Optimizing Analytics Storage Strategies for Search Engines and Wiki Platforms

Master's Defense
by **Sujith Kakarlapudi**

Advisor : **Dr. Chris Pollett**
Committee : **Dr. Navrati Saxena**
Committee : **Dr. Thomas Austin**

Outline

1. Introduction
2. Background
3. Implementation
4. Comparative Analysis
5. Results
6. Conclusion

Problem Statement

- ➔ Small to medium scale search engines like DocFetcher, MediaWiki rely on SQLite for Analytics.
- ➔ Analytics events such as page views, edits and clicks are directly inserted into SQLite tables.
- ➔ Under heavy write loads, this leads to random I/O operations, journal overhead, and lock contention.
- ➔ Consequently, ingestion throughput stalls and storage requirements increases, delaying real-time analytics.

Objectives

- ➔ Integrate an append-only, size-rotated log storage mechanism into Yioop.
- ➔ Extend the aggregation routine to consume those log partitions and populate summary tables.
- ➔ Benchmark and compare the log-based and SQL-based pipelines on write throughput, storage footprint, aggregation latency, and query performance.

Introduction

- Relational Analytics Storage
  - Uses B-tree indices, rollback journals (e.g., SQLite) or MVCC (e.g., PostgreSQL)
  - Suffers random I/O and lock contention under heavy writes
- Log-Based Storage
  - Appends events sequentially to files
  - Enables partitioning by time or size for parallelism and bounded reads
- Key Trade-Offs
  - Write performance & storage efficiency vs. query flexibility

SQlite

Rollback-Journal & B-Tree Storage

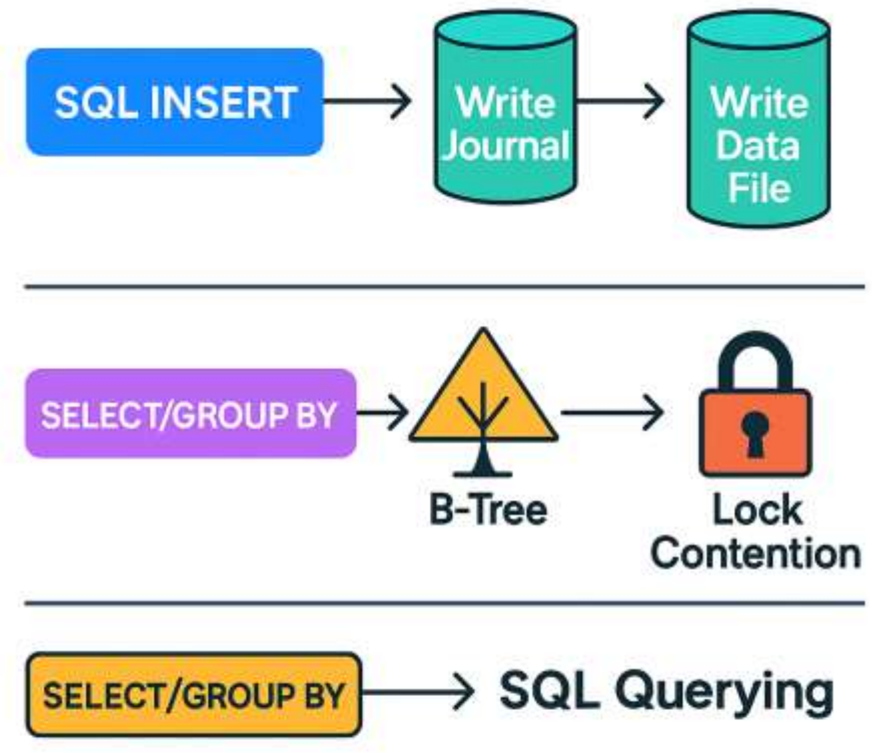
- Two writes per insert (DB file + journal)
- Tables/indexes in B-trees ⇒ random I/O

SQL Querying

- SELECT/GROUP BY traverse B-trees in native code
- Reads can block on write locks

