Benchmarking Analytical Query Performance

An Empirical Study of Execution Models and Optimizer Stability on a modern laptop

Junaid Hasan

University of Washington
CSE 544: Principles of DBMS

December 3, 2025

Agenda

Context: The “Mid-Size” Data Problem

The Scenario

- Data Science workflows are frequently executed on local hardware before cloud deployment.
- Datasets often fall into the “Awkward Size”: Too large for Excel, yet small enough to theoretically fit on a laptop SSD (1GB – 100GB).

The Challenge

- Practitioners must choose between mature RDBMS (Postgres) and modern Columnar engines (DuckDB).
- Performance on local hardware (e.g., Apple M1) is often constrained by RAM and I/O rather than CPU.

Research Questions

This study empirically investigates:

1. **Setup Overhead:** What is the cost of ETL into a Heap File versus Zero-Copy ingestion?
2. **Execution Models:** How does the **Volcano Iterator Model** (Row-Store) compare to **Vectorized Processing** (Column-Store) on modern hardware?
3. **Optimizer Stability:** How do optimizers handle correlated predicates where the **Independence Assumption** fails?

Hardware Data

Hardware Environment:

- **System:** Apple MacBook Air (M1 Silicon)
- **Memory:** 16 GB Unified Memory (Shared)
- **Storage:** NVMe SSD

Dataset: NYC Yellow Taxi Trip Records (2024)

- **Period:** Jan 2024 – Dec 2024 (12 Months)
- **Row Count:** 41,169,720 rows
- **Storage:** $\approx$ 800 MB (Parquet/Snappy) $\rightarrow$ $\approx$ **6.7 GB** (Uncompressed DB)

Experimental Systems

System	**Version**	**Paradigm**
PostgreSQL	18.0	Client-Server Row-Store (Volcano)
SQLite	3.43.2	Embedded Row-Store
Pandas	2.3.3	In-Memory DataFrame (Eager)
DuckDB	1.4.2	In-Process Column-Store (Vectorized)
Polars	1.35.2	DataFrame Library (Lazy)

The “Setup Tax” (Results)

We measured the wall-clock time required to load the 41M row dataset from Parquet into a queryable state.

System	Time (Seconds)	Analysis
PostgreSQL	**2,008.00 s**	33.5 Minutes (ETL Overhead)
SQLite	**211.00 s**	3.5 Minutes
Pandas	2.60 s	Transient Load (RAM only)
DuckDB	**0.11 s**	Zero-Copy View
Polars	**0.11 s**	Zero-Copy Scan

Analysis: Ingestion

Why the 18,000x difference?

- **Row Stores (Postgres):** Must parse, deserialize, and rewrite data into Heap Files and Write-Ahead Logs (WAL) for ACID compliance.
- **Column Stores (DuckDB):** Read the Parquet metadata footer and query the file in-place ("Zero-Copy").

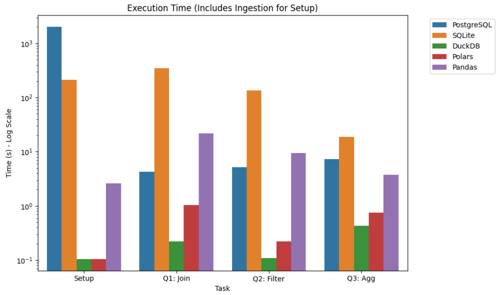
Implication: For ad-hoc analytics, Postgres imposes a 30-minute penalty before the first query can be run.

Task 1: The “Bridge & Tunnel” Join

Query: Count trips starting in 'Manhattan' and ending in 'Queens'.

- Requires **two joins** to the lookup table.
- Tests execution engine efficiency on large scans.

