crime_df = crime_df.filter(crime_df["Vict Descent"].isNotNull())

# Filter data for the year 2015
crime_2015_df = crime_df.filter(year("DATE OCC") == 2015)

# Rename the "ZIPcode" column in coordinates_df to "Zip Code"
revgeo_df = revgeo_df.withColumnRenamed("ZIPcode", "Zip Code")

# Define a regular expression pattern to extract the first set of
digits
pattern = r'^(\d+)'

# Apply the regexp_extract function to the "zip_code" column
revgeo_df = revgeo_df.withColumn(
    "Zip Code",
    regexp_extract(col("Zip Code"), pattern, 1)
)

# Perform a left semi-join based on "Zip Code" column,
if hint_type != 'None' and mode != 'None':
    real_incomes_df = incomes_df.join(revgeo_df.hint(hint_type),
    "Zip Code", "left_semi")
    real_incomes_df.explain(mode = mode)
else:
    real_incomes_df = incomes_df.join(revgeo_df, "Zip Code",
    "left_semi")

# Select necessary columns for both top and bottom salary
DataFrames
top_zipcodes_df = real_incomes_df.orderBy(col("Estimated Median
Income").desc()).limit(3).select("Zip Code")
```

```

bottom_zipcodes_df = real_incomes_df.orderBy(col("Estimated Median
Income")).limit(3).select("Zip Code")

# Combine the two DataFrames into one
zipcodes_df = top_zipcodes_df.union(bottom_zipcodes_df)

# Perform a left semi-join based on "Zip Code" column
#filtered_revgeo_df = revgeo_df.join(zipcodes_df, "Zip Code",
"left_semi")
if hint_type != 'None' and mode != 'None':
    filtered_revgeo_df =
revgeo_df.join(zipcodes_df.hint(hint_type), "Zip Code",
"left_semi")
    filtered_revgeo_df.explain(mode = mode)
else:
    filtered_revgeo_df = revgeo_df.join(zipcodes_df, "Zip Code",
"left_semi")

# Left semi join between crime_2015_df and filtered_revgeo_df
based on Latitude and Longitude
#result_df = crime_2015_df.join(filtered_revgeo_df, ["LAT",
"LON"], "left_semi")
if hint_type != 'None' and mode != 'None':
    result_df =
crime_2015_df.join(filtered_revgeo_df.hint(hint_type), ["LAT",
"LON"], "left_semi")
    result_df.explain(mode = mode)
else:
    result_df = crime_2015_df.join(filtered_revgeo_df, ["LAT",
"LON"], "left_semi")

# Group by "Vict Descent" and count the number of victims
victim_counts_df = result_df.groupBy("Vict
Descent").agg(count("*").alias("Victim Count"))

# Order the result in descending order based on "Victim Count"
victim_counts_df = victim_counts_df.orderBy(desc("Victim Count"))

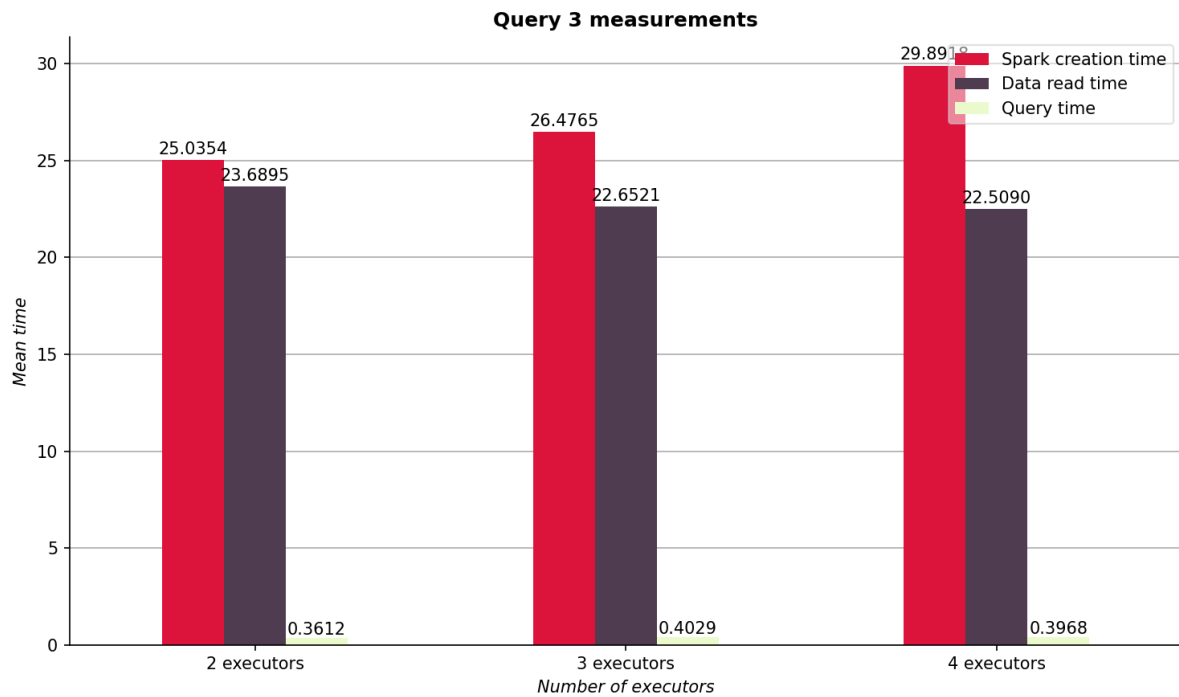
```

### *Αποτελέσματα*

-----Spark Dataframe-----

Vict Descent	Victim Count
White	2219
Other	543
Hispanic/Latin/Mexican	418
Unknown	281
Black	250
Other Asian	125
Filipino	3
Japanese	1

### Χρόνοι εκτέλεσης



Όπως παρατηρούμε από το plot των χρόνων για το query 3, όλοι οι χρόνοι είναι αρκετά παρόμοιοι. Παρατηρώντας τις επόμενες δύο εικόνες, οι οποίες καταγράφουν τα tasks ανά Job που εκτελεί το spark, βλέπουμε ότι δεν εκτελείται παράλληλα σχεδόν ποτέ πάνω από 1 task. Κάποια jobs έχουν 6 tasks τα οποία όμως εκτελούνται αρκετά γρήγορα ήδη, οπότε με το να προσθέσουμε παραπάνω executors δεν θα επηρεάσει ιδιαίτερα την απόδοση. Το όποιο κέρδος παίρνουμε με την παραλληλοποίηση, χάνεται λόγω το overhead για την πρόσβαση στη μνήμη και τη μεταφορά δεδομένων.

*Spark history tasks*

Tasks (for all stages): Succeeded/Total	
1/1	6/6
	1/1
3/3	2/2
1/1	1/1
1/1	1/1
2/2	1/1
1/1	1/1
1/1	1/1
1/1	1/1
1/1 (6 skipped)	6/6
	1/1

## Query 4

Σημείωση: Ο κώδικας που ακολουθεί δεν είναι ο πλήρης κώδικας, καθώς κομμάτια όπως ο ορισμός των dataframes, ή το ξεκίνημα ενός spark session δεν περιλαμβάνονται για εξοικονόμηση χώρου. Για τον πλήρη κώδικα υπάρχει στην αρχή του εγγράφου το link στο github repository.

### Κώδικας

#### -----Query 4.1-----

