Team 13: Incremental Data Processing for NYC Taxi Ride Analytics

Lubin Ye, Meriç Bulca, Laxman Laxman, Yusa Bababemez, Emad Mshalwat

ABSTRACT

Modern data-intensive applications increasingly require low-latency analytics over continuously evolving datasets, yet traditional pipelines rely on costly full recomputation when data changes. This raises the open question of whether incremental data processing can provide meaningful performance advantages over full recomputation for large-scale, dynamic analytical workloads.

In this project, we investigate this question by designing and implementing an end-to-end incremental data processing pipeline for real-time urban mobility analytics, using New York City taxi trip records as a representative case study. The system captures fine-grained data changes and incrementally maintains analytical aggregates through a streaming architecture, avoiding full recomputation. Our evaluation against a full recomputation baseline shows that incremental processing achieves orders-of-magnitude improvements in latency and CPU efficiency at scale, while remaining robust to increasing update sizes. These results demonstrate that incremental data processing provides a scalable and resource-efficient alternative to full recomputation analytics for continuously evolving datasets.

1 INTRODUCTION

Urban mobility datasets are growing rapidly in both size and velocity, driven by the widespread adoption of digital transportation platforms and sensor-based data collection [4]. City-scale taxi trip records, such as those published by the New York City Taxi and Limousine Commission (TLC) [1], are continuously updated as new trips are completed, corrected, or canceled. Analyzing such data in near real-time enables valuable insights into traffic patterns, demand fluctuations, and revenue distribution across different regions of a city.

However, most existing analytical pipelines rely on full recomputation data processing paradigms, where metrics are recomputed from scratch whenever new data arrives. While conceptually simple, this approach does not scale well to interactive or near real-time analytics when datasets grow to tens or hundreds of millions of records [2], as it introduces substantial latency and unnecessary resource consumption.

In this project, we investigate the open research question of whether incremental data processing can outperform full recomputation for large-scale, continuously evolving datasets. As a concrete use case, we explore an incremental data processing pipeline approach for real-time NYC taxi ride analytics. Instead of recomputing aggregates over the entire dataset, our system incrementally updates analytical results as data changes occur. We focus on common analytical tasks such as computing trip counts, average fares, and revenue distributions across time windows and geographic regions.

Regarding the evaluation method, we empirically evaluate this pipeline against a traditional full recomputation baseline through a benchmark to assess whether incremental processing provides measurable advantages.

The remainder of this paper is structured as follows. Section 2 formalizes the problem and outlines our approach. Section 3 presents the experimental setup and evaluation methodology. Finally, Section 4 discusses limitations, lessons learned, and directions for future work.

Contributions.

This project makes the following contributions:

- We design and implement an end-to-end incremental data processing pipeline for real-time analytics that supports insert-only workloads without full dataset recomputation.
- We demonstrate improved performance in terms of latency and computational efficiency under realistic workloads by using incremental data processing pipeline.

2 PROBLEM & APPROACH

Problem Statement.

This project investigates whether incremental data processing can serve as an effective alternative to full recomputation for large-scale, dynamic analytical workloads. While traditional data-processing systems recompute analytics from scratch after each data update, incremental processing propagates only fine-grained changes, potentially reducing latency and computational cost. However, there remains limited empirical evaluation of end-to-end incremental pipelines applied to realistic large-scale datasets. The challenge addressed in this work is enabling low-latency analytical queries over continuously evolving NYC yellow taxi trip data, where new ride records arrive over time and analytical aggregates—such as ride counts, average fares, and regional revenue—must remain correct and up to date without full recomputation.

Approach.. To address this challenge, we adopt an incremental data processing architecture that decomposes analytical computations into updateable components. Rather than periodically recomputing aggregates over the entire dataset, the system propagates only the delta changes introduced by newly arriving records through the pipeline. The approach is organized as a layered architecture consisting of four logical components, illustrated in Figures 1–4, which handle data ingestion, change propagation, analytical processing, and visualization. This design enables low-latency responses to data updates while minimizing redundant computation, and its modular structure makes it extensible and well suited for continuously evolving datasets.