Results (Time in Seconds):



- **DuckDB:** 0.22 s | **Postgres:** 4.26 s | **SQLite:** 349.39 s

Task 1 Implementation Details

SQL Query (Postgres / DuckDB / Polars 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'
```

Pandas Query (Eager Execution):

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

Note: Pandas materializes intermediate tables ‘m1‘ and ‘m2‘, increasing memory pressure.

Task 1 Forensics: SQLite

The Investigation:

```
EXPLAIN QUERY PLAN
SELECT COUNT(*) FROM trips ... -- (Join Query)
```

The Verdict (Naive Execution):

```
> SEARCH trips USING INDEX idx_pu ...
```

Analysis:

- The optimizer saw an index and blindly used it.
- For a Join touching 10%+ of rows, a **Sequential Scan** (or Hash Join) is usually faster due to sequential I/O.
- SQLite failed to account for the high selectivity, triggering millions of random disk seeks (Nested Loop).

Task 1 Forensics: Architecture & Execution

Why was Postgres 20x slower than DuckDB? Postgres correctly used a `Hash Join`, but failed on architecture.

1. **The I/O Bottleneck:**
   - Postgres is a Row-Store. Accessing specific attributes requires fetching the entire tuple (PAX/Blob storage not utilized).
2. **The Execution Bottleneck (Lecture 13):**
   - Postgres uses the **Volcano Iterator Model** (Pull).
   - Every tuple requires an ‘open() / next()‘ virtual function call.
   - **DuckDB** uses **Vectorized Execution**, processing batches (e.g., 1024 tuples) per call, amortizing instruction cache overhead.

Task 2: The Correlation Trap

Query: Count rows where `payment_type=2` (Cash) AND `trip_distance > 10`.

- These attributes are negatively correlated.
- We test if the optimizer assumes independence: $P(A \wedge B) = P(A)P(B)$.

Results (Time in Seconds):

- **DuckDB:** 0.11 s
- **Polars:** 0.22 s
- **Postgres:** 5.21 s
- **SQLite:** 135.97 s

Task 2 Implementation Details

SQL Query (Postgres / DuckDB):

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

Polars (Lazy API):

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

Task 2 Forensics: Independence Assumption

The Result:

- Postgres Estimated: 169,508 rows.
- Actual: 163,189 rows.

Analysis (Lecture 15):

- Postgres **avoided** the Independence Assumption trap.
- It likely utilized Multi-Column Statistics or MCV (Most Common Value) lists to model the correlation accurately.
- However, execution time (5.2s) was still dominated by the Row-Store I/O penalty.

Task 3: Aggregation & Blocking Operators

Query: Average speed by Zone for “Rush Hour” trips (5-7 PM).

The “Smoking Gun”: Postgres performance degraded (7.26s). The logs revealed why:

```
Disk: 91784kB
```

Results (Time): DuckDB 0.43s vs Postgres 7.26s.

Task 3 Implementation Details

SQL Query (Postgres / DuckDB):

```
SELECT z.service_zone, AVG(trip_distance)
FROM trips t JOIN zones z ON t."PULocationID" = z."LocationID"
WHERE EXTRACT(HOUR FROM tpep_pickup_datetime) BETWEEN 17 AND 19
GROUP BY z.service_zone
```

Pandas Query:

```
df_filt = trips[trips['tpep_pickup_datetime'].dt.hour.between(17, 19)]
merged = df_filt.merge(zones, left_on="PULocationID", ...)
merged.groupby('service_zone')['trip_distance'].mean()
```

Task 3 Forensics: Blocking Operators

The Investigation:

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT ... GROUP BY service_zone;
```

The "Smoking Gun":

```
-> Sort Method: external merge  Disk: 91784kB
```

Analysis:

- The `Sort` operator is **Blocking** (Lecture 13).
- It must consume all inputs before producing outputs.
- The state size (91 MB) exceeded `work_mem`, forcing the OS to swap pages to the physical SSD.

