Προχωρημένα Θέματα Βάσεων Δεδομένων
Αναφορά Εξαμηνιαίας Εργασίας

Ακαδημαϊκό Έτος 2023-24, 9ο Εξάμηνο

Στοιχεία ομάδας:

Team ID: 10
Αλέξανδρος Ιονίτσα (03119193) | alexandros.ionitsa@gmail.com
Εμμανουήλ Εμμανουηλίδης (03119435) | manos.emmanouilidis05@gmail.com
GitHub Repository

Τίτλος εργασίας: LA Crime Data Analysis using Apache Spark and Apache Hadoop.

Περιγραφή εργασίας: Στην παρούσα εργασία ζητείται η ανάλυση σε (μεγάλα) σύνολα δεδομένων εφαρμόζοντας επεξεργασία με τεχνικές που εφαρμόζονται σε data science projects. Πιο συγκεκριμένα, γίνεται ανάλυση στα παρακάτω σύνολα δεδομένων:

- **Los Angeles Crime Data:** Περιλαμβάνει δεδομένα καταγραφής εγκλημάτων για το Los Angeles από το 2010 μέχρι και σήμερα και αποτελεί το βασικό σύνολο δεδομένων (dataset).
- **LA Police Stations:** Περιλαμβάνει δεδομένα σχετικά με την τοποθεσία των 21 αστυνομικών τμημάτων που βρίσκονται στην πόλη του Los Angeles.
- **Median Household Income by Zip Code (Los Angeles County):** Περιέχει δεδομένα σχετικά με το μέσο εισόδημα ανά νοικοκυριό και ταχυδρομικό κώδικα (ZIP Code) στην Κομητεία του Los Angeles.
- **Reverse Geocoding:** Περιλαμβάνει στοιχεία αντιστοίχισης συντεταγμένων (latitude, longitude) σε ταχυδρομικούς κώδικες (ZIP Codes) εντός της πόλης του Los Angeles.

Για την υλοποίηση της εργασίας, έγινε χρήση των εργαλείων Apache Hadoop και Apache Spark, καθώς επίσης και χρήση πόρων/εικονικών μηχανών από το okeanos-knossos.

Σημείωση: Όλα τα datasets μεταφέρθηκαν στο HDFS χρησιμοποιώντας την ακόλουθη εντολή:

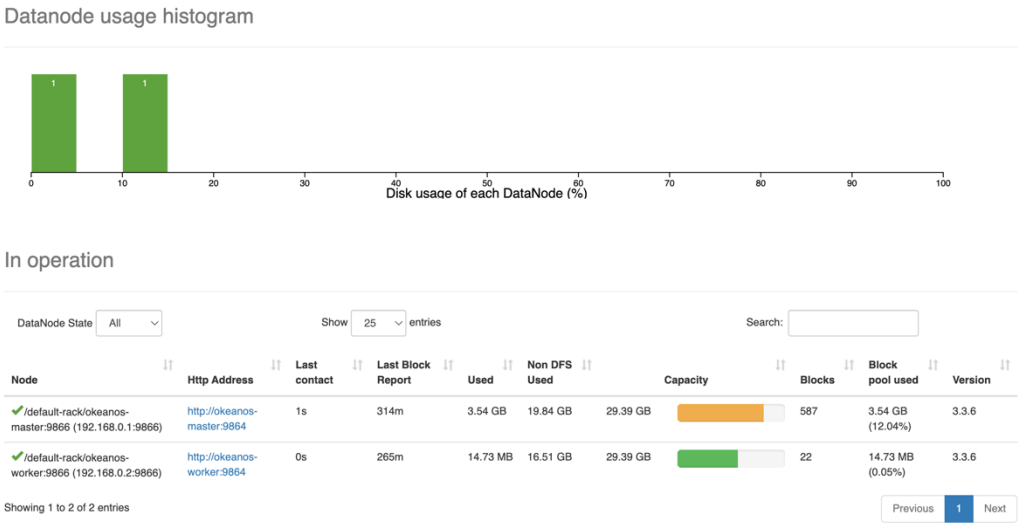
hadoop fs -put /local-file-path /hdfs-file-path

Ζητούμενο 1: Εγκατάσταση και διαμόρφωση της πλατφόρμας εκτέλεσης Apache Spark ώστε να εκτελείται πάνω από τον διαχειριστή πόρων του Apache Hadoop, YARN.

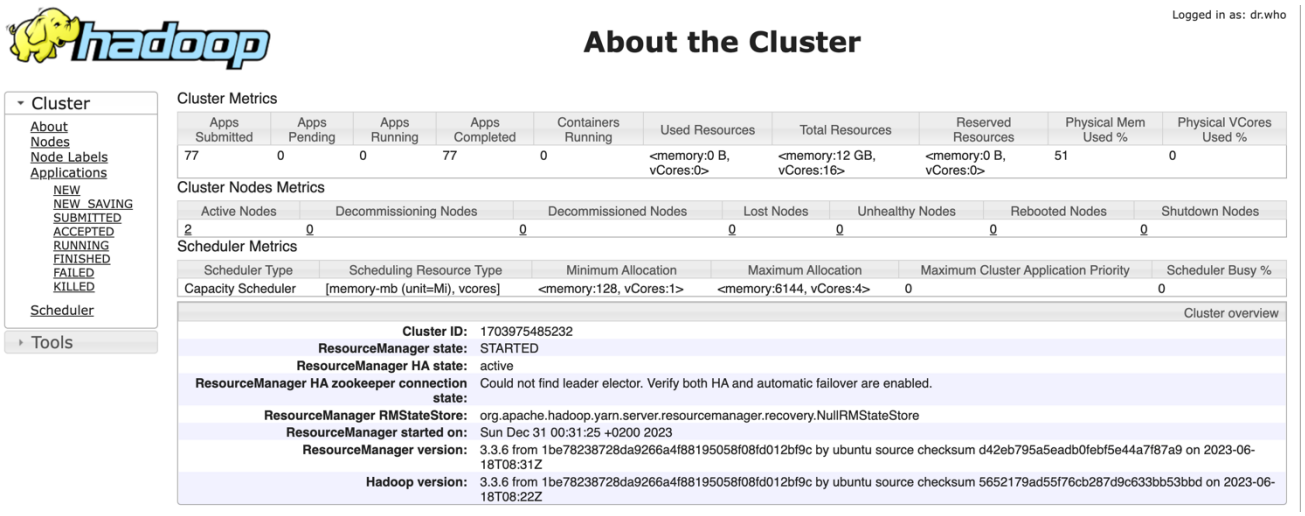
Κατόπιν δημιουργίας των οικιακών μηχανών μέσω του okeanos-knossos, ακολουθήσαμε τον σχετικό οδηγό εγκατάστασης του περιβάλλοντος λειτουργίας, που δόθηκε. Πιο συγκεκριμένα, αναβαθμίσαμε τους servers και κάναμε set up τα HDFS & Apache Spark. Στη συνέχεια, παραθέτουμε τα διαχειριστικά endpoints όπου φαίνεται η σωστή εγκατάσταση και λειτουργία του περιβάλλοντος:

ΣΗΜΑΝΤΙΚΟ!!! Για λόγους ασφάλειας, έχουμε κάνει set up ένα firewall, οπότε η (δημόσια) πρόσβαση στα παρακάτω endpoints είναι περιορισμένη και διαθέσιμη μόνο από τα δικά μας μηχανήματα. Σε περίπτωση που χρειαστείτε πρόσβαση ενημερώστε μας.

