2025

Αναφορά Εξαμηνιαίας Εργασίας

Προχωρημένα Θέματα Βάσεων Δεδομένων

ΠΑΠΑΔΟΠΟΥΛΟΣ ΣΠΥΡΙΔΩΝ (ΑΜ):03120033
ΕΜΜΑΝΟΥΗΛ ΞΕΝΟΣ (ΑΜ):03120850

Επισημάνσεις

Το link για το GitHub repo με τον κώδικα μας.

Σε αυτό το σημείο θα αναφέρουμε κάποιες πρακτικές που εφαρμόσαμε στο σύνολο της παρούσας εργασίας:

- Σε οποιοδήποτε σημείο δεν ζητήθηκε να τεθεί κάποιο συγκεκριμένο configuration για τον cluster, έχουμε αφήσει το spark να θέσει το δικό του default configuration
- Σε οποιοδήποτε ζητούμενο ζητήθηκε να συγκρίνουμε χρόνους ανάμεσα σε διαφορετικά configurations ή υλοποιήσεις ανάμεσα στην εκτέλεση των υλοποιήσεων πάντα γίνεται restart ο kernel. Αυτό έχει ως στόχο την ελαχιστοποίηση των περιπτώσεων όπου τα δεδομένα μπορεί να είναι κάπου ήδη φορτωμένα(cached στην cache ή στην main memory) όταν εκτελέσουμε κάποιο άλλο configuration ή υλοποίηση. Το αποτέλεσμα της εκτέλεσης (χρόνος εκτέλεσης) γράφεται σε ένα json αρχείο στο S3 bucket της ομάδας μας(ομάδα 45) και όταν θέλουμε να παρουσιάσουμε τα συγκριτικά αποτελέσματα διαβάζουμε το json αυτό και το παραθέτουμε.
- Σε όλα τα reads θέτουμε την παράμετρο inferSchema=True για να μην γίνει lazily evaluated το read και να μπορέσουμε να λάβουμε το read time(ιδιαίτερα σημαντικό για το δεύτερο ζητούμενο) ή σε άλλα query, όπως το πρώτο να μην επηρεάζει η ανάγνωση του αρχείου τον πραγματικό χρόνο εκτέλεσης του ερωτήματος.

Πρώτο Ζητούμενο

Στο πρώτο ζητούμενο μας ζητείται να χρησιμοποιήσουμε το Los Angeles Crime Data Dataset, προκειμένου να κατατάξουμε τα περιστατικά «βαριάς σωματικής βλάβης σε 4 ηλικιακές ομάδες. Για το ζητούμενο Query θα δημιουργηθούν δύο υλοποιήσεις, όπου η μία θα χρησιμοποιεί το RDD API και η άλλη θα χρησιμοποιεί το Dataframe API. Για να λάβουμε ορθά τα αποτελέσματα εκτελούμε, από τον φάκελο Q1 του αποθετηρίου, πρώτα το αρχείο RDD.ipynb μετά DataFrame.ipynb. Στην συνέχεια παρουσιάζουμε τα αποτελέσματα της εκτέλεσης του κώδικα μας και με τα 2 APIs :

```
Executor Instances: 4
Executor Memory: 4743M
Executor Cores: 2
+------------+------+
|       group| count|
+------------+------+
|      Adults|121093|
|    Children| 10830|
|     Elderly|  5985|
|Young Adults| 33605|
+------------+------+

+-----------------+-----------------+
|Data_Abstractions|   Execution_Time|
+-----------------+-----------------+
|        Dataframe|5.989339351654053|
|              RDD|7.777620315551758|
+-----------------+-----------------+
```

Αρχικά, φαίνεται το configuration της εκτέλεσης(στην εκφώνηση ζητείται η εκτέλεση με 4 spark executors) και στην συνέχεια το αποτέλεσμα του Query. Αυτά είναι κοινά και για τα 2 APIs. Τέλος, βλέπουμε την σύγκριση τους χρόνους εκτέλεσης και με τα 2 API. Όπως φαίνεται και στον κώδικα, στους παραπάνω χρόνους εκτέλεσης χωρίς να συμπεριλαμβάνεται και το read του αρχείου(καθώς εισήγαγε «θόρυβο»). Παρατηρούμε, διαφορά στην επίδοση των δύο APIs, καθώς το RDD, όπως αναμέναμε, έχει μικρότερο χρόνο εκτέλεσης .Αυτό οφείλεται στο γεγονός ότι, πρώτον το Spark δεν δύναται να γνωρίζει τι operations έχει ο χρήστης σκοπό να πραγματοποιήσει ή τον τύπο των δεδομένων εντός του RDD. Αυτό έχει ως αποτέλεσμα να μην μπορεί το Spark να εκμεταλλευτεί τον Catalyst optimizer, ο οποίος εφαρμόζει μία σειρά από βελτιστοποιήσεις, αναλύοντας τον κώδικα, και δημιουργώντας ένα βέλτιστο λογικό και

