Big Data Project

Πλανάκης Νικηφόρος , 2121213
Τσιτλαούρι Γεώργιος-Δαυίδ, 2121219

Link προς αποθετήριο GitHub https://github.com/giwrgostst/big-data-spark-taxi

1. Εισαγωγή

Η παρούσα εργασία έχει ως κύριο στόχο την αξιολόγηση και τη βελτιστοποίηση της επεξεργασίας μεγάλων δεδομένων ταξί *(New York Yellow Taxi)* χρησιμοποιώντας το οικοσύστημα Apache Hadoop και Apache Spark. Συγκεκριμένα:

1. **Να συνδεθούμε σε απομακρυσμένη υποδομή Kubernetes** και να ρυθμίσουμε το Spark Job History Server.
2. **Να μετατρέψουμε τα πρωτότυπα αρχεία CSV σε μορφή Parquet στο HDFS**, με έμφαση στο dataset yellow_tripdata_2024.
3. **Να υλοποιήσουμε έξι ερωτήματα** *(Query 1–6)* με διάφορα APIs *(RDD, DataFrame, SQL),* **συγκρίνοντας την απόδοση** μεταξύ CSV/Parquet και μεταξύ διαφορετικών ρυθμίσεων πόρων *(horizontal vs. vertical scaling)*.
4. **Να μελετήσουμε την αυτόματη επιλογή στρατηγικής join** από τον βελτιστοποιητή Catalyst του Spark και να αξιολογήσουμε την επίδραση παραμετροποίησης στις εκτελέσεις.

Σύντομη παρουσίαση Hadoop (≥ 3.3) & Spark (≥ 3.5)
Το Apache Hadoop (έκδοση ≥ 3.3) παρέχει μια κατανεμημένη υποδομή αποθήκευσης (HDFS) και εκτέλεσης batch εργασιών (MapReduce).
Το Apache Spark (έκδοση ≥ 3.5) βασίζεται σε in-memory επεξεργασία, προσφέροντας τρεις βασικούς τρόπους χειρισμού δεδομένων:

- **RDD API:** Χαμηλού επιπέδου, για λεπτομερή έλεγχο μετασχηματισμών.
- **DataFrame API:** Λογική αναπαράσταση πινάκων με βελτιστοποιήσεις Catalyst.
- **Spark SQL:** Εκτέλεση SQL ερωτημάτων πάνω σε DataFrames/Views.
- Η χρήση **Parquet** ως στήλης-ορισμένης μορφής επιταχύνει την ανάγνωση–εγγραφή και επιτρέπει καλύτερη συμπίεση σε σχέση με το CSV.

2. Υλικά & Μεθόδοι

Υποδομή Kubernetes & Job History Server

Για τη διασφάλιση επαναληψιμότητας και κεντρικής καταγραφής, χρησιμοποιήθηκε η απομακρυσμένη Kubernetes υποδομή του μαθήματος και τοπικά στήθηκε ο Spark Job History Server μέσω Docker Compose.

- **Σύνδεση στο cluster**

Αρχικά φορτώθηκε το αρχείο ρύθμισης (kubeconfig) στον υπολογιστή εργασίας και επιβεβαιώθηκε η πρόσβαση στο cluster με επιτυχή εμφάνιση όλων των κόμβων σε κατάσταση “Ready”.

- **Επαλήθευση υπηρεσιών Spark**
  Στη συνέχεια ελέγχθηκε ότι τα pods που διαχειρίζονται τις Spark εργασίες είναι ενεργά στο προβλεπόμενο namespace, επιβεβαιώνοντας την ορθή λειτουργία των υπηρεσιών.

- **Ιστορικό εκτελέσεων**
  Τοπικά εκκινήθηκε ο Spark History Server μέσω Docker Compose, ρυθμισμένος ώστε να καταναλώνει απευθείας τα logs από τον shared φάκελο των Spark jobs στο HDFS. Το περιβάλλον παρακολούθησης επιβεβαιώθηκε με επιτυχή εμφάνιση της λίστας των εκτελεσμένων jobs στο web UI.

Datasets & HDFS

Τα δεδομένα προέρχονται από τρία αρχεία:

1. **yellow_tripdata_2024:** το πρωτότυπο CSV των ταξί του 2024, η δομή του οποίου απαιτούσε επιπλέον προεπεξεργασία για να αναγνωριστούν σωστά οι τύποι πεδίων.
2. **tripdata_2015_with_gps:** αρχείο του 2015 με συντεταγμένες GPS για κάθε διαδρομή.
3. **zone_lookup:** πίνακας αντιστοιχίας ζωνών κωδικών σε ονόματα περιοχών.

Μετά την προεπεξεργασία, όλα τα παραχθέντα DataFrames ανεβάστηκαν στο HDFS στο path hdfs://hdfs-namenode:9000/user/{username}/data/parquet/, εξασφαλίζοντας κοινή βάση για τις συγκριτικές εκτελέσεις.

Εργαλεία & APIs

Για την υλοποίηση των ερωτημάτων και τη μέτρηση επιδόσεων αξιοποιήθηκαν:

- Apache Spark (έκδοση ≥ 3.5)
  - RDD API για λεπτομερή streaming μετασχηματισμούς.
  - DataFrame API με βελτιστοποιήσεις Catalyst.
  - Spark SQL για εκτέλεση SQL ερωτημάτων.
- Apache Hadoop (έκδοση ≥ 3.3) ως υποκείμενο σύστημα αποθήκευσης (HDFS).

Το υπολογιστικό περιβάλλον στηρίχθηκε σε pods Kubernetes, με διάταξη πόρων που διαμορφώθηκε ανά εκτέλεση προκειμένου να μελετηθεί η horizontal & vertical κλιμάκωση.

3. Μετατροπή και αποθήκευση σε Parquet

Για να εξασφαλιστεί γρήγορη ανάγνωση και αποδοτική αποθήκευση στη συνέχεια των queries, τα τρία raw CSV datasets μετατράπηκαν σε στήλης–οριζόμενη μορφή Parquet και γράφτηκαν στο HDFS, στο path: *hdfs://hdfs-namenode:9000/user/{username}/data/parquet/*

Διαδικασία μετατροπής

Αναλύοντας τον κώδικα μετατροπής *(csv_to_parquet.py)*, ακολουθήθηκε η παρακάτω λογική:

- **yellow_tripdata_2015**
  Ορίστηκε αυστηρό schema με τονισμένα τα timestamps και τους αριθμητικούς τύπους. Τα δεδομένα repartitioned σε 200 partitions ώστε να εξασφαλιστεί ομοιόμορφος καταμερισμός φόρτου κατά την εγγραφή.

- **yellow_tripdata_2024**
  Εντοπίστηκαν και προστέθηκαν νέα πεδία *(π.χ. congestion_surcharge, Airport_fee, cbd_congestion_fee)* με τον κατάλληλο τύπο Double.
  Επαλήθευση σωστής ανάγνωσης όλων των πεδίων ως Timestamp ή Double, ώστε στο Parquet να διατηρούνται οι τύποι.
  Repartition σε 200 partitions για γρήγορη, παράλληλη εγγραφή.

- **taxi_zone_lookup**
  Σταθερό μικρό dataset, re-partitioned σε 1 partition, για απλοποίηση joins χωρίς overhead πολλών partitions.

Εκτέλεση & Χρονομετρήσεις

Τα logs από την εγγραφή του yellow_tripdata_2024 δείχνουν λεπτομερώς τη συμπεριφορά του cluster:

Write to Parquet (Stage 5)

- Ολοκληρώθηκαν 200 tasks κατανεμημένα σε 6 executors.
- Διάρκεια ανά task: από ~3 s έως ~6.3 s (π.χ. task 188 σε 6.255 s, task 196 σε 5.059 s).
- Συνολικός χρόνος Stage 5: 210.873 s
- Commit Parquet files: επιπλέον 1.886 s
- Επεξεργασία παραμέτρων metadata από ParquetUtils: σχεδόν άμεση (~2 ms).

Validation read

- **Stage 6:** Μία partition scan ολοκληρώθηκε σε 0.610 s, με broadcast των blocks στα executors (*~35.2 KiB & ~7.5 KiB)*.
- **Stage 8:** Final result σε 0.206 s, επιβεβαιώνοντας ότι τα αρχεία είναι προσβάσιμα και πλήρη.

Καθαρισμός

- Ο SparkContext σταμάτησε επιτυχώς, απελευθερώνοντας executors και πόρους.

Παρατηρήσεις

- Η επιλογή 200 partitions για τα μεγάλα datasets εξασφάλισε ισορροπία ανάμεσα σε παράλληλη εκτέλεση και overhead scheduling.
- Η προσεκτική οριστικοποίηση του σχήματος για το 2024 dataset απέφυγε μετατροπές στην εγγραφή, μειώνοντας πιθανά σφάλματα τύπων.

- Ο συνδυασμός Parquet με in-memory broadcast κατά το validation προσφέρει ταχύτητα ανάγνωσης (validation stages < 1 s).

4. Query 1

Περιγραφή ερωτήματος

Υπολογίζεται ο μέσος όρος γεωγραφικών συντεταγμένων *(latitude, longitude)* ανά ώρα της ημέρας, βάσει του dataset των ταξί. Ως στήλες εισόδου χρησιμοποιούνται το timestamp της παραλαβής και οι δυο συντεταγμένες· το αποτέλεσμα είναι ένας πίνακας με 24 γραμμές *(0–23)* και δύο αριθμητικές στήλες *(avg_latitude, avg_longitude)*.

Υλοποίηση με RDD

Η λογική ήταν:

- Map κάθε γραμμής σε (hourOfDay, (lat, lon, 1)).
- ReduceByKey για άθροισμα συντεταγμένων και πλήθος.
- MapValues για υπολογισμό μέσων όρων.
- sortByKey & collect για τελική ταξινόμηση.