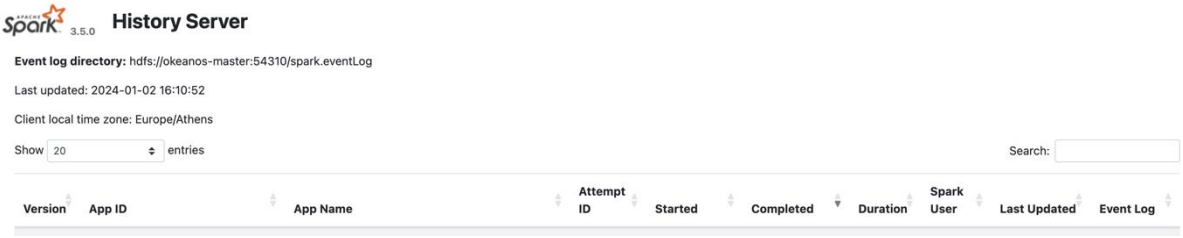
HDFS



Hadoop Cluster



Spark History Server



Ζητούμενο 2: Δημιουργία ενός DataFrame από το βασικό σύνολο δεδομένων.

Η υλοποίηση του συγκεκριμένου ζητήματος εντοπίζεται στα ακόλουθα αρχεία:

- **/utils/import_data.py:** στο συγκεκριμένο module, ορίζονται ξεχωριστά import functions για κάθε ένα από τα datasets. Στην περίπτωσή μας, η **import_crime_data** είναι υπεύθυνη για τη δημιουργία του DataFrame από το βασικό σύνολο δεδομένων και για την κατάλληλη προσαρμογή των ζητούμενων τύπων των δεδομένων, σύμφωνα με την εκφώνηση.
- **/utils/SparkSession.py:** στο συγκεκριμένο module, ορίζονται δύο συναρτήσεις που είναι υπεύθυνες για τη δημιουργία ενός SparkSession. Εδώ μας ενδιαφέρει η **create_spark_session.**
- **print_crime_df_metadata.py:** χρησιμοποιεί το παραπάνω module και τυπώνει στο standard output τον συνολικό αριθμό γραμμών του συγκεκριμένου συνόλου δεδομένων, καθώς και τον τύπο κάθε στήλης, δηλαδή το σχήμα. Τρέχοντας την εντολή **spark-submit –py-files ./utils/SparkSession.py,./utils/import_data.py print_crime_df_metadata.py** παίρνουμε το ακόλουθο αποτέλεσμα:

```
DataFrame Rows count : 2993433

|-- DR_NO: string (nullable = true)
|-- Date Rptd: date (nullable = true)
|-- DATE OCC: date (nullable = true)
|-- TIME OCC: string (nullable = true)
|-- AREA: string (nullable = true)
|-- AREA NAME: string (nullable = true)
|-- Rpt Dist No: string (nullable = true)
|-- Part 1-2: string (nullable = true)
|-- Crm Cd: string (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: string (nullable = true)
|-- Premis Desc: string (nullable = true)
|-- Weapon Used Cd: string (nullable = true)
|-- Weapon Desc: string (nullable = true)
|-- Status: string (nullable = true)
|-- Status Desc: string (nullable = true)
|-- Crm Cd 1: string (nullable = true)
|-- Crm Cd 2: string (nullable = true)
|-- Crm Cd 3: string (nullable = true)
|-- Crm Cd 4: string (nullable = true)
|-- LOCATION: string (nullable = true)
|-- Cross Street: string (nullable = true)
|-- LAT: double (nullable = true)
|-- LON: double (nullable = true)
```

Ζητούμενο 3: Υλοποίηση Query 1 με χρήση DataFrame API & SQL API και εκτέλεση με 4 executors.

Στο Query 1, ζητείται να βρεθούν, για κάθε έτος οι 3 μήνες με τον υψηλότερο αριθμό καταγεγραμμένων εγκλημάτων καθώς και το πλήθος αυτών.

Για πιο εύκολη εκτέλεση των διάφορων υλοποιήσεων, έχουν δημιουργηθεί αντίστοιχα shell scripts κάτω από το directory “scripts”.

Για την εκτέλεση του Query 1 με 4 Spark executors, έχουμε ορίσει στο script, την επιλογή –num-executors 4. Έτσι, με την εντολή:

./scripts/run_query1.sh DF cluster

εκτελούμε την υλοποίηση με DataFrame API, σε cluster mode, με 4 Spark executors και τα αποτελέσματα αποθηκεύονται στο hdfs. Με ανάλογο τρόπο εκτελούμε και την υλοποίηση με χρήση SQL API.

Σε περίπτωση που το query τρέχει σε cluster mode, τα αποτελέσματα είναι προσβάσιμα μόνο μέσω ενός output αρχείου στο hdfs. Σε περίπτωση που το query τρέχει σε client mode, τα αποτελέσματα εμφανίζονται και στο standard output του μηχανήματος στο οποίο τρέχει.

Τα αποτελέσματα είναι ίδια και για τις δύο υλοποιήσεις και είναι τα εξής:

```
+----+-----+-----------+----------+
|year|month|crimes_total|crime_rank|
+----+-----+-----------+----------+
|2010|    1|       19515|         1|
|2010|    3|       18131|         2|
|2010|    7|       17856|         3|
|2011|    1|       18135|         1|
|2011|    7|       17283|         2|
|2011|   10|       17034|         3|
|2012|    1|       17943|         1|
|2012|    8|       17661|         2|
|2012|    5|       17502|         3|
|2013|    8|       17440|         1|
|2013|    1|       16820|         2|
|2013|    7|       16644|         3|
|2014|    7|       13531|         1|
|2014|   10|       13362|         2|
|2014|    8|       13317|         3|
|2015|   10|       19219|         1|
|2015|    8|       19011|         2|
|2015|    7|       18709|         3|
|2016|   10|       19659|         1|
|2016|    8|       19490|         2|
|2016|    7|       19448|         3|
+----+-----+-----------+----------+

|2017|   10|       20431|         1|
|2017|    7|       20192|         2|
|2017|    1|       19833|         3|
|2018|    5|       19973|         1|
|2018|    7|       19875|         2|
|2018|    8|       19761|         3|
|2019|    7|       19121|         1|
|2019|    8|       18979|         2|
|2019|    3|       18856|         3|
|2020|    1|       18498|         1|
|2020|    2|       17256|         2|
|2020|    5|       17205|         3|
|2021|   12|       25453|         1|
|2021|   10|       24653|         2|
|2021|   11|       24276|         3|
|2022|    5|       20419|         1|
|2022|   10|       20276|         2|
|2022|    6|       20204|         3|
|2023|    8|       19772|         1|
|2023|    7|       19709|         2|
|2023|    1|       19637|         3|
+----+-----+-----------+----------+
```

Όπως βλέπουμε, για κάθε χρόνο, εμφανίζονται οι 3 μήνες με τα περισσότερα εγκλήματα, καθώς και το πλήθος αυτών.

Σχετικά με την επίδοση των δύο APIs αναφέρουμε τις εξής παρατηρήσεις. Αρχικά, όπως βλέπουμε μέσα από το History Server, ο χρόνος εκτέλεσης και για τις δύο υλοποιήσεις είναι ο ίδιος (40 sec). Βεβαίως, μέσω ενός μόνο παραδείγματος, δεν μπορούμε να αποφανθούμε για την συγκριτική απόδοση των DataFrame & SQL APIs. Ωστόσο, πραγματοποιώντας κάποιες ακόμα μετρήσεις (και στα επόμενα queries), βλέπουμε πως, πράγματι, οι χρόνοι εκτέλεσης είναι σχεδόν ίδιοι κάθε φορά. Αυτό είναι αναμενόμενο, διότι τα δύο APIs μοιράζονται το ίδιο execution engine και το Spark παράγει το ίδιο logical & physical plan. Συνεπώς, οι επίδοση μεταξύ των δύο APIs είναι η ίδια.

Ζητούμενο 4: Υλοποίηση Query 2 με χρήση DataFrame/SQL και RDD API.