φυσικό πλάνο εκτέλεσης για το query. Αντίθετα το Spark μπορεί να χρησιμοποιήσει τον Catalyst Optimizer για να βελτιστοποιήσει το πλάνο εκτέλεσης ενός query υλοποιημένου με το Dataframe API. Επιπλέον, το RDD API είναι αργό σε non-JVM(Java Virtual Machine) γλώσσες όπως η Python. Αυτό οφείλεται ότι το περιβάλλον εκτέλεσης του κώδικά μας είναι το Pyspark . Το Pyspark λειτουργεί σαν ένας Wrapper καλώντας το JVM για την εκτέλεση των Queries.Έτσι παρότι τα RDDs είναι βελτιστοποιημένα για την εκτέλεση σε JVM περιβάλλοντα, το Pyspark δεν μπορεί να αλληλοεπιδράσει άμεσα με τα RDDs, αλλά πρέπει να επικοινωνήσει με το JVM. Αυτό το κόστος επικοινωνίας έχει ως αποτέλεσμα μεγάλες καθυστερήσεις.

Δεύτερο Ζητούμενο

Σε αυτό το ζητούμενο έχουμε προσθέσει στα αποτελέσματα μας και ξεχωριστά τον χρόνο ανάγνωσης και εκτέλεσης, καθώς στο Β μέρος αυτού θα συγκρίνουμε τα file formats csv και parquet. Συνεπώς, για την πλήρη σύγκριση αυτών θέλουμε ιδανικά να γνωρίζουμε και το read time για κάθε περίπτωση. Για να λάβουμε ορθά τα αποτελέσματα εκτελούμε, στον φάκελο Q2, πρώτα το αρχείο query_df.ipynb μετά το sql_csv.ipynb και τέλος για το Β ερώτημα το sql_parquet.ipynb.

Α) Στο δεύτερο ζητούμενο μας ζητείται να βρεθούν, για κάθε έτος, τα 3 Αστυνομικά τμήματα με το υψηλότερο ποσοστό κλεισμένων υποθέσεων. Για το ζητούμενο Query θα δημιουργηθούν δύο υλοποιήσεις, όπου η μία θα χρησιμοποιεί το DataFrame API και η άλλη θα χρησιμοποιεί το SQL API. Και οι δύο υλοποιήσεις μπορούν να βρεθούν στο αρχείο Query_2.ipynb του αποθετηρίου μας. Στην συνέχεια παρουσιάζουμε τα αποτελέσματα της εκτέλεσης του κώδικα μας και με τα 2 APIs:

```
Executor Instances: None
Executor Memory: 4743M
Executor Cores: 2
+----+-----------+------------------+----+
|year|  area name|              rate|rank|
+----+-----------+------------------+----+
|2010|    Rampart| 32.84713448949121|   1|
|2010|    Olympic|31.515289821999087|   2|
|2010|     Harbor| 29.36028339237341|   3|
|2011|    Olympic|  35.0400600901352|   1|
|2011|    Rampart|32.496447181430604|   2|
|2011|     Harbor|28.513362463164313|   3|
|2012|    Olympic| 34.29708533302119|   1|
|2012|    Rampart| 32.46000463714352|   2|
|2012|     Harbor| 29.50958584895668|   3|
|2013|    Olympic| 33.58217940999398|   1|
|2013|    Rampart|  32.1060382916053|   2|
|2013|     Harbor|29.723638951488553|   3|
|2014|   Van Nuys|  32.0215235281705|   1|
|2014|West Valley| 31.49754809505847|   2|
|2014|    Mission| 31.22493985565357|   3|
|2015|   Van Nuys|32.265140677157845|   1|
|2015|    Mission|30.463762673676303|   2|
|2015|   Foothill|30.353001803658852|   3|
|2016|   Van Nuys|32.194518462124094|   1|
|2016|West Valley| 31.40146437042384|   2|
+----+-----------+------------------+----+
only showing top 20 rows
```

```
+---------+-----------+---------+--------------+----------+
|      API|File_Format|Read_time|Execution_Time|Total_time|
+---------+-----------+---------+--------------+----------+
|Dataframe|        CSV|   17.656|        12.201|    29.857|
|      SQL|        CSV|   18.664|         8.087|    26.751|
+---------+-----------+---------+--------------+----------+
```

Αρχικά, φαίνεται το configuration της εκτέλεσης(το configuration επιλέχθηκε αυθαίρετα) και στην συνέχεια το αποτέλεσμα του Query. Αυτά είναι κοινά και για τα 2 APIs. Στην συνέχεια μπορούμε να δούμε τους χρόνους εκτέλεσης και των δύο υλοποιήσεων. Παρατηρούμε ότι οι χρόνοι εκτέλεσης και των 2 είναι παρόμοιοι, το οποίο είναι αναμενόμενο, καθώς και για τα 2 APIs το Pyspark χρησιμοποιεί τον catalyst optimizer για να βελτιστοποιήσει το πλάνο εκτέλεσης. Η πολύ μικρή διαφορά στον χρόνο ανάγνωσης είναι στα όρια του σφάλματος και εξαρτάται σε ένα βαθμό από τις τρέχουσες συνθήκες εκτέλεσης(συνθήκες του ίδιου του συστήματος).

