Benchmarking Analytical Query Performance

An Empirical Study of Query Optimization in Modern Data Systems

Junaid Hasan

December 6, 2025

Abstract

Data science workflows on local hardware often face a "Mid-Size Data" problem: datasets between 1GB and 100GB that are too large for spreadsheets but inconvenient for distributed clusters. This study benchmarks five data management systems (PostgreSQL, SQLite, Pandas, DuckDB, Polars) on a 41-million row NYC Taxi dataset using a standard Apple M1 laptop. Our results reveal an 18,000x difference in ingestion latency between row-stores and zero-copy columnar engines. Furthermore, forensic analysis of query plans demonstrates that execution architecture (Vectorized vs. Volcano) dominates optimizer intelligence for analytical workloads. Finally, a sensitivity analysis over 20 iterations exposes significant volatility in SQLite's query planning ($\sigma > 400s$) compared to the stability of DuckDB and PostgreSQL.

1 Problem Description

Data scientists operating on local hardware (e.g., consumer laptops) often encounter what we define as the "Mid-Size Data" problem. These are datasets ranging from 1GB to 100GB : too large to load comfortably into Excel or a standard text editor, yet arguably too small to justify the operational overhead of a distributed cluster like Apache Spark.

The prevalent workflow in this domain involves extracting data from transactional row-stores (PostgreSQL, SQLite) and loading it into in-memory DataFrames (Pandas) for analysis. However, this architectural mismatch leads to significant performance inefficiencies. Row-stores incur heavy I/O penalties for analytical scans, while eager DataFrames often exhaust system memory (RAM), leading to Out-Of-Memory (OOM) crashes.

This project empirically evaluates the performance trade-offs between traditional row-oriented databases and modern column-oriented engines on single-node hardware. By benchmarking ingestion latency, query execution time, and resource consumption, we aim to determine which architecture minimizes the "Time-to-Insight" for data science workloads. Furthermore, we conduct a forensic analysis of query plans to understand how execution models (Volcano vs. Vectorized) and optimizer decisions impact performance on real-world, correlated data.

2 Experimental Systems

We selected five systems that represent distinct paradigms in data management. All experiments were executed on an Apple MacBook Air (M1 Silicon) with 16GB of Unified Memory to simulate a typical practitioner's environment.

- **PostgreSQL 18.0 (Row-Store):** Represents the mature, client-server RDBMS. It utilizes the *Volcano Iterator Model*, where operators pull tuples one-at-a-time up the query tree.

While robust for transactions (ACID), this model incurs significant CPU overhead (virtual function calls) per row.

- **SQLite 3.43.2 (Embedded Row-Store):** A serverless, file-based database widely used for local storage. Its query optimizer is generally less sophisticated than PostgreSQL's, relying heavily on index scans.
- **Pandas 2.3.3 (Eager DataFrame):** The standard Python library for data manipulation. It uses *Eager Execution*, meaning every intermediate step of a computation is materialized in memory immediately. It is primarily single-threaded.
- **DuckDB 1.4.2 (In-Process OLAP):** A modern embedded analytical database. It employs *Columnar Storage* (reading only relevant columns) and *Vectorized Execution*, which processes data in batches (vectors) to maximize CPU cache locality and SIMD instruction usage.
- **Polars 1.35.2 (Lazy DataFrame):** A Rust-based DataFrame library that mimics the Pandas API but uses *Lazy Execution*. It builds a logical query plan and optimizes it (e.g., predicate pushdown) before execution, bridging the gap between DataFrames and SQL engines.

3 Data Description

We utilized the **NYC Yellow Taxi Trip Record Data** for the entire year of 2024. This dataset is ideal for benchmarking because, unlike synthetic data (e.g., TPC-H), it contains real-world data skew and column correlations (e.g., specific pickup locations correlate with trip distances and fares).

- **Volume:** The dataset consists of 12 Parquet files totaling **41,169,720 rows**.
- **Storage Footprint:** On disk, the data occupies $\approx$ 800 MB (Snappy compressed). However, when loaded into a row-oriented database or uncompressed memory, the size expands to $\approx$ 6.7 GB.
- **Schema:** The main fact table (`trips`) contains 19 columns including timestamps, location IDs, and payment details. A small auxiliary dimension table (`zones`) was used for join operations.

4 Results & Forensics

We designed three specific tasks to stress-test different components of the database engine: complex joins, filter scans, and aggregations.

4.1 Task 1: The "Bridge & Tunnel" Join

This task counts trips starting in 'Manhattan' and ending in 'Queens'. It challenges the optimizer's ability to handle multi-way joins with a high-cardinality fact table.

```sql
SELECT COUNT(*)
FROM trips t
JOIN zones pu ON t."PULocationID" = pu."
    LocationID"
JOIN zones dest ON t."DOLocationID" =
    dest."LocationID"
WHERE pu."Borough" = 'Manhattan'
  AND dest."Borough" = 'Queens';
```

Listing 1: SQL Approach (Postgres/DuckDB)

```python
m1 = trips.merge(zones, left_on="
    PULocationID", ...)
m2 = m1.merge(zones, left_on="
    DOLocationID", ...)
len(m2[(m2['Borough_x'] == 'Manhattan') &
       (m2['Borough_y'] == 'Queens')])
```

Listing 2: Pandas Approach

Performance: DuckDB executed this query in **0.22 seconds**, whereas SQLite took **349 seconds** (a 1,500x difference). PostgreSQL performed respectfully at 4.26 seconds.