Στο Query 2 ζητείται να ταξινομηθούν τα τμήματα της ημέρας ανάλογα με τις καταγραφές εγκλημάτων που έλαβαν χώρα στο δρόμο.
Ακολουθώντας παρόμοια βήματα με πριν, εκτελούμε τις διαφορετικές υλοποιήσεις του Query 2, με 4 Spark executors και παίρνουμε τα ακόλουθα αποτελέσματα (που είναι ίδια για όλες τις υλοποιήσεις):

```
+-----------+-----------+
|DAY SEGMENT|crime_total|
+-----------+-----------+
|      NIGHT|     237605|
|    EVENING|     187306|
|  AFTERNOON|     148180|
|    MORNING|     123846|
+-----------+-----------+
```

Οι χρόνοι εκτέλεσης για 4 spark executors, όπως προκύπτουν από το History Server είναι:

Implementation	Duration
DataFrame API	38 sec
SQL API	37 sec
RDD API	33 sec

Όπως βλέπουμε τα DataFrame και SQL APIs έχουν πάλι παρόμοια απόδοση, όπως είναι αναμενόμενο, ενώ η υλοποίηση με RDD API φαίνεται να είναι ελαφρώς ταχύτερη. Παρ' όλ' αυτά, δεν μπορούμε να εξάγουμε γενικά συμπεράσματα, καθώς οι χρόνοι και στις 3 περιπτώσεις είναι σχετικά μικροί και επομένως, οι διαφορές μπορεί να οφείλονται σε καλύτερη ή χειρότερη υλοποίηση στον εκάστοτε κώδικα.

Ζητούμενο 5: Υλοποίηση Query 3 με χρήση DataFrame/SQL API, με 2,3 και 4 Spark executors.

Στο Query 3, ζητείται να βρεθεί η καταγωγή (descent) των καταγεγραμμένων θυμάτων εγκλημάτων στο Los Angeles για το έτος 2015 στις 3 περιοχές (ZIP Codes) με το υψηλότερο και τις 3 περιοχές (ZIP Codes) με το χαμηλότερο εισόδημα ανά νοικοκυριό.

Όπως και πριν, ακολουθούμε τα ίδια βήματα με μόνη διαφορά ότι τώρα, στο αντίστοιχο script προσδιορίζουμε και τον αριθμό των Spark executors. Τα αποτέλεσμα έχουν ως εξής:

```
+--------------+------------+
|Victim Descent|total_crimes|
+--------------+------------+
|             H|        1427|
|             B|        1082|
|             W|         988|
|             O|         377|
|             A|         110|
|             K|           7|
|             I|           3|
|             J|           2|
|             F|           1|
|             C|           1|
+--------------+------------+
```

Οι χρόνοι εκτέλεσης για 2, 3 και 4 Spark executors, όπως προκύπτει από το History Server, είναι οι εξής:

# executors	Duration	
	DataFrame API	**SQL API**
2	1.1 min	1.1 min
3	1 min	58 sec
4	55 sec	52 sec

Όπως θα αναμέναμε, όσο αυξάνεται το πλήθος των executors, τόσο μικρότερος γίνεται ο χρόνος εκτέλεσης. Αυτό οφείλεται στο γεγονός ότι, όσο αυξάνουμε το πλήθος των executors, τόσο περισσότερο παραλληλισμό επιτυγχάνουμε, οδηγώντας σε καλύτερο utilization του cluster. Βέβαια, αυτό ισχύει μέχρι ένα συγκεκριμένο πλήθος executors, αφού από ένα σημείο-bottleneck και μετά, η περεταίρω αύξηση δεν θα οδηγήσει σε καμία βελτίωση.

Ζητούμενο 6: Υλοποίηση Query 4 με χρήση DataFrame/SQL API.

Το Query 4, αποτελείται από 2 ζεύγη παρόμοιων ερωτημάτων, τα οποία έχουν ως στόχο να εξεταστεί κατά πόσον τα εγκλήματα που καταγράφονται στην πόλη του Los Angeles αντιμετωπίζονται από το πλησιέστερο στον τόπο εγκλήματος αστυνομικό τμήμα ή όχι.

Query 4a1: Να υπολογιστεί ανά έτος ο αριθμός εγκλημάτων με καταγραφή χρήσης οποιασδήποτε μορφής πυροβόλων οπλών και η μέση απόσταση (σε km) των σημείων όπου αυτά έλαβαν χώρα από το αστυνομικό τμήμα που ανέλαβε την ερευνά για το περιστατικό. Τα αποτελέσματα να εμφανιστούν ταξινομημένα κατά έτος σε αύξουσα σειρά.

```
+----+----------------+------------+
|year|average_distance|total_crimes|
+----+----------------+------------+
|2010|        2.783 km|        8212|
|2011|        2.793 km|        7232|
|2012|        2.836 km|        6532|
|2013|        2.826 km|        5838|
|2014|        2.773 km|        4526|
|2015|        2.706 km|        6763|
|2016|        2.717 km|        8100|
|2017|        2.724 km|        7786|
```

year	average_distance	total_crimes
2018	2.732 km	7413
2019	2.739 km	7129
2020	2.69 km	8487
2021	2.692 km	12324
2022	2.608 km	10025
2023	2.548 km	8896

Query 4b1: Να υπολογιστεί ανά αστυνομικό τμήμα ο αριθμός εγκλημάτων με καταγραφή χρήσης οποιαδήποτε μορφής όπλων που του ανατέθηκε καθώς και η μέση απόσταση του εκάστοτε τόπου εγκλήματος από το αστυνομικό τμήμα. Τα αποτελέσματα να εμφανιστούν ταξινομημένα κατά αριθμό περιστατικών, με φθίνουσα σειρά.

division	average_distance	total_crimes
77TH STREET	2.646 km	94600
SOUTHEAST	2.092 km	77814
SOUTHWEST	2.61 km	72590
CENTRAL	1.008 km	63363
NEWTON	2.053 km	61242
RAMPART	1.532 km	55680
OLYMPIC	1.755 km	52861
HOLLYWOOD	1.435 km	50992
MISSION	4.704 km	43528
PACIFIC	3.877 km	42825
HOLLENBECK	2.592 km	41422
HARBOR	3.934 km	40713
NORTH HOLLYWOOD	2.546 km	40297
WILSHIRE	2.402 km	37789
NORTHEAST	3.993 km	37184
FOOTHILL	4.237 km	36868
VAN NUYS	2.135 km	36121
TOPANGA	3.511 km	34694
WEST VALLEY	3.387 km	33797
DEVONSHIRE	3.981 km	32447

Query 4a2: Να υπολογιστεί ανά έτος ο αριθμός εγκλημάτων με καταγραφή χρήσης οποιασδήποτε μορφής πυροβόλων όπλων και η μέση απόσταση (σε km) των σημείων όπου αυτά έλαβαν χωρά από το πλησιέστερο σε αυτά αστυνομικό τμήμα. Τα αποτελέσματα να εμφανιστούν ταξινομημένα κατά έτος σε αύξουσα σειρά.

year	average_distance	total_crimes
2010	2.434 km	8212
2011	2.461 km	7232
2012	2.506 km	6532
2013	2.456 km	5838
2014	2.388 km	4526
2015	2.387 km	6763
2016	2.428 km	8100
2017	2.392 km	7786
2018	2.408 km	7413
2019	2.429 km	7129
2020	2.384 km	8487
2021	2.407 km	12324

```
|2022|        2.312 km|       10025|
|2023|        2.266 km|        8896|
+----+----------------+------------+
```

Συγκρίνοντας τα αποτελέσματα μεταξύ του 4a1 και 4a2 παρατηρούμε ότι η μέση απόσταση στο 4a2 είναι μικρότερη από αυτή του 4a1 (περίπου στα 300μ.) οπότε σίγουρα ανά έτος τα εγκλήματα δεν αντιμετωπίζονται κατά κανόνα από το κοντινότερο αστυνομικό τμήμα.

