Project Documentation

NYC Yellow Taxi Analytics

Course: Big Data Processing and Analysis - INF615
Instructor: Garofalakis Minos
Department: Electrical and Computer Engineering, Technical University of Crete

ΠΟΛΥΤΕΧΝΕΙΟ
ΚΡΗΤΗΣ /
TECHNICAL
UNIVERSITY
OF CRETE

September 13, 2025

Student:
Karalis Asterinos ID: 2020030107

Contents

1. Introduction

1.1. About this project.

This project focuses on processing and analyzing the NYC Yellow Taxi dataset using Apache Spark. We implement a set of analytical queries that provide insights into passenger behavior, fare structures, zone-based patterns, and urban mobility. Due to the dataset's large size, we also explore approximate techniques such as Reservoir Sampling and Count-Min Sketch to improve performance while maintaining acceptable accuracy.

1.2. Project setup

The setup of this project was a Windows 11 laptop with local installations of Apache Spark, Hadoop HDFS. The app was written in Scala.

2. Dataset

2.1. About the data.

The dataset used is the NYC Yellow Taxi Trip Record Data provided by the New York City Taxi and Limousine Commission (TLC). It contains detailed trip-level data for yellow taxi rides in New York City between 2021 and 2023. The total volume is around 9 GB in CSV format and includes millions of rows.

2.2. Regarding data's structure.

Each record contains various attributes such as:

- `tpep_pickup_datetime`, `tpep_dropoff_datetime` –Pickup and drop-off timestamps
- `passenger_count` – Number of passengers
- `trip_distance` – Distance of the trip in miles
- `PULocationID`, `DOLocationID` – Pickup and drop-off zones (Taxi Zone IDs)
- `fare_amount`, `tip_amount`, `total_amount` – Fare-related financial metrics
- `congestion_surcharge`, `airport_fee`, `mta_tax` – Additional surcharges

The dataset is read using a predefined schema in Spark to ensure proper data typing and parsing.

Timestamps are converted to 'timestamp' type for temporal queries, and filters are applied to remove invalid or null entries.

3. Queries

This section describes the six analytical queries designed to extract insights from the NYC Yellow Taxi dataset for the period 2021–2023. Each query focuses on a distinct aspect of passenger behavior, financial metrics, or traffic patterns.

3.1. Query 1: Top 20 Busiest Pickup Locations

Objective: Identify pickup locations (zones) with the highest trip counts.

- Filter out trips with null or invalid pickup location IDs.
- Group the data by `PULocationID`.
- Compute the count of trips (`count(*)`) for each pickup location.
- Sort the results in descending order of trip count and return the top 20.

3.2. Query 2: Top 50 Weekday Morning Drop-offs

Objective: Identify the most frequent drop-off zones during weekday mornings (6 AM to 11 AM).

- Extract the hour and day of the week from the `dropoff_datetime`.
- Keep only trips that occurred Monday through Friday, between 6:00 and 11:59.
- Group the data by `DOLocationID` and hour.
- Compute the count of drop-offs and order the results in descending order.
- Return the top 50 drop-off locations by volume.

3.3. Query 3: Busiest Taxi Routes (Pickup-Dropoff Pair)

Objective: Identify the most frequently traveled routes, defined by a specific pickup and drop-off zone pair.

- Group the data by the combination of `PULocationID` and `DOLocationID`.
- Compute the count of trips for each unique route pair.
- Sort the results by trip count in descending order and display the top 20 busiest routes.

3.4. Query 4: Passenger Count Distribution

Objective: Analyze the distribution of passenger counts per trip.

- Filter trips with valid passenger counts (e.g., between 1 and 8).
- Group the data by `passenger_count`.
- Compute the total number of trips for each passenger count.
- Order the results by passenger count to show the distribution.

3.5. Query 5: Payment Type Analysis

Objective: Determine the frequency of different payment types.

- Create a user-defined function (UDF) to map numerical '$payment - type$' IDs to descriptive strings (e.g., "1" to "Credit card").
- Group the data by the descriptive payment type.
- Compute the count of trips for each payment type.
- Order the results by trip count in descending order.

3.6. Query 6: Top 10 Most Frequent Tip Amounts

Objective: Identify the most commonly occurring tip amounts.

- Filter trips to include only positive tip amounts.
- Group the data by `tip_amount`.
- Compute the frequency of each unique tip amount.
- Sort the results in descending order of frequency and display the top 10.

4. Sampling Techniques

This project explores two approximate data summarization techniques to handle large-scale NYC Taxi data: **Algorithm R (Reservoir Sampling)** and **Count-Min Sketch**.

4.1. Reservoir Sampling - Algorithm R

Algorithm R is designed to select a uniform sample of a fixed size (k) from a stream of data where the total number of items is unknown or too large to fit in memory. In this project, we implemented our own version of **Algorithm R** to generate a small but representative sample of 100,000 rows from the full dataset (73 million rows):

```
private def reservoirSample[T: ClassTag](input: Iterator[T], k:
   Int): Array[T] = {
  val reservoir = new Array[T](k)
  val random = new Random()
  for ((elem, i) <- input.zipWithIndex) {
    if (i < k) {
      reservoir(i) = elem
    } else {
      val j = random.nextInt(i + 1)
      if (j < k) {
        reservoir(j) = elem
      }
    }
  }
  reservoir
}
```

The algorithm works as follows:

1. The first k elements from the data stream are placed directly into a fixed-size array called the *reservoir*.

2. For each subsequent element (the $(i+1)^{th}$ element), a random integer j is generated between 0 and i.

3. If j is less than k, the $(j+1)^{th}$ element in the reservoir is replaced with the new element from the stream.

This process ensures that at any point, the reservoir contains a uniform random sample of all the items seen so far.
To make the sampled results comparable to the full dataset queries, we also apply a **scale factor**, i.e., multiply the sampled counts by $\frac{N}{k}$ (where N is the dataset size and k the sample size). This ensures that the reservoir estimates are projected to the same magnitude as the exact results.

4.2. Count-Min Sketch

Count-Min Sketch is a probabilistic data structure that provides approximate counts for events in a data stream. It is particularly effective for queries involving frequency estimation, such as identifying the most frequent items. In this project, we used Spark's built-in `CountMinSketch` utility to efficiently approximate the counts of categorical data without performing a full scan or storing all unique items.
The implementation was achieved through a helper function, `buildCmsForKey`, which performs a distributed aggregation of sketches. The process is as follows:

1. A `CountMinSketch` instance is created with a relative error (ϵ), a confidence, and a fixed random seed of 1 for reproducibility.

2. Each partition of the input `DataFrame` creates its own local `CountMinSketch`. The key for the sketch is a string representation of the column(s) being analyzed (e.g., `PULocationID` or a concatenation of `PULocationID` and `DOLocationID`).

3. All local sketches are then merged into a single, global `CountMinSketch` using the `mergeInPlace` method, which combines the frequency arrays from each local sketch.

4. Finally, the global sketch is used to estimate the count for specific items identified by the exact calculations. This allows for a fast and memory-efficient approximation of item frequencies.

This approach was applied to several analytical queries to estimate frequencies, including:

- **Top 20 Busiest Pickup Locations**: The sketch was built on the `PULocationID` column to estimate the trip counts.

- **Top 50 Weekday Morning Dropoffs**: The sketch was built on a combined key of `DOLocationID` and the hour of drop-off.

- **Busiest Taxi Routes**: The sketch was built on a concatenated key of `PULocationID` and `DOLocationID` to approximate the frequency of specific routes.

- **Passenger Count Distribution**: The sketch was built on the passenger count as a string to estimate the distribution.
- **Payment Type Analysis**: The sketch was used on the payment type to approximate the frequency of each payment method.

5. Evaluation

The following tables compare exact results with approximations. **Reservoir Sampling** provides reasonable estimates but with natural variance, especially for rare items, since only a small fraction of the data is used. **Count-Min Sketch (CMS)** achieves high accuracy for frequent items, with small overestimation from hash collisions. Stricter parameters reduce error further but at the cost of slower runtime and higher memory use.

5.1. Query 1: Top 20 Busiest Pickup Locations

PULocationID	exact_trip_count	sampled_trip_count	cms_0.01	cms_0.005
237	3610790	3688770	3610790	3610790
132	3276396	3215813	3276396	3276396
236	3242518	3212871	3242518	3242518
161	2914716	2914975	2914716	2914716
186	2494725	2533961	2494725	2494725
162	2430674	2418480	2430674	2430674
142	2408971	2368463	2408971	2408971
170	2289514	2241949	2289988	2289514
48	2166756	2128674	2166756	2166756
239	2143349	2144856	2143349	2143349
230	2110760	2105137	2110760	2110760
163	2047396	2094104	2047396	2047396
141	2026770	2000689	2026770	2026770
234	1980368	1952143	1980368	1980368
138	1962350	1938903	1962350	1962350
79	1866637	1880795	1866637	1866637
68	1806085	1807240	1806085	1806085
107	1752292	1718974	1753285	1752435
263	1633389	1671899	1633389	1633389
238	1610885	1653511	1610885	1610885

Reservoir sampling gives close but slightly shifted counts due to scaling. CMS with $\epsilon = 0.01$ already produces exact matches for heavy hitters except for minor deviations (e.g., PULocationID 170, 107). CMS with $\epsilon = 0.005$ fully stabilizes and matches exact results, though runtime is slower.

5.2. Query 2: Top 20 Weekday Morning Dropoffs

DOLocationID_hour	exact_dropoff_count	sampled_dropoff_count	cms_0.01	cms_0.005
161_8	182224	188300	274924	216459
161_9	181986	159614	462275	228119
237_11	176449	189036	482824	303837
161_10	165401	158143	411763	229020
236_11	164165	179474	380969	294682
237_10	160191	150787	358898	274903
161_11	152977	141225	267758	207558
236_10	147587	141225	401734	208080
161_7	147089	152258	319454	178741
237_9	141323	140490	266127	232535
162_9	126820	130192	276085	172360
236_9	126158	141225	192865	160576
162_8	123445	133870	373075	240124
236_8	120488	130192	269710	237566
237_8	114627	100770	272881	157959
162_10	107057	122836	235882	154100
170_9	99112	101506	283674	135527
162_11	96732	103712	213330	134568
170_10	96641	85324	457378	272336
170_11	94331	94150	374942	167125

Here we observe that reservoir sampling approximates the shape but with some variance (sometimes under/over counts). CMS with $\epsilon = 0.01$ inflates values heavily, while $\epsilon = 0.005$ is more balanced. The strict $\epsilon = 0.001$ (not shown) gave exact matches but required much higher runtime.

5.3. Query 3: Busiest Taxi Routes (Top 20)

Route	exact_trip_count	sampled_trip_count	cms_0.01	cms_0.005
237_236	515644	533272	772358	661779
236_237	439282	435444	636830	517134
264_264	362424	358947	467801	419859
237_237	348073	366303	584223	452640
236_236	337529	322905	646734	467760
237_161	211275	217722	456607	239646
161_237	198775	185358	413365	314074
239_142	191500	210367	459187	282484
239_238	190937	187565	473352	346717
142_239	190845	201540	402282	321294
141_236	186454	192713	381155	286689
161_236	176914	197862	457064	300813
237_162	172811	161820	444680	251914
263_141	164618	159614	376075	263731

236_161	162524	155201	332452	204884
238_239	162468	172854	362788	245017
236_141	158537	158143	358589	183596
142_238	155657	162556	261427	206413
132_132	154955	163292	271283	214411
237_141	153171	—	341843	276895

Reservoir sampling approximates route volumes but with higher variance. CMS with $\epsilon = 0.01$ inflates counts significantly due to collisions. CMS with $\epsilon = 0.005$ reduces error but still has some overestimation. At $\epsilon = 0.001$, results matched exactly but were very slow.

5.4. Query 4: Passenger Count Distribution

passenger_count	exact_trip_count	sampled_trip_count	cms_0.01	cms_0.005
1.0	55498039	5.5467633E7	55498039	55498039
2.0	11344249	1.1498078E7	11344249	11344249
3.0	2949291	2887024	2949291	2949291
4.0	1301703	1286472	1301703	1301703
5.0	1491540	1483599	1491540	1491540
6.0	969299	931203	969299	969299
7.0	315	736	315	315
8.0	215	—	215	215

Reservoir sampling provides a good distribution approximation but slightly misestimates rare counts (e.g., 7 and 8 passengers). Both CMS configurations reproduce exact results for frequent counts. At $\epsilon = 0.005$, the results are exact with moderate runtime.

5.5. Query 5: Payment Type Analysis

payment_type_desc	exact_trip_count	sampled_trip_count	cms_0.01	cms_0.005
Credit card	57503192	5.7524224E7	57503192	57503192
Cash	15231547	1.5199353E7	15231547	15231547
Dispute	455132	486197	455132	455132
No charge	364866	344972	364866	364866
Unknown	8	—	8	8

Reservoir sampling slightly shifts counts but keeps proportions correct. CMS at both $\epsilon = 0.01$ and $\epsilon = 0.005$ matches the exact results exactly for all payment types.

5.6. Query 6: Top 10 Most Frequent Tip Amounts

tip_amount	exact_trip_count	sampled_trip_count	cms_0.01	cms_0.005
1.0	3591476	3536512	3646458	3591695
2.0	3340672	3347476	3343383	3341716

3.0	1177898	1199678	1211547	1179175
2.16	954664	982691	962282	957548
1.96	936245	934145	998331	944954
2.06	933732	921641	971547	937539
2.26	909922	927525	962820	918242
2.36	872635	—	1040674	887802
2.46	860187	862062	880709	861688
1.5	856062	856913	866008	860626

Reservoir sampling preserves the rank of most frequent tips but adds noise. CMS with $\epsilon = 0.01$ slightly inflates counts, while $\epsilon = 0.005$ closely matches exact. At $\epsilon = 0.001$, results matched exactly but execution was slow.

Overall Result

Reservoir Sampling gives reasonable approximations but with some variance, especially for rare items. CMS with $\epsilon = 0.01$ is fast but often overestimates due to collisions. CMS with $\epsilon = 0.005$ greatly improves accuracy with moderate slowdown. With stricter $\epsilon = 0.001$, CMS matches exact counts but becomes very slow and memory-heavy. Overall, Reservoir is lightweight, while CMS offers tunable trade-offs between speed and accuracy, making it the better choice for heavy-hitter analysis.

6. References

- Vitter, J. S. (1985). Random sampling with a reservoir. *ACM Transactions on Mathematical Software*.
- Professor's lecture on Algorithm R
- NYC Yellow Taxi 2023 records.
- NYC Yellow Taxi 2022 records.
- NYC Yellow Taxi 2021 records.
- Spark's CountMinSketch Documentation.