```
# calculate the distance between two points [lat1,long1],
[lat2,long2] in km
@F.udf(DoubleType())
def get_distance (lat1, long1, lat2, long2) :
    return geodesic((lat1,long1),(lat2,long2)).km

#filter out rows where LAT or LON is equal to 0
crime_df = crime_df.filter((col("LAT") != 0) & (col("LON") != 0))

#count crimes where "Weapon Used Cd" starts with 1, in order to
keep crimes
#where the weapon was a firearm
crimes_with_firearms = crime_df.filter(col("Weapon Used
Cd").startswith("1"))

'''
Note: There is a difference in how the two datasets refer to some
divisions
This is why these division names were manually changed to match so
that data
would not be lost during the join operation
'''
crimes_with_firearms= crimes_with_firearms.withColumnRenamed('AREA
NAME', 'DIVISION')
crimes_with_firearms = crimes_with_firearms.withColumn(
    "DIVISION",
    when(col("DIVISION") == 'N Hollywood', 'NORTH HOLLYWOOD')\
    .when(col("DIVISION") == 'West LA', 'WEST LOS ANGELES')\
    .otherwise(col("DIVISION"))
)
crimes_with_firearms = crimes_with_firearms.withColumn("DIVISION",
F.upper(col("DIVISION")))
```

```

#crimes_with_firearms.select('DIVISION').distinct().show()

#keep the "X", "Y", "DIVISION" columns as these are the relevant
columns for this query
## This is done to save memory and time (due to the cross join
that follows)
police_dpt_df = police_dpt_df.select("DIVISION", "X", "Y")
#police_dpt_df.select('DIVISION').distinct().show()

#cross join the two dataframes
if hint_type != 'None' and mode != 'None':
    joined_df =
crimes_with_firearms.join(police_dpt_df.hint(hint_type), 'DIVISION'
, 'left')
    joined_df.explain(mode = mode)
else:
    joined_df = crimes_with_firearms.join(police_dpt_df,
'DIVISION', 'left')

joined_df = joined_df.withColumn("distance", get_distance('LAT',
'LAT', 'LON', 'Y', 'X'))

# Find nearest police department for each crime and keep that
nearest_place_df = joined_df.groupBy("DR_NO").agg(
    F.min("distance").alias("min_distance"),
    F.first("DIVISION").alias("nearest_place")
)

if hint_type != 'None' and mode != 'None':
    result_df = joined_df.join(nearest_place_df.hint(hint_type),
"DR_NO", "left")
    result_df.explain(mode = mode)
else:
    result_df = joined_df.join(nearest_place_df, "DR_NO", "left")

#result_df.select("min_distance").describe().show()

# Group by 'year' and calculate the count and average distance
avg_per_year_df = result_df.groupBy(year("DATE
OCC")).alias("year")).agg(
    F.mean("min_distance").alias("average_distance"),

```



```

    F.count("*").alias("crime_count")
).orderBy("year")

avg_per_year_df.show()

# Calculate average distance and count of crimes per police
department
avg_per_division_df = result_df.groupBy("nearest_place").agg(
    F.count("*").alias("crime_count"),
    F.mean("min_distance").alias("average_distance")
).orderBy("crime_count", ascending=False)

avg_per_division_df.show()

```

#### -----Query 4.2-----

```

# calculate the distance between two points [lat1,long1],
[lat2,long2] in km
@F.udf(DoubleType())
def get_distance (lat1, long1, lat2, long2) :
    return geodesic((lat1,long1),(lat2,long2)).km

#filter out rows where LAT or LON is equal to 0
crime_df = crime_df.filter((col("LAT") != 0) & (col("LON") != 0))

#count crimes where "Weapon Used Cd" starts with 1, in order to
keep crimes
#where the weapon was a firearm
crimes_with_firearms = crime_df.filter(col("Weapon Used
Cd").startswith("1"))

...

Note: There is a difference in how the two datasets refer to some
divisions
This is why these division names were manually changed to match so
that data
would not be lost during the join operation
...

crimes_with_firearms = crimes_with_firearms.withColumn(
    "AREA NAME",
    when(col("AREA NAME") == 'N Hollywood', 'NORTH
HOLLYWOOD')\

```

```

        .when(col("AREA NAME") == 'West LA', 'WEST LOS ANGELES')\
        .otherwise(col("AREA NAME"))
    )
    crimes_with_firearms = crimes_with_firearms.withColumn("AREA
NAME", F.upper(col("AREA NAME")))
#crimes_with_firearms.select('DIVISION').distinct().show()

#keep the "X", "Y", "DIVISION" columns as these are the relevant
columns for this query
## This is done to save memory and time (due to the cross join
that follows)
police_dpt_df = police_dpt_df.select("X", "Y", "DIVISION")

#cross join the two dataframes
if hint_type != 'None' and mode != 'None':
    joined_df =
    crimes_with_firearms.crossJoin(police_dpt_df.hint(hint_type))
    joined_df.explain(mode = mode)
else:
    joined_df = crimes_with_firearms.crossJoin(police_dpt_df)

#calculate distance for each pair of crime location and police
department
joined_df = joined_df.withColumn("distance",
get_distance('LAT', 'LON', 'Y', 'X'))

#define a window to rank distances for each crime
window_spec = Window.partitionBy("DR_NO").orderBy("distance")

#rank the distances for each crime
ranked_df = joined_df.withColumn("rank",
F.row_number().over(window_spec))

#filter for rows where rank is 1 (minimum distance)
nearest_place_df = ranked_df.filter("rank = 1").select("DR_NO",
"DIVISION", "distance")

if hint_type != 'None' and mode != 'None':
    result_df =
    crimes_with_firearms.join(nearest_place_df.hint(hint_type),
"DR_NO", "left")
    result_df.explain(mode = mode)

```

```

else:
    result_df = crimes_with_firearms.join(nearest_place_df,
"DR_NO", "left")