Query 4b2: Επίσης, να υπολογιστεί ανά αστυνομικό τμήμα ο αριθμός εγκληματών με καταγραφή χρήσης οποιαδήποτε μορφής όπλων που έλαβαν χωρά πλησιέστερα σε αυτό καθώς και η μέση απόστασή των σημείων από το εκάστοτε αστυνομικό τμήμα. Τα αποτελέσματα να εμφανιστούν ταξινομημένα κατά αριθμό περιστατικών, με φθίνουσα σειρά.

```
+----------------+----------------+------------+
|        DIVISION|average_distance|total_crimes|
+----------------+----------------+------------+
|     77TH STREET|        1.673 km|       79625|
|       SOUTHWEST|        2.161 km|       78216|
|       SOUTHEAST|        2.212 km|       71083|
|       HOLLYWOOD|        1.924 km|       71069|
|         OLYMPIC|         1.66 km|       64550|
|         CENTRAL|        0.864 km|       59364|
|        WILSHIRE|        2.479 km|       58315|
|         RAMPART|         1.36 km|       56496|
|        VAN NUYS|        2.823 km|       56076|
|          NEWTON|        1.599 km|       45507|
|        FOOTHILL|        3.979 km|       42893|
|      HOLLENBECK|        2.584 km|       42120|
| NORTH HOLLYWOOD|        2.619 km|       41461|
|          PACIFIC|        3.847 km|       40484|
|          HARBOR|         3.68 km|       39541|
|         TOPANGA|         3.06 km|       35708|
|     WEST VALLEY|        2.848 km|       35349|
|         MISSION|         3.77 km|       29213|
|       NORTHEAST|        3.769 km|       27359|
|WEST LOS ANGELES|        2.714 km|       22462|
|      DEVONSHIRE|        2.835 km|       16899|
+----------------+----------------+------------+
```

Συγκρίνοντας τα αποτελέσματα του 4b1 και 4b2 βλέπουμε αρχικά ότι υπάρχουν αστυνομικά τμήματα που λαμβάνουν κατά πολύ περισσότερα εγκλήματα απ' ότι τους αναλογεί με βάσει την μικρότερη απόσταση. Αντίστοιχα, υπάρχουν τμήματα που λαμβάνουν πολύ λιγότερα. Για παράδειγμα, το Van Nuys τμήμα αναλαμβάνει μόλις 36121 εγκλήματα έναντι των 56076 που του αναλογούν γεωγραφικά. Επίσης αυτό το γεγονός έχει ως αποτέλεσμα ότι η μέση απόσταση των εγκλημάτων από κάθε αστυνομικό τμήμα να μην είναι η μικρότερη δυνατή. Για παράδειγμα, το 77th Street τμήμα το οποίο αναλαμβάνει κατά 15 χιλιάδες παραπάνω εγκλήματα ανεβάζει τον μέσο όρο απόστασης κατά 1 χιλιόμετρο.

Ζητούμενο 7: Μελέτη και σχολιασμός των διαθέσιμων στρατηγικών join του Spark, μέσω των Queries 3 & 4.

Σημείωση: Εφόσον, όπως αναφέραμε νωρίτερα, δεν υπάρχει ουσιαστική διαφορά, στην επίδοση, μεταξύ DataFrame & SQL API, στο συγκεκριμένο ζητούμενο, λαμβάνουμε υπόψη τις υλοποιήσεις με DataFrame API.

Όσον αφορά τα joins που πραγματοποιούνται στα συγκεκριμένα queries, αναφέρουμε τα εξής:

Στο Query 3, αρχικά γίνεται equi-join μεταξύ των income & revgecoding dataframes με βάση το zip code και στη συνέχεια, το παραγόμενο dataframe, γίνεται equi-join με το LA crime dataframe (το οποίο έχει προηγουμένως φιλτραριστεί, ώστε να κρατάει μόνο τα επιθυμητά δεδομένα), με βάση τις συντεταγμένες LAT & LON. Έπειτα, υπολογίζουμε τα zip codes που αντιστοιχούν στις 3 περιοχές με το υψηλότερο και τις 3 περιοχές με το χαμηλότερο εισόδημα ανά νοικοκυριό και κάνουμε άλλο ένα equi-join μεταξύ αυτών και του dataframe που προέκυψε προηγουμένως, με βάση το zip code και εφαρμόζουμε κατάλληλες ενέργειες για να παράγουμε το επιθυμητό αποτέλεσμα.
Έτσι λοιπόν, στο συγκεκριμένο query, εκτελούμε συνολικά 3 equi-joins.

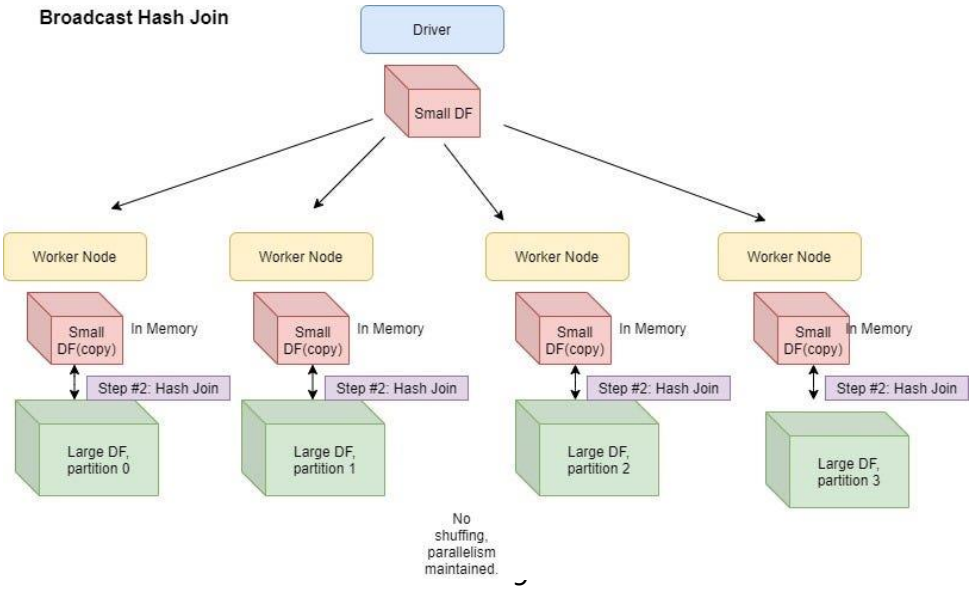
Το query 4, όπως αναφέρεται και νωρίτερα, αποτελείται από δύο ζεύγη παρόμοιων ερωτημάτων όπου οι κύριες διαφορές εντοπίζονται στο αρχικό φιλτράρισμα των δεδομένων που αφορούν τα εγκλήματα και στην τελική ομαδοποίηση (group by).

Στο 1ο ζεύγος, εκτελείται ένα equijoin μεταξύ των κατάλληλα φιλτραρισμένων εγκλημάτων και των αστυνομικών τμημάτων, ώστε να ενωθούν τα εγκλήματα με τα αντίστοιχα αστυνομικά τμήματα που τα ανέλαβαν.

Στο 2ο ζεύγος, για κάθε ένα από τα φιλτραρισμένα εγκλήματα, αναζητούμε το πλησιέστερο αστυνομικό τμήμα και πιο συγκεκριμένα, την απόσταση από αυτό. Για να το πετύχουμε αυτό, χρειάζεται να υπολογίζουμε τις αποστάσεις από κάθε τμήμα και να επιλέξουμε την μικρότερη. Συνεπώς, χρειαζόμαστε καρτεσιανό γινόμενο, οπότε χρησιμοποιούμε crossjoin, μεταξύ των εγκλημάτων και των αστυνομικών τμημάτων.