Methodology: Monte Carlo Sensitivity Test

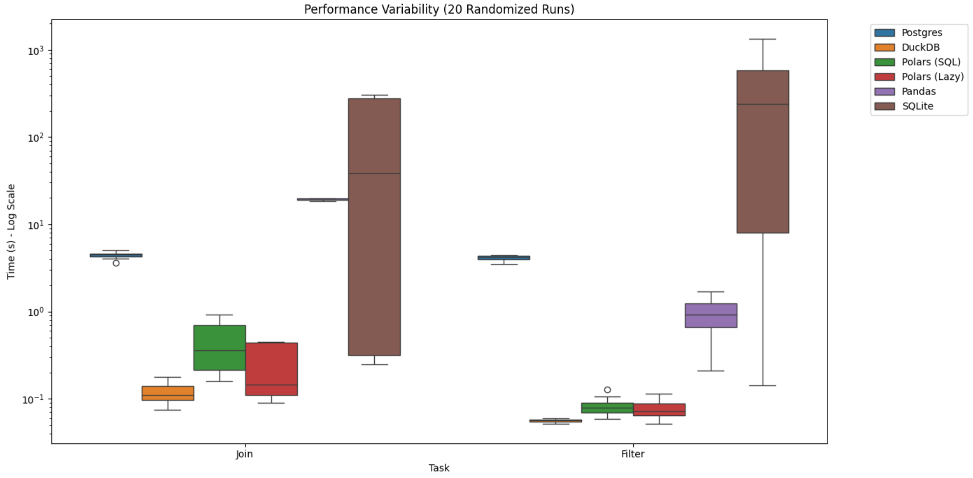
A single benchmark run can be misleading due to caching or “lucky” parameters. We performed a **Sensitivity Analysis** to measure stability.

The Algorithm (n=20 Iterations):

1. **Randomize:** Sample parameters P from a distribution.
   - *Distance:* $\{0.5, 1.5, \ldots, 50.0\}$ miles.
   - *Borough:* $\{Manhattan, Queens, \ldots\}$.
2. **Execute:** Run query $Q(P)$ on all 5 systems.
3. **Measure:** Record Wall-Clock Time.
4. **Trace:** Capture the Query Plan (Index Scan vs. Seq Scan).

Goal: Determine if variance (σ) is driven by data distribution or optimizer instability.

Results: Stability vs. Volatility



Sensitivity Findings: SQLite

The Volatility Problem

- **Mean Time:** 361 s
- **Standard Deviation:** 401 s

Interpretation:

- SQLite consistently chose the **Index Search** plan (20/20 times).
- This was disastrous for low-selectivity queries (short distances) but fast for high-selectivity ones.
- The optimizer failed to adapt to data distribution, making performance unpredictable.

Sensitivity Findings: PostgreSQL

The Consistency Story

- **Mean Time:** 4.1 s
- **Standard Deviation:** 0.29 s

Interpretation:

- Postgres chose **Seq Scan** 20/20 times.
- It correctly identified that for this dataset size, a parallel scan is safer and more predictable than a random index lookup. It sacrificed peak speed for stability.

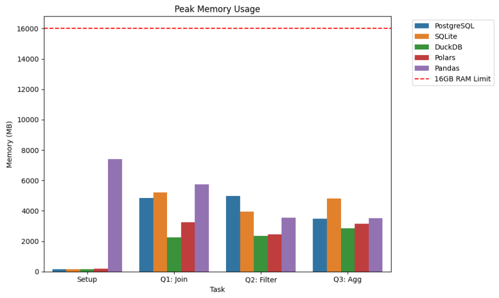
Sensitivity Findings: Polars

API Overhead: SQL vs. Native We benchmarked both the SQL and Lazy DataFrame APIs.

Task	**Polars (Native)**	**Polars (SQL)**
Filter Scan	0.07 s	0.08 s
Complex Join	**0.22 s**	0.43 s

Insight: For complex operations, the SQL translation layer in Polars adds a $\approx 2x$ overhead compared to the optimized Native API.

The Memory Wall



- Pandas required **7.4 GB** of RAM to load the 2024 dataset.
- This represents 50% of the system memory for just 1 year of data.
- Eager execution models are fragile on consumer hardware; Streaming models (DuckDB/Polars) are essential for scalability.

Summary Recommendations

1. **Avoid SQLite for Analytics:** High volatility ($\sigma > \mu$) makes it unsuitable for interactive data science.
2. **Postgres is Reliable but Heavy:** 33-minute setup and row-store I/O limits speed, even with a robust optimizer.
3. **DuckDB/Polars are the Standard:** Sub-second performance, zero setup, and predictable stability make them the clear choice for local analytics.

Thank You

`https://github.com/junaid-hasan/database-benchmark`