#Group by 'year' and calculate the mean distance and count of rows
avg_per_year_df = result_df.groupBy(year("DATE
OCC").alias("year")).agg(
    F.mean("distance").alias("average_distance"),
    F.count("*").alias("crime_count")
).orderBy("Year")

avg_per_year_df.show()

# Calculate average distance and count of crimes per police
department
avg_per_division_df = result_df.groupBy("DIVISION").agg(
    F.mean("distance").alias("average_distance"),
    F.count("*").alias("crime_count")
).orderBy("crime_count", ascending=False)

avg_per_division_df.show()

```

## Αποτελέσματα

### -----Query 4.1.a-----

year	average_distance	crime_count
2010	2.783223420612492	8212
2011	2.792627779940297	7232
2012	2.835754848654567	6532
2013	2.826132495314073	5838
2014	2.772898516477533	4526
2015	2.7058134773177893	6763
2016	2.717126699483445	8100
2017	2.723956102818232	7786
2018	2.73230651155873	7413
2019	2.7394299000869644	7129
2020	2.6897964608047538	8487
2021	2.6918773938364216	12324
2022	2.608112987208764	10025
2023	2.5475714817765347	8896

-----Query 4.1.b-----

nearest_place	crime_count	average_distance
77TH STREET	16567	2.697221022730226
SOUTHEAST	12917	2.1027902716277453
NEWTON	9617	2.0146354569413476
SOUTHWEST	8641	2.699419915321537
HOLLENBECK	6113	2.6491807867280888
HARBOR	5444	4.08369941390368
RAMPART	4998	1.578602693862525
MISSION	4463	4.71750031436258
OLYMPIC	4336	1.833579483721724
FOOTHILL	3943	3.803698582865336
NORTHEAST	3848	3.905631374560966
HOLLYWOOD	3560	1.454911784758709
CENTRAL	3485	1.1352127099684708
WILSHIRE	3428	2.3208496365459346
NORTH HOLLYWOOD	3394	2.719621897901707
WEST VALLEY	2789	3.525605238376684
VAN NUYS	2649	2.2148398439548074
PACIFIC	2646	3.732975873043782
DEVONSHIRE	2602	4.019167427465698
TOPANGA	2313	3.4792441420192675
WEST LOS ANGELES	1510	4.263889486464344

-----Query 4.2.a-----

year	average_distance	crime_count
2010	2.4342351310730477	8212
2011	2.461005078431005	7232
2012	2.505525574337147	6532
2013	2.4555437568989573	5838
2014	2.3879294419713832	4526
2015	2.387261320024826	6763
2016	2.428195035737651	8100
2017	2.391618932774622	7786
2018	2.4082079737438558	7413
2019	2.4294088109777476	7129
2020	2.383615837920006	8487
2021	2.40694993550755	12324
2022	2.312096005221259	10025
2023	2.2659285230178985	8896

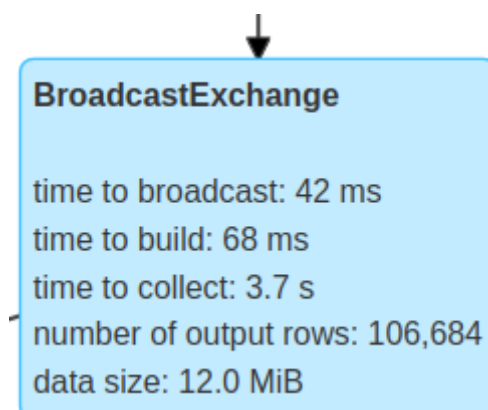
-----Query 4.2.b-----

DIVISION	average_distance	crime_count
77TH STREET	1.7215559668318008	13489
SOUTHEAST	2.1955986195059825	11816
SOUTHWEST	2.279343343133375	11209
NEWTON	1.5693029438285195	7161
WILSHIRE	2.4446505649478225	6253
HOLLENBECK	2.636843036493343	6174
OLYMPIC	1.665470250718678	5415
HOLLYWOOD	2.0086559103864285	5378
HARBOR	3.8982379354204326	5322
RAMPART	1.3978586849187002	4700
FOOTHILL	3.6003578272454986	4693
VAN NUYS	2.9734701269529067	4673
CENTRAL	1.017698301999662	3584
NORTH HOLLYWOOD	2.745580642931433	3389
NORTHEAST	3.7555638718187887	3096
MISSION	3.806759164814483	2853