We evaluate our approach using real-world NYC taxi trip data, described in detail in Section 3.

Data Cleaning. As part of the data cleaning and preprocessing stage, we apply a SQL-based normalization and validation process to transform raw ingested records into analysis-ready tuples. Numeric attributes, including identifiers, counts, distances, and monetary

values, are safely cast to their expected data types using tolerant casting to prevent pipeline failures caused by malformed inputs. Basic data quality corrections are applied, including replacing negative numeric values with zero and mapping unexpected categorical values to null to maintain schema consistency. Timestamp fields are normalized into standard timestamp formats to ensure temporal consistency across analyses. In addition, Change data capture (CDC) metadata is used to track new record insertions, enabling correct incremental propagation of data through the pipeline. Records with missing primary identifiers or invalid change metadata are filtered out, ensuring that only valid and semantically meaningful updates enter the incremental processing pipeline.

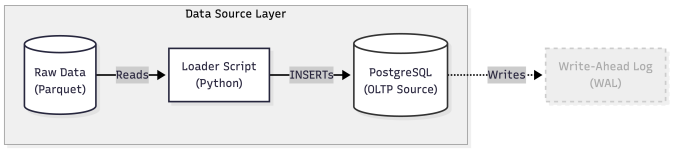

Figure 1: *Data source layer: Taxi trip data is ingested into PostgreSQL, where insert operations generate WAL entries used for downstream change capture.*

Implementation. We implement the proposed pipeline using a combination of open-source data processing tools, following the layered architecture shown in Figures 1–4. Taxi trip data is initially loaded from raw Parquet files into PostgreSQL, which serves as the transactional data source to simulate new incoming real-time data (Figure 1). A Python-based loader script performs insert operations into the database, triggering changes that are durably recorded in PostgreSQL's write-ahead log (WAL).

To track data changes, we employ Debezium to capture row-level events directly from the WAL. As illustrated in Figure 2, Debezium publishes these changes as structured JSON events to an Apache Kafka topic, which acts as the central transport layer. Kafka provides durability, ordering guarantees, and scalable delivery of change events to downstream analytical components.

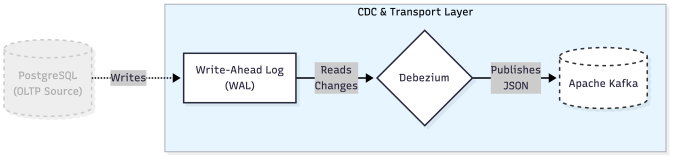

Figure 2: CDC and transport layer: WAL changes are captured by Debezium and published as ordered JSON events to Apache Kafka.

Analytical processing is performed by DuckDB, which serves as the incremental analytical storage engine (Figure 3). DuckDB is used as the analytical database due to its efficient columnar execution and native support for analytical SQL queries. Change events are consumed from Kafka by a Python client (which also serves as the DuckDB client) and applied incrementally to DuckDB using explicit delta semantics: each event encodes the operation type (insert, update, or delete) along with the relevant record state, allowing analytical aggregates to be updated without recomputing them from scratch.

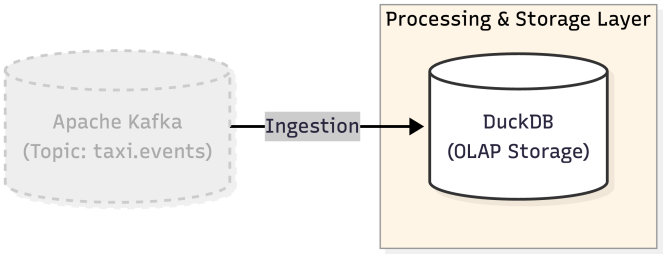

Figure 3: Processing and storage layer: DuckDB ingests change events from Kafka and incrementally maintains analytical aggregates using delta semantics.

