ΣΥΣΤΗΜΑΤΑ ΔΙΑΧΕΙΡΙΣΗΣ ΜΕΓΑΛΟΥ ΟΓΚΟΥ ΔΕΔΟΜΕΝΩΝ

ΜΙΧΑΗΛ ΣΟΛΟΜΩΝΙΔΗΣ - ΑΜ 2121063
ΚΩΝΣΤΑΝΤΙΝΟΣ ΚΙΟΥΣΗΣ - ΑΜ 2121129

Εισαγωγή

Η παρούσα εργασία εκπονήθηκε στο πλαίσιο του μαθήματος «Συστήματα Διαχείρισης Μεγάλου Όγκου Δεδομένων» και έχει ως στόχο την υλοποίηση και αξιολόγηση δεδομένων μεγάλου όγκου με χρήση του Apache Spark σε περιβάλλον Kubernetes. Αντικείμενο της εργασίας είναι η πειραματική διερεύνηση διαφορετικών τρόπων υλοποίησης queries (με RDDs, DataFrames και SQL), η σύγκριση διαφορετικών μορφών αποθήκευσης (π.χ. CSV vs Parquet), και η ανάλυση των στρατηγικών που χρησιμοποιεί ο Spark Catalyst Optimizer για την εκτέλεση joins.

Υλοποίηση εργασίας με χρήση γλώσσας προγραμματισμού Python, ενώ η επεξεργασία των queries έγινε μέσω Spark APIs. Καταγράφηκαν μετρήσεις χρόνου εκτέλεσης για διαφορετικές υλοποιήσεις και μορφές αρχείων, αναλύθηκαν οι επιλογές στρατηγικών joins, και διατυπώθηκαν τεκμηριωμένα συμπεράσματα σχετικά με την απόδοση κάθε προσέγγισης. Η αναφορά αυτή περιλαμβάνει τα αποτελέσματα των πειραμάτων, θεωρητική αξιολόγηση των επιλογών του Catalyst Optimizer, και τις σχετικές τεχνικές αποφάσεις, ενώ υπάρχει ανεβασμένο αποθετήριο με τους κώδικες στο GitHub.

Περιγραφή Υλοποίησης

Υπάρχουν 6 queries (Q1–Q6) κάθε ένα από αυτά υλοποιήθηκε με διαφορετικές μεθόδους και εργαλεία του Apache Spark: RDDs, DataFrames και Spark SQL. Επίσης δοκιμάσαμε και τις δύο μορφές αποθήκευσης: αρχεία CSV και Parquet.

Q1 – Φιλτράρισμα και Ομαδοποίηση

Χρόνος Εκτέλεσης

Q1-RDD: ήταν 245 sec

Q1-DF: 434 sec.

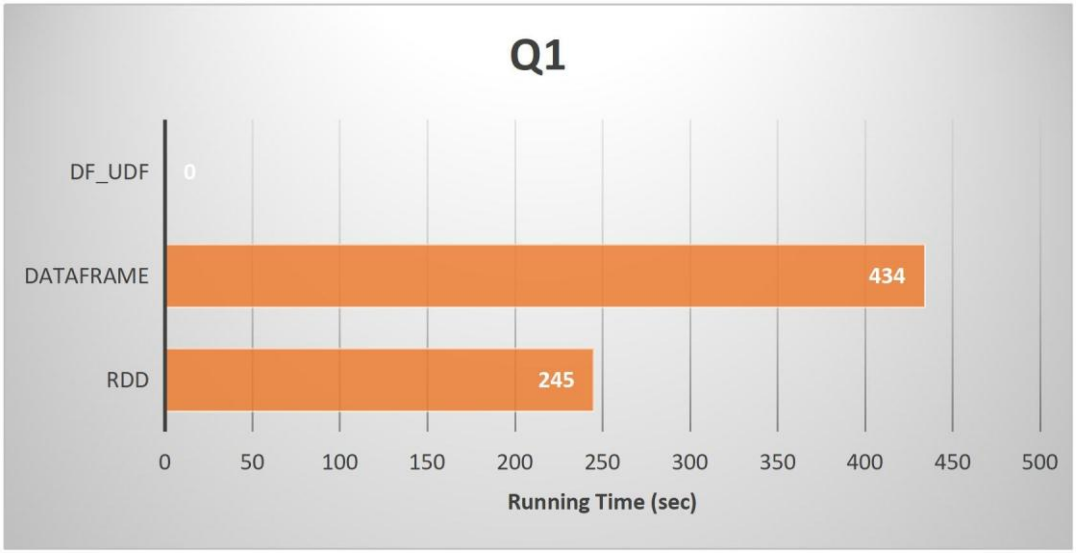
Η σταθερότητα της DF υλοποίησης δείχνει καλύτερο καταμερισμό εργασίας και αποδοτικότερο scheduling, χάρη στο Catalyst Optimizer.