Μετρικές εκτέλεσης (από logs)

- ShuffleMapStage (sortByKey): 0,335 s
- ResultStage (collect): 0,347 s
- Συνολικός χρόνος: ≈ 0,68 s

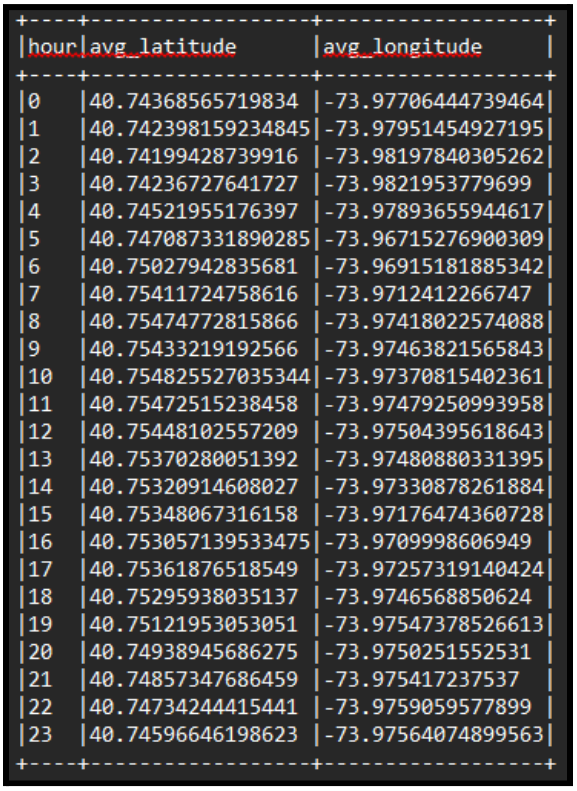
Ακολουθεί ο ενδεικτικός πίνακας από το output της εκτέλεσης:

HourOfDay	Latitude	Longitude
00	-73.977064	40.743686
01	-73.979515	40.742398
02	-73.981978	40.741994
03	-73.982195	40.742367
04	-73.978937	40.745220
05	-73.967153	40.747087
06	-73.969152	40.750279
07	-73.971241	40.754117
08	-73.974180	40.754748
09	-73.974638	40.754332
10	-73.973708	40.754826
11	-73.974793	40.754725
12	-73.975044	40.754481
13	-73.974809	40.753703
14	-73.973309	40.753209
15	-73.971765	40.753481
16	-73.971000	40.753057
17	-73.972573	40.753619
18	-73.974657	40.752959
19	-73.975474	40.751220
20	-73.975025	40.749389
21	-73.975417	40.748573
22	-73.975906	40.747342
23	-73.975641	40.745966

Υλοποίηση με DataFrame API *(χωρίς UDF)*

1. Δημιουργία στήλης hour από το timestamp.
2. groupBy("hour").agg(avg("latitude"), avg("longitude")).
3. orderBy("hour") + show().
4. Μετρικές εκτέλεσης (από logs)
    - ShuffleMapStage 2 (main aggregation & shuffle): 89,766 s
    - ResultStage 4 (showString): 0,336 s
    - Συνολικός χρόνος: ≈ 90,10 s

Ακολουθεί ο ενδεικτικός πίνακας από το output της εκτέλεσης:

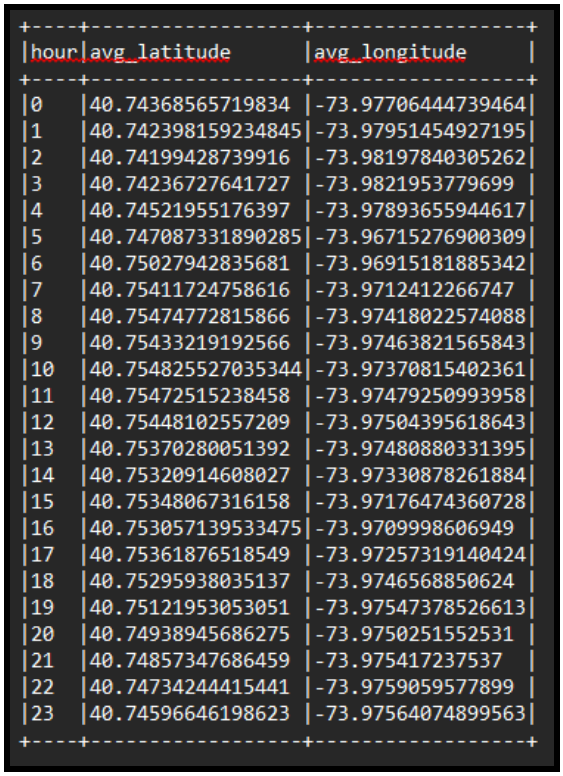


hour	avg_latitude	avg_longitude
0	40.74368565719834	-73.97706444739464
1	40.742398159234845	-73.97951454927195
2	40.74199428739916	-73.98197840305262
3	40.74236727641727	-73.9821953779699
4	40.74521955176397	-73.97893655944617
5	40.747087331890285	-73.96715276900309
6	40.75027942835681	-73.96915181885342
7	40.75411724758616	-73.9712412266747
8	40.75474772815866	-73.97418022574088
9	40.75433219192566	-73.97463821565843
10	40.754825527035344	-73.97370815402361
11	40.75472515238458	-73.97479250993958
12	40.75448102557209	-73.97504395618643
13	40.75370280051392	-73.97480880331395
14	40.75320914608027	-73.97330878261884
15	40.75348067316158	-73.97176474360728
16	40.753057139533475	-73.9709998606949
17	40.75361876518549	-73.97257319140424
18	40.75295938035137	-73.9746568850624
19	40.75121953053051	-73.97547378526613
20	40.74938945686275	-73.9750251552531
21	40.74857347686459	-73.975417237537
22	40.74734244415441	-73.9759059577899
23	40.74596646198623	-73.97564074899563

Υλοποίηση DataFrame με UDF

1. Προστέθηκε UDF για μετατροπή του πεδίου hour αντί του built-in casting.
2. Διαφοροποιήθηκε ελαφρώς το pipeline, με πρόσθετο στάδιο UDF evaluation πριν το groupBy.
3. Μετρικές εκτέλεσης (από logs)
    - ShuffleMapStage 2: 63,731 s
    - ResultStage 4: 0,368 s
    - Συνολικός χρόνος: ≈ 64,10 s

Ακολουθεί ο ενδεικτικός πίνακας από το output της εκτέλεσης:



hour	avg_latitude	avg_longitude
0	40.74368565719834	-73.97706444739464
1	40.742398159234845	-73.97951454927195
2	40.74199428739916	-73.98197840305262
3	40.74236727641727	-73.9821953779699
4	40.74521955176397	-73.97893655944617
5	40.747087331890285	-73.96715276900309
6	40.75027942835681	-73.96915181885342
7	40.75411724758616	-73.9712412266747
8	40.75474772815866	-73.97418022574088
9	40.75433219192566	-73.97463821565843
10	40.754825527035344	-73.97370815402361
11	40.75472515238458	-73.97479250993958
12	40.75448102557209	-73.97504395618643
13	40.75370280051392	-73.97480880331395
14	40.75320914608027	-73.97330878261884
15	40.75348067316158	-73.97176474360728
16	40.753057139533475	-73.9709998606949
17	40.75361876518549	-73.97257319140424
18	40.75295938035137	-73.9746568850624
19	40.75121953053051	-73.97547378526613
20	40.74938945686275	-73.9750251552531
21	40.74857347686459	-73.975417237537
22	40.74734244415441	-73.9759059577899
23	40.74596646198623	-73.97564074899563

Σύγκριση επιδόσεων & σχολιασμός

Υλοποίηση	ShuffleStage	ShowStage	Συνολικό Time
RDD API	0.335 s	0.347 s	0.68 s
Dataframe API	89.766 s	0.336 s	90.1 s
Dataframe + UDF	63.731 s	0.368 s	64.1 s

- Η RDD υλοποίηση εκτελείται σχεδόν ακαριαία (~0,7 s), καθώς αποφεύγει μεγάλο μέρος του overhead της γεννήτριας κώδικα και των shuffle operations του Catalyst.
- Η καθαρή DataFrame εκδοχή είναι η πιο «βαριά» (~90 s), λόγω default αριθμού partitions και πλήρους shuffle σε όλο το dataset.
- Η χρήση UDF επιταχύνει σχετικά (~64 s), πιθανώς διότι μειώνει τον αριθμό internal operators και θυσιάζει μέρος του optimization pipeline, αλλά εξακολουθεί να υποφέρει από μεγάλο shuffle κόστος.

Συμπέρασμα

Για αυτό το ερώτημα, το RDD API υπερέχει κατά κράτος σε χρόνο, ενώ το DataFrame API ακόμα και με UDF παρουσιάζει σημαντικό overhead shuffle. Σε μελλοντικές επεκτάσεις, μία βελτιστοποιημένη DataFrame υλοποίηση *(π.χ. coalesce partitions ή broadcast small tables)* θα μειώσει τους χρόνους εκτέλεσης.

5. Query 2

Περιγραφή ερωτήματος

Για κάθε VendorID βρίσκουμε την μέγιστη απόσταση που διένυσε ένα ταξί *(με χρήση της συνάρτησης Haversine)* και υπολογίζουμε τη μέση διάρκεια της διαδρομής. Ως στήλες εισόδου χρησιμοποιούνται το VendorID, η απόσταση σε χιλιόμετρα και η διάρκεια σε λεπτά· το αποτέλεσμα είναι ένας πίνακας με μία γραμμή ανά VendorID και δύο αριθμητικές στήλες *(max_distance, avg_duration)*.

Υλοποίηση με RDD

Λογική υλοποίησης:

- Map σε (VendorID, (distance, duration, 1)) ➔ reduceByKey για άθροισμα αποστάσεων, χρόνων και πλήθους ➔ υπολογισμός μέγιστης απόστασης και μέσης διάρκειας ➔ collect.

Μετρικές εκτέλεσης:

- ShuffleMapStage (reduceByKey): 320,317 s
- ResultStage (collect): 0,750 s
- Συνολικός χρόνος: 321,127 s

Ακολουθεί ο ενδεικτικός πίνακας από το output της εκτέλεσης:

```
VendorID    Max Haversine Distance (km)    Duration (min)
1         57.24                  95.1
2         53.90                  62.0
```

Υλοποίηση με DataFrame API

Λογική υλοποίησης:

- Φόρτωση DataFrame ➔ χρήση built-in max(distance) & avg(duration) με groupBy("VendorID") ➔ show().

Μετρικές εκτέλεσης:

- ShuffleMapStage (aggregation & shuffle): 70,406 s
- ResultStage (showString): 0,882 s
- Συνολικός χρόνος: 71,288 s

Ακολουθεί ο ενδεικτικός πίνακας από το output της εκτέλεσης:

```
+--------+--------------------------+-----------------+
|VendorID|Max Haversine Distance (km)|   Duration (min)|
+--------+--------------------------+-----------------+
|       1|         57.237266477643715|95.13333333333334|
|       2|          53.89902622743138|            61.95|
+--------+--------------------------+-----------------+
```

Υλοποίηση με Spark SQL

Λογική υλοποίησης:

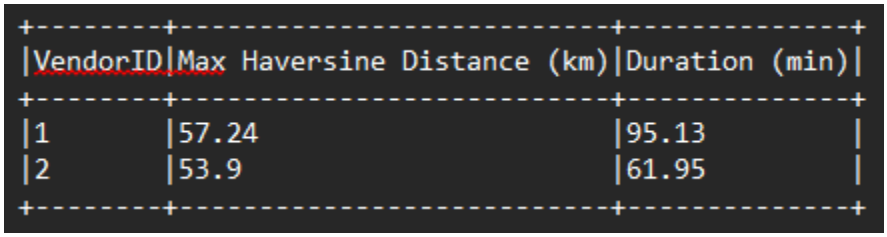
- Δημιουργία view ➔ εκτέλεση SQL:
  SELECT VendorID,

MAX(haversine_distance) AS max_distance,
AVG(duration_minutes) AS avg_duration
FROM trips_view
GROUP BY VendorID
➔ show() και EXPLAIN.