WEST VALLEY	2.7933743074835	2755
PACIFIC	3.7014554679410585	2522
TOPANGA	3.0519644506900643	2435
DEVONSHIRE	2.9846027225677347	1332
WEST LOS ANGELES	2.768471576203468	1014

## Join hint and explain

### Broadcast Hash Join

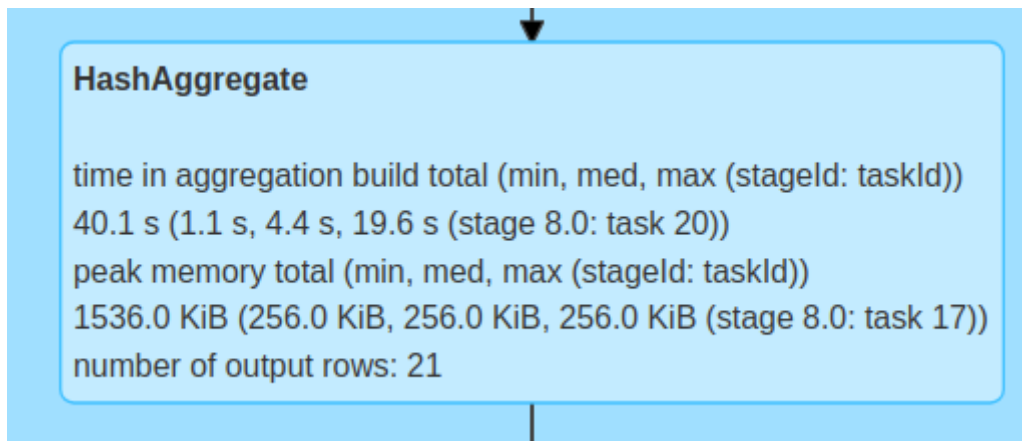
Θα εξετάσουμε την απόδοση των τριών query, του 4.2, του 4.1 και του 3. Το 4.1 θα το εξετάσουμε σε 2 διαφορετικές εκτελέσεις καθώς παρατηρήθηκε διαφορά στους χρόνους που αυτό έτρεξε, εξηγώντας τον λόγο για την διαφορά αυτή. Γενικότερα για να είναι αποδοτικό το broadcast hash join, απαιτείται το dataframe το οποίο θα γίνει broadcasted να είναι σχετικά μικρό, και επομένως να χωράει στη μνήμη του κάθε executor. Το spark στα configurations του έχει ένα standard threshold της τάξης των 10 mb. Πλοηγούμαστε στο spark ui, στο sql/dataframe tab, στο ακριβότερο απο τα queries. Εκεί παρατηρούμε τον physical operator BroadcastExchange, ο οποίος μας λέει πληροφορίες για το broadcasted dataframe. Για το query 3 βλέπουμε ότι το μέγιστο μέγεθος dataframe που γίνεται broadcasted είναι 4mb, και για τα 4.1, 4.2 είναι 12 mb, τα οποία είναι estimated μεγέθη ικανοποιητικά ώστε να γίνουν broadcasted.



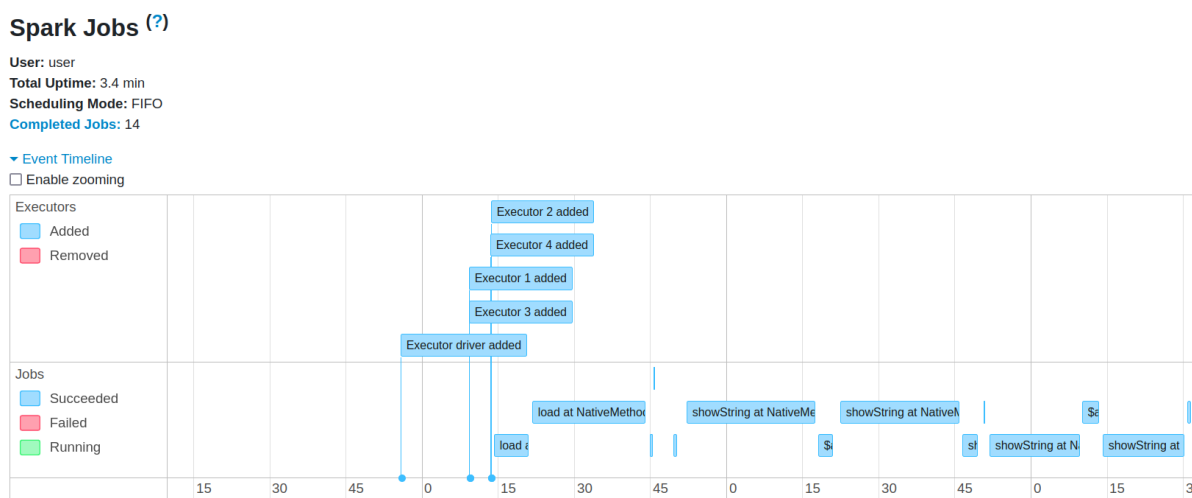
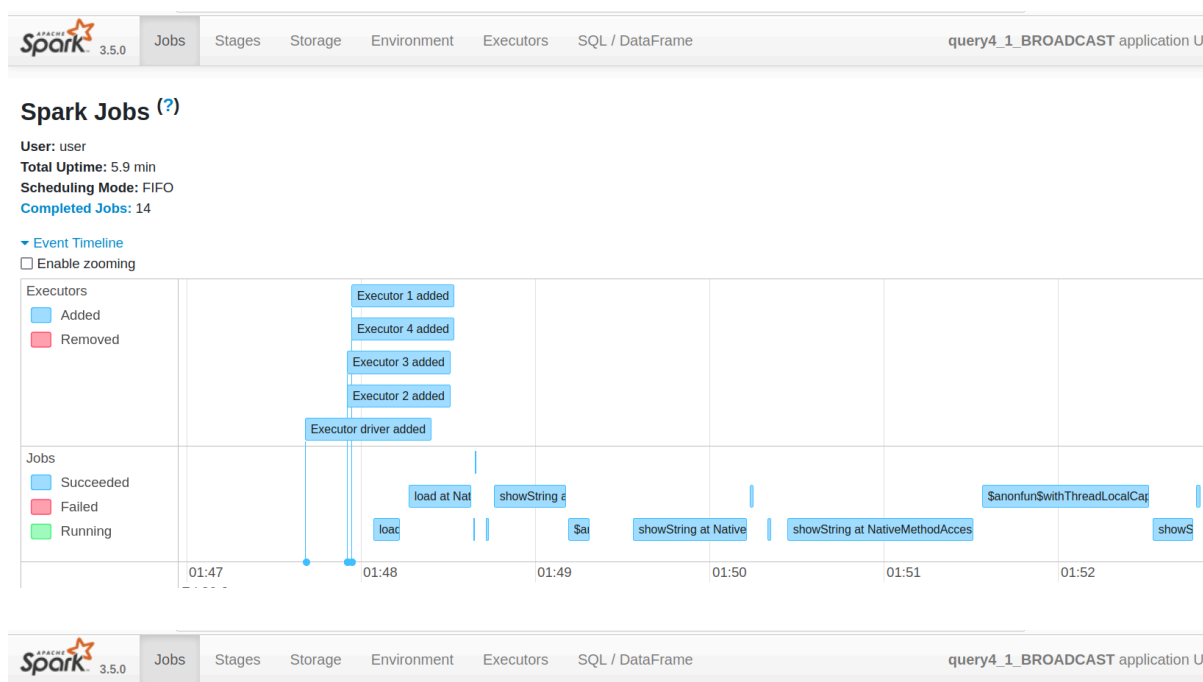
Παραπάνω βλέπουμε τον physical operator BroadcastExchange για το 4.2, για το ακριβότερο query στο sql/dataframe tab όπως ακριβώς εξηγήσαμε και παραπάνω. Βλέπουμε λοιπόν το estimated data size το οποίο είναι 12 mb.

Παρ'όλα αυτά παρατηρούμε αντίστοιχα στα ίδια DAG τον physical operator HashAggregate, ο οποίος μας δίνει πληροφορίες για το χρόνο τον οποίο χρειάζεται για να δημιουργηθεί τοπικά σε κάθε executor το hash table. Παρατηρούμε πως για όλα τα query ο χρόνος που απαιτείται δεν είναι αμελητέος, κάτι το οποίο είναι λογικό καθώς γενικότερα η διαδικασία του hashing απαιτεί χρόνο.

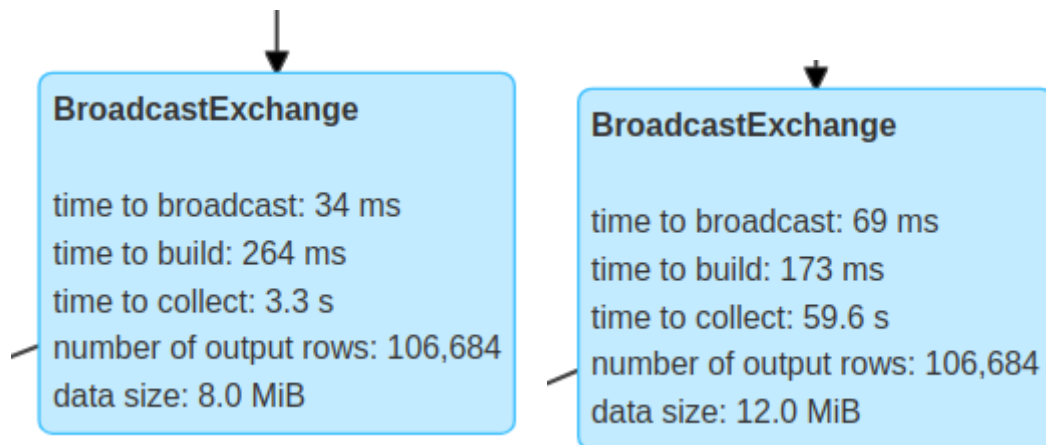




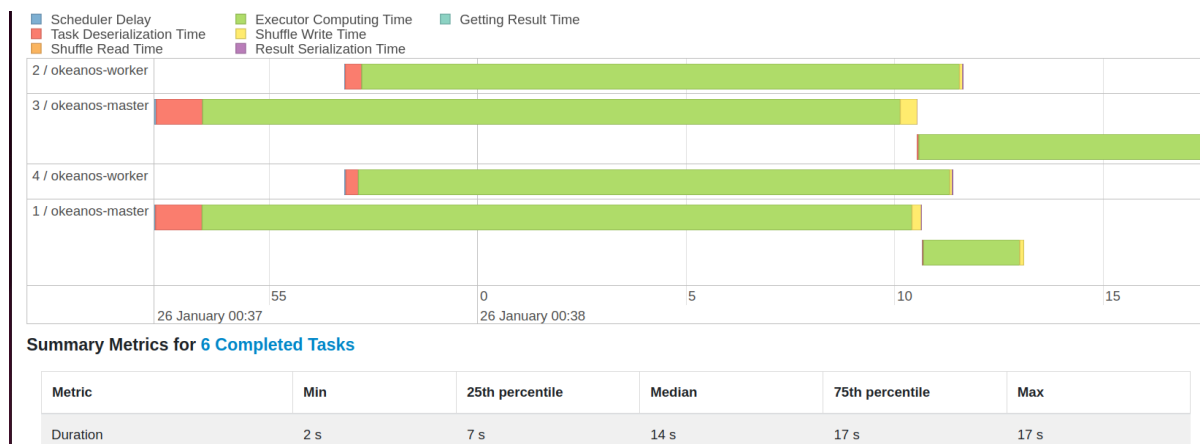
Αντίστοιχα βλέπουμε τον physical operator HashAggregate στο ίδιο dag.

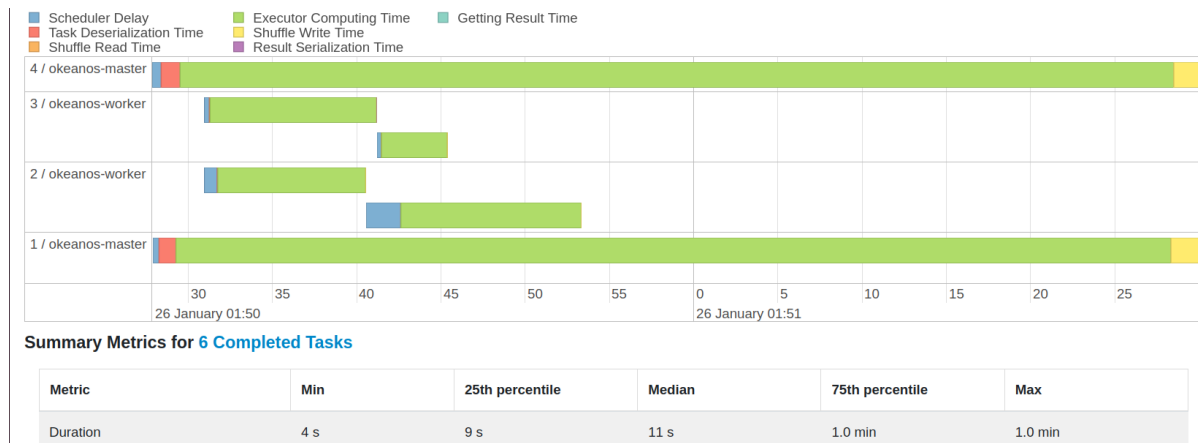


Παραπάνω βλέπουμε τις 2 διαφορετικές εκτελέσεις για το query 4.1, και τη διαφορά που έχουν αυτές στο Total Uptime. Αυτό συμβαίνει λόγω skewed data, τα οποία παρατηρούνται μέσω του spark ui. Αυτό σημαίνει πως τα δεδομένα μας δεν είναι ομοιόμορφα κατανομημένα μεταξύ των executors, και ότι κάποιοι executors έχουν λάβει σαν είσοδο μεγαλύτερα partitions σε σχέση με άλλους, άρα κάποια tasks θα έχουν μεγαλύτερο data load.



Αρχικά παρατηρούμε, ενώ είμαστε στο ίδιο tab που περιγράφηκε και στην αρχή, πως το time to collect της εκτέλεσης που διαρκεί παραπάνω είναι υπερβολικά αυξημένο σε σχέση με της εκτέλεσης των 3.4 min. Αυτό δεν αποτελεί την απόδειξη για τα skewed data, παρά μόνο μια ακόμα ένδειξη. Στη συνέχεια δείχνουμε την απόδειξη. Πλοηγούμαστε στο stages tab. Εκεί επιλέγουμε για την κάθε εκτέλεση το stage με το μεγαλύτερο duration. Έτσι έχουμε τα εξής:





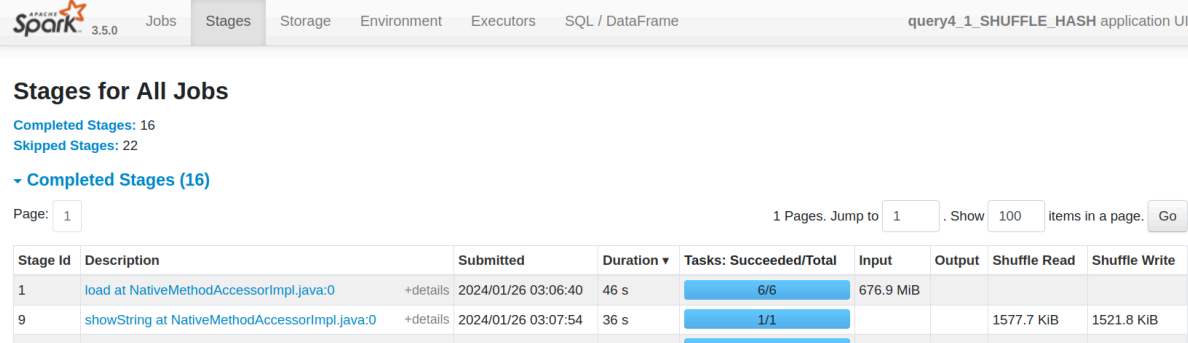
Πλέον το data skewness είναι προφανές και από το event timeline αλλά και από το duration (που βρίσκεται κάτω κάτω) και τις διαφορές που υπάρχουν στη 2η εικόνα, που είναι και η εκτέλεση η οποία έχει skewed data. Εδώ βλέπουμε τη διαφορά που υπάρχει στο Median και στο Max, δηλαδή ότι υπάρχει ένα τουλάχιστον task το οποίο διαρκεί ένα λεπτό, ενώ τα περισσότερα tasks διαρκούν 11 sec κατά μέσο όρο. Παράλληλα βλέπουμε πως οι executors 1, 4 είναι πολύ περισσότερο απασχολημένοι σε σχέση με τους 2, 3. Αντίστοιχα στην από πάνω εκτέλεση παρατηρούμε πως όλοι οι executors είναι εξίσου απασχολημένοι και οι χρόνοι στο Duration, οι Median, Max είναι σχεδόν ίδιοι, δηλαδή ότι το ακριβότερο task διαρκεί όσο διαρκεί και ο μέσος όρος των task όσον αφορά τον χρόνο. Σε σύγκριση με τα υπόλοιπα είδη join, βλέπουμε ότι το συγκεκριμένο έχει λίγο καλύτερη επίδοση στο 4.1, μαζί με το merge join, από ότι έχουν τα άλλα. Στα υπόλοιπα 2 query πετυχαίνει πανομοιότυπους χρόνους με τα άλλα. Επομένως καταλήγουμε ότι όταν τα partitions είναι ομοιόμορφα κατανομημένα μεταξύ των executors, άρα όταν δεν υπάρχει data skewness, το Broadcast Hash Join πετυχαίνει καλή επίδοση.

Σημείωση: Μιας και δεν αναγραφόταν στην εκφώνηση, οπότε έβγαине εκτός score του πρότζεκτ αυτού, τα cores ανά executors είχαν παντού την ίδια τιμή, δηλαδή 1 (όπως δόθηκε στο spark default config file). Αυτό, ωστόσο, δεν βοηθάει τις εφαρμογές στην επίδοσή τους.

## Shuffle Hash, Shuffle Sort Merge Join


Το συγκεκριμένο είδος join, είναι κατάλληλο όταν και τα 2 tables είναι μεγάλα σε μέγεθος, σε αντίθεση με το Broadcast Hash Join. Παρ'ότι το shuffle hash θεωρείται ένα όχι και τόσο αποδοτικό join, βλέπουμε ότι τα πάει καλά σε επίδοση (εκτός από το 4.1), ενώ σε κάποια έχει ακόμα και την καλύτερη επίδοση με διαφορά μερικών sec. Θεωρείται από τα ακριβότερα είδη join, λόγω των 2 ακριβών φάσεων που έχει, το shuffle και το hash. Επιπλέον και τα 2 αυτά είδη join απαιτούν επικοινωνία μεταξύ των executors, κάτι το οποίο το broadcast hash join το παρέκαμπτε. Ακόμη, για να γίνει merge join, απαιτείται τα joinkeys να είναι sortable, κάτι το οποίο δε χρειάζεται προφανώς στο shuffle hash join.

Ας μιλήσουμε αρχικά για το query 4.1 και το shuffle hash join, το οποίο παρατηρούμε πως σε σύγκριση με τα υπόλοιπα δεν έχει τόσο καλή επίδοση. Η απόδοση αυτών των 2 μεθόδων join παρατηρούμε ότι έγκειται στον αριθμό των data partitions που θα καθορίσει το spark ότι χρειάζεται για το aqeShuffleRead. Υπάρχουν stages στα οποία γίνεται shuffle read, άλλα τα data δε γίνονται καν partitioned, επομένως έχουμε ένα task να τρέχει σε έναν executor, κάτι το οποίο είναι ένδειξη του χειριστου data skew, και πετυχαίνει κακή επίδοση. Όταν έχουμε έστω και 2 data partitions, επομένως 2 tasks να τρέχουν σε 2 executors, πετυχαίνουμε παραλληλισμό επομένως τα πράγματα τότε είναι καλύτερα.



Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	load at NativeMethodAccessorImpl.java:0	2024/01/26 03:06:40	46 s	6/6	676.9 MIB			
9	showString at NativeMethodAccessorImpl.java:0	2024/01/26 03:07:54	36 s	1/1			1577.7 KiB	1521.8 KiB
26	showString at NativeMethodAccessorImpl.java:0	2024/01/26 03:09:09	27 s	1/1			1577.7 KiB	1968.0 KiB

Όπως βλέπουμε παραπάνω το 2ο και το 3ο ακριβότερο stage για το query 4\_1 δεν γίνονται τα data τους partitioned, και αναλαμβάνει το ένα task ένας executor. Το ίδιο συμβαίνει και με το shuffle sort merge, το οποίο παρ'ότι πετυχαίνει πολύ καλή επίδοση, τα 2 ακριβότερα stages του είναι τα shuffle read στα οποία ξανά όπως και πριν, τα 1,5 MB data δε γίνονται partitioned και βλέπουμε παρακάτω τα αποτελέσματα:



3.5.0

Jobs

Stages

Storage

Environment

Executors

SQL / DataFrame

query4\_1\_SHUFFLE\_HASH application UI

## Stages for All Jobs

Completed Stages: 16

Skipped Stages: 22

▼ Completed Stages (16)

Page: 1

1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id	Description	Submitted	Duration ▼	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
26	<a href="#">showString at NativeMethodAccessorImpl.java:0</a> <a href="#">+details</a>	2024/02/02 19:55:10	24 s	1/1			1577.7 KiB	1968.0 KiB
9	<a href="#">showString at NativeMethodAccessorImpl.java:0</a> <a href="#">+details</a>	2024/02/02 19:54:35	23 s	1/1			1577.7 KiB	1521.9 KiB
1	<a href="#">load at NativeMethodAccessorImpl.java:0</a> <a href="#">+details</a>	2024/02/02 19:54:05	16 s	6/6	676.9 MiB			
4	<a href="#">showString at NativeMethodAccessorImpl.java:0</a> <a href="#">+details</a>	2024/02/02 19:54:24	7 s	6/6	676.9 MiB			1215.9 KiB
21	<a href="#">showString at NativeMethodAccessorImpl.java:0</a> <a href="#">+details</a>	2024/02/02 19:55:00	6 s	6/6	676.9 MiB			655.1 KiB
5	<a href="#">showString at NativeMethodAccessorImpl.java:0</a> <a href="#">+details</a>	2024/02/02 19:54:24	5 s	6/6	676.9 MiB			1576.0 KiB
22	<a href="#">showString at NativeMethodAccessorImpl.java:0</a> <a href="#">+details</a>	2024/02/02 19:55:00	5 s	6/6	676.9 MiB			1576.0 KiB
0	<a href="#">load at NativeMethodAccessorImpl.java:0</a> <a href="#">+details</a>	2024/02/02 19:54:02	2 s	1/1	64.0 KiB			
14	<a href="#">showString at NativeMethodAccessorImpl.java:0</a> <a href="#">+details</a>	2024/02/02 19:54:58	1 s	2/2			2.7 MiB	2.1 KiB
31	<a href="#">showString at NativeMethodAccessorImpl.java:0</a> <a href="#">+details</a>	2024/02/02 19:55:34	0.9 s	2/2			2.6 MiB	3.4 KiB

3.5.0
Jobs
Stages
Storage
Environment
Executors
SQL / DataFrame
query4\_1\_SHUFFLE\_HASH application UI

## Details for Stage 9 (Attempt 0)

**Resource Profile Id:** 0  
**Total Time Across All Tasks:** 34 s  
**Locality Level Summary:** Node local: 1  
**Shuffle Read Size / Records:** 1577.7 KiB / 109284  
**Shuffle Write Size / Records:** 1521.8 KiB / 106684  
**Associated Job Ids:** [7](#)

▼ DAG Visualization