Επιπλέον, η υλοποίηση με UDF σε DF απέτυχε σε πολλές περιπτώσεις με OOMKilled, δείχνοντας ότι custom συναρτήσεις σε Spark χωρίς optimization μπορούν να είναι επικίνδυνες σε μεγάλους όγκους.

Οι UDFs συχνά παρακάμπτουν τις βελτιστοποιήσεις του Catalyst, και αυτό φάνηκε έντονα εδώ.

Συμπέρασμα

Η DataFrame υλοποίηση χωρίς UDF είναι μακράν η πιο αποδοτική, σταθερή και ασφαλής. Οι RDDs είναι χρήσιμες για εκπαιδευτικούς σκοπούς και βαθύτερο έλεγχο, αλλά χάνουν στην απόδοση και την οικονομία πόρων.



Q2 – Υπολογισμός Μοναδικών Τιμών

Το Query 2 είχε ως στόχο να εντοπίσει, για κάθε εταιρεία ταξί (VendorID), τη διαδρομή με τη μεγαλύτερη ευκλείδεια απόσταση (Haversine), υπό την προϋπόθεση ότι:

1)η διάρκεια της διαδρομής ήταν πάνω από 10 λεπτά,

2)η απόσταση δεν ξεπερνούσε τα 50 χλμ.

Η υλοποίηση έγινε με τρεις τρόπους: RDD, DataFrame, και SQL πάνω σε DataFrame.

Χρόνος Εκτέλεσης

Σύμφωνα με τις μετρήσεις:

RDD: 135 δευτερόλεπτα

SQL DataFrame: 311 δευτερόλεπτα

DataFrame με UDF: 528 δευτερόλεπτα

Η RDD ήταν σχεδόν 4 φορές ταχύτερη από τη DF υλοποίηση, παρόλο που συνήθως οι DF είναι ταχύτερες.

DataFrame με UDF:

Ο υπολογισμός Haversine έγινε μέσω udf().

Οι UDFs δεν είναι vectorized: επεξεργάζονται γραμμή-γραμμή και οδηγούν σε μεγάλη καθυστέρηση.

Επιπλέον, χρησιμοποιούνται πολλές withColumn() μετατροπές που είναι ακριβές.

SQL πάνω σε DataFrame:

Εδώ το haversine ήταν UDF αλλά χρησιμοποιήθηκε μέσα σε SQL query.

Αν και λίγο καλύτερη από την προηγούμενη, παραμένει αργή λόγω της ίδιας UDF λογικής.

Επιδεικνύει καλύτερο performance πιθανώς λόγω ROW_NUMBER() optimization.

RDD:

Η υλοποίηση έγινε σε χαμηλότερο επίπεδο, με καθαρή Python λογική.

Αν και δεν γίνεται καμία optimization από το Spark, ο λογικός διαχωρισμός και η γρήγορη χρήση .reduceByKey() οδήγησε σε χαμηλό χρόνο και σχετικά χαμηλή χρήση πόρων.

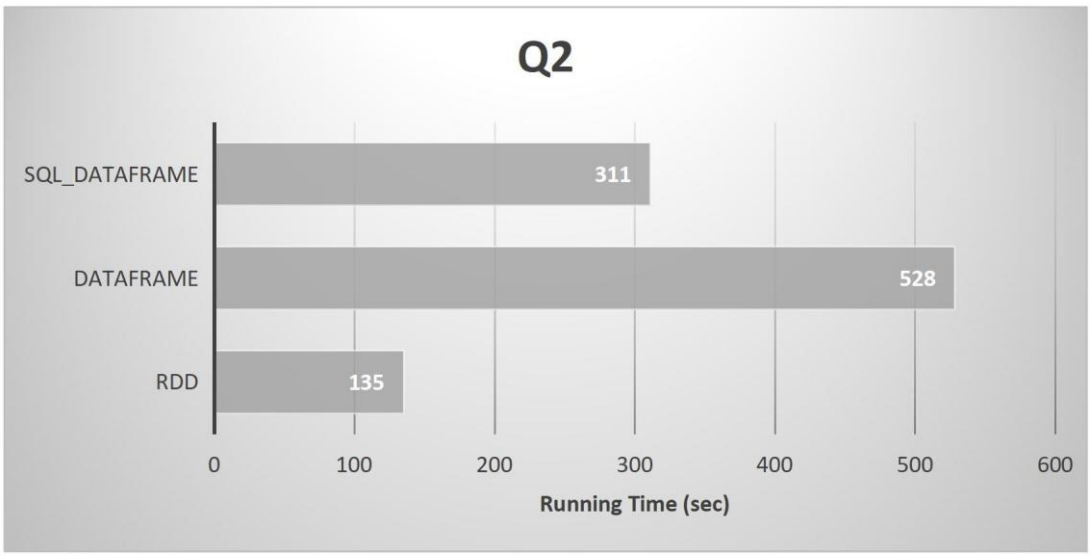
Συμπέρασμα

Σε αυτή την περίπτωση, ο RDD κώδικας αποδείχτηκε ταχύτερος, παρότι γενικά θεωρείται λιγότερο αποδοτικός. Ο κύριος λόγος είναι η χρήση User Defined Functions (UDFs) στις άλλες δύο εκδοχές, που παρακάμπτουν τις εσωτερικές βελτιστοποιήσεις του Spark.

Επομένως:

Αν δεν μπορούμε να αποφύγουμε UDFs, ίσως συμφέρει RDD για μαθηματικά υπολογιστικά queries.

Αν χρησιμοποιηθεί SQL χωρίς UDF ή με built-in συναρτήσεις, τότε το DataFrame/SQL υπερέχει.



Parquet

Υλοποιήθηκε κώδικας που μετατρέπει τα csv αρχεία (yellow_tripdata_2015.csv, yellow_tripdata_2024.csv, taxi_zone_lookup.csv) σε αρχεία .parquet Apache-Spark με χρήση εντολών από τον οδηγό του Apache-Spark και τα αποθηκεύει στο path:

https://hdfs-namenode:9870/user/kkiousis/data/parquet

Η εκτέλεση αυτού του κώδικα διήρκησε 376 sec.

Q3 – Join και Aggregation

Το Query 3 στόχευε στον υπολογισμό του αριθμού διαδρομών που ξεκινούν και τελειώνουν στην ίδια γεωγραφική περιοχή (Borough). Αξιοποιήθηκαν τα πεδία PULocationID, DOLocationID από το αρχείο διαδρομών, και ο πίνακας taxi_zone_lookup.

Το query υλοποιήθηκε με τέσσερις τρόπους:

1)DataFrame API με CSV

2)SQL API με CSV

3)DataFrame API με Parquet

4)SQL API με Parquet

DataFrame με CSV – 948s

Χρησιμοποιεί διπλό join σε δύο αντίγραφα του πίνακα περιοχών.

Όλα τα δεδομένα διαβάζονται στη μνήμη.

Καθώς τα CSV αρχεία δεν είναι columnar, το Spark πρέπει να κάνει parsing σε κάθε στήλη.

Το query είναι το ίδιο λειτουργικά, αλλά εκτελείται πιο "βαριά" σε σχέση με τα υπόλοιπα.

SQL με CSV – 471s

Ίδιο query, αλλά σε SQL συντακτικό.

Ο Catalyst Optimizer φαίνεται να κάνει κάποιες βελτιστοποιήσεις.

Σχεδόν 2 φορές ταχύτερο από το DataFrame API, παρότι το format είναι ίδιο.

DataFrame με Parquet – 60s

Η χρήση Parquet μειώνει δραστικά το I/O.

Διαβάζονται μόνο οι απαιτούμενες στήλες.

Ο συνδυασμός columnar storage + optimized joins οδηγεί σε τεράστια μείωση χρόνου.

SQL με Parquet – 55s

Η πιο αποδοτική υλοποίηση.

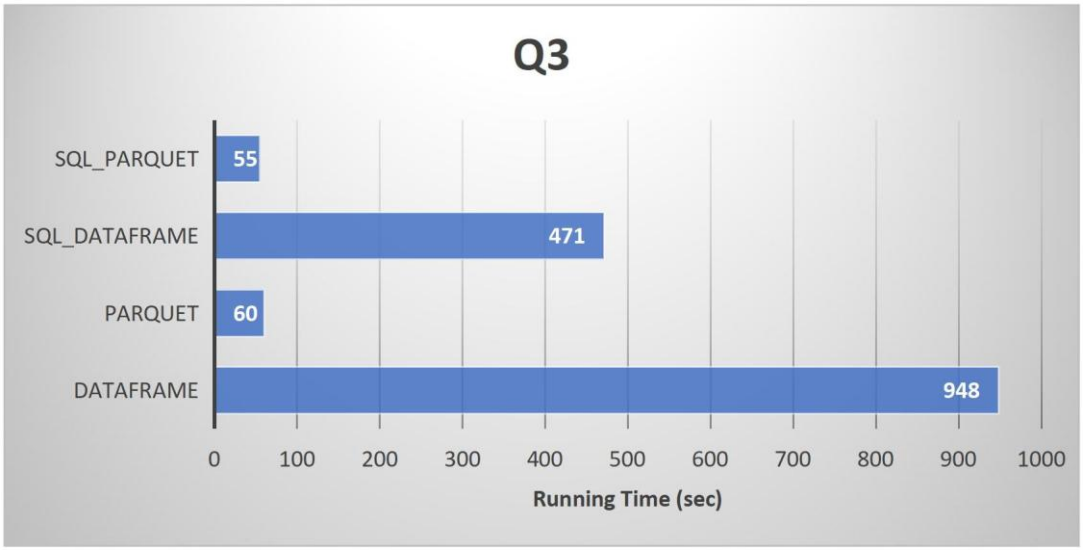
Χρησιμοποιεί temporary views και SQL syntax.

Ο Catalyst Optimizer αξιοποιεί πλήρως το Parquet format για να κάνει pushdown filters, project μόνο τις στήλες που χρειάζονται, και αποφεύγει shuffle όσο γίνεται.

Ιδανική επιλογή για queries με joins και groupings.

Συμπέρασμα

Το Query 3 είναι χαρακτηριστικό παράδειγμα του πόσο σημαντική είναι η επιλογή μορφής αρχείων (CSV vs Parquet), και η χρήση του κατάλληλου API. Ο πιο γρήγορος τρόπος εκτέλεσης ήταν το SQL + Parquet (55s), ενώ ο πιο αργός το DataFrame + CSV (948s).
Η διαφορά αγγίζει τα 17x, τονίζοντας τη σημασία του σωστού storage format και του optimized API.



Q4 – Join με Projection

(Δεν χρησιμοποιήθηκαν joins αλλά η εντολή GROUP BY)

Το Query 4 έχει ως σκοπό να υπολογίσει τον αριθμό των διαδρομών που πραγματοποιήθηκαν τις νυχτερινές ώρες (23:00 – 06:59) για κάθε εταιρεία ταξί (VendorID). Η υλοποίηση έγινε εξ ολοκλήρου με SQL και εξετάσαμε δύο μορφές εισόδου:

1)CSV αρχείο

2)Parquet αρχείο

Υλοποίηση με SQL + CSV

Διαβάστηκε το αρχείο yellow_tripdata_2024.csv ως CSV.

Ορίστηκε προσωρινό SQL view (night_rides).

Εκτελέστηκε SQL query με φίλτρο σε ώρες και ομαδοποίηση ανά VendorID.

Χρόνος Εκτέλεσης: 414 δευτερόλεπτα

Η χρήση CSV έχει μεγάλο I/O overhead: κάθε στήλη διαβάζεται ανεξαρτήτως του αν τη χρειαζόμαστε, και το parsing του χρόνου είναι αργό.

Υλοποίηση με SQL + Parquet

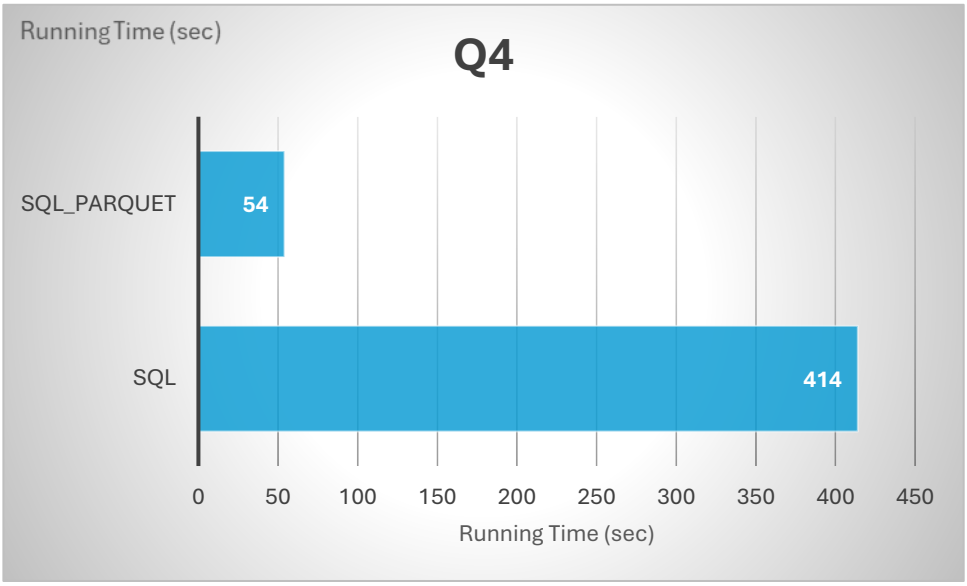
Τα δεδομένα διαβάστηκαν από Parquet format.

Εκτελέστηκε ίδιο SQL query.

Χρόνος Εκτέλεσης: 54 δευτερόλεπτα

Το columnar format του Parquet επιτρέπει στο Spark να διαβάσει μόνο τις στήλες tpep_pickup_datetime και VendorID, με αποτέλεσμα πολύ πιο αποδοτική εκτέλεση.

Το ίδιο ακριβώς SQL query, όταν εφαρμόζεται σε Parquet δεδομένα, είναι πολλαπλάσια πιο γρήγορο. Αυτό οφείλεται στη φύση του Parquet που αποθηκεύει τις στήλες χωριστά και επιτρέπει αποδοτική ανάγνωση μόνο των απαιτούμενων πεδίων. Αντιθέτως, το CSV απαιτεί πλήρη ανάγνωση και parsing όλου του αρχείου, κάτι που καθυστερεί σημαντικά την εκτέλεση.



Q5 – Σύνθετο Join με Πολλαπλά Κριτήρια

Το Query 5 ζητούσε τον υπολογισμό του πλήθους διαδρομών που ξεκίνησαν και κατέληξαν σε διαφορετικές ζώνες (Zones), εντός των ίδιων γεωγραφικών περιοχών (Boroughs), για κάθε συνδυασμό Pickup Zone και Dropoff Zone. Ήταν ένα query με δύο join, filtering και ομαδοποίηση. Το υλοποιήσαμε δύο φορές, με τη μόνη διαφορά να είναι ο τρόπος εισαγωγής των δεδομένων: CSV vs Parquet.

DataFrame με CSV – 579 sec

Το Spark χρειάζεται να κάνει parsing όλων των πεδίων, και αυτό επιβαρύνει τη διαδικασία, ειδικά όταν το join εφαρμόζεται δύο φορές. Δεν υπάρχει καμία δυνατότητα για pushdown filters ή partition pruning.

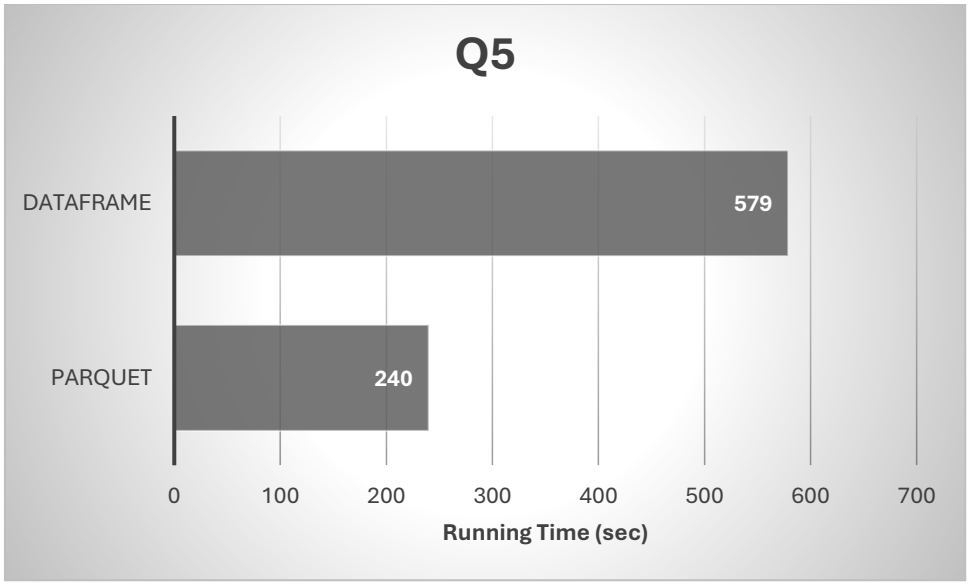
Το query λειτουργεί σωστά, αλλά είναι σαφώς βαρύ λόγω format.

DataFrame με Parquet – 240 sec

Τα Parquet αρχεία είναι columnar, οπότε το Spark διαβάζει μόνο τις στήλες PULocationID, DOLocationID, LocationID, Zone, Borough. Επιπλέον, χάρη στο optimized Catalyst execution plan, το Spark μπορεί να εκτελέσει joins και aggregations πιο αποδοτικά. Το ίδιο query ολοκληρώνεται σε λιγότερο από τη μισή ώρα, με σημαντική εξοικονόμηση σε μνήμη και χρόνο.

Συμπέρασμα

Η χρήση Parquet format στο Query 5 είχε ως αποτέλεσμα μείωση του χρόνου εκτέλεσης κατά 58%, συγκριτικά με το CSV. Αυτό αποδεικνύει για ακόμη μία φορά ότι η επιλογή αποθηκευτικού format είναι κρίσιμη και ότι οι Parquet files είναι ιδανικές για queries με join, filtering και aggregation – ειδικά όταν χρησιμοποιούνται με το DataFrame API.



Q6 – Παράλληλη Εκτέλεση

Το Query 6 ζητούσε να υπολογιστεί ο αριθμός και ο μέσος όρος απόστασης διαδρομών ανά μήνα, αξιοποιώντας το πεδίο tpep_pickup_datetime για grouping και το trip_distance για aggregation. Η υλοποίηση έγινε με το DataFrame API, και ο στόχος ήταν να μελετήσουμε πώς επηρεάζουν οι διαφορετικές στρατηγικές κατανομής πόρων την απόδοση της ίδιας επεξεργασίας.

2x4x8 – Ισχυροί αλλά λίγοι executors

Κάθε executor είχε πολύ υπολογιστική ισχύ (4 cores) και μνήμη (8 GB).

Παρότι ισχυροί, ήταν μόνο 2, άρα περιορισμένος παραλληλισμός σε data partitioning.

Το I/O overhead δεν διαμοιράζεται καλά με λίγους executors.

Ανθεκτικό, αλλά όχι αποδοτικό για μεγάλα datasets.

4x2x4 – Ενδιάμεση λύση

4 executors, με 2 cores και 4GB ο καθένας.

Παρουσίασε καλύτερο καταμερισμό φόρτου και μειωμένο χρόνο εκτέλεσης σε σχέση με 2x4x8.

Ισορροπημένη λύση, καλύτερη εκμετάλλευση παράλληλης επεξεργασίας.

8x1x2 – Πολλοί μικροί executors

Η πιο λεπτομερής κατανομή εργασίας: 8 executors με 1 core και 2GB.

Η αυξημένη παραλληλία οδήγησε σε καλύτερο pipelining και μικρότερο χρόνο εκτέλεσης.

Ειδικά σε queries με I/O και απλό aggregation (όπως αυτό), η “ποσότητα” εκτελεστών υπερτερεί της “ποιότητας”.

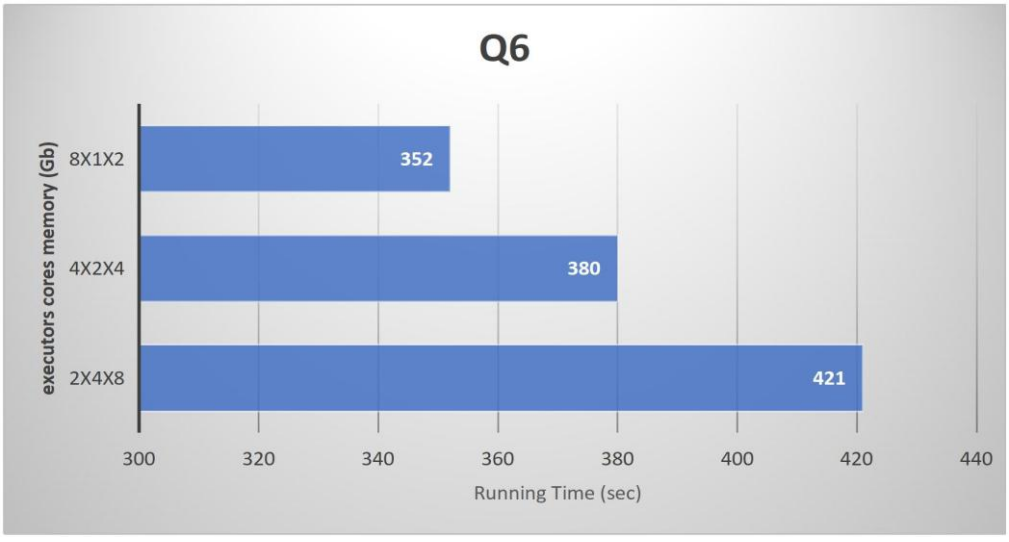
Η πιο αποδοτική λύση – καλύτερη αξιοποίηση των διαθέσιμων πόρων.

Συμπέρασμα

Η κάθετη κλιμάκωση (fewer, stronger executors) δεν λειτούργησε καλύτερα από την οριζόντια κλιμάκωση (περισσότεροι, ελαφρύτεροι executors). Στο Q6, το σύστημα επωφελήθηκε από τη μεγαλύτερη παραλληλία του 8x1x2, η οποία επέτρεψε ταχύτερη ανάγνωση δεδομένων, αποτελεσματικό filtering και groupBy, και καλύτερη κατανομή φόρτου.

Αυτό δείχνει ότι σε πολλές περιπτώσεις:

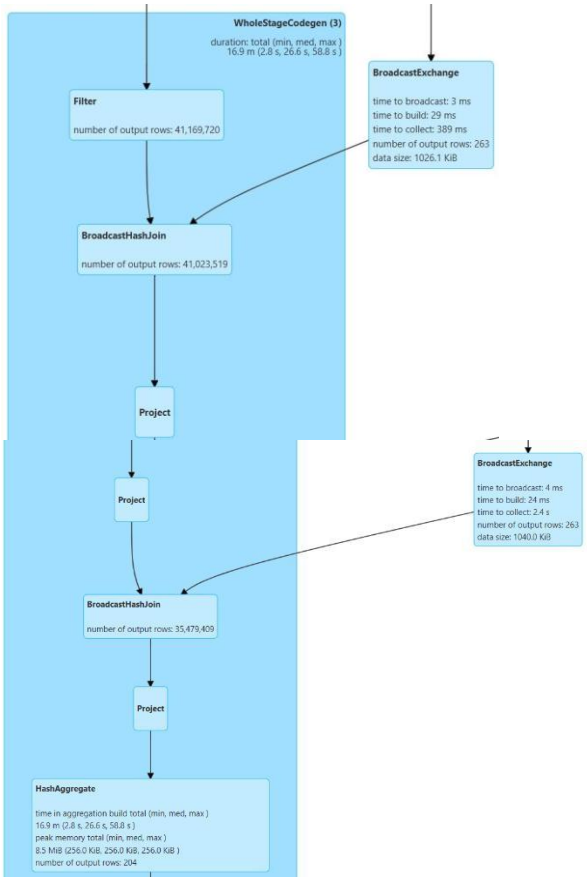
"Περισσότερα μικρά" > "Λίγα δυνατά", όταν το query είναι embarrassingly parallel όπως εδώ.



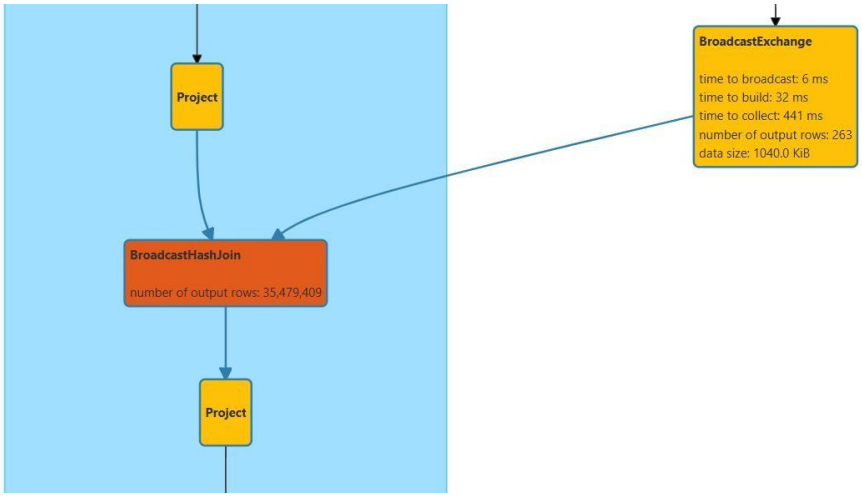
Q3–Q5: Εκτίμηση join στρατηγικών

Για τα παρακάτω Queries , ο Catalyst Optimizer επέλεξε τη στρατηγική BroadcastHashJoin για την εκτέλεση των joins (όπου χρειάστηκε). Και στις δύο περιπτώσεις, το build side ήταν το δεξί dataset (BuildRight), το οποίο είχε μικρό μέγεθος (περίπου 1MB), όπως φαίνεται στο Physical Plan (Statistics(sizeInBytes ≃ 1 MB)).
Η επιλογή του BroadcastHashJoin είναι θεωρητικά δικαιολογημένη, καθώς τα μικρού μεγέθους datasets μπορούν να μεταδοθούν αποδοτικά σε όλους τους executors, αποφεύγοντας το κόστος του shuffle που απαιτείται σε στρατηγικές όπως το SortMergeJoin. Συνεπώς, η στρατηγική που επιλέχθηκε είναι σωστή και βελτιστοποιεί τον χρόνο εκτέλεσης.

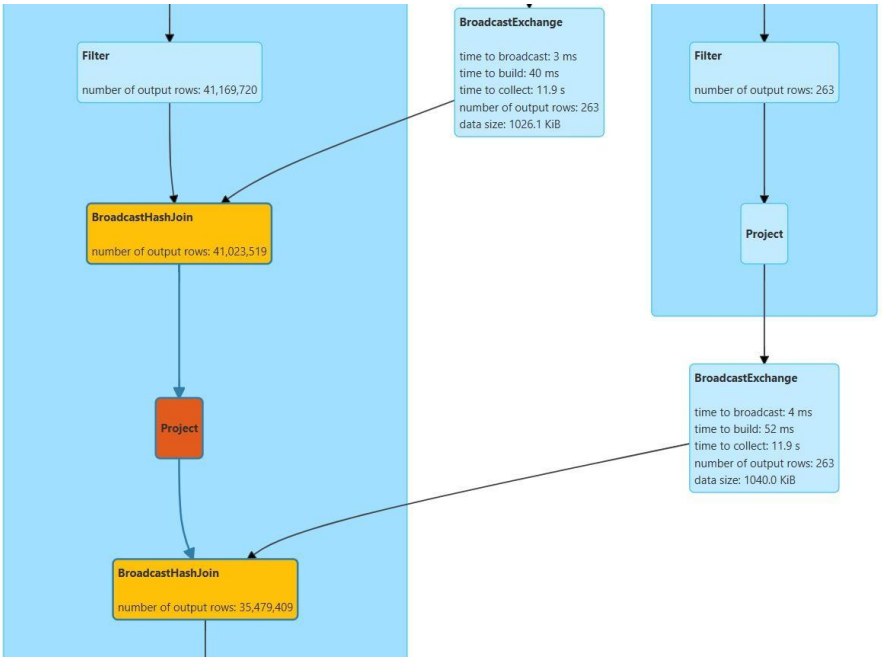
Q3_DF:



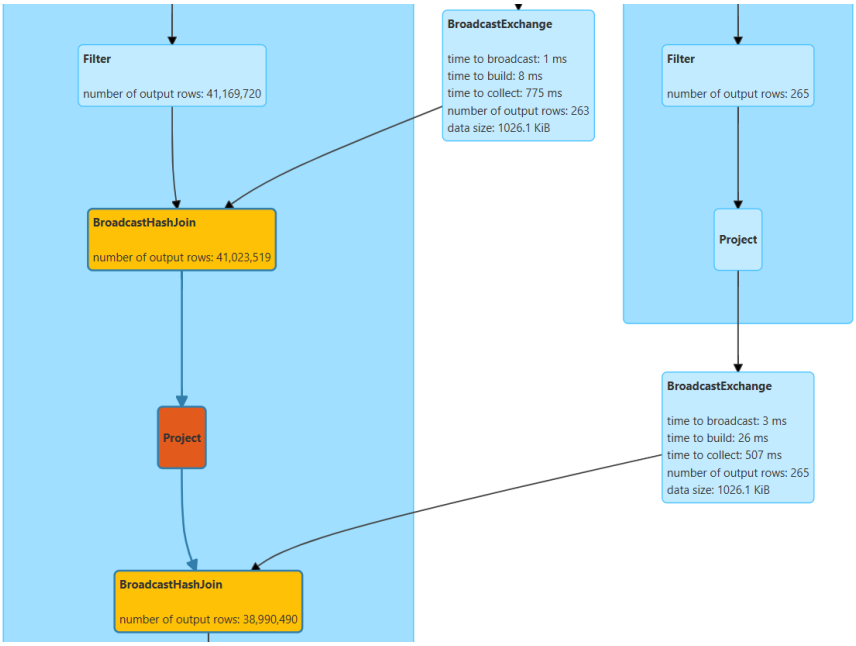
Q3_DF_SQL:



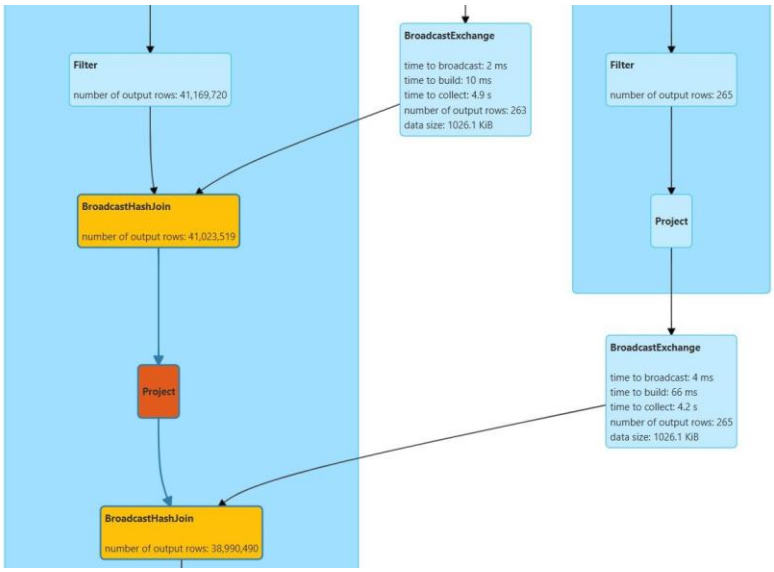
Q3_PARQ:



Q5_DF:


Filter
number of output rows: 41,169,720
BroadcastExchange
time to broadcast: 1 ms
time to build: 8 ms
time to collect: 775 ms
number of output rows: 263
data size: 1026.1 KiB
Filter
number of output rows: 265
BroadcastHashJoin
number of output rows: 41,023,519
Project
Project
BroadcastExchange
time to broadcast: 3 ms
time to build: 26 ms
time to collect: 507 ms
number of output rows: 265
data size: 1026.1 KiB
BroadcastHashJoin
number of output rows: 38,990,490

Q5_PARQ:


Filter
number of output rows: 41,169,720
BroadcastExchange
time to broadcast: 2 ms
time to build: 10 ms
time to collect: 4.9 s
number of output rows: 263
data size: 1026.1 KiB
Filter
number of output rows: 265
BroadcastHashJoin
number of output rows: 41,023,519
Project
Project
BroadcastExchange
time to broadcast: 4 ms
time to build: 66 ms
time to collect: 4.2 s
number of output rows: 265
data size: 1026.1 KiB
BroadcastHashJoin
number of output rows: 38,990,490