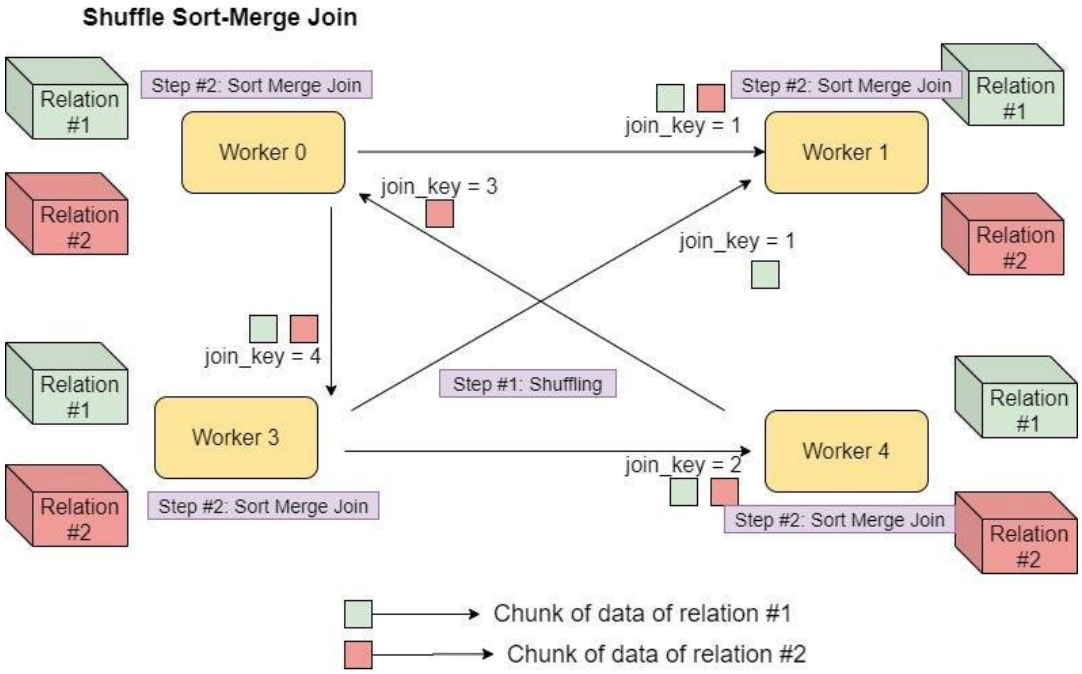
Μελέτη και Αξιολόγηση Στρατηγικών join:

Broadcast Hash Join (BHJ):



Στο broadcast hash join, αντίγραφο από μία από τις δύο σχέσεις που γίνονται join, μεταδίδεται (broadcast) σε όλους τους worker κόμβους, εξοικονομώντας κόστος από το shuffling. Αυτό είναι χρήσιμο σε περιπτώσεις όπου γίνονται join ένα μεγάλο σύνολο δεδομένων με ένα μικρότερο. Επίσης, για να δουλέψει αυτή η στρατηγική, η μεταδιδόμενη σχέση, πρέπει να χωράει πλήρως στη μνήμη του κάθε executor (default μέγιστη τιμή είναι τα 10 mb), αλλά και του driver και ακόμη, πρέπει να έχουμε equi-join. Συνεπώς, το Broadcast Hash Join δεν είναι διαθέσιμο στο 2ο ζεύγος ερωτημάτων του query 4.

(Shuffle Sort) Merge Join (SMJ):

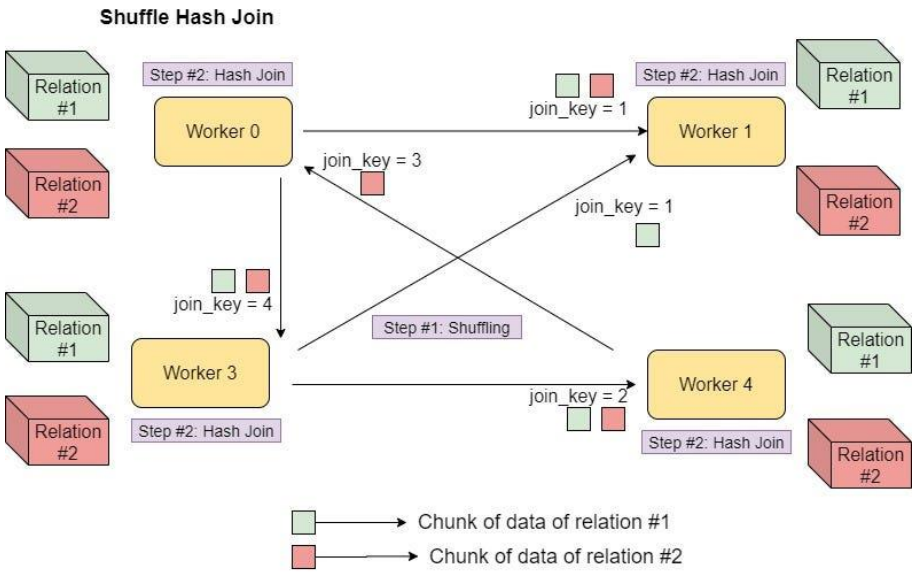


Στο Spark το Shuffle Sort-Merge Join αποτελείται από 3 βήματα:

- **Shuffle:** Τα δεδομένα και από τους δύο πίνακες, χωρίζονται με βάση το join key, έτσι ώστε οι εγγραφές με το ίδιο join key να σταλούν στον ίδιο worker.
- **Sort:** Τα δεδομένα σε κάθε worker, ταξινομούνται με βάση το join key.
- **Merge:** Τα ταξινομημένα δεδομένα συγχωνεύονται, ώστε να εκτελεστεί η λειτουργία του join.

Η συγκεκριμένη στρατηγική χρησιμοποιείται σε joins μεταξύ μεγάλων συνόλων δεδομένων, όπου δεν μπορούν να χωρέσουν στη μνήμη ενός μόνο κόμβου. Επίσης, όπως είναι προφανές, προκειμένου να μπορέσει να λειτουργήσει, απαιτείται, αφενός να έχουμε equi-join και αφετέρου, τα join keys να μπορούν να ταξινομηθούν. Συνεπώς, ούτε αυτή η στρατηγική είναι διαθέσιμη στο 2ο ζεύγος ερωτημάτων του query 4.

Shuffle Hash Join (SHJ):



Στο Spark, το Shuffle Hash Join αποτελεί μία από τις default στρατηγικές join, όταν έχουμε να κάνουμε join δύο μεγάλα σύνολα δεδομένων. Αποτελείται από δύο κύρια βήματα:

- **Shuffling:** Σε αυτό το βήμα, τα δεδομένα από τα join tables χωρίζονται με βάση το join key και κατανέμονται στους διάφορους workers, έτσι ώστε οι εγγραφές με τα ίδια join keys να καταλήξουν στους ίδιους workers.
- **Hash join:** Το κλασικό βήμα, όπου αρχικά, δημιουργείται ένα Hash table με βάση το join key του μικρότερου συνόλου και στη συνέχεια γίνεται loop πάνω στο μεγαλύτερο σύνολο, ώστε να γίνει ταίριασμα με το join key του μεγαλύτερου συνόλου.

Οπότε, αρχικά τα δεδομένα με ίδιο join key μεταφέρονται στον ίδιο executor κόμβο, μέσω του shuffling, και στη συνέχεια συνδυάζονται με hash join.
Αυτή η στρατηγική, όπως και η BHJ υποστηρίζεται μόνο για equi-join. Ακόμη, χρησιμοποιείται σε περιπτώσεις όπου τα δεδομένα είναι αρκετά μεγάλα ώστε να χωρέσουν στη μνήμη ενός μόνο κόμβου. Τέλος, το join key, δεν είναι απαραίτητο να μπορεί να ταξινομηθεί. Ούτε αυτή η στρατηγική είναι διαθέσιμη στο 2^{ο} ζεύγος ερωτημάτων του query 4.