Β) Στο δεύτερο μέρος του τρέχοντος ζητήματος θέλουμε να μετατρέψουμε το κυρίως dataset σε parquet format (τα δεδομένα να είναι αποθηκευμένα ανά στήλες , αντί ανά σειρές) σε ένα αρχείο και στην συνέχεια να εφαρμόσουμε μία εκ των παραπάνω υλοποιήσεων στο νέο αρχείο που αποθηκεύσαμε στο S3 bucket μας. Για να αποθηκευτεί σε ένα αρχείο χρησιμοποιούμε την εντολή coalesced(1) (Η εγγραφή του parquet αρχείου γίνεται στο τέλος του sql_csv.ipynb, ώστε να είμαστε βέβαιοι ότι κατά την εκτέλεση του sql_parquet.ipynb δεν θα υπάρχουν δεδομένα του στην μνήμη του executor).

```
# Reduce to a single partition
crime_data_df.coalesce(1).write.parquet(parquet_path, mode="overwrite")
```

Στην συνέχεια εκτελούμε το query χρησιμοποιώντας το SQL API και λαμβάνουμε τα ακόλουθα συνολικά αποτελέσματα για τον χρόνο:

```
+---------+-----------+---------+--------------+----------+
|      API|File_Format|Read_time|Execution_Time|Total_time|
+---------+-----------+---------+--------------+----------+
|Dataframe|        CSV|   17.656|        12.201|    29.857|
|      SQL|    Parquet|    5.730|        12.051|    17.781|
|      SQL|        CSV|   18.664|         8.087|    26.751|
+---------+-----------+---------+--------------+----------+
```

Παραπάνω, μπορούμε να πραγματοποιήσουμε δύο αξιοσημείωτες παρατηρήσεις. Πρώτον, η ανάγνωση των δεδομένων χρησιμοποιώντας το parquet file format είναι σημαντικά ταχύτερη.(στο paquet format τα δεδομένα αποθηκεύονται ανά στήλες και όχι ανα γραμμές, όπως στα csv) Αυτό οφείλεται στο γεγονός ότι αφενός τα parquet αρχεία υποστηρίζουν καλύτερους αλγόριθμους συμπίεσης, καθώς επιβάλουν ένα αυστηρό σχήμα για τα δεδομένα, με επακόλουθο να απαιτείται λιγότερο IO για το διάβασμα ενός parquet αρχείου(αντίθετα το csv αρχείο αποθηκεύεται σαν plain text).Αφετέρου τα parquet αρχεία διατηρούν μεταδεδομένα για το σχήμα των δεδομένων που έχουν αποθηκευτεί, με αποτέλεσμα να επιτυγχάνεται ταχύτερη ανάγνωση των δεδομένων σε σχέση με το csv, όπου το σχήμα πρέπει να γίνει infer (στον κώδικα μας βάλαμε την εντολή inferSchema και στο read.parquet προκειμένου να αποφύγουμε το lazy evaluation). Αναφορικά με την χειρότερη επίδοση του parquet σε σχέση με το ίδιο implementation σε CSV μπορεί να οφείλονται διάφοροι λόγοι. Γενικότερα, τα parquet προσφέρουν μεγαλύτερη επίδοση στην επεξεργασία των δεδομένων, αφού δίνουν την δυνατότητα να επενεργούμε σε συγκεκριμένα columns, χωρίς να χρειάζεται να διαβαστεί ολόκληρο το dataset(γεγονός ιδιαίτερα χρήσιμο για where clauses / filters). Εντούτοις, στην συγκεκριμένη περίπτωση βλέπουμε την υλοποίηση με το SQL API και CSV file format να έχει τον ταχύτερο χρόνο εκτέλεσης, κάτι που μπορεί να οφείλεται στις συνθήκες του συστήματος κατά την εκτέλεση(π.χ να υπάρχει πολύ traffic, να έχουμε κάποιον strangler κατά την εκτέλεση κλπ) ή στο γεγονός ότι η επεξεργασία που πραγματοποιείται στα δεδομένα είναι σχετικά απλή και δεν επωφελείται από τα μεταδεδομένα και τον τρόπο αποθήκευσης ανά στήλες που παρέχει το parquet format. Σε κάθε περίπτωση, βλέπουμε ότι η χρήση του parquet file format δίνει τον βέλτιστο συνολικό χρόνο εκτέλεσης.