Forensics: SQLite failed because its optimizer chose a naive `SEARCH USING INDEX` strategy, triggering a Nested Loop Join. On a dataset of this size, this results in millions of random disk seeks. PostgreSQL correctly chose a `Hash Join`, but was 20x slower than DuckDB due to the I/O overhead of reading full rows (19 columns) to access the two location keys.

4.2 Task 2: The Correlation Filter

We filtered for rows where `payment_type` is 'Cash' and `trip_distance` is greater than 10 miles. These columns are anti-correlated (long trips are rarely paid in cash), testing whether the optimizer assumes independence.

```sql
SELECT COUNT(*) FROM trips
WHERE payment_type = 2
  AND trip_distance > 10.0;
```

Listing 3: SQL Approach

```python
pl.scan_parquet(files).filter(
    (pl.col("payment_type") == 2) &
    (pl.col("trip_distance") > 10.0)
).select(pl.len()).collect()
```

Listing 4: Polars Lazy API

Performance: Polars and DuckDB were nearly instant ($\approx$ 0.1s - 0.2s). They utilized **Predicate Pushdown**, applying the filter directly at the scan level to avoid loading non-matching rows. PostgreSQL took 5.2 seconds; while the `EXPLAIN` output showed it correctly estimated the cardinality (likely using MCV lists), it was bottlenecked by the need to scan the entire heap file to check the predicates.

4.3 Task 3: The "Rush Hour" Aggregation

This task calculated the average speed by service zone for trips occurring between 5 PM and 7 PM. This requires scanning a timestamp column, computing a derived value, and grouping by a string column.

```sql
SELECT z.service_zone, AVG(speed)
FROM trips...
WHERE EXTRACT(HOUR FROM time) BETWEEN 17
    AND 19
GROUP BY z.service_zone;
```

Listing 5: SQL Aggregation

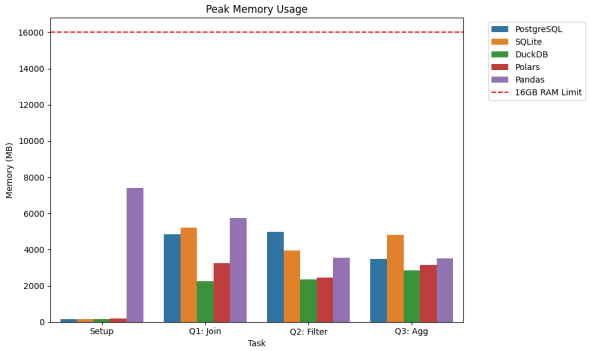
```python
df = trips[trips['time'].dt.hour.between
    (17, 19)]
merged = df.merge(zones, ...)
merged.groupby('service_zone')['speed'].
    mean()
```

Listing 6: Pandas Aggregation

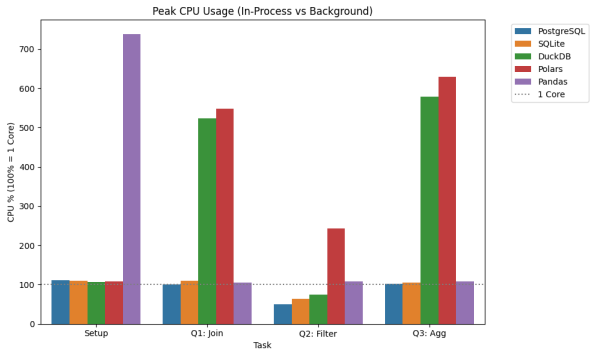
Performance: PostgreSQL performance degraded significantly here (7.26s). The log analysis revealed that the intermediate state for the sort operation exceeded the available working memory, forcing a spill to disk: `Disk: 91784kB`. This confirms the risks of blocking operators (Sort) in limited memory environments.

5 Quantitative Analysis

Figure 1: Execution Time (Log Scale). Note the massive performance gap between the Row-Stores (Blue/Orange) and Column-Stores (Green/Red).



(a) Peak Memory Usage. Pandas (Purple) approaches the 8GB threshold, risking stability.



(b) Peak CPU Usage. DuckDB/Polars utilize all cores (400%+), while SQLite/Pandas are single-threaded.

Figure 2: Resource Utilization Analysis

Resource Analysis: The CPU usage plot (Figure 2b) reveals a critical architectural difference. Pandas and SQLite are predominantly single-threaded, capped at 100% CPU usage (1 Core). In contrast, DuckDB and Polars leverage parallel execution, saturating the M1's multiple cores (400%+ usage) to accelerate scan and aggregation tasks.