Shuffle and Replication Nested Loop Join or Cartesian Product Join (CPJ)

Αυτή η στρατηγική, παράγει το καρτεσιανό γινόμενο μεταξύ των δύο συνόλων δεδομένων. Όπως γίνεται εύκολα κατανοητό, αυτή η λειτουργία είναι πολύ ακριβή και συνεπώς πρέπει να χρησιμοποιείται μόνο όπου είναι απαραίτητο. Στην περίπτωσή μας, θα τη χρησιμοποιήσουμε στο 2^{ο} ζεύγος ερωτημάτων του query 4, όπου χρησιμοποιούμε cross join.

Συνοψίζοντας όλα τα παραπάνω, αρχικά, βλέπουμε πως για το 2ο ζεύγος ερωτημάτων του query 4, από τις 4 στρατηγικές join, μόνο η τελευταία είναι διαθέσιμη. Αντίθετα, για το 1ο ζεύγος, αλλά και για το query 3, είναι διαθέσιμες όλες οι στρατηγικές. Ωστόσο, γίνεται εύκολα αντιληπτό, πως η υλοποίηση με καρτεσιανό γινόμενο (CPJ) δεν είναι καθόλου αποδοτική για τα συγκεκριμένα ερωτήματα αλλά θα την εξετάσουμε για λόγους πληρότητας. Έτσι λοιπόν, θα εξετάσουμε τους ακόλουθους συνδυασμούς:

Query	Broadcast (BHJ)	Merge (SMJ)	Shuffle Hash Join	Shuffle Replicate NL
3	✓	✓	✓	✓
4a1	✓	✓	✓	✓
4b1	✓	✓	✓	✓
4a2				✓
4b2				✓

Στα ζητούμενα ερωτήματα, χρειαζόμαστε τουλάχιστον ένα join μεταξύ του crime-data dataset, το μέγεθος του οποίου είναι αρκετά μεγάλο, και ενός από τα υπόλοιπα dataset, τα οποία είναι αρκετά μικρά, ώστε να χωρέσουν στη μνήμη ενός μόνο worker. Έτσι, μπορούμε να κάνουμε μία αρχική πρόβλεψη και να πούμε ότι για αυτά τα joins, καλύτερη στρατηγική μάλλον είναι η Broadcast Hash Join. Αλλά ακόμη και σε joins μεταξύ μικρών dataset, πιθανώς πάλι αυτή να είναι αποδοτικότερη, καθώς οι Shuffle Hash Join και Sort Merge Join, χρειάζονται κάποιο παραπάνω χρόνο για το shuffling. Βέβαια, αυτές οι προβλέψεις/εκτιμήσεις μπορούν να επιβεβαιωθούν εξετάζοντας το execution plan στο Spark History UI, καθώς, αν δεν ορίσουμε εμείς κάποια συγκεκριμένη στρατηγική join, τότε το Spark, και συγκεκριμένα ο Optimizer, θα βρει και θα εφαρμόσει την βέλτιστη στρατηγική.

Για να ορίσουμε μία συγκεκριμένη στρατηγική (πχ shuffle hash), μπορούμε να χρησιμοποιήσουμε τη μέθοδο hint σε ένα dataframe με τον εξής τρόπο:

```
df.hint("shuffle_hash")
```

Επίσης, με τη μέθοδο explain μπορούμε να πάρουμε το logical & physical plan στην κονσόλα για περαιτέρω ανάλυση.

Query3:

Εξετάζοντας τους χρόνους εκτέλεσης για τις διάφορες στρατηγικές join βλέπουμε τα εξής:

Query	Broadcast (BHJ)	Merge (SMJ)	Shuffle Hash Join	Shuffle Replicate NL
3	1m	58s	52s	35min

Όπως βλέπουμε οι χρόνοι για όλες τις περιπτώσεις πέρα από την περίπτωση του Shuffle Replicate NL (καρτεσιανό γινόμενο) είναι αντίστοιχοι. Παρατηρώντας το execution plan από το History Server στην περίπτωση του καρτεσιανού γινομένου μπορούμε γρήγορα να καταλάβουμε γιατί η υλοποίηση είναι σαφώς χειρότερη. Τα datasets τα οποία συμμετέχουν στον join έχουν πολλές εγγραφές (μερικές χιλιάδες) και τα άσκοπα tuples που παράγονται από το καρτεσιανό γινόμενο συνεισφέρουν αρνητικά στην απόδοση. Στις υπόλοιπες 3 υλοποιήσεις παρατηρούμε ότι δεν υπάρχει κάποια σαφώς προτιμητέα. Οι χρόνοι είναι

παρεμφερείς και παρότι εκ πρώτης όψεως το Broadcast Hash Join φαίνεται ως η καταλληλότερη επιλογή βλέπουμε ότι δεν έχει απαραίτητα τον ταχύτερο χρόνο. Βέβαια έχουμε συναντήσει αρκετά μεγάλη διαβάθμιση των χρόνων οπότε θεωρούμε τις τρεις στρατηγικές ισάξιες.

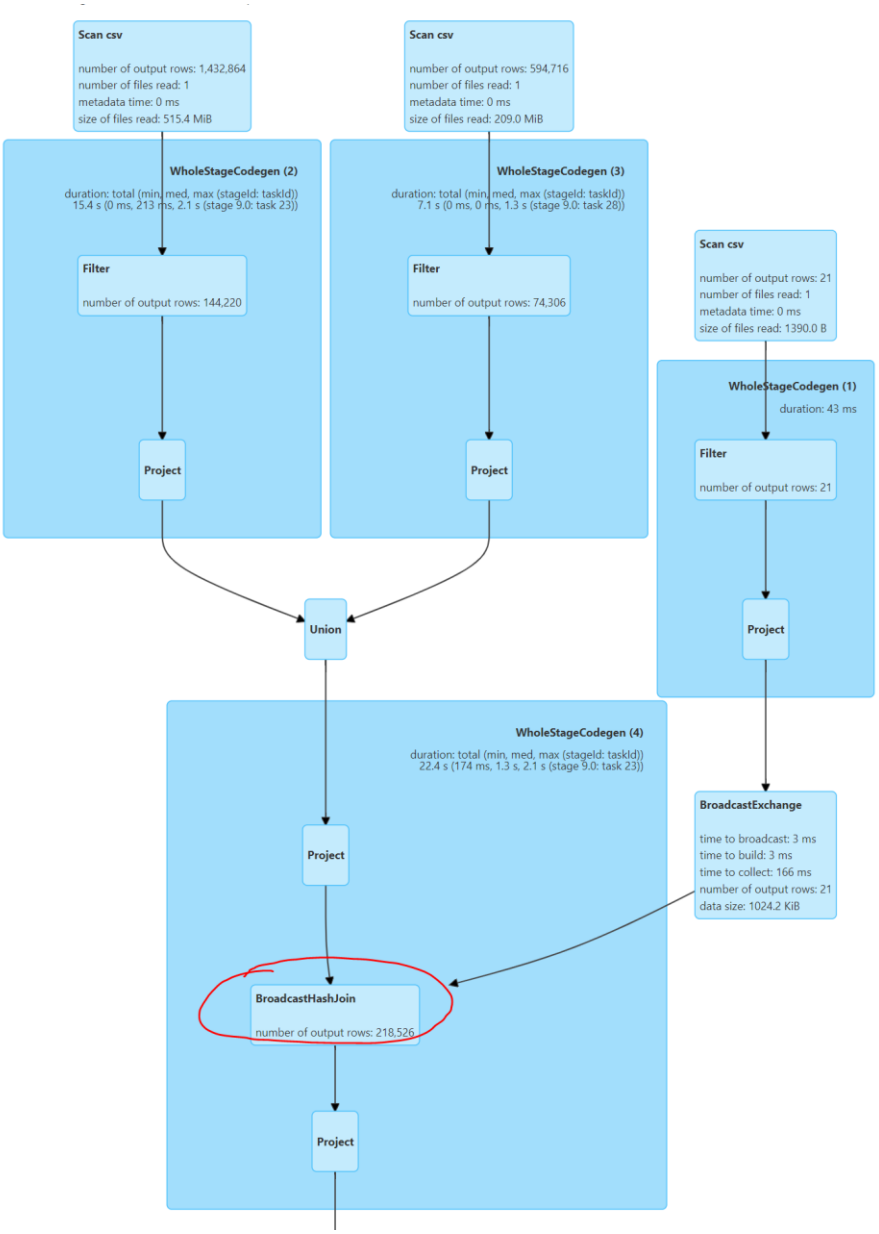
Query4:

A1-B1)

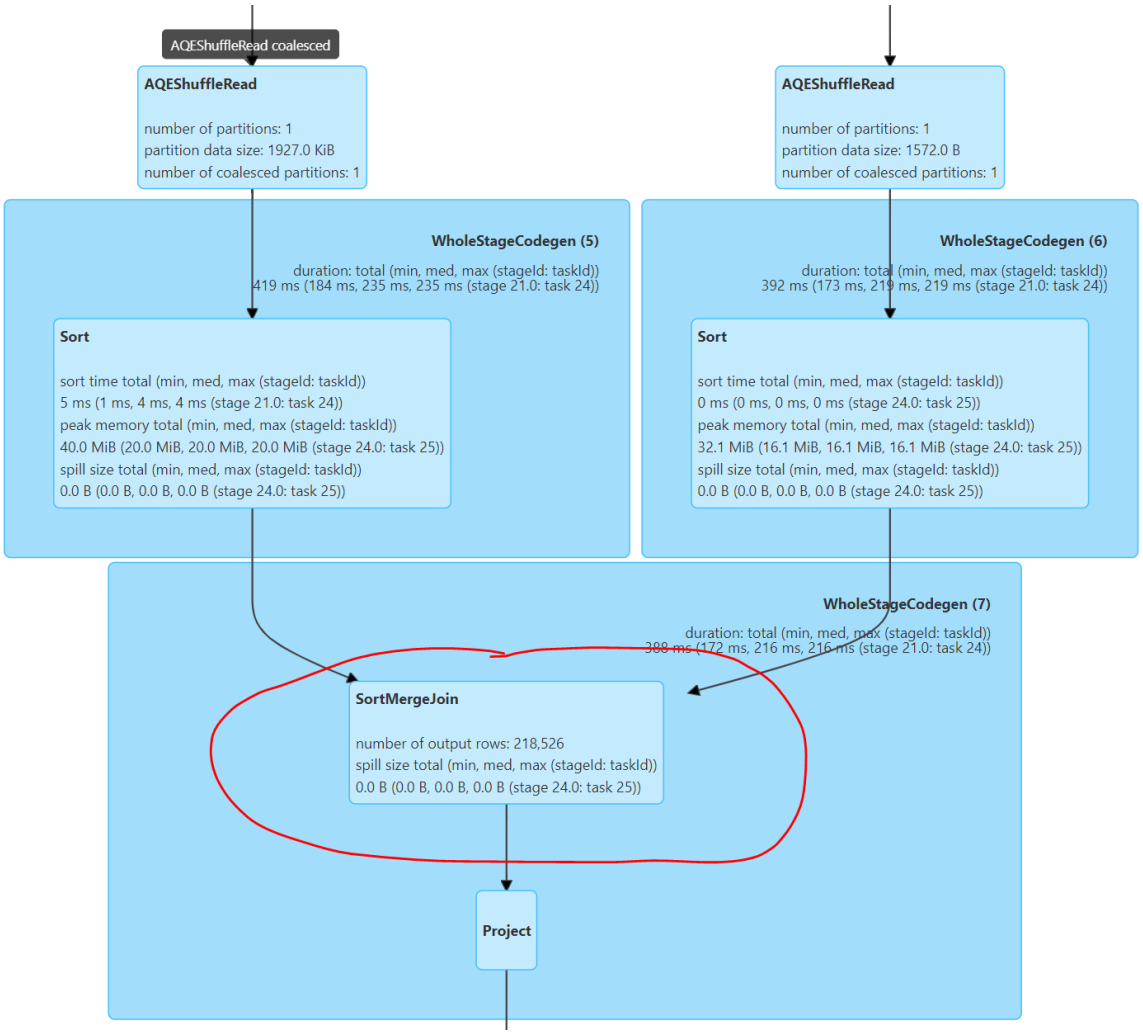
Query	Broadcast (BHJ)	Merge (SMJ)	Shuffle Hash Join	Shuffle Replicate NL
4a1	1.4m	1.4m	1.5m	1.3m
4b1	2.3m	2.9m	2.9m	2.3m

Ενδεικτικά, παρουσιάζουμε και το execution (physical plan) της κάθε στρατηγικής join για το συγκεκριμένο query. Ίδια συμπεράσματα από το physical plan μπορούμε να δούμε και για τα υπόλοιπα queries. Για τον λόγο αυτό παραθέτουμε μόνο για το παρόν query:

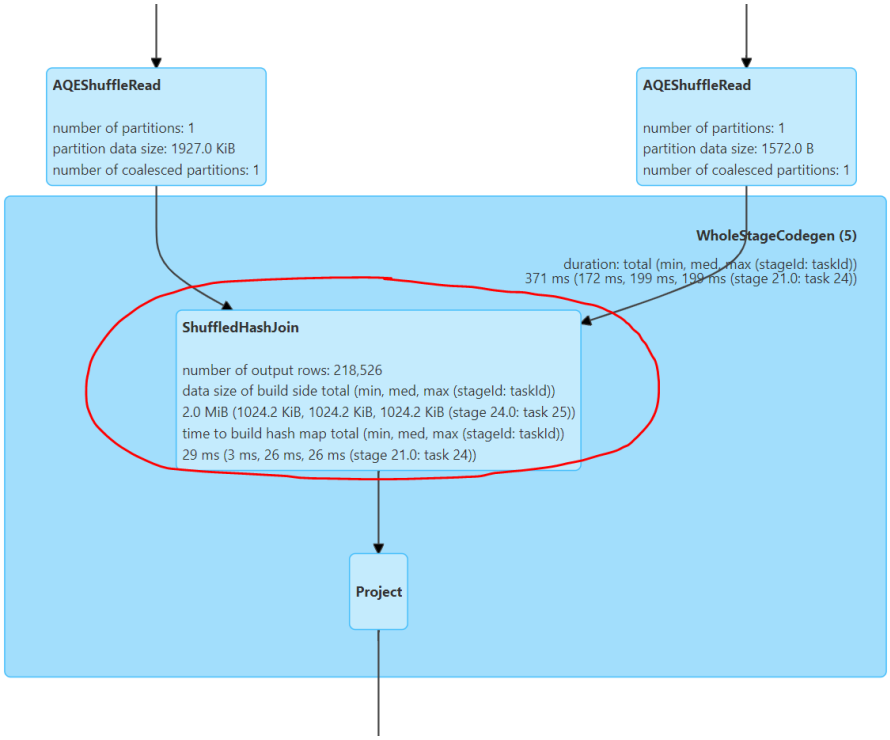
BHJ:



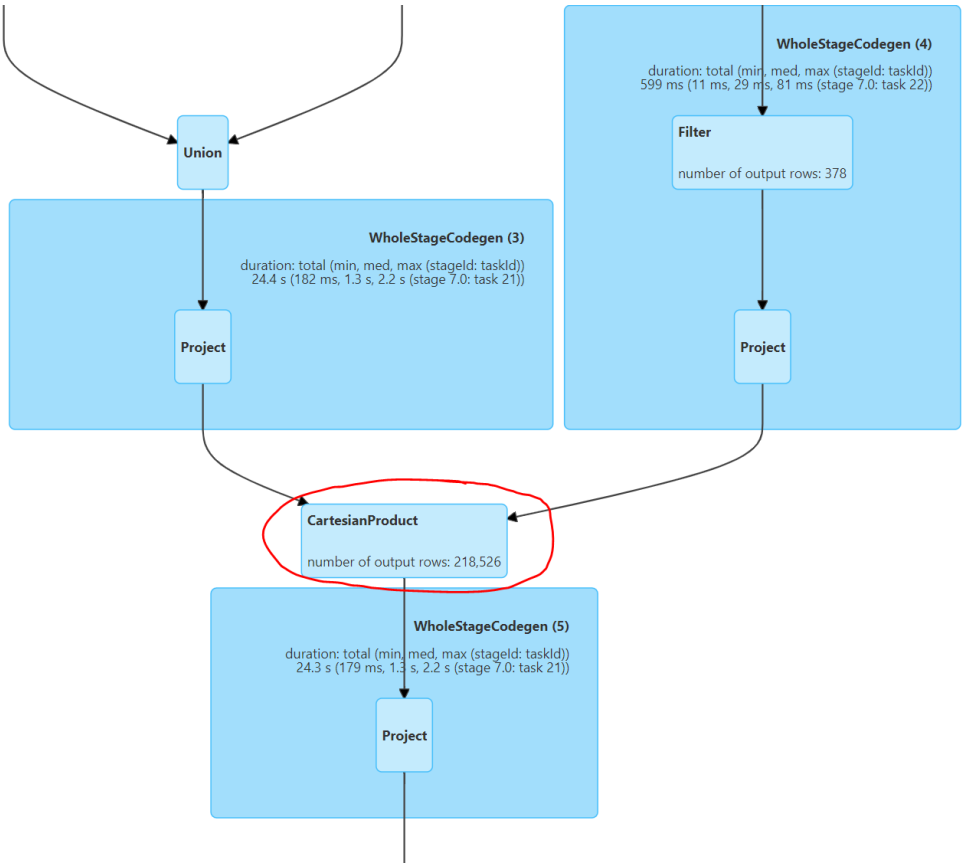
SMJ:


AQEShuffleRead coalesced
AQEShuffleRead
number of partitions: 1
partition data size: 1927.0 KiB
number of coalesced partitions: 1
AQEShuffleRead
number of partitions: 1
partition data size: 1572.0 B
number of coalesced partitions: 1
WholeStageCodegen (5)
duration: total (min, med, max (stageId: taskId))
419 ms (184 ms, 235 ms, 235 ms (stage 21.0: task 24))
WholeStageCodegen (6)
duration: total (min, med, max (stageId: taskId))
392 ms (173 ms, 219 ms, 219 ms (stage 21.0: task 24))
Sort
sort time total (min, med, max (stageId: taskId))
5 ms (1 ms, 4 ms, 4 ms (stage 21.0: task 24))
peak memory total (min, med, max (stageId: taskId))
40.0 MiB (20.0 MiB, 20.0 MiB, 20.0 MiB (stage 24.0: task 25))
spill size total (min, med, max (stageId: taskId))
0.0 B (0.0 B, 0.0 B, 0.0 B (stage 24.0: task 25))
Sort
sort time total (min, med, max (stageId: taskId))
0 ms (0 ms, 0 ms, 0 ms (stage 24.0: task 25))
peak memory total (min, med, max (stageId: taskId))
32.1 MiB (16.1 MiB, 16.1 MiB, 16.1 MiB (stage 24.0: task 25))
spill size total (min, med, max (stageId: taskId))
0.0 B (0.0 B, 0.0 B, 0.0 B (stage 24.0: task 25))
WholeStageCodegen (7)
duration: total (min, med, max (stageId: taskId))
388 ms (172 ms, 216 ms, 216 ms (stage 21.0: task 24))
SortMergeJoin
number of output rows: 218,526
spill size total (min, med, max (stageId: taskId))
0.0 B (0.0 B, 0.0 B, 0.0 B (stage 24.0: task 25))
Project

SHJ:


AQEShuffleRead
number of partitions: 1
partition data size: 1927.0 KiB
number of coalesced partitions: 1
AQEShuffleRead
number of partitions: 1
partition data size: 1572.0 B
number of coalesced partitions: 1
WholeStageCodegen (5)
duration: total (min, med, max (stageId: taskId))
371 ms (172 ms, 199 ms, 199 ms (stage 21.0: task 24))
ShuffledHashJoin
number of output rows: 218,526
data size of build side total (min, med, max (stageId: taskId))
2.0 MiB (1024.2 KiB, 1024.2 KiB, 1024.2 KiB (stage 24.0: task 25))
time to build hash map total (min, med, max (stageId: taskId))
29 ms (3 ms, 26 ms, 26 ms (stage 21.0: task 24))
Project

SRNL:



Στα παραπάνω διαγράμματα βλέπουμε τις διαφορές μεταξύ των στρατηγικών αλλά και των διαφορετικών βημάτων που ακολουθούνται κατά την εκτέλεση (execution plan). Για παράδειγμα στο broadcast hash join βλέπουμε το broadcast του μικρότερου dataset στους υπόλοιπους κόμβους, στο sort merge join παρατηρούμε shuffle και κατόπιν sort, στο shuffle hash join παρατηρούμε shuffle και μετά απευθείας hash join ενώ στο shuffle replicate nested loop παρατηρούμε το καρτεσιανό γινόμενο.

Στις δύο αυτές περιπτώσεις έχουμε join μεταξύ ενός μεγάλου dataset και μεταξύ ενός πολύ μικρού dataset. Από αυτά τα χαρακτηριστικά καταλαβαίνουμε ότι η καταλληλότερη στρατηγική είναι αυτή του broadcast hash join. Βέβαια, το μη αναμενόμενο είναι ότι και στη περίπτωση του Shuffle Replicate NL η απόδοση είναι αντίστοιχη αυτού του BHJ. Πιθανότατα, το καρτεσιανό γινόμενο είναι υλοποιημένο ώστε να είναι αποδοτικό σε περίπτωση μικρών tables με μικρή ανταλλαγή πληροφορίας που θα έριχνε την απόδοση. Αντίθετα, στις περιπτώσεις SMJ & SHJ η απόδοση είναι χειρότερη με κύρια αιτία τις μεγαλύτερες ανάγκες μεταφοράς δεδομένων.

A2-B2) Σε αυτή την περίπτωση αν δεν ορίσουμε κάποια συγκεκριμένη στρατηγική join, ο optimizer επιλέγει το broadcast nested loop join που υλοποιεί και αυτό αντίστοιχα καρτεσιανό γινόμενο. Αν ορίσουμε το Shuffle Replicate NL η στρατηγική που ακολουθείται είναι αυτή (Cartesian Product). Σε οποιαδήποτε άλλη περίπτωση δεδομένου ότι οι υπόλοιπες στρατηγικές δεν είναι διαθέσιμες (λόγω της αναγκαιότητας ύπαρξης ενός equi-join) το execution plan καταφεύγει και πάλι στο Broadcast Nested Loop Join. Οι δύο στρατηγικές καρτεσιανού γινομένου (δηλαδή η Broadcast Nested Loop Join και η Shuffle Replicate NL) δεν

παρουσιάζουν σημαντικές χρονικές διαφορές ωστόσο εκτιμούμε ότι η Broadcast Nested Loop Join είναι η καταλληλότερη λόγω της αντιστοίχισής της με το Broadcast Hash Join και την παρουσία μικρών συνόλων δεδομένων στην περίπτωση των query μας.

Query	Broadcast (BHJ)	Merge (SMJ)	Shuffle Hash Join	Shuffle Replicate NL	Broadcast NL
4a2	-	-	-	36m	43m
4b2	-	-	-	35m	38m