Τρίτο Ζητούμενο

Στο τρίτο ζητούμενο θέλουμε να υπολογίσουμε για κάθε περιοχή του Los Angeles, το μέσο ετήσιο εισόδημα ανά περιοχή (με τον όρο περιοχή εννοούμε εδώ την στήλη COMM του Census Blocks dataset) και την αναλογία συνολικού αριθμού εγκλημάτων ανά άτομο. Για την πραγματοποίηση αυτού του Query πρέπει να εφαρμόσουμε ένα αυθαίρετο φιλτράρισμα στο Census Dataset. Συγκεκριμένα, πραγματοποιούμε το ακόλουθο filtering:

- Περιοριζόμαστε στην πόλη του Los Angeles θέτοντας city == “Los Angeles”
- Θεωρούμε πως ένα entry της απογραφής είναι έγκυρο αν έχει Housing_10>0 και POP_10>0. Entries για τα οποία δεν ισχύει το παραπάνω κρίνουμε ότι δεν εξυπηρετούν την ανάλυση εισοδήματος που θέλουμε να πραγματοποιήσουμε, καθώς το εισόδημα μας δίνεται με βάση τα νοικοκυριά(housing) και θέλουμε να υπολογίσουμε το κατά κεφαλήν εισόδημα(με βάση το POP). Συνεπώς, περιοχές χωρίς νοικοκυριά είναι «άχρηστες» καθώς το μέσο εισόδημα ανά νοικοκυριό, όπως φαίνεται στον κώδικα μας, πολλαπλασιάζεται με το housing για να υπολογίσουμε το συνολικό εισόδημα ανά ZIP. Αντίστοιχα περιοχές χωρίς πληθυσμό αλλά με Housing δεν θεωρούμε ότι νοηματικά συνάδουν και για αυτό τις αποκλείσαμε (θεωρούμε ότι τέτοια entries θα προσέθεταν θόρυβο στον υπολογισμό μας).
- Προφανώς, αποκλείσαμε entries που δεν είχαν ZIP ή COMM.

Μία σημείωση για την υλοποίηση μας είναι ότι εκμεταλλευτήκαμε την δυνατότητα του apache Sedona και πραγματοποιήσαμε join των δεδομένων περιοχής με τα δεδομένα των εγκλημάτων, ταιριάζοντας κάθε σημείο εγκλήματος με την γεωμετρία της περιοχής στην οποία το σημείο αυτό ανήκει. Οι υπόλοιποι υπολογισμοί είναι προφανείς(ουσιαστικά είναι απλοί υπολογισμοί μέσων όρων).Αρχικά, παραθέτουμε το αποτέλεσμα της εκτέλεσης του κώδικα μας, όπου φαίνεται το configuration(default),ο χρόνος εκτέλεσης του πλάνου εκτέλεσης που σχεδίασε ο catalyst optimizer(να σημειωθεί ότι είναι μετά από restart του kernel και σε αντίθεση με τις επόμενες μετρήσεις συνδυασμών join αλγορίθμων, που πιθανώς να έχουν κάποια δεδομένα στην μνήμη, σε αυτή την εκτέλεση έχουν διαβαστεί απλώς οι πίνακες),καθώς και οι πρώτες 20 γραμμές του ζητούμενου πίνακα:

```
Executor Instances: None
Executor Memory: 4743M
Executor Cores: 2
+--------------------+----------+-----------------------+
|                COMM|Crime rate|Estimated_Median_Income|
+--------------------+----------+-----------------------+
|     Adams-Normandie|    0.7149|                 $8,791|
|              Alsace|    0.5416|                $11,240|
|Angeles National ...|    0.4118|                $33,080|
|    Angelino Heights|    0.5733|                $18,427|
|              Arleta|    0.4265|                $12,111|
|     Atwater Village|    0.5288|                $28,481|
|       Baldwin Hills|    0.9950|                $17,304|
|             Bel Air|    0.3992|                $63,041|
|       Beverly Crest|    0.3690|                $60,947|
|         Beverlywood|    0.5085|                $29,268|
|       Boyle Heights|    0.6172|                 $8,494|
|           Brentwood|    0.4059|                $60,847|
|           Brookside|    0.8857|                $18,139|
|    Cadillac-Corning|    0.5817|                $19,573|
|         Canoga Park|    0.5506|                $19,660|
|             Carthay|    0.7629|                $49,849|
|             Central|    0.6594|                 $6,973|
|        Century City|    0.6330|                $45,618|
|  Century Palms/Cove|    1.1446|                 $8,610|
|          Chatsworth|    0.5281|                $30,695|
+--------------------+----------+-----------------------+
only showing top 20 rows

Execution Time : 27.029760599136353 seconds
```