Finally, aggregated metrics stored in DuckDB are exposed through a FastAPI-based REST interface and visualized using a Streamlit dashboard, as shown in Figure 4. The dashboard issues analytical queries against the incrementally maintained aggregates and refreshes automatically, enabling users to observe changes in ride volume, average fare, and total revenue as new data arrives.

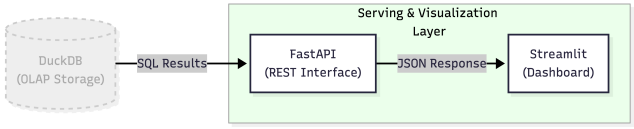

Figure 4: Serving and visualization layer: Aggregates stored in DuckDB are served via FastAPI and visualized through a Streamlit dashboard.

3 EVALUATION

Experimental Setup.

Infrastructure. All experiments were conducted on a constrained Docker environment to emulate a realistic production deployment. The pipeline consists of PostgreSQL 15 as the transactional data store, Apache Kafka (Confluent Platform 7.5.0) as the streaming backbone, Debezium 2.4 for Change Data Capture (CDC), DuckDB (version ≥1.0.0) as the analytical engine, and FastAPI and Streamlit for data serving and visualization. The complete system runs on a single machine with shared resources across all components. Core Python dependencies include Pandas ≥2.0.0, NumPy ≥1.24.0, PyArrow ≥14.0.0, and kafka-python ≥2.0.2. Experiments were run on a MacBook Pro (2021) with an Apple M1 Pro processor, 16 GB RAM, and a 512 GB SSD.

Workload and Methodology. To simulate an operational analytics workload, we generate a stream of data via insert operations on a PostgreSQL dataset. These changes are propagated through the pipeline and reflected in downstream analytical queries.

Each experiment applies the same order of incoming data to both systems and is executed under identical hardware and software conditions to ensure a fair comparison while measuring their performance.

We compare the following two processing strategies:

(1) **Baseline (Batch Full Recomputation):** This approach represents a traditional batch-oriented analytics workflow. Upon the arrival of new data, the system triggers a full recomputation by executing aggregation queries over the entire dataset stored in the analytical engine.
(2) **Incremental Data Processing:** The proposed incremental pipeline that maintains aggregates using delta-based updates. See Section 2.

To empirically demonstrate the advantages of incremental processing over full recomputation, we use the following metrics:

1. Latency. Time (in milliseconds) to compute and materialize all analytical aggregates in DuckDB after the base table has been updated, i.e., aggregation processing time only. This metric does not include base-table write/commit time or any downstream dashboard refresh.

2. Resource Utilization (CPU time usage). To assess the overhead introduced by maintaining incremental state, we measure the CPU time footprint of the aggregation tables in DuckDB as the base dataset grows. This metric evaluates whether the incremental approach incurs additional time costs compared to full recomputation.

3. Delta Scalability (Sensitivity to Update Size). Analysis of how aggregation performance responds to varying update magnitudes by measuring aggregation latency when applying different fractions of the dataset in a single transaction (1%, 3%, 5%, and 10% of rows). This evaluates the robustness of the pipeline as the size of each update grows and serves as a proxy for bursty or high-velocity ingestion scenarios.

Datasets.

We use the 2024 New York City Yellow Taxi trip data published by the NYC Taxi and Limousine Commission [1]. The dataset contains trip-level records including pickup and drop-off times, locations, fares, and payment types, with geographic aggregation based on official NYC borough lookup tables.

Our experiments use a six-month subset of the 2024 data, comprising approximately 20 million trips, stored as monthly Parquet files.

Experiments and Results.

Latency. We first evaluate scalability under a fixed ingestion workload by growing the dataset from 10^4 to approximately 2×10^7 rows and applying a constant batch of 1,000 new records at each scale.