Key Challenges

- Lock contention under heavy writes
- Growing storage footprint



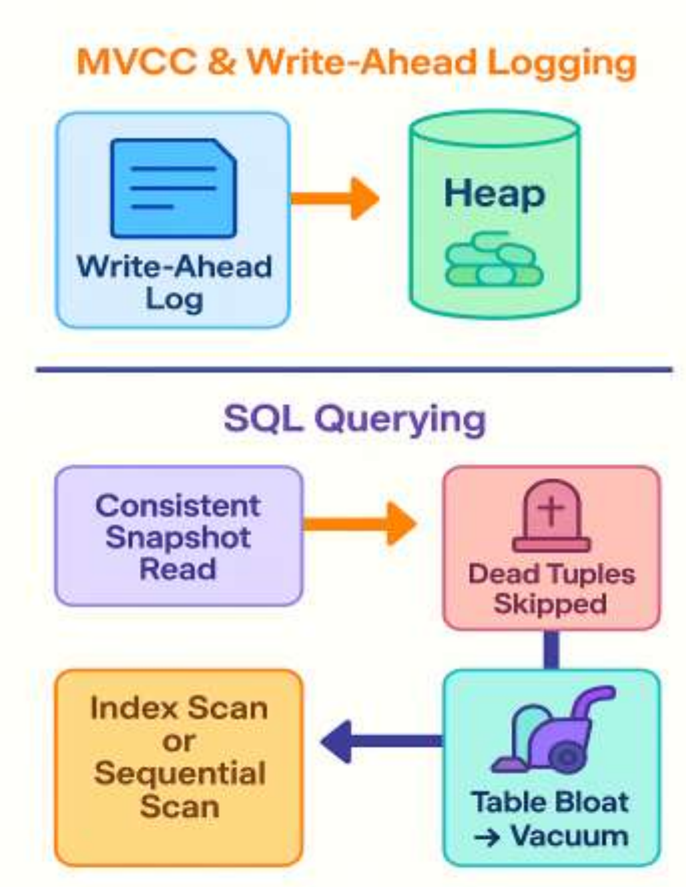
PostgreSQL

MVCC & Write-Ahead Logging

- Writes first append to the WAL, then data files on checkpoint
- Heap stores multiversion tuples; B-tree (and other) indexes maintain pointers
- Queries use consistent-snapshot reads via MVCC
- Index scans and sequential scans skip dead tuples

Key Challenges

- Table/index bloat requiring frequent vacuuming
- Checkpoint-driven I/O spikes and write amplification



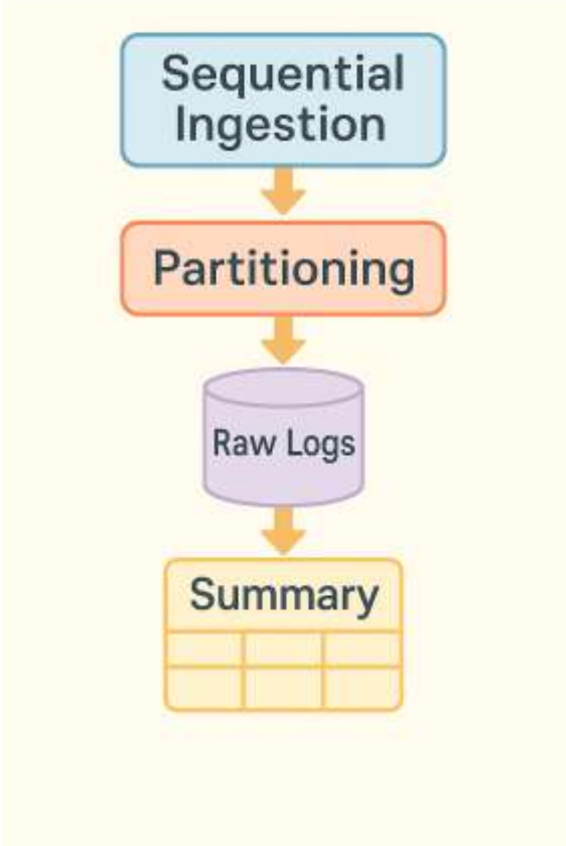
Log Based

Sequential Ingestion & Partitioning

- Append-only writes eliminate random seeks and lock contention
- Rotate logs by time or size to keep file sizes bounded and enable parallel reads

Compaction & Querying

- Background roll-ups merge partitions into summary tables for sub-second lookups
- Raw log scans support deep forensic analysis alongside real-time dashboards



Implementation - Log Redirection & Append Pipeline

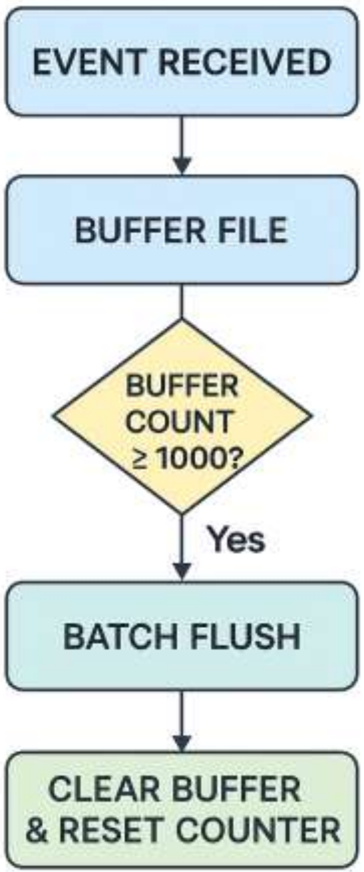
Purpose:

Batch and buffer incoming analytics events into an append-only log to minimize per-event I/O and lock contention.