Στην συνέχεια χρησιμοποιούμε την εντολή explain όπως φαίνεται ακολούθως, προκειμένου να λάβουμε το φυσικό πλάνο εκτέλεσης που έφτιαξε ο catalyst optimizer για εμάς:

```
== Physical Plan ==
AdaptiveSparkPlan (29)
+- Sort (28)
   +- Exchange (27)
      +- HashAggregate (26)
         +- Exchange (25)
            +- HashAggregate (24)
               +- Project (23)
                  +- RangeJoin (22)
                     :- Filter (18)
                     :  +- ObjectHashAggregate (17)
                     :     +- Exchange (16)
                     :        +- ObjectHashAggregate (15)
                     :           +- Project (14)
                     :              +- BroadcastHashJoin Inner BuildRight (13)
                     :                 :- ObjectHashAggregate (8)
                     :                 :  +- Exchange (7)
                     :                 :     +- ObjectHashAggregate (6)
                     :                 :        +- Project (5)
                     :                 :           +- Filter (4)
                     :                 :              +- Generate (3)
                     :                 :                 +- Filter (2)
                     :                 :                    +- Scan geojson  (1)
                     :                 +- BroadcastExchange (12)
                     :                    +- Project (11)
                     :                       +- Filter (10)
                     :                          +- Scan csv  (9)
                     +- Project (21)
                        +- Filter (20)
                           +- Scan csv  (19)
```

Από το παραπάνω φυσικό πλάνο μπορούμε να δούμε την σειρά εκτέλεσης των operations, αλλά και την μέθοδο που θα χρησιμοποιηθεί για να εκτελεστούν αυτά. Προφανώς, μπορούμε να δούμε τα operations όπως τα ορίσαμε και στον κώδικα μας(χαρακτηριστικό παράδειγμα είναι ότι (1) είναι το scan geojson η πρώτη εντολή στον κώδικα μας, που θα εκτελεστεί από το spark. Αναλόγως η τελευταία είναι η sort, καθώς για την παρουσίαση των αποτελεσμάτων χρησιμοποιούμε το transformation orderBy (στην πραγματικότητα στο τέλος εκτελούμε το action show(), αλλά δεν φαίνεται στο φυσικό πλάνο αυτό). Η πλειονότητα των points στο physician plan είναι προφανή(π.χ όπου έχουμε το transformation agg εκτελεί ObjectHashAggregate, όπου έχουμε filter εκτελεί filter κ.λ.π). Στο σημείο που θα εστιάσουμε από αυτό το σημείο και έπειτα είναι η επιλογή των join αλγορίθμων από τον Catalyst optimizer. Το πρώτο join που εφαρμόζουμε στον κώδικα μας φαίνεται ακολούθως(μετά από αυτό πραγματοποιούνται και άλλα transformations που δεν φαίνονται γιατί δεν μας ενδιαφέρουν για αυτή την ανάλυση):

```
joined_income_dataset = zip_comm_population.join(
    income_data_cleaned,
    zip_comm_population["ZCTA10"] == income_data_cleaned["Zip Code"],
    "inner"
    ) \
```

Σε αυτό το σημείο βλέπουμε να πραγματοποιείται ένα inner join μεταξύ των income_data (φιλτραρισμένα δεδομένα LA_income_2015) και του zip_comm_population (φιλτραρισμένα δεδομένα 2010_Census_Blocks). Για αυτό το join ο catalyst optimizer επιλέγει να εφαρμόζει ένα Broadcast Hash join με build το δεξί argument (δηλαδή το income_data_cleaned). Ο Catalyst Optimizer επιλέγει αυτό το είδος join, διότι το μέγεθος του income_data_cleaned dataframe είναι αρκετά μικρό(όπως θα φαίνεται από την ακόλουθη εικόνα έχει μόνο 284 entries):

```
num_rows = income_data_cleaned.count()
num_columns = len(income_data_cleaned.columns)
print(f"Income_data_cleaned number of columns: {num_columns}")
print(f"Income_data_cleaned number of entries: {num_rows}")
```

```
Income_data_cleaned number of columns: 2
Income_data_cleaned number of entries: 284
```

Όταν πραγματοποιούμε ένα broadcast hash join ένα αντίγραφο της μικρής σε μέγεθος σχέσης αποστέλλεται σε όλους τους worker nodes, όπου εκεί κάθε ένας εκτελεί ένα hash join ανάμεσα σε ολόκληρο το μικρό dataset και στο κομμάτι του μεγάλου dataset που διαθέτει διαθέσιμο στην μνήμη του. Ο catalyst optimizer προτιμά αυτή την στρατηγική όταν έχουμε κάποιο μικρό dataset, διότι με τις υπόλοιπες στρατηγικές έχουμε ένα shuffling κόστος (μεταφορά δεδομένων των οποίων το κλειδί δίνει το ίδιο αποτέλεσμα μετά από το πέρασμα από μία hash function), το οποίο μπορεί να υπερβαίνει το κόστος της απλής αποστολής ολόκληρου του μικρού dataset σε όλους τους κόμβους(π.χ εδώ αν εκτελέσουμε shuffle στο μεγάλο dataset zip_comm_population μπορεί να δημιουργήσουμε πολλά παραπάνω transfers στο Interconnection network). Είναι πλέον προφανές γιατί ο catalyst optimizer επέλεξε αυτή την μέθοδο join για το πρώτο Join του κώδικα μας. Συνεχίζουμε παραθέτοντας το δεύτερο join:

```
joined_income_dataset.join(
crime_data_with_geometry,
ST_Within(crime_data_with_geometry.geom,
joined_income_dataset.geometry), "inner")
```