```

graph TD
    A["AQEShuffleRead  
ShuffledRowRDD [32] [Unordered]  
showString at NativeMethodAccessorImpl.java:0"] --> D["WholeStageCodegen (4)"]
    B["AQEShuffleRead  
ShuffledRowRDD [33] [Unordered]  
showString at NativeMethodAccessorImpl.java:0"] --> D
  
```

Καταλήγοντας για αυτά τα 2 είδη join, η επίδοση τους όταν γίνονται partitioned τα data στο shuffle read stage είναι πολύ καλή, ενώ όταν δεν γίνεται αυτό η επίδοση τους είναι ίδια με τις άλλες μεθόδους join, είτε χειρότερη. Το 4\_2 αποτελεί εξαίρεση καθώς εκεί όλα τα query έχουν πανομοιότυπες επιδόσεις ενώ τα data γίνονται partitioned στο shuffle read stage για τα 2 είδη join τα οποία μας απασχολούν.

## Shuffle\_Replicate\_Nested\_Loop\_Join or Cartesian Join

Το συγκεκριμένο είδος join φαίνεται να είναι το πιο απρόβλεπτο απ' όλα. Η λογική του είναι ότι τα partition του dataset που έχει ο κάθε executor γίνονται replicate σε όλους τους υπόλοιπους executors, επομένως γίνεται all to all communication. Κάτι τέτοιο προφανώς έχει αρκετό ρίσκο για data skew στο dataset, εξού και η μη προβλεψιμότητα. Πχ για το 4\_1 το εκτελούμε τη μια φορά και τρέχει σε 1.5 min και την επόμενη τρέχει σε 5.5 min.

Page: 1

1 Pages. Jump to 1

Show 100 items in a page. Go

Stage Id	Description		Submitted	Duration ▾	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
16	showString at NativeMethodAccessorImpl.java:0	+details	2024/01/26 02:05:56	51 s	6/6	674.9 MiB			2.2 MiB
6	showString at NativeMethodAccessorImpl.java:0	+details	2024/01/26 02:04:14	36 s	6/6	674.9 MiB			1658.0 KiB
18	\$anonfun\$withThreadLocalCaptured\$1 at FutureTask.java:264	+details	2024/01/26 02:06:57	30 s	2/2			2.2 MiB	
1	load at NativeMethodAccessorImpl.java:0	+details	2024/01/26 02:03:41	22 s	6/6	676.9 MiB			
15	showString at NativeMethodAccessorImpl.java:0	+details	2024/01/26 02:05:45	21 s	6/6	676.9 MiB			655.1 KiB
8	\$anonfun\$withThreadLocalCaptured\$1 at FutureTask.java:264	+details	2024/01/26 02:05:07	18 s	1/1			1658.0 KiB	
5	showString at NativeMethodAccessorImpl.java:0	+details	2024/01/26 02:04:10	12 s	6/6	676.9 MiB			1215.9 KiB
20	showString at NativeMethodAccessorImpl.java:0	+details	2024/01/26 02:07:36	10 s	1/1			655.1 KiB	1747.0 B
0	load at NativeMethodAccessorImpl.java:0	+details	2024/01/26 02:03:33	8 s	1/1	64.0 KiB			
14	\$anonfun\$withThreadLocalCaptured\$1 at FutureTask.java:264	+details	2024/01/26 02:05:45	4 s	1/1	1390.0 B			
10	showString at NativeMethodAccessorImpl.java:0	+details	2024/01/26 02:05:29	3 s	1/1			1215.9 KiB	1078.0 B
13	showString at NativeMethodAccessorImpl.java:0	+details	2024/01/26 02:05:38	3 s	1/1			1078.0 B	
4	\$anonfun\$withThreadLocalCaptured\$1 at FutureTask.java:264	+details	2024/01/26 02:04:10	0.8 s	1/1	1390.0 B			
23	showString at NativeMethodAccessorImpl.java:0	+details	2024/01/26 02:07:47	0.6 s	1/1			1747.0 B	
3	load at NativeMethodAccessorImpl.java:0	+details	2024/01/26 02:04:05	0.2 s	1/1	1390.0 B			
2	load at NativeMethodAccessorImpl.java:0	+details	2024/01/26 02:04:05	0.2 s	1/1	1390.0 B			

Page: 1

1 Pages. Jump to 1

Show 100 items in a page. Go

Page: 1

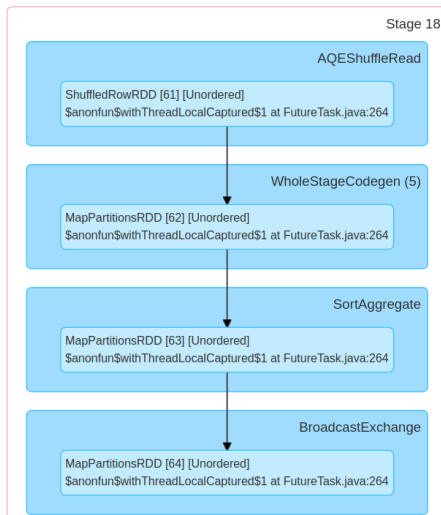