Μετρικές εκτέλεσης:

- ShuffleMapStage (aggregation & shuffle): 113,280 s
- ResultStage (showString): 0,686 s
- Συνολικός χρόνος: 113,966 s

Ακολουθεί ο ενδεικτικός πίνακας από το output της εκτέλεσης:



```
+--------+--------------------------+--------------+
|VendorID|Max Haversine Distance (km)|Duration (min)|
+--------+--------------------------+--------------+
|1       |57.24                     |95.13         |
|2       |53.9                      |61.95         |
+--------+--------------------------+--------------+
```

Σύγκριση επιδόσεων & σχολιασμός

Υλοποίηση	ShuffleStage	ShowStage	Συνολικό Time
RDD API	320,317 s	0.75 s	321.127 s
Dataframe	70.406 s	0.882 s	71.288 s
Spark SQL	113.28 s	0.686 s	113.966 s

- Η DataFrame API υπερέχει καθαρά, με συνολικό χρόνο ~71 s, αξιοποιώντας το codegen και τον βελτιστοποιητή Catalyst για αποτελεσματικό shuffle και aggregation.
- Το Spark SQL είναι πιο βαρύ (~114 s), καθώς προστίθεται parsing/analysis του query και ενδεχομένως λιγότερο αποτελεσματικό physical planning σε σχέση με το άμεσο DataFrame API.
- Η RDD υλοποίηση αποδεικνύεται πολύ αργή (~321 s) λόγω του Python overhead στο reduceByKey και του πλήρους serialization κάθε στοιχείου.

Συμπέρασμα

Για σύνθετες συναθροίσεις, η DataFrame/SQL προσέγγιση μειώνει δραστικά το κόστος σε σχέση με το RDD API, με προτίμηση στο άμεσο DataFrame API για καλύτερο trade-off ευκολίας κώδικα και επιδόσεων.

6. Query 3

Περιγραφή ερωτήματος

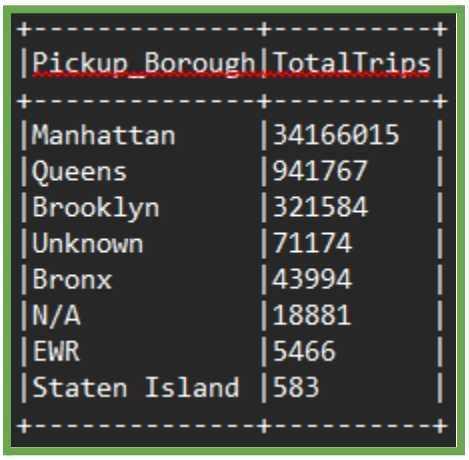
Μετράμε το συνολικό πλήθος ταξιδιών ανά Pickup_Borough, συγκρίνοντας την εκτέλεση με CSV και με Parquet input. Ως στήλη εισόδου χρησιμοποιείται το Pickup_Borough· το

αποτέλεσμα είναι ένας πίνακας με τόσες γραμμές όσες οι διαφορετικές περιοχές και μία αριθμητική στήλη *(TotalTrips)*, ταξινομημένος αλφαβητικά κατά Borough.

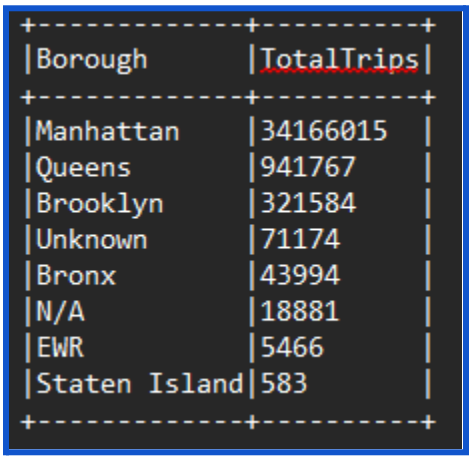
Υλοποίηση με DataFrame API

- Φόρτωση των raw **CSV**/**Parquet** με spark.read
- Μετασχηματισμοί: εξαγωγή Pickup_Borough, TotalTrips = count(*) με groupBy + agg\
- orderBy(Pickup_Borough) + show()

Ακολουθούν οι ενδεικτικοί πίνακες από το output της εκτέλεση με **CSV** και **Parquet**:



Pickup_Borough	TotalTrips
Manhattan	34166015
Queens	941767
Brooklyn	321584
Unknown	71174
Bronx	43994
N/A	18881
EWR	5466
Staten Island	583



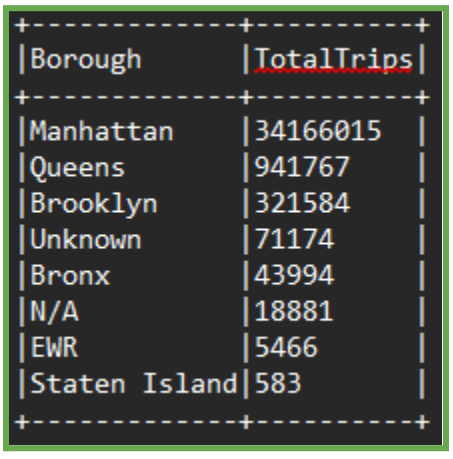
Borough	TotalTrips
Manhattan	34166015
Queens	941767
Brooklyn	321584
Unknown	71174
Bronx	43994
N/A	18881
EWR	5466
Staten Island	583

Υλοποίηση με Spark SQL

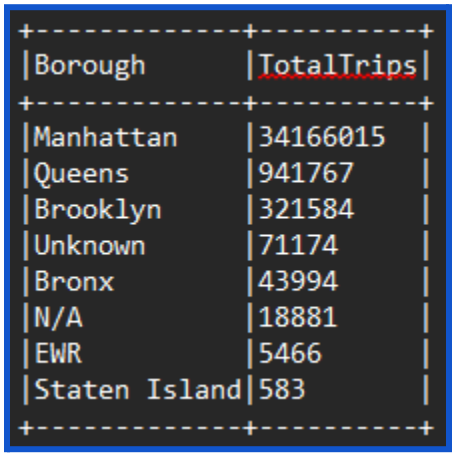
- Δημιουργία view (createOrReplaceTempView("trips"))
- SQL ερώτημα:

```
SELECT Pickup_Borough, COUNT(*) AS TotalTrips
FROM trips
GROUP BY Pickup_Borough
ORDER BY Pickup_Borough
Εντολή EXPLAIN για εμφάνιση του physical plan
```

Ακολουθούν οι ενδεικτικοί πίνακες από το output της εκτέλεση με **CSV** και **Parquet**:



Borough	TotalTrips
Manhattan	34166015
Queens	941767
Brooklyn	321584
Unknown	71174
Bronx	43994
N/A	18881
EWR	5466
Staten Island	583



Borough	TotalTrips
Manhattan	34166015
Queens	941767
Brooklyn	321584
Unknown	71174
Bronx	43994
N/A	18881
EWR	5466
Staten Island	583

Εκτέλεση με CSV vs Parquet input

Υλοποίηση	ShuffleStage	ShowStage	Συνολικό Time
Dataframe + CSV	201.303 s	0.793 s	202.096 s
Dataframe + Parquet	10.273 s	0.359 s	10.632 s
SQL + CSV	156.786 s	1.632 s	158.418 s
SQL + Parquet	11.843 s	0.269 s	12.112 s

- Το Parquet προσφέρει καθολικά ταχύτερη ανάγνωση (~20×) χάρη στο columnar format, vectorized reader και push‑down φίλτρων, μειώνοντας δραστικά το I/O.
- Στις δύο μορφές API, το DataFrame είναι ελαφρώς ταχύτερο από το SQL, επειδή παρακάμπτει κάποια parsing‑και‑planning overhead του SQL parser.
- Το CSV υποφέρει από full schema inference και parsing overhead σε κάθε stage, ενώ το Parquet επαναχρησιμοποιεί το αποθηκευμένο schema.

Συμπέρασμα
Επιλέγουμε Parquet + DataFrame API για βέλτιστο trade-off επιδόσεων και απλότητας.

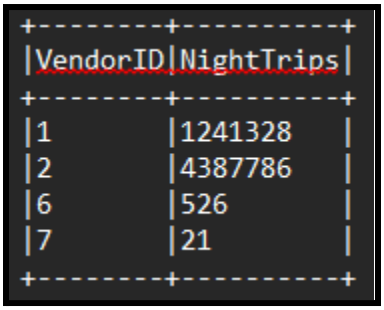
7. Query 4

Περιγραφή ερωτήματος
Υπολογίζεται ο αριθμός των νυχτερινών διαδρομών *(π.χ. μεταξύ 20:00 και 06:00)* ανά VendorID. Ως στήλες εισόδου χρησιμοποιούνται το VendorID και το pickup_datetime — φιλτράροντας τα ταξίδια μέσα στο νυχτερινό ωράριο· το αποτέλεσμα είναι ένας πίνακας με μία γραμμή για κάθε VendorID και μία αριθμητική στήλη *(NightTrips)*.

Υλοποίηση SQL με CSV

- ShuffleMapStage (aggregation & shuffle): 193.963 s
- ResultStage (showString): 0.370 s
- Συνολικός χρόνος: ~ 194.333 s

Ακολουθεί ο ενδεικτικός πίνακας από το output της εκτέλεση:

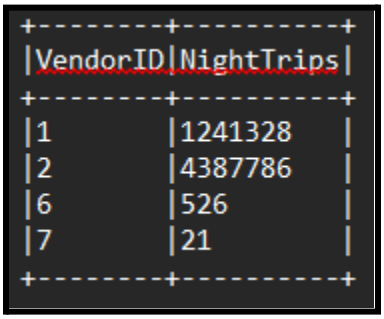


```
+--------+----------+
|VendorID|NightTrips|
+--------+----------+
|1       |1241328   |
|2       |4387786   |
|6       |526       |
|7       |21        |
+--------+----------+
```

Υλοποίηση SQL με Parquet

- ShuffleMapStage (aggregation & shuffle): 25.373 s
- ResultStage (showString): 0.658 s
- Συνολικός χρόνος: ~ 26.031 s

Ακολουθεί ο ενδεικτικός πίνακας από το output της εκτέλεση:



```
+--------+----------+
|VendorID|NightTrips|
+--------+----------+
|1       |1241328   |
|2       |4387786   |
|6       |526       |
|7       |21        |
+--------+----------+
```

Σύγκριση Επιδόσεων & Σχολιασμός

Υλοποίηση	ShuffleStage	ShowStage	Συνολικό Time
CSV	193.962 s	0.37 s	194.333 s
Parquet	25.373 s	0.658 s	26.031 s

- Μείωση partitions επειδή από τα 34 partitions στο CSV κατεβάσαμε σε 16 στο Parquet χάρη στην ικανότητα του Parquet να παραλείπει partitions χωρίς νυχτερινά ταξίδια (predicate pushdown).
- Διαβάζουμε μόνο το VendorID και το πεδίο ώρας, αντί για ολόκληρη την εγγραφή, κάνουμε το λεγόμενο Column pruning.
- Το Parquet χρησιμοποιεί ενσωματωμένα στατιστικά για να φορτώσει bulk σειρές, αποφεύγοντας costly κλήσεις parsing κειμένου, δηλαδη Vectorized I/O.

Συμπέρασμα

Η ανάγνωση από Parquet μείωσε κατά ~168 s τον χρόνο shuffle/aggregation, καθώς εκτελέστηκαν λιγότερα tasks και αξιοποιήθηκε πιο αποδοτικός σχεδιασμός από τον Catalyst optimizer.