Key Steps:

- Buffer each event as a CSV line in impression_buffer.log
- On 1,000 buffered events, load them and call PartitionDocumentBundle::put($rows)
- Clear the buffer file and reset the counter

Result: Thousands of individual writes collapse into a single, high-throughput log append per batch, dramatically reducing disk seeks.



PartitionDocumentBundle – Creation & Rotation

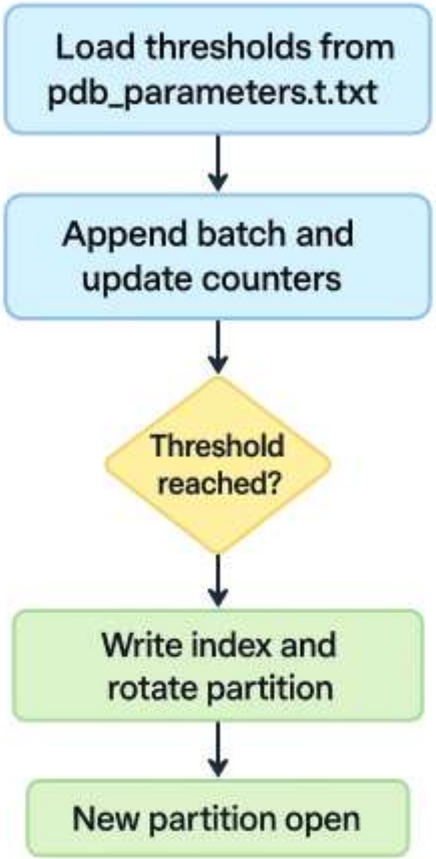
Purpose:

Keep log files bounded in size and rows, with a fast in-RAM index for reads.

Key Steps:

- Load maxRows & maxBytes thresholds from pdb_parameters.txt.
- After each batch append, update row/byte counters
- When a threshold is reached:
    - Atomically write the in-RAM index to disk.
    - Increment the partition number and open a new log file.

Result: Predictable, bounded partitions with constant-memory indexing and atomic rotation to avoid half-written or corrupted logs.



PackedTableTools – Binary Encoding

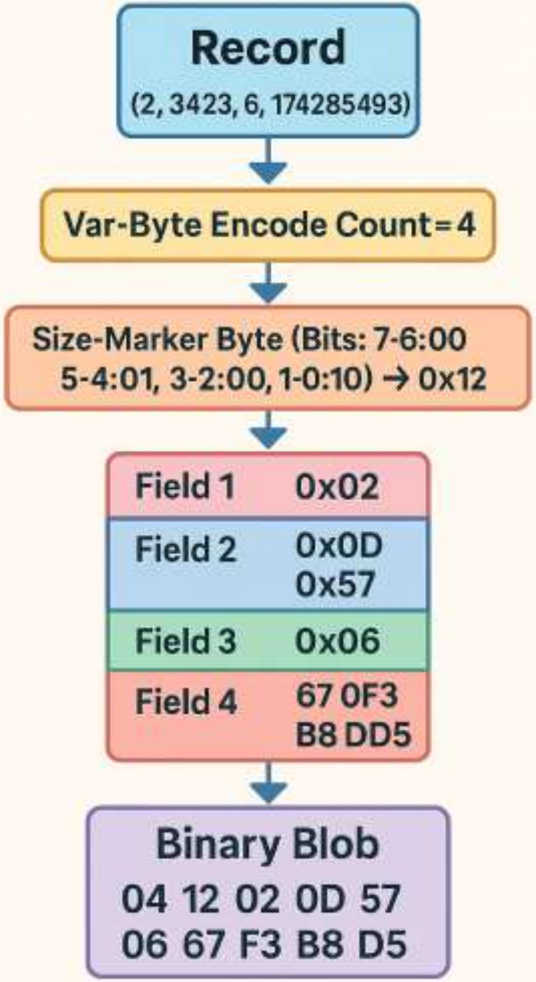
Purpose:

Minimize on-disk footprint and parsing cost with a compact binary format.

Key Steps:

- Batch prefix: Var-byte encode the number of records
- Null flags: Pack up to 8 boolean fields into a single bitmap byte
- Integer fields: Use 2-bit headers to select 1/2/4/8-byte encoding per value
- Text fields: Store a length byte followed by UTF-8 data
- Optional compression: Apply LZ4 to the full buffer for further size reduction.

Result: Adaptive integer encoding shrinks each value to 1–8 bytes (vs. fixed 8), cutting average record size by ~50%.



Synthetic Workload Generation