1 Pages. Jump to 1

Show 100 items in a page.

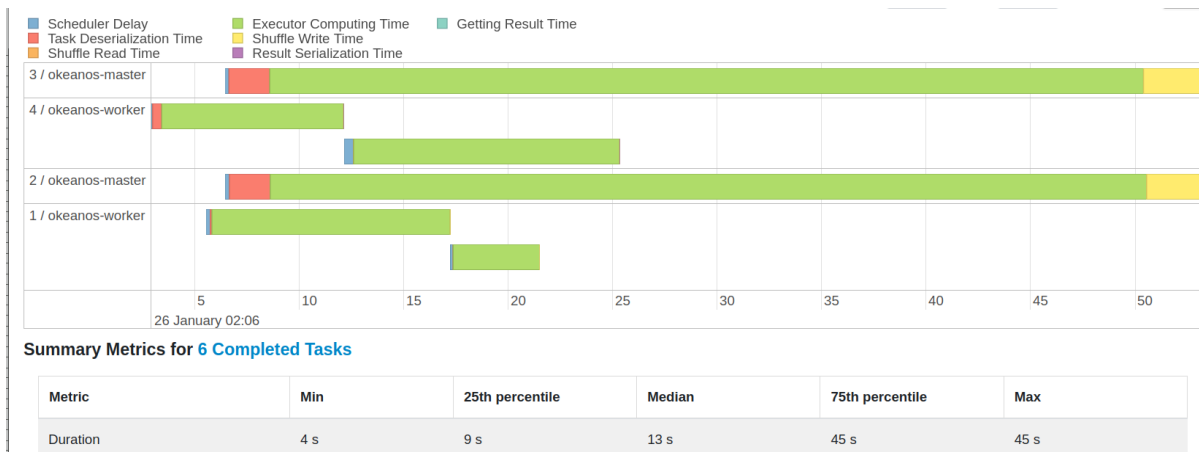
Go

Stage Id	Description		Submitted	Duration ▾	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
6	showString at NativeMethodAccessorImpl.java:0	+details	2024/02/02 19:56:33	14 s	6/6	674.9 MiB			1658.0 KiB
1	load at NativeMethodAccessorImpl.java:0	+details	2024/02/02 19:56:16	13 s	6/6	676.9 MiB			
16	showString at NativeMethodAccessorImpl.java:0	+details	2024/02/02 19:56:56	10 s	6/6	674.9 MiB			2.2 MiB
5	showString at NativeMethodAccessorImpl.java:0	+details	2024/02/02 19:56:32	8 s	6/6	676.9 MiB			1215.9 KiB
15	showString at NativeMethodAccessorImpl.java:0	+details	2024/02/02 19:56:55	5 s	6/6	676.9 MiB			655.1 KiB
0	load at NativeMethodAccessorImpl.java:0	+details	2024/02/02 19:56:13	3 s	1/1	64.0 KiB			
8	\$anonfun\$withThreadLocalCaptured\$1 at FutureTask.java:264	+details	2024/02/02 19:56:52	1.0 s	1/1			1658.0 KiB	
10	showString at NativeMethodAccessorImpl.java:0	+details	2024/02/02 19:56:53	0.8 s	1/1			1215.9 KiB	1078.0 B
18	\$anonfun\$withThreadLocalCaptured\$1 at FutureTask.java:264	+details	2024/02/02 19:57:10	0.6 s	2/2			2.2 MiB	
20	showString at NativeMethodAccessorImpl.java:0	+details	2024/02/02 19:57:11	0.5 s	1/1			655.1 KiB	1739.0 B
4	\$anonfun\$withThreadLocalCaptured\$1 at FutureTask.java:264	+details	2024/02/02 19:56:31	0.4 s	1/1	1390.0 B			
14	\$anonfun\$withThreadLocalCaptured\$1 at FutureTask.java:264	+details	2024/02/02 19:56:55	0.2 s	1/1	1390.0 B			
13	showString at NativeMethodAccessorImpl.java:0	+details	2024/02/02 19:56:54	0.2 s	1/1			1078.0 B	
23	showString at NativeMethodAccessorImpl.java:0	+details	2024/02/02 19:57:11	92 ms	1/1			1739.0 B	
3	load at NativeMethodAccessorImpl.java:0	+details	2024/02/02 19:56:30	82 ms	1/1	1390.0 B			
2	load at NativeMethodAccessorImpl.java:0	+details	2024/02/02 19:56:30	74 ms	1/1	1390.0 B			

Η πάνω φωτογραφία είναι για την costly εκτέλεση και η κάτω για την αποδοτική. Σε όσα stages της πάνω τρέχουν 6 tasks άλλα βλέπουμε μεγάλους χρόνους, έχουμε skewed data. Επιπλέον καθυστερούν αρκετά τα shuffle read stages, στα οποία δε γίνεται ακριβώς shuffle καθώς περιγράψαμε τον τρόπο που δουλεύει το συγκεκριμένο είδος join προηγουμένως



Παραπάνω βλέπουμε το stage του “shuffle”.



Εδώ βλέπουμε τα skewed data για το πιο κοστοβόρο stage της εκτέλεσης που διαρκεί 5.5 min, το stage της οποίας διαρκεί 51 sec.

Τα ίδια συμπεράσματα προκύπτουν και για το 3ο query.

### Συμπεράσματα

Παρατηρούμε πως για τα query 3, 4\_1 σε μη skewed data τα shuffle joins και το cartesian product join έχουν καλή επίδοση σε σύγκριση με το broadcast join. Παρ όλα αυτά αυτά τα join έχουν κάποιο ρίσκο για τη δημιουργία skewed data, το cartesian product έχει το περισσότερο ρίσκο απ όλα. Το broadcast hash join έχει το μικρότερο ρίσκο, και επιπλέον τις φορές που τα άλλα join δημιουργούν skewed data, το broadcast join έχει καλύτερη επίδοση, εφόσον τα data του δεν είναι skewed. Για το 4\_2 δε μπορούμε να βγάλουμε κάποιο συμπέρασμα, καθώς σε όλα τα είδη join δε παρατηρούνται skewed data, άλλα λόγω του cartesian join η εκτέλεση είναι εξίσου costly σε όλα.