Παρατηρούμε, ότι αυτό το Join γίνεται με βάση την γεωμετρία. Η στήλη αυτή για το joined_income_dataset ορίζει κάποια πολύγωνα, των οποίων ο χώρος που καλύπτουν αντιστοιχεί σε ένα COMM, ενώ για το crime_data_with_geometry η αντίστοιχή στήλη ορίζει το σημείο του εγκλήματος. Το join αυτό ορίζει ότι ένα σημείο εγκλήματος θα αντιστοιχηθεί με το πολύγωνο(δηλαδή το COMM) εντός του οποίου βρίσκεται. Προφανώς, λοιπόν το join αυτό ,για να ελέγξει που ανήκει ένα σημείο, πραγματοποιεί κάποιους ελέγχους ανισοτήτων. Το Spark πραγματοποιεί βέλτιστα Join με βάση ανισότητες χρησιμοποιώντας το RangeJoin και για αυτό ακριβώς τον λόγο χρησιμοποιεί αυτή την μέθοδο και στην δική μας περίπτωση.

Εφόσον πλέον εξηγήσαμε τις επιλογές του Catalyst Optimizer για τα join operation μπορούμε να προχωρήσουμε σε μία συγκριτική ανάλυση των εξής join μεθόδων για τα 2 join που πραγματοποιούμε στον κώδικά μας:

- BROADCAST
- MERGE
- SHUFFLE_HASH
- SHUFFLE_REPLICATE_NL

Ουσιαστικά εξηγήσαμε ήδη την λειτουργία του broadcast join και η λειτουργία των merge και shuffle hash είναι προφανείς αφού εξηγήσαμε το hash. Αυτές οι μέθοδοι πραγματοποιούν ένα Hashing στην αρχή με βάση το attribute που θα γίνει το join στέλνοντας στον ίδιο worker entries με το ίδιο αποτέλεσμα του hash. Σε κάθε worker πραγματοποιείται ή merge join ή hash join αντίστοιχα. Τέλος, για το shuffle replicate Nested loop join δημιουργείται ένα αντίγραφο του μικρού dataset (replicate) σε όλους τους workers, ενώ το μεγάλο dataset γίνεται shuffle, ώστε να είναι διαθέσιμο σε πολλούς κόμβους. Στην συνέχεια σε κάθε κόμβο εκτελείται ένα nested loop join(εξωτερικό loop το μεγάλο dataset και εσωτερικά το μικρό). Πλέον, εφόσον εξηγήθηκε η αρχή λειτουργίας κάθε join μεθόδου μπορούμε να σχολιάσουμε τα ακόλουθα πειραματικά αποτελέσματα(Σημειώνουμε ότι για την καλύτερη μέτρηση των αποτελεσμάτων προτείνεται η πραγματοποίηση επανεκκίνησης του kernel πριν την εκτέλεση κάθε νέου configuration):

First_Join_Strategy	Second_Join_Strategy	Execution_Time
BROADCAST	SHUFFLE_HASH	18.962
SHUFFLE_HASH	MERGE	20.181
SHUFFLE_REPLICATE_NL	BROADCAST	20.278
BROADCAST	MERGE	20.745
SHUFFLE_REPLICATE_NL	SHUFFLE_REPLICATE_NL	21.029
MERGE	SHUFFLE_HASH	21.050
SHUFFLE_HASH	SHUFFLE_HASH	21.169
BROADCAST	SHUFFLE_REPLICATE_NL	21.249
SHUFFLE_REPLICATE_NL	MERGE	21.499
SHUFFLE_HASH	BROADCAST	21.720
SHUFFLE_HASH	SHUFFLE_REPLICATE_NL	22.807
MERGE	SHUFFLE_REPLICATE_NL	25.419
BROADCAST	BROADCAST	25.811
MERGE	BROADCAST	26.330
MERGE	MERGE	28.349
SHUFFLE_REPLICATE_NL	SHUFFLE_HASH	32.894