As shown in Figure 5, the full recomputation baseline exhibits linear latency growth with dataset size, reflecting its $O(N)$ cost. In contrast, the incremental pipeline maintains near-constant latency, as it processes only the incoming deltas. At 20 million rows, incremental processing achieves a latency of 15.8,ms compared to approximately 711,ms for full recomputation, corresponding to a 45× speedup.

Delta Sensitivity (Sensitivity to Update). To assess robustness under bursty workloads, we vary the update size as a fraction

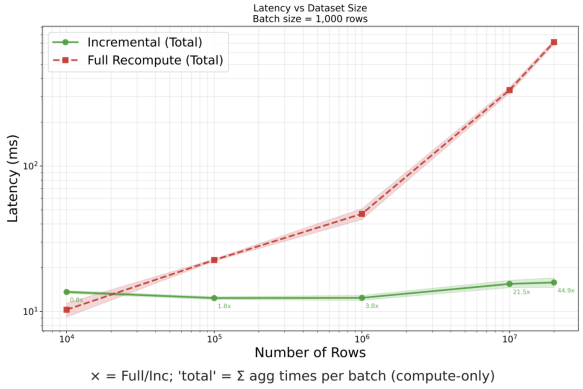

Figure 5: Latency vs. dataset size with a fixed 1,000-row batch. Incremental processing remains flat, while full recomputation scales linearly.

of the dataset ($\delta \in 1\%, 3\%, 5\%, 10\%$). Figure 6 shows that full recomputation remains dominated by dataset size, incurring high latency regardless of update magnitude.

The incremental pipeline exhibits a modest increase in latency as δ grows, which is expected since larger fractions correspond to more records being processed. However, even at 20 million rows, incremental latency remains between 35.1,ms ($\delta = 1\%$) and 40.1,ms ($\delta = 10\%$), compared to 786.7,ms and 930.1,ms for full recomputation, yielding 22–23× speedups. The tight clustering of incremental curves indicates strong robustness to increasing update intensity.

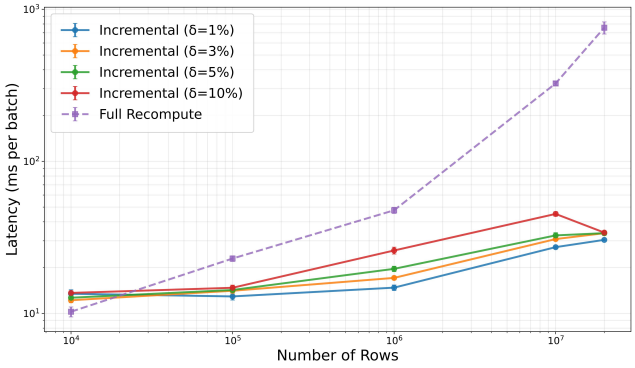

Figure 6: Latency vs. dataset size for varying update fractions δ. Incremental processing remains significantly faster across all update magnitudes.

Resource Utilization (CPU time). We isolate computational cost by measuring CPU time during aggregation under the same fixed ingestion workload. As shown in Figure 7, CPU usage for full recomputation grows rapidly with dataset size, while incremental processing remains constant.

At 20 million rows, incremental aggregation requires approximately 30,ms of CPU time, compared to 3,122,ms for full recomputation, corresponding to a reduction of over 100×. This confirms

that the incremental pipeline scales with the update size ($O(\Delta)$) rather than total dataset size.

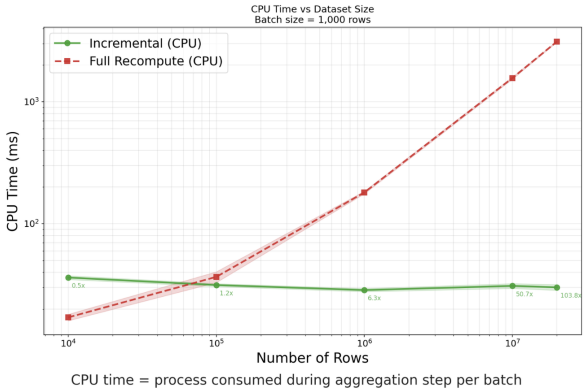

Figure 7: CPU time vs. dataset size. Incremental processing achieves over 100× lower CPU cost at scale.