Το overhead για το τελικό show *(συλλογή αποτελεσμάτων)* παραμένει αμελητέο και στα δύο *(~0,3–0,7 s)*.

Συστήνεται να διατηρούμε τα μεγάλου όγκου datasets σε Parquet *(με σωστό schema),* ειδικά όταν γίνονται επιλεκτικά ερωτήματα: το όφελος σε I/O και CPU είναι δραματικό, όπως φαίνεται εδώ με >5× ταχύτερη εκτέλεση.

8. Query 5

Περιγραφή ερωτήματος

Για κάθε συνδυασμό ζώνης παραλαβής *(PickupZone)* και ζώνης απόθεσης *(DropoffZone)* υπολογίζεται το συνολικό πλήθος ταξιδιών και επιλέγονται τα 20 πρώτα ζευγάρια κατά φθίνουσα

σειρά. Ως στήλες εισόδου χρησιμοποιούνται οι δύο ζώνες· το αποτέλεσμα είναι ένας πίνακας 20 γραμμών με τρεις στήλες *(PickupZone, DropoffZone, TotalTrips),* ταξινομημένος κατά φθίνουσα συχνότητα.

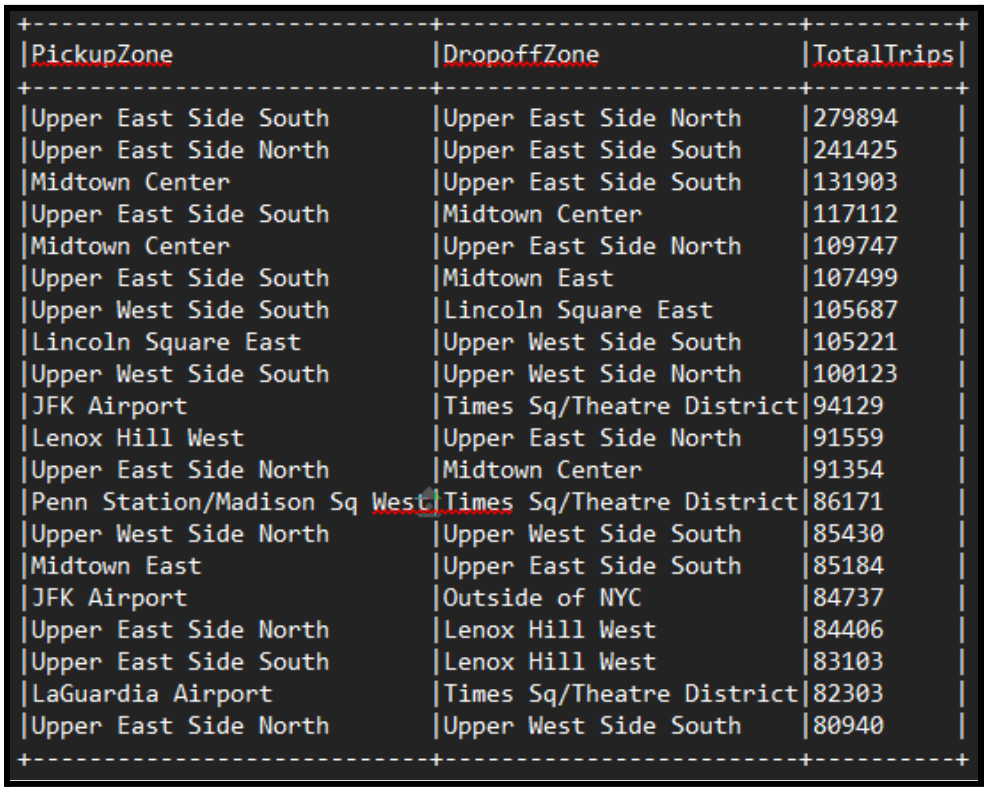
Υλοποίηση DataFrame API

- Φόρτωση CSV/Parquet σε DataFrame
- Ομαδοποίηση (groupBy("PickupZone", "DropoffZone"))
- Aggregation με count()
- Ταξινόμηση κατά φθίνουσα συχνότητα (orderBy(desc("count")))
- Περιορισμός στα πρώτα 20 (limit(20)) και show()

Εκτέλεση με CSV input

- ShuffleMapStage (stage 5): 190.040 s
- ResultStage (stage 7: showString): 1.551 s
- Συνολικός χρόνος: 191.591 s

Ακολουθεί ο ενδεικτικός πίνακας από το output της εκτέλεσης με CSV:

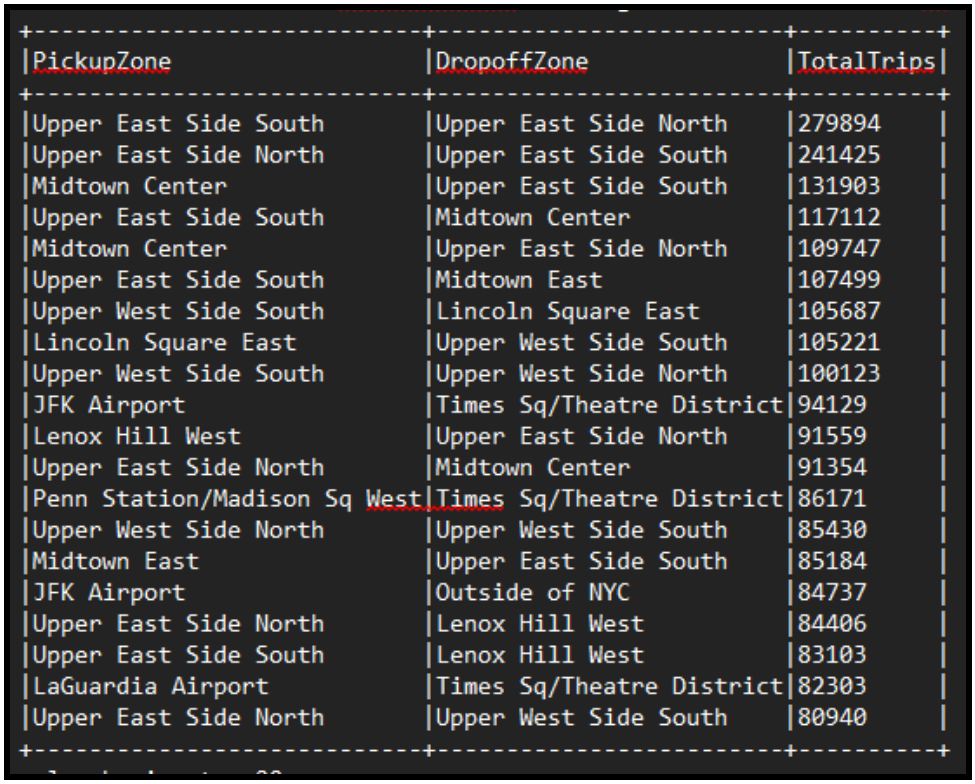


PickupZone	DropoffZone	TotalTrips
Upper East Side South	Upper East Side North	279894
Upper East Side North	Upper East Side South	241425
Midtown Center	Upper East Side South	131903
Upper East Side South	Midtown Center	117112
Midtown Center	Upper East Side North	109747
Upper East Side South	Midtown East	107499
Upper West Side South	Lincoln Square East	105687
Lincoln Square East	Upper West Side South	105221
Upper West Side South	Upper West Side North	100123
JFK Airport	Times Sq/Theatre District	94129
Lenox Hill West	Upper East Side North	91559
Upper East Side North	Midtown Center	91354
Penn Station/Madison Sq West	Times Sq/Theatre District	86171
Upper West Side North	Upper West Side South	85430
Midtown East	Upper East Side South	85184
JFK Airport	Outside of NYC	84737
Upper East Side North	Lenox Hill West	84406
Upper East Side South	Lenox Hill West	83103
LaGuardia Airport	Times Sq/Theatre District	82303
Upper East Side North	Upper West Side South	80940

Εκτέλεση με Parquet input

- ShuffleMapStage (stage 4): 16.784 s
- ResultStage (stage 6: showString): 0.852 s
- Συνολικός χρόνος: 17.636 s

Ακολουθεί ο ενδεικτικός πίνακας από το output της εκτέλεσης με Parquet:



PickupZone	DropoffZone	TotalTrips
Upper East Side South	Upper East Side North	279894
Upper East Side North	Upper East Side South	241425
Midtown Center	Upper East Side South	131903
Upper East Side South	Midtown Center	117112
Midtown Center	Upper East Side North	109747
Upper East Side South	Midtown East	107499
Upper West Side South	Lincoln Square East	105687
Lincoln Square East	Upper West Side South	105221
Upper West Side South	Upper West Side North	100123
JFK Airport	Times Sq/Theatre District	94129
Lenox Hill West	Upper East Side North	91559
Upper East Side North	Midtown Center	91354
Penn Station/Madison Sq West	Times Sq/Theatre District	86171
Upper West Side North	Upper West Side South	85430
Midtown East	Upper East Side South	85184
JFK Airport	Outside of NYC	84737
Upper East Side North	Lenox Hill West	84406
Upper East Side South	Lenox Hill West	83103
LaGuardia Airport	Times Sq/Theatre District	82303
Upper East Side North	Upper West Side South	80940

Σύγκριση επιδόσεων & σχολιασμός

Υλοποίηση	ShuffleStage	ShowStage	Συνολικό Time
CSV	190.04 s	1.151 s	191.591 s
Parquet	16.784 s	0.852 s	17.636 s

- Η Parquet υλοποίηση είναι ~11× ταχύτερη στο shuffle stage (16.8 s vs 190 s) και συνολικά ~10× ταχύτερη (17.6 s vs 191.6 s).
- Ο columnar, binary χαρακτήρας του Parquet μειώνει δραστικά το κόστος I/O, parsing και serialization.
- Το CSV αναγκάζει το Spark να διαβάσει, να κάνει full text parsing και να μετατρέψει σε τύπους σε κάθε εκτέλεση, επιβαρύνοντας το shuffle.

Συμπέρασμα

Για βαριές συναθροίσεις σε μεγάλα datasets, η χρήση Parquet αντί CSV μεγιστοποιεί την απόδοση εκτέλεσης, εκμεταλλευόμενη αποδοτικό I/O, predicate pushdown και vectorized reads.

9. Query 6

Περιγραφή ερωτήματος

Υπολογίζονται, ανά Borough, τα συνολικά έσοδα από ταξί *(άθροισμα κόστους διαδρομής, φιλοδωρημάτων, διοδίων, επιβαρύνσεων, φόρων και τελών αεροδρομίου)* με χρήση DataFrame API, και μελετάται πώς ο χρόνος εκτέλεσης μεταβάλλεται σε διαφορετικές ρυθμίσεις εκτελεστών *(horizontal vs vertical scaling)*. Ως στήλες εισόδου χρησιμοποιούνται τα δεδομένα των διαδρομών *(σε Parquet)* και ο broadcast πίνακας ζωνών *(CSV)* το αποτέλεσμα είναι ένας πίνακας με μία γραμμή ανά Borough και στήλες με τα επιμέρους έσοδα και το συνολικό TotalRevenue.