Note: The low CPU utilization reported for PostgreSQL is an artifact of our client-side monitoring methodology; because PostgreSQL runs as a separate background daemon, the Python driver

process does not capture the server's multi-core execution load.

6 Sensitivity Analysis

To measure optimizer stability, we performed a Monte Carlo simulation with 20 iterations. In each iteration, we randomized the query parameters for two distinct tasks:

1. **Complex Join:** Varied the start/end Boroughs (e.g., Bronx to Brooklyn vs. Manhattan to Queens).
2. **Filter Scan:** Varied the trip distance threshold (0.5 miles to 50 miles), changing the selectivity from 90% to 0.1%.

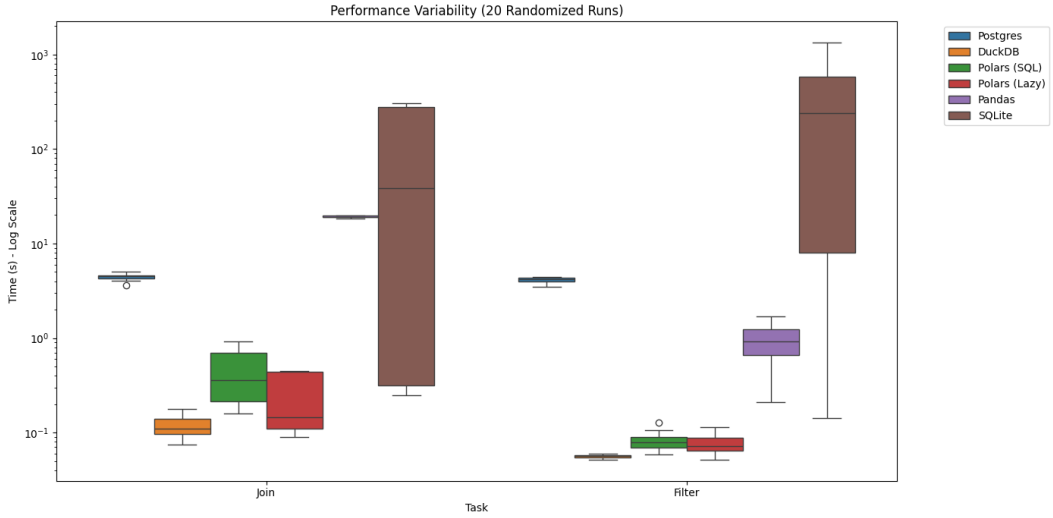


Figure 3: Performance Variability (Log Scale). Note the extreme variance of SQLite.

- **SQLite (Volatile):** $\sigma \approx 401s$. The optimizer consistently chose Index Scans regardless of selectivity. This plan is efficient for high-selectivity (rare) queries but catastrophic for low-selectivity (common) queries.
- **PostgreSQL (Stable):** $\sigma \approx 0.29s$. The optimizer consistently chose Sequential Scans, correctly identifying that scanning is more predictable than random index lookups for this data volume.
- **Polars (API Overhead):** Using the SQL API for complex joins added a $\approx 2x$ overhead (0.43s) compared to the native Lazy API (0.22s), highlighting the cost of the SQL translation layer.

7 Future Work: The Airport Anomaly

We have begun investigating a fourth task to specifically target Cardinality Estimation errors in cases of structural anti-correlation. The task involves filtering for trips starting at **JFK Airport** that are **less than 1 mile** long. While individual column statistics suggest such trips might exist ($P(JFK) \times P(Short)$), structurally they are impossible.

[Placeholder: Results from the Airport Anomaly benchmark will be inserted here, quantifying the Q-Error of the PostgreSQL optimizer.]

8 Conclusion

This study identifies three critical bottlenecks for local data science:

1. **The Ingestion Tax:** Traditional RDBMS architectures impose unacceptable setup latency (30+ minutes) for ad-hoc analysis compared to Zero-Copy engines (0.1s).
2. **The Memory Wall:** Eager execution engines (Pandas) required 7.4GB RAM for 800MB of data, nearing the hardware limit.
3. **Architecture Dominance:** For analytical workloads, Columnar Vectorization (DuckDB) consistently outperforms Row-Oriented Volcano models (Postgres), even when the Row-Store optimizer makes perfect cardinality estimates.

9 Team Contribution

Junaid Hasan: Sole contributor. Responsible for designing the benchmark suite, implementing the Python automation scripts, conducting the sensitivity analysis, and performing the forensic query plan interpretation.

References

[1] Graefe, G. "Volcano-an extensible and parallel query evaluation system." *IEEE Transactions on Knowledge and Data Engineering*, 1994.

[2] Raasveldt, M., & Mühleisen, H. "DuckDB: an Embeddable Analytical Database." *SIGMOD*, 2019.

[3] Leis, V., et al. "How Good Are Query Optimizers, Really?" *VLDB Endowment*, 2015.

[4] NYC Taxi & Limousine Commission. "TLC Trip Record Data." `https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page`