Από το παραπάνω διάγραμμα βλέπουμε το χρόνο εκτέλεσης κάθε πιθανού συνδυασμού μεθόδων για τα 2 join που πραγματοποιούνται. Την καλύτερη επίδοση φαίνεται να έχει η χρήση του broadcast join σαν στρατηγική για το πρώτο join και η shuffle_hash για το δεύτερο. Αυτό ήταν σχετικά αναμενόμενο, καθώς και ο catalyst optimizer χρησιμοποιεί για το πρώτο του join το broadcast(για λόγους που ήδη εξηγήσαμε). Για το δεύτερο join βλέπουμε ότι γενικά καλύτερη επίδοση έχουν οι μέθοδοι shuffle merge join και shuffle hash join. Οι άλλες δύο μέθοδοι δεν είναι το ίδιο αποτελεσματικές για το δεύτερο Join, διότι αφενός και τα δύο dataset δεν είναι πάρα πολύ μικρά(ειδικά το crimes είναι πολύ μεγάλο) και συνεπώς το broadcast δεν μπορεί να οδηγήσει απαραίτητα σε καλύτερη απόδοση(κάποιες φορές εξαρτάται και από το πως είναι κατανεμημένα τα δεδομένα). Για τον ίδιο λόγο (το μέγεθος των dataset) δεν είναι τόσο αποτελεσματικό και το shuffle replicate, καθώς πρέπει να δημιουργήσει ένα αντίγραφο του μικρού dataset σε κάθε κόμβο. Αναφορικά με την σύγκριση των άλλων δύο μεθόδων αμφότερες εκτελούν ένα shuffling και στην συνέχεια εκτελούν merge ή hash join σε κάθε κόμβο αντίστοιχα. Οπότε η διαφορά στην επίδοση τους βρίσκεται στο ποια μέθοδος είναι πιο αποτελεσματική για τα δικά μας dataset. Το merge join δεν είναι τόσο αποδοτικό όσο το hash join, για μεγάλα dataset, τα οποία είναι μη ταξινομημένα ως προς το join attribute, καθώς χρειάζονται ταξινόμηση ως προς αυτό. Για αυτούς τους λόγους είναι πιθανό να βλέπουμε το shuffle hash join να δίνει την καλύτερη επίδοση για το δεύτερο join σε σχέση με τα υπόλοιπα join strategies.

Τέταρτο Ζητούμενο

Στο Τέταρτο Query μας ζητήθηκε να βρεθεί το φυλετικό προφίλ των καταγεγραμμένων θυμάτων εγκλημάτων στο Los Angeles (city) για το έτος 2015 στις 3 περιοχές με το υψηλότερο κατά κεφαλήν εισόδημα και στις 3 περιοχές με το χαμηλότερο κατά κεφαλήν εισόδημα. Για την εύρεση των περιοχών με το χαμηλότερο και το υψηλότερο κατά κεφαλήν εισόδημα εφαρμόσαμε την ίδια λογική και περιορισμούς με αυτούς που αναλύσαμε στο ζητούμενο 3.Για την υλοποίηση αυτού του Query χρησιμοποιούμε το DataFrame API. Στην συνέχεια παρουσιάζουμε τα αποτελέσματα της εκτέλεσης του ζητούμενου Query για τα 3 διαφορετικά configurations:

Top 3:

Victim Descent	#
White	649
Other	72
Hispanic/Latin/Me...	66
Unknown	38
Black	37
Other Asian	21
American Indian/A...	1
Chinese	1

Bottom 3:

Victim Descent	#
Hispanic/Latin/Me...	2815
Black	761
White	330
Other	187
Other Asian	113
Unknown	22
American Indian/A...	21
Korean	5
Chinese	3
AsianIndian	1
Filipino	1

spark_executor_cores	spark_executor_memory	execution_time
4	8g	60.674782514572144
2	4g	70.12353825569153
1	2g	79.3983838558197

Παραπάνω, μπορούμε να δούμε το αποτέλεσμα της κλιμάκωσης των υπολογιστικών πόρων στο χρόνο εκτέλεσης. Σε κάθε περίπτωση χρησιμοποιούμε 2 spark executors για την εκτέλεση του query και σταδιακά κλιμακώνουμε τους πόρους στους οποίους έχει πρόσβαση κάθε executor, διπλασιάζοντας κάθε φορά το σύνολο των

πυρήνων και της μνήμης στην οποία έχει πρόσβαση. Αυτό προφανώς οδηγεί σε καλύτερους χρόνους εκτέλεσης, καθώς όσο αυξάνονται οι υπολογιστικοί πόροι, αυξάνονται τα task που μπορεί να εκτελέσει παράλληλα (αυτό ισχύει και γιατί η αναλογία πυρήνων μνήμης παραμένει σταθερή).Επιπλέον, περισσότερη μνήμη ανά executor συνεπάγεται ότι περισσότερα δεδομένα θα μπορούν να είναι ανα πάσα στιγμή στην μνήμη , μειώνοντας τις περιπτώσεις όπου δεδομένα δεν θα χωρούν στην μνήμη και θα πρέπει να αποθηκευτούν και να διαβαστούν από τον δίσκο(memory spills). Ωστόσο, παρότι όντως έχουμε βελτίωση στο χρόνο εκτέλεσης δεν παρατηρούμε linear speedup, όπως θα ήταν το ιδανικό.Αυτό μπορεί να οφείλεται σε διάφορους λόγους όπως:

- Οι επεξεργαστές μοιράζονται τους πόρους του συστήματος με αποτέλεσμα να δημιουργείται κάποιο overhead λόγω του ανταγωνισμού αυτών για την απόκτηση των πόρων
- Κόστος επικοινωνίας τόσο μεταξύ των επεξεργαστών που αυξάνονται όσο και μεταξύ των executors(π.χ για την χρήση locks,επικοινωνία μέσω Interconnection network)

Πέμπτο Ζητούμενο

Στο Πέμπτο Query μας ζητείται να υπολογιστεί η ο αριθμός εγκλημάτων που έλαβαν χώρα πλησιέστερα σε αυτό, καθώς και η μέση απόστασή του από τις τοποθεσίες όπου σημειώθηκαν τα συγκεκριμένα περιστατικά. Για την υλοποίηση αυτού του Query χρησιμοποιούμε το DataFrame API. Αρχικά, παρουσιάζουμε το αποτέλεσμα αυτού του Query, καθώς και τους χρόνους εκτέλεσης για τα 3 διαφορετικά configurations:

Division	average_distance_km	crime_count
HOLLYWOOD	2.0763	224340
VAN NUYS	2.9534	210134
SOUTHWEST	2.1914	188901
WILSHIRE	2.5927	185996
77TH STREET	1.7165	171827
OLYMPIC	1.7236	170897
NORTH HOLLYWOOD	2.6430	167854
PACIFIC	3.8501	161359
CENTRAL	0.9925	153871
RAMPART	1.5345	152736
SOUTHEAST	2.4219	152176
WEST VALLEY	3.0357	138643
TOPANGA	3.2970	138217
FOOTHILL	4.2509	134896
HARBOR	3.7026	126747
HOLLENBECK	2.6802	115837
WEST LOS ANGELES	2.7925	115781
NEWTON	1.6346	111110
NORTHEAST	3.6237	108109
MISSION	3.6909	103355

Executor instances	cores	memory	time
2	4	8g	31.710615396499634
4	2	4g	37.794593811035156
8	1	2g	54.164450883865356

Σε αντίθεση με το προηγούμενο ζητούμενο, εδώ δοκιμάζουμε διαφορετικούς συνδυασμούς executor instances, cores και memory διατηρώντας σταθερό το συνολικό αριθμό πυρήνων και συνολικής μνήμης. Αυτό που παρατηρούμε είναι

ότι το configuration με λίγους αλλά “ισχυρότερους” executors(περισσότερα cores και μνήμη) αποδίδει καλύτερα από τα αντίστοιχα configurations όπου είναι πιο διαμοιρασμένοι οι πόροι(η πρώτη περίπτωση είναι περίπτωση coarse grained parallel machine, ενώ η δεύτερη fine grained parallel machine). Η καλύτερη επίδοση ενός coarse grained parallel machine μπορεί να οφείλεται σε διάφορους λόγους . Πιο συγκεκριμένα:

- Η ύπαρξη περισσότερων spark executor instances έχει ως αποτέλεσμα την ύπαρξη Overhead για την χρονοδρομολόγηση και τον συντονισμό των instances από τον cluster manager.
- Η πραγματοποίηση transformation, που απαιτούν shuffling των δεδομένων στον cluster έχουν μεγαλύτερο κόστος (σε χρόνο) εκτέλεσης στα συστήματα με περισσότερους executors, καθώς απαιτείται η μεταφορά των δεδομένων ανάμεσα σε αυτούς.(π.χ για την εκτέλεση ενός shuffle-hash join απαιτείται μεταφορά μέρους του dataset ανάμεσα στους executors).Αντίθετα σε συστήματα, όπου έχουμε μικρότερο διαμοιρασμό πόρων κάθε executor μπορεί να εκτελέσει μεγαλύτερα tasks, με αποτέλεσμα να ελαχιστοποιείται το κόστος μεταφορών δεδομένων μέσω του interconnection network.
- Στο configuration με τα 8gb ανά executor, κάθε executor έχει περισσότερη μνήμη διαθέσιμη με αποτέλεσμα να μπορεί να διατηρεί κάθε στιγμή περισσότερα δεδομένα στην μνήμη του αποφεύγοντας τα memory spills(περιπτώσεις όπου η μνήμη πλέον δεν χωράει όλα τα δεδομένα και πρέπει να πραγματοποιηθούν προσβάσεις στον δίσκο).