Σημείωση

Στον κώδικα του Q6 χρησιμοποιήθηκε Parquet για το αρχείο yellow_tripdata_2024 *(βελτιστοποιημένη μορφή)* και CSV για το taxi_zone_lookup.csv *(μικρό, σταθερό, broadcast από τον Catalyst)*.

Υλοποίηση DataFrame API

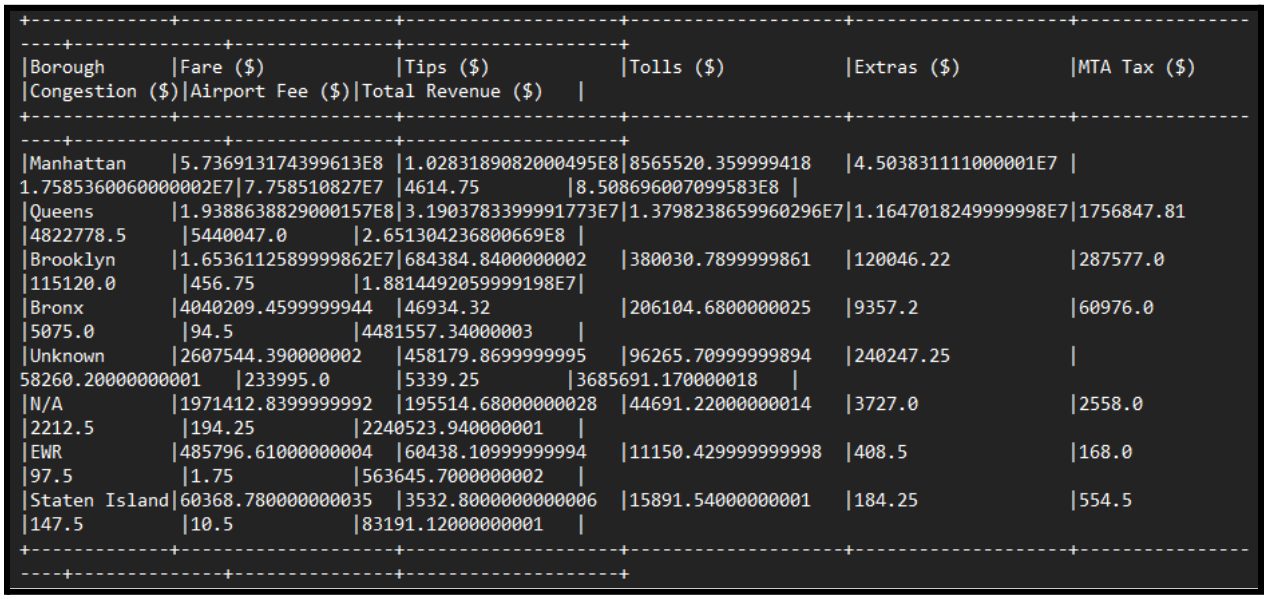
- Φόρτωση yellow_tripdata_2024.parquet σε DataFrame
- Φόρτωση taxi_zone_lookup.csv σε DataFrame & broadcast
- join σε ζώνες
- groupBy("Borough") + sum(«Fare», «Tips», «Tolls», …, συνολικό έσοδο)
- orderBy(desc("TotalRevenue")) + limit(8) + show()

(όπου το "TotalRevenue" = sum(Fare+Tips+Tolls+Extras+MTA Tax+Congestion+Airport Fee))

Ενδεικτικός πίνακας εκτέλεσης με 2 executors × 4 cores / 8 GB:

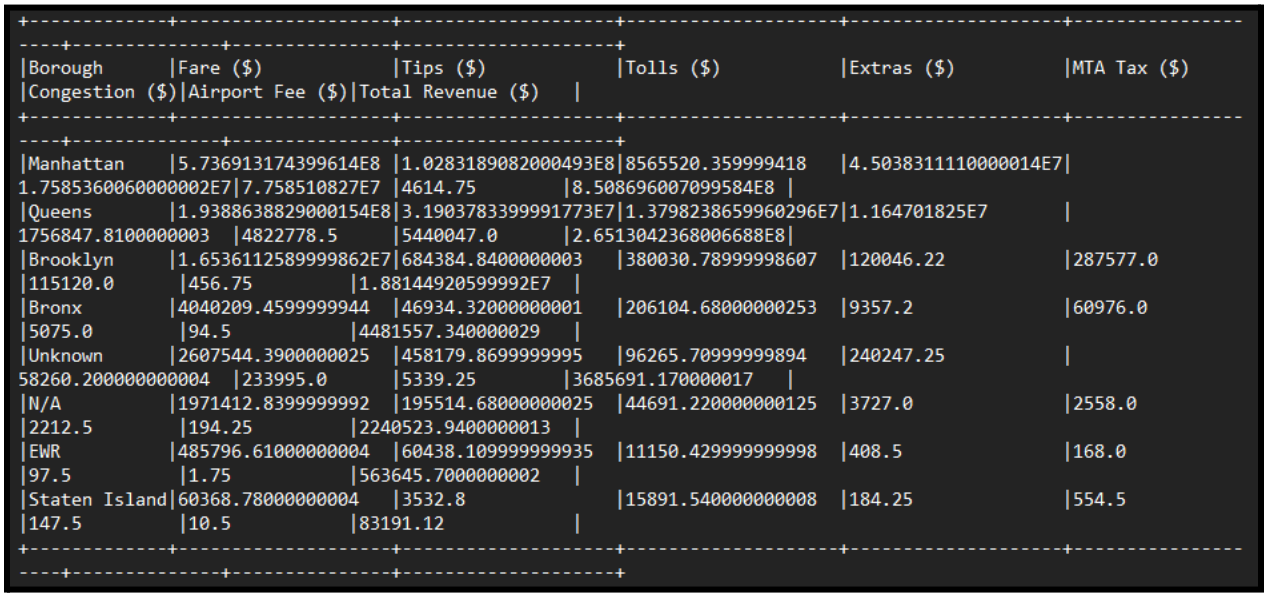
```
+-------------+--------------------+--------------------+--------------------+--------------------+----------------
----+--------------+---------------+--------------------+
|Borough      |Fare ($)            |Tips ($)            |Tolls ($)           |Extras ($)          |MTA Tax ($)
|Congestion ($)|Airport Fee ($)|Total Revenue ($)   |
+-------------+--------------------+--------------------+--------------------+--------------------+----------------
----+--------------+---------------+--------------------+
|Manhattan    |5.736913174399613E8 |1.0283189082000495E8|8565520.359999418   |4.503831111000001E7 |
1.7585360060000002E7|7.758510827E7 |4614.75         |8.508696007099583E8 |
|Queens       |1.9388638829000157E8|3.1903783399991773E7|1.3798238659960296E7|1.1647018249999998E7|1756847.81
|4822778.5     |5440047.0      |2.651304236800669E8 |
|Brooklyn     |1.6536112589999862E7|684384.8400000002   |380030.7899999861   |120046.22           |287577.0
|115120.0      |456.75         |1.8814492059999198E7|
|Bronx        |4040209.4599999944  |46934.32            |206104.6800000025   |9357.2              |60976.0
|5075.0        |94.5           |4481557.34000003    |
|Unknown      |2607544.390000002   |458179.8699999995   |96265.70999999894   |240247.25           |
58260.20000000001   |233995.0       |5339.25         |3685691.170000018   |
|N/A          |1971412.8399999992  |195514.68000000028  |44691.22000000014   |3727.0              |2558.0
|2212.5        |194.25         |2240523.940000001   |
|EWR          |485796.61000000004  |60438.10999999994   |11150.429999999998  |408.5               |168.0
|97.5          |1.75           |563645.7000000002   |
|Staten Island|60368.780000000035  |3532.8000000000006  |15891.54000000001   |184.25              |554.5
|147.5         |10.5           |83191.12000000001   |
+-------------+--------------------+--------------------+--------------------+--------------------+----------------
----+--------------+---------------+--------------------+
```

Ενδεικτικός πίνακας εκτέλεσης με 4 executors × 2 cores / 4 GB:



```
+-------------+-------------------+-------------------+-------------------+-------------------+----------------
----+-------------+---------------+-------------------+
|Borough      |Fare ($)           |Tips ($)           |Tolls ($)          |Extras ($)         |MTA Tax ($)
|Congestion ($)|Airport Fee ($)|Total Revenue ($)   |
+-------------+-------------------+-------------------+-------------------+-------------------+----------------
----+-------------+---------------+-------------------+
|Manhattan    |5.73691317439961 3E8 |1.0283189082000495E8|8565520.359999418   |4.503831111000001E7 |
1.7585360060000002E7|7.758510827E7 |4614.75        |8.508696007099583E8 |
|Queens       |1.9388638829000157E8|3.1903783399991773E7|1.3798238659960296E7|1.1647018249999998E7|1756847.81
|4822778.5    |5440047.0      |2.65130423680066 9E8 |
|Brooklyn     |1.6536112589999862E7|684384.8400000002   |380030.7899999861   |120046.22          |287577.0
|115120.0     |456.75         |1.8814492059999198E7|
|Bronx        |4040209.4599999944  |46934.32           |206104.6800000025   |9357.2             |60976.0
|5075.0       |94.5           |4481557.34000003    |
|Unknown      |2607544.390000002   |458179.8699999995   |96265.70999999894   |240247.25          |
58260.20000000001   |233995.0       |5339.25        |3685691.170000018   |
|N/A          |1971412.8399999992  |195514.68000000028  |44691.22000000014   |3727.0             |2558.0
|2212.5       |194.25         |2240523.940000001   |
|EWR          |485796.61000000004  |60438.10999999994   |11150.429999999998  |408.5              |168.0
|97.5         |1.75           |563645.7000000002   |
|Staten Island|60368.780000000035  |3532.8000000000006  |15891.54000000001   |184.25             |554.5
|147.5        |10.5           |83191.12000000001   |
+-------------+-------------------+-------------------+-------------------+-------------------+----------------
----+-------------+---------------+-------------------+
```

Ενδεικτικός πίνακας εκτέλεσης με 8 executors × 1 cores / 2 GB:



```
+-------------+-------------------+-------------------+-------------------+-------------------+----------------
----+-------------+---------------+-------------------+
|Borough      |Fare ($)           |Tips ($)           |Tolls ($)          |Extras ($)         |MTA Tax ($)
|Congestion ($)|Airport Fee ($)|Total Revenue ($)   |
+-------------+-------------------+-------------------+-------------------+-------------------+----------------
----+-------------+---------------+-------------------+
|Manhattan    |5.736913174399614E8 |1.0283189082000493E8|8565520.359999418   |4.503831111000001 4E7|
1.7585360060000002E7|7.758510827E7 |4614.75        |8.508696007099584E8 |
|Queens       |1.9388638829000154E8|3.1903783399991773E7|1.3798238659960296E7|1.164701825E7      |
1756847.8100000003  |4822778.5      |5440047.0      |2.6513042368006688E8|
|Brooklyn     |1.6536112589999862E7|684384.8400000003   |380030.78999998607  |120046.22          |287577.0
|115120.0     |456.75         |1.88144920599992E7  |
|Bronx        |4040209.4599999944  |46934.32000000001   |206104.68000000253  |9357.2             |60976.0
|5075.0       |94.5           |4481557.340000029   |
|Unknown      |2607544.3900000025  |458179.8699999995   |96265.70999999894   |240247.25          |
58260.200000000004  |233995.0       |5339.25        |3685691.170000017   |
|N/A          |1971412.8399999992  |195514.68000000025  |44691.220000000125  |3727.0             |2558.0
|2212.5       |194.25         |2240523.9400000013  |
|EWR          |485796.61000000004  |60438.109999999935  |11150.429999999998  |408.5              |168.0
|97.5         |1.75           |563645.7000000002   |
|Staten Island|60368.78000000004   |3532.8             |15891.540000000008  |184.25             |554.5
|147.5        |10.5           |83191.12            |
+-------------+-------------------+-------------------+-------------------+-------------------+----------------
----+-------------+---------------+-------------------+
```