Overall, these results show that incremental processing eliminates the scalability bottlenecks of batch-oriented analytics. It delivers near-constant latency (at 20 million rows, incremental updates achieve up to 45× lower latency), remains robust under large update fractions (maintain 22–23× speedups across varying update fractions), and achieves these gains with drastically lower CPU cost (reduce CPU usage by over 100× compared to full recomputation), making it well suited for continuously evolving analytical workloads.

4 DISCUSSION AND CONCLUSION

In this project, we demonstrated the feasibility and effectiveness of incremental data processing for low-latency analytics over batch-oriented full recomputation mechanisms. By propagating fine-grained data changes through a streaming pipeline, our system avoids costly full recomputation and enables timely analytical insights over large-scale urban mobility data. Using New York City taxi trips as a real-world case study, we show that incremental updates maintain near-constant latency, achieving 45× lower end-to-end latency and >100× reduction in CPU usage compared to full recomputation at 20 million rows.

While the incremental approach substantially improves responsiveness, it also introduces additional system complexity, particularly in the correct handling of updates and deletions. Preserving aggregate correctness requires careful bookkeeping and explicit delta semantics, increasing implementation effort compared to batch-oriented pipelines. Furthermore, our current prototype focuses on a restricted set of aggregate queries, and extending the approach to more complex analytical workloads (e.g., joins or higher-order aggregates) remains an open challenge.

Future work could investigate tighter integration with advanced stream processing frameworks, improved fault tolerance, and scaling the pipeline to support multiple concurrent data sources. Evaluating incremental techniques on such datasets could further validate their practical relevance. Finally, incorporating predictive or learning-based analytics on top of incrementally maintained state represents a promising avenue for continued research.

Detailed Contributions.

This project utilizes the CRediT (Contributor Roles Taxonomy) [3] to acknowledge individual contributions to the research and development process.

- **Lubin Ye**: Contributed to *Methodology* by leading the overall methodological design and coordinating the research workflow (Sections 2, 1). Contributed to *Software*, *Writing – Original Draft*, and *Presentation* by designing the data processing pipeline, structuring the written report, and preparing the final presentation (Sections 3, 4).
- **Meriç Bulca**: Contributed to *Software* through implementation of the data processing pipeline and benchmarking experiments (Section 3). Contributed to *Visualization* and *Validation* by creating benchmark plots and assisting with performance evaluation (Sections 3, 4).
- **Laxman Laxman**: Contributed to *Software* by implementing core components for data processing and benchmarking (Section 3). Contributed to *Data Curation* and *Validation* through data cleaning, preprocessing, and assistance with benchmark evaluation and debugging (Sections 2, 4).
- **Yusa Bababemez**: Contributed to *Data Curation* by collecting, organizing, and preparing datasets for analysis (Section 2). Contributed to *Presentation* by supporting dataset documentation and preparation of presentation materials (Section 4).
- **Emad Mshalwat**: Contributed to *Software* and *Visualization* through the design and implementation of the project dashboard (Section 3). Contributed to *Presentation* by assisting with presentation preparation and communicating technical results clearly (Section 4).

PROJECT RESOURCES

GitHub Repository: UVA Yellow Taxi Project Repository

NYC Yellow Taxi Dashboard Recording : Demo Link

REFERENCES

[1] New York City Taxi and Limousine Commission [n.d.]. *NYC Taxi and Limousine Commission Trip Record Data.* New York City Taxi and Limousine Commission. https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page Accessed: 2026-01-17.
[2] Martin Kleppmann. 2017. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.* " O'Reilly Media, Inc.".
[3] NISO. 2024. CRediT (Contributor Roles Taxonomy). https://credit.niso.org/
[4] Guozhang Wang. 2021. Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing. *IEEE Data Eng. Bull.* 44, 2 (2021), 1–12.