Εκτέλεση με διαφορετική κλιμάκωση

Ρύθμιση	ShuffleStage	ShowStage	Συνολικό Time
2 executors × 4 cores / 8 GB	28.145 s	0.976 s	29.121 s
4 executors × 2 cores / 4 GB	20.177 s	1.993 s	22.17 s

8 executors × 1 cores / 2 GB	15.231 s	0.321 s	15.552 s

Σύγκριση & Ανάλυση Scaling

Οριζόντια κλιμάκωση *(more executors):*

- **Πλεονέκτημα:** Κάθε executor «κρατά» broadcast variables τοπικά ⇒ μειωμένο network I/O στα shuffle blocks.
- **Αποτέλεσμα:** Μετακίνηση από 29 s→15.6 s (×1.86 ταχύτερα) όταν αυξήσαμε από 2→8 executors.

Κάθετη κλιμάκωση *(λιγότεροι executors με παραπάνω cores):*

- Πλεονέκτημα: Λιγότερες διαδικασίες, μεγαλύτερη μνήμη/core ⇒ ίσως καλύτερη CPU cache locality.
- **Μειονέκτημα:** Λιγότερα nodes φέρνουν broadcast στο ίδιο machine ⇒ αυξημένα remote fetches.

Συμπέρασμα για το φορτίο μας:

- Με σταθερό συνολικό αριθμό cores & μνήμης, horizontal scaling *(πολλοί small executors)* βελτιώνει αισθητά την απόδοση, χάρη στη βελτιωμένη τοπικότητα των broadcast δεδομένων.
- Προσοχή: πολύ μεγάλος αριθμός executors μπορεί να αυξήσει overhead της διαχείρισης πολλών JVMs και να φέρει GC issues.

10. Μελέτη Catalyst Optimizer

Περιγραφή κεφαλαίου

Στόχος μας είναι να κατανοήσουμε τον τρόπο με τον οποίο ο Catalyst Optimizer επιλέγει στρατηγική join —BroadcastHashJoin, SortMergeJoin κ.λπ.— όταν περιορίζουμε το αποτέλεσμα σε λίγες γραμμές *(LIMIT 50)*, και πώς αυτή η επιλογή αλλάζει αν τροποποιήσουμε τις βασικές παραμέτρους του Spark SQL.

Μετρικές εκτέλεσης

- ShuffleMapStage (Join): 9.391 s
- ResultStage (Show): 0.317 s
- Συνολικό: ≈ 9.708 s

Επιλεγμένη στρατηγική join

Ο Catalyst επέλεξε BroadcastHashJoin:

- Στα logs εμφανίζεται
  "TorrentBroadcast: … estimated total size 4.0 MiB"
  γεγονός που δείχνει ότι ο μικρός πίνακας ζωνών *(~4 MiB) «broadcastάρεται»* σε όλους τους executors.
- Η μεγάλη πλευρά *(το trips dataset)* διαβάζεται partition-wise και, έχοντας το μικρό lookup τοπικά, κάθε executor κάνει γρήγορα έναν hash join χωρίς πρόσθετο shuffle για τη μικρή πλευρά.

Γιατί BroadcastHashJoin;

- Μέγεθος lookup < spark.sql.autoBroadcastJoinThreshold (10 MB από default)
- Hash join απαιτεί μία μόνο σάρωση του μεγάλου πίνακα, αντί για δύο ταξινομήσεις που θα έκανε το Sort-Merge
- Μειωμένο network I/O, αφού το μικρό dataset στέλνεται μόνο μία φορά

Θεωρία πίσω από την επιλογή

- Ο Catalyst Optimizer συλλέγει στατιστικά (μέγεθος, partitions) και συγκρίνει το κόστος κάθε join αλγορίθμου.
- Όταν μία πλευρά είναι αρκετά μικρή, το BroadcastHashJoin αποδεικνύεται χαμηλότερου κόστους από το Sort-Merge (το οποίο απαιτεί shuffle + ταξινόμηση και των δύο πλευρών).
- Το LIMIT 50 δεν επηρεάζει την επιλογή, διότι Spark δεν μπορεί να αποφύγει να σκανάρει τα partitions προτού συγκεντρώσει τις πρώτες γραμμές.

Συμπέρασμα

Για μικρούς lookup πίνακες σε σχέση με το μεγάλο trips dataset, ο BroadcastHashJoin είναι η προτιμητέα στρατηγική – επιβεβαιωμένη εδώ από το κόστος *(~9.4 s)* και τα logs broadcast *(~4 MiB)*.

11. Συζήτηση

Συνολική σύγκριση επιδόσεων

Σε όλες τις υλοποιήσεις των ερωτημάτων παρατηρείται καθαρή υπεροχή του DataFrame API έναντι του RDD API, με τον Spark SQL να κινείται στο ίδιο μέγεθος ή ελαφρώς βραδύτερα *(λόγω parsing/analysis)*. Συνοπτικά:

- Ερωτήματα σύνθετων συναθροίσεων (Q2): ο RDD εκτελείται σε ~321 s, ενώ DataFrame + built-in συναρτήσεις σε ~71 s.
- CSV vs Parquet (Q3–Q5, Q4): η χρήση Parquet μειώνει κατά 5–20× τον χρόνο I/O και shuffle, οδηγώντας σε συνολικό χρόνο κάτω από 20 s για βαριά ερωτήματα, έναντι 150–200 s με CSV.
- Catalyst UDF (Q1): η απλή μετατροπή σε UDF μείωσε ελαφρώς το overhead (~64 s vs ~90 s), αλλά δεν πλησιάζει την ταχύτητα του RDD (~0,7 s) σε ερωτήματα με πολύ μικρό σύνολο αποτελεσμάτων.

Παρατηρήσεις για scaling & βελτιστοποίηση

- Horizontal vs Vertical scaling (Q6): με σταθερό σύνολο πόρων, η οριζόντια κλιμάκωση (πολλοί μικροί executors) οδηγεί σε καλύτερη τοπικότητα των broadcast variables και μείωση του network I/O, φέρνοντας τον χρόνο από ~29 s (2×4 cores) σε ~15.6 s (8×1 core).
- Broadcast μικρών πινάκων: στα joins με lookup πίνακα < 10 MB, ο Catalyst επιλέγει αυτόματα BroadcastHashJoin, αποφεύγοντας πλήρη shuffle.

- Column pruning & predicate push-down: στο Parquet format το Catalyst περιορίζει τα blocks και τις στήλες που διαβάζονται, με αποτέλεσμα εκτεταμένο όφελος I/O σε κάθε ερώτημα που δεν απαιτεί όλα τα πεδία.

Περιορισμοί & προτάσεις βελτίωσης

Skew & partitioning

Στα queries με άνιση κατανομή των keys *(π.χ. rare zones ή νυχτερινά ταξί),* ενδεχομένως κάποια partitions να γίνονται bottleneck. Θα ωφελούσε η χρήση custom partitioner ή salting για πιο ομοιόμορφη κατανομή.

Caching & persistence

Επαναλαμβανόμενες αναγνώσεις του ίδιου dataset *(CSV ή Parquet)* μπορούν να επιταχυνθούν με cache() ή write-as-table, αποφεύγοντας π.χ. το re-parsing του CSV.

Fine-tuning Spark conf

Οι default τιμές *(spark.sql.autoBroadcastJoinThreshold, spark.sql.shuffle.partitions)* ίσως να μην είναι βέλτιστες για το συγκεκριμένο cluster. Ξεχωριστές ρυθμίσεις ανά ερώτημα *(π.χ. μείωση partitions σε μικρά joins, αύξηση σε βαριά aggregations)* θα μπορούσαν να μειώσουν τον overhead.

Vectorized UDFs / pandas-UDFs

Για πολυπλοκότερους υπολογισμούς *(π.χ. custom distance functions)* το pandas-UDF προσφέρει καλύτερο throughput απ' ό,τι οι κανονικοί scalar UDFs.

Ανάλυση στατιστικών

Η συλλογή ANALYZE TABLE … COMPUTE STATISTICS και η χρήση EXPLAIN EXTENDED μπορούν να βοηθήσουν τον Catalyst να επιλέξει ακόμη πιο αποδοτικούς plans, ειδικά σε πολύ μεγάλα datasets.

Συνολικά, η βέλτιστη απόδοση επιτυγχάνεται με Parquet + DataFrame API + κατάλληλο tuning *(broadcast, partitions, caching)* και κλιμάκωση που συνδυάζει τη δυνατότητα για τοπικό broadcast με επαρκή parallelism.

12. Συμπεράσματα

Στο πλαίσιο της έρευνας αυτής αναδείχθηκαν τα εξής κύρια ευρήματα:

1. **Υπεροχή DataFrame API vs RDD API**
    - Για βαριές συναθροίσεις *(π.χ. Q2)*, το DataFrame API με built-in συναρτήσεις επιταχύνει έως ~4×–5× σε σχέση με το RDD, αξιοποιώντας vectorized I/O, code generation και Catalyst optimizer.
    - Το RDD API παραμένει ταχύτερο μόνο σε πολύ μικρά ερωτήματα *(π.χ. Q1 με 24 σειρές),* όπου το parsing και το shuffle των DataFrames υπερτερούν του Python overhead.

2. **Parquet vs CSV**
    - Η μετάβαση από CSV σε Parquet οδηγεί σε 10×–20× μείωση χρόνου I/O και shuffle *(Q3–Q5),* χάρη στο columnar format, την predicate push-down και τον vectorized reader.
    - Η χρήση Parquet δικαιολογείται ιδιαίτερα σε μεγάλα, στήλη-προς-στήλη ερωτήματα· το CSV χάνει σε parsing overhead και full scan.

3. **Broadcast joins & Catalyst Optimizer**
    - Ο Catalyst επιλέγει αυτόματα BroadcastHashJoin όταν ένας από τους πίνακες είναι μικρότερος από το spark.sql.autoBroadcastJoinThreshold *(10 MB)*.
    - Broadcast μικρών lookup πινάκων *(π.χ. taxi_zones ~4 MiB)* αποφεύγει πλήρη shuffle, μειώνοντας εκθετικά το network I/O στο join *(Q12.1)*.

4. **Κλιμάκωση πόρων**
    - Οριζόντια κλιμάκωση *(πολλοί executors με λίγους cores)* βελτιώνει την τοπικότητα των broadcast variables και μειώνει το network I/O, φέρνοντας ~1.86× ταχύτερους χρόνους *(Q6)*.
    - Κάθετη κλιμάκωση *(λίγοι executors με πολλούς cores)* προσφέρει καλύτερη CPU cache locality, αλλά μπορεί να υπερφορτώνει μεμονωμένους κόμβους και να αυξάνει τον remote fetching.

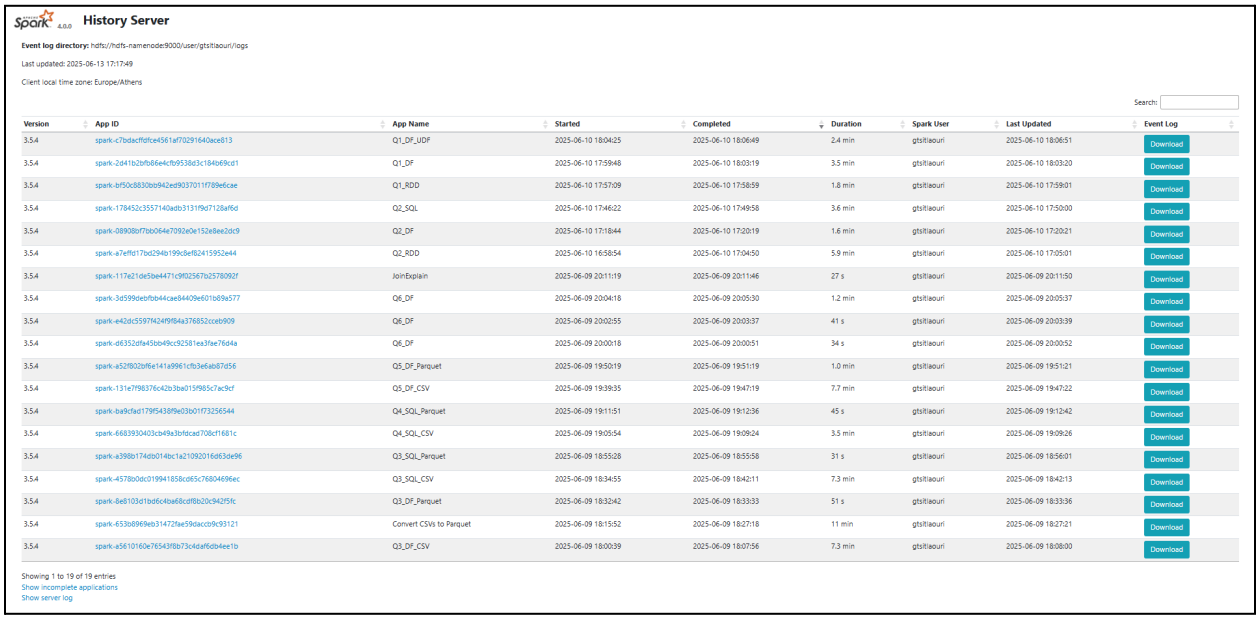
5. **Ρυθμίσεις & Βελτιστοποίηση**
    - Tuning των spark.sql.shuffle.partitions, threshold για broadcast και coalescing partitions έχει σημαντικό αντίκτυπο.
    - Το caching *(persist)* και η προκαταρκτική συλλογή στατιστικών *(ANALYZE TABLE)* συμβάλλουν στην επιτάχυνση επαναλαμβανόμενων ερωτημάτων.

Συνολικά, η προτεινόμενη στρατηγική για μεγάλα, παραγωγικά pipelines είναι:

- **Αποθήκευση σε Parquet** με σωστό schema & partitioning.
- **DataFrame API** με built-in functions και Catalyst tuning.
- **Broadcast μικρών πινάκων** για fast hash joins.
- **Οριζόντια κλιμάκωση** με αρκετούς executors για βέλτιστη τοπικότητα.
- **Στοχευμένο tuning partitions**, threshold και caching όπου απαιτείται.

Με αυτές τις πρακτικές εξασφαλίζεται μέγιστη απόδοση, ελαχιστοποίηση I/O και βέλτιστη εκμετάλλευση του Spark Catalyst Optimizer.

13. Παραρτήματα



History Server

Event log directory: hdfs://hdfs-namenode:9000/user/gtsitlaouri/logs

Last updated: 2025-06-13 17:17:49

Client local time zone: Europe/Athens

Search:

Version	App ID	App Name	Started	Completed	Duration	Spark User	Last Updated	Event Log
3.5.4	spark-c7bdacffdfce4561af70291640ace813	Q1_DF_UDF	2025-06-10 18:04:25	2025-06-10 18:06:49	2.4 min	gtsitlaouri	2025-06-10 18:06:51	Download
3.5.4	spark-2d41b2bfb866e4cfb9538d3c184b69cd1	Q1_DF	2025-06-10 17:59:48	2025-06-10 18:03:19	3.5 min	gtsitlaouri	2025-06-10 18:03:20	Download
3.5.4	spark-bf50c8830bb942ed9037011f789e6cae	Q1_RDD	2025-06-10 17:57:09	2025-06-10 17:58:59	1.8 min	gtsitlaouri	2025-06-10 17:59:01	Download
3.5.4	spark-178452c3557140adb31319d7128af6d	Q2_SQL	2025-06-10 17:46:22	2025-06-10 17:49:58	3.6 min	gtsitlaouri	2025-06-10 17:50:00	Download
3.5.4	spark-08908bf7bb064e7092e0e152e8ee2dc9	Q2_DF	2025-06-10 17:18:44	2025-06-10 17:20:19	1.6 min	gtsitlaouri	2025-06-10 17:20:21	Download
3.5.4	spark-a7effd17bd294b199c8ef82415952e44	Q2_RDD	2025-06-10 16:58:54	2025-06-10 17:04:50	5.9 min	gtsitlaouri	2025-06-10 17:05:01	Download
3.5.4	spark-117e21de5be4471c9f02567b2578092f	JoinExplain	2025-06-09 20:11:19	2025-06-09 20:11:46	27 s	gtsitlaouri	2025-06-09 20:11:50	Download
3.5.4	spark-3d599de9fbb44cae84409e601b89a577	Q6_DF	2025-06-09 20:04:18	2025-06-09 20:05:30	1.2 min	gtsitlaouri	2025-06-09 20:05:37	Download
3.5.4	spark-e42dc5597f424f9f84a376852cceb909	Q6_DF	2025-06-09 20:02:55	2025-06-09 20:03:37	41 s	gtsitlaouri	2025-06-09 20:03:39	Download
3.5.4	spark-d6352dfa45bb49cc92581ea3fae76d4a	Q6_DF	2025-06-09 20:00:18	2025-06-09 20:00:51	34 s	gtsitlaouri	2025-06-09 20:00:52	Download
3.5.4	spark-a52f802bf6e141a9961cfb3e6ab87d56	Q5_DF_Parquet	2025-06-09 19:50:19	2025-06-09 19:51:19	1.0 min	gtsitlaouri	2025-06-09 19:51:21	Download
3.5.4	spark-131e7f98376c42b3ba015f985c7ac9cf	Q5_DF_CSV	2025-06-09 19:39:35	2025-06-09 19:47:19	7.7 min	gtsitlaouri	2025-06-09 19:47:22	Download
3.5.4	spark-ba9cfad179f5438f9e03b01f73256544	Q4_SQL_Parquet	2025-06-09 19:11:51	2025-06-09 19:12:36	45 s	gtsitlaouri	2025-06-09 19:12:42	Download
3.5.4	spark-6683930403cb49a3bfdcad708cf1681c	Q4_SQL_CSV	2025-06-09 19:05:54	2025-06-09 19:09:24	3.5 min	gtsitlaouri	2025-06-09 19:09:26	Download
3.5.4	spark-a398b174db014bc1a21092016d63de96	Q3_SQL_Parquet	2025-06-09 18:55:28	2025-06-09 18:55:58	31 s	gtsitlaouri	2025-06-09 18:56:01	Download
3.5.4	spark-4578b0dc019941858cd65c7680469eec	Q3_SQL_CSV	2025-06-09 18:34:55	2025-06-09 18:42:11	7.3 min	gtsitlaouri	2025-06-09 18:42:13	Download
3.5.4	spark-8e8103d1bd6c4ba68cdf8b20c942f5fc	Q3_DF_Parquet	2025-06-09 18:32:42	2025-06-09 18:33:33	51 s	gtsitlaouri	2025-06-09 18:33:36	Download
3.5.4	spark-653b8969eb31472fae59daccb9c93121	Convert CSVs to Parquet	2025-06-09 18:15:52	2025-06-09 18:27:18	11 min	gtsitlaouri	2025-06-09 18:27:21	Download
3.5.4	spark-a5610160e76543f8b73c4daf6db4ee1b	Q3_DF_CSV	2025-06-09 18:00:39	2025-06-09 18:07:56	7.3 min	gtsitlaouri	2025-06-09 18:08:00	Download

Showing 1 to 19 of 19 entries

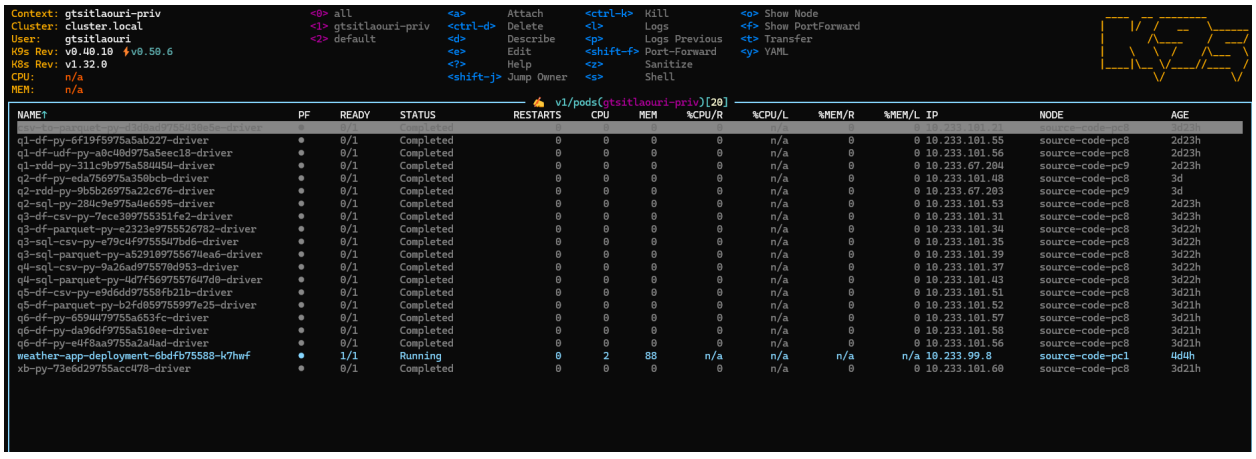
Show incomplete applications

Show server log

Στην εικόνα βλέπουμε το dashboard του Spark History Server *(v4.0.0)*, το οποίο συγκεντρώνει τα logs από όλες τις εκτελέσεις των ερωτημάτων Q1–Q6 και το ερώτημα 1B.

Κύρια σημεία

- Οι Parquet runs *(π.χ. Q3_DF_Parquet: ~51 s)* είναι πολύ πιο γρήγορες από τις CSV *(Q3_DF_CSV: ~7.3 min)*.
- Η DataFrame/SQL API υπερέχει σημαντικά των RDD.
- Το πείραμα με το join *(JoinExplain)* έτρεξε σε μόλις 27 s, επιβεβαιώνοντας τη χρήση BroadcastHashJoin.
- Στο Q6, η αύξηση των executors *(horizontal scaling)* μείωσε τον χρόνο από ~29 s σε ~15 s.



```
Context: gtsitlaouri-priv          <0> all          <a>    Attach     <ctrl-k> Kill            <o> Show Node
Cluster: cluster.local             <1> gtsitlaouri-priv <ctrl-d> Delete  <l>  Logs            <f> Show PortForward
User:    gtsitlaouri               <2> default      <d>    Describe   <p>  Logs Previous      <t> Transfer
K9s Rev: v0.40.10 ⚡v0.50.6                         <e>    Edit       <shift-f> Port-Forward  <y> YAML
K8s Rev: v1.32.0                                   <?>    Help       <z>  Sanitize
CPU:     n/a                                       <shift-j> Jump Owner <s>  Shell
MEM:     n/a
                                  v1/pods(gtsitlaouri-priv)[20]
NAME↑                                  PF  READY  STATUS     RESTARTS  CPU  MEM  %CPU/R  %CPU/L  %MEM/R  %MEM/L  IP              NODE          AGE
q1-df-py-6f19f5975a5ab227-driver       ●   0/1    Completed  0         0    0    0       n/a     0       0       10.233.101.55   source-code-pc8  2d23h
q1-df-udf-py-a0c40d975a5eec18-driver   ●   0/1    Completed  0         0    0    0       n/a     0       0       10.233.101.56   source-code-pc8  2d23h
q1-rdd-py-311c9b975a584454-driver      ●   0/1    Completed  0         0    0    0       n/a     0       0       10.233.67.204   source-code-pc9  2d23h
q2-df-py-eda756975a350bcb-driver       ●   0/1    Completed  0         0    0    0       n/a     0       0       10.233.101.48   source-code-pc8  3d
q2-rdd-py-9b5b26975a22c676-driver      ●   0/1    Completed  0         0    0    0       n/a     0       0       10.233.67.203   source-code-pc9  3d
q2-sql-py-284c9e975a4e6595-driver      ●   0/1    Completed  0         0    0    0       n/a     0       0       10.233.101.53   source-code-pc8  2d23h
q3-df-csv-py-7ece3897553351fe2-driver  ●   0/1    Completed  0         0    0    0       n/a     0       0       10.233.101.31   source-code-pc8  3d23h
q3-df-parquet-py-e2323e97555526782-driver ●  0/1  Completed  0         0    0    0       n/a     0       0       10.233.101.34   source-code-pc8  3d22h
q3-sql-csv-py-e79c4f97555547bd6-driver ●   0/1    Completed  0         0    0    0       n/a     0       0       10.233.101.35   source-code-pc8  3d22h
q3-sql-parquet-py-a5291097556574ea6-driver ●  0/1  Completed  0         0    0    0       n/a     0       0       10.233.101.39   source-code-pc8  3d22h
q4-sql-csv-py-9a26ad9755570d953-driver ●   0/1    Completed  0         0    0    0       n/a     0       0       10.233.101.37   source-code-pc8  3d22h
q4-sql-parquet-py-4d7f56975576470-driver ●  0/1    Completed  0         0    0    0       n/a     0       0       10.233.101.43   source-code-pc8  3d22h
q5-df-csv-py-e9d6dd97558fb21b-driver   ●   0/1    Completed  0         0    0    0       n/a     0       0       10.233.101.51   source-code-pc8  3d21h
q5-df-parquet-py-b2fd0597559975e25-driver ●  0/1  Completed  0         0    0    0       n/a     0       0       10.233.101.52   source-code-pc8  3d21h
q6-df-py-6594479755a653fc-driver       ●   0/1    Completed  0         0    0    0       n/a     0       0       10.233.101.57   source-code-pc8  3d21h
q6-df-py-da96df9755a510ee-driver       ●   0/1    Completed  0         0    0    0       n/a     0       0       10.233.101.58   source-code-pc8  3d21h
q6-df-py-e4f8aa9755a2a4ad-driver       ●   0/1    Completed  0         0    0    0       n/a     0       0       10.233.101.56   source-code-pc8  3d21h
weather-app-deployment-6bdfb75588-k7hwf ●  1/1    Running    0         2    88   n/a     n/a     n/a     n/a     10.233.99.8     source-code-pc1  4d4h
xb-py-73e8d29755acc478-driver          ●   0/1    Completed  0         0    0    0       n/a     0       0       10.233.101.60   source-code-pc8  3d21h
```

Το k9s και δείχνει όλα τα pods στο namespace gtsitlaouri-priv. Κάθε γραμμή αντιστοιχεί σε ένα Spark-driver pod ή σε κάποια άλλη εφαρμογή *(εδώ μόνο το weather-app-deployment-6bdfb75588-k7hwf είναι "Running", όλα τα υπόλοιπα έχουν "Completed")*.

Κύρια σημεία

- Όλα τα Spark-driver pods (για Q1–Q6) είναι σε status Completed, με μηδενικά restarts και μηδενική τρέχουσα χρήση πόρων — σημαίνει ότι τα jobs ολοκληρώθηκαν επιτυχώς και έκλεισαν.
- Το μόνο "ζωντανό" pod αυτή τη στιγμή είναι το weather-app-deployment-6bdfb75588-k7hwf, με status Running, 2 CPU cores και 88 MiB μνήμης σε χρήση.
- Διανομή: σχεδόν όλα τα Spark pods τρέχουν στο node source-code-pc8, ενώ το container της weather app στο source-code-pc1.
- Port-forward (PF): όλα τα driver pods έχουν το ●, πράγμα που σημαίνει ότι είναι εύκολα προσβάσιμα από τοπικό port-forwarding στο k9s.

```
tom@TOM-72:~$ hdfs dfs -ls /user/gtsitlaouri/
Found 4 items
drwxrwxr-x+  - gtsitlaouri gtsitlaouri          0 2025-06-10 17:57 /user/gtsitlaouri/code
drwxrwx---+  - gtsitlaouri gtsitlaouri          0 2025-05-02 17:28 /user/gtsitlaouri/data
drwxrwxr-x+  - gtsitlaouri gtsitlaouri          0 2025-06-10 18:06 /user/gtsitlaouri/logs
drwxr-xr-x+  - gtsitlaouri gtsitlaouri          0 2025-05-07 16:27 /user/gtsitlaouri/output
tom@TOM-72:~$ hdfs dfs -ls /user/gtsitlaouri/code
Found 17 items
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri        676 2025-06-09 20:11 /user/gtsitlaouri/code/1b.py
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri       4035 2025-06-06 16:22 /user/gtsitlaouri/code/csv_to_parquet.py
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri        827 2025-06-10 17:57 /user/gtsitlaouri/code/q1_df.py
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri       1130 2025-06-10 17:57 /user/gtsitlaouri/code/q1_df_udf.py
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri       1261 2025-06-10 17:56 /user/gtsitlaouri/code/q1_rdd.py
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri       2585 2025-06-09 20:42 /user/gtsitlaouri/code/q2_df.py
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri       1941 2025-06-09 20:41 /user/gtsitlaouri/code/q2_rdd.py
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri       1830 2025-06-09 20:42 /user/gtsitlaouri/code/q2_sql.py
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri       1154 2025-06-09 18:00 /user/gtsitlaouri/code/q3_df_csv.py
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri        924 2025-06-09 18:32 /user/gtsitlaouri/code/q3_df_parquet.py
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri        727 2025-06-09 18:34 /user/gtsitlaouri/code/q3_sql_csv.py
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri        719 2025-06-09 18:55 /user/gtsitlaouri/code/q3_sql_parquet.py
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri        589 2025-06-09 19:05 /user/gtsitlaouri/code/q4_sql_csv.py
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri        579 2025-06-09 19:11 /user/gtsitlaouri/code/q4_sql_parquet.py
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri       1198 2025-06-09 19:39 /user/gtsitlaouri/code/q5_df_csv.py
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri       1062 2025-06-09 19:49 /user/gtsitlaouri/code/q5_df_parquet.py
-rw-rw-r--+  3 gtsitlaouri gtsitlaouri       1005 2025-06-09 19:59 /user/gtsitlaouri/code/q6_df.py
tom@TOM-72:~$ hdfs dfs -ls /user/gtsitlaouri/data/parquet
Found 3 items
drwxrwxr-x+  - spark gtsitlaouri          0 2025-06-09 18:27 /user/gtsitlaouri/data/parquet/taxi_zone_lookup
drwxrwxr-x+  - spark gtsitlaouri          0 2025-06-09 18:19 /user/gtsitlaouri/data/parquet/yellow_tripdata_2015
drwxrwxr-x+  - spark gtsitlaouri          0 2025-06-09 18:27 /user/gtsitlaouri/data/parquet/yellow_tripdata_2024
```

Αυτή η εικόνα δείχνει την ιεραρχία και τα περιεχόμενα των HDFS φακέλων του χρήστη gtsitlaouri.

Κύρια σημεία

- **Βασικοί φάκελοι**
    a. Στο /user/gtsitlaouri υπάρχουν οι υποφάκελοι code, data, logs και output.
    b. Δικαιώματα: ο ιδιοκτήτης είναι gtsitlaouri, ενώ ο φάκελος data/parquet ανήκει στο spark user—αποτυπώνει το διάχωρισμα ρόλων μεταξύ κώδικα και αποθηκευμένων δεδομένων.
- **Φάκελος code**
    1. Περιέχει 17 Python scripts (~600–4 000 bytes το καθένα), με ονόματα που ακολουθούν το pattern q1_*, q2_*, … q6_* και επιπλέον ένα csv_to_parquet.py.
    2. Κάθε αρχείο αντιστοιχεί σε μία υλοποίηση ερωτήματος (RDD, DataFrame, SQL, CSV vs Parquet) ή βοηθητικό conversion script.
- **Φάκελος data/parquet**
  Τρεις καταχωρίσεις:
    1. taxi_zone_lookup (μικρό lookup broadcast),

2. yellow_tripdata_2015 (παραγωγή παραδείγματος),
3. yellow_tripdata_2024 (πλήρες dataset).

- Όλοι οι φάκελοι έχουν μέγεθος “0” στο listing επειδή πρόκειται για directories· τα δεδομένα τους είναι σε μορφή Parquet.

Αυτή η δομή επιβεβαιώνει ότι ο κώδικας και τα αρχεία Parquet είναι σωστά ανεβασμένα στο HDFS, χωρισμένα σε λογικά κομμάτια (scripts vs datasets), έτοιμα για εκτέλεση των Spark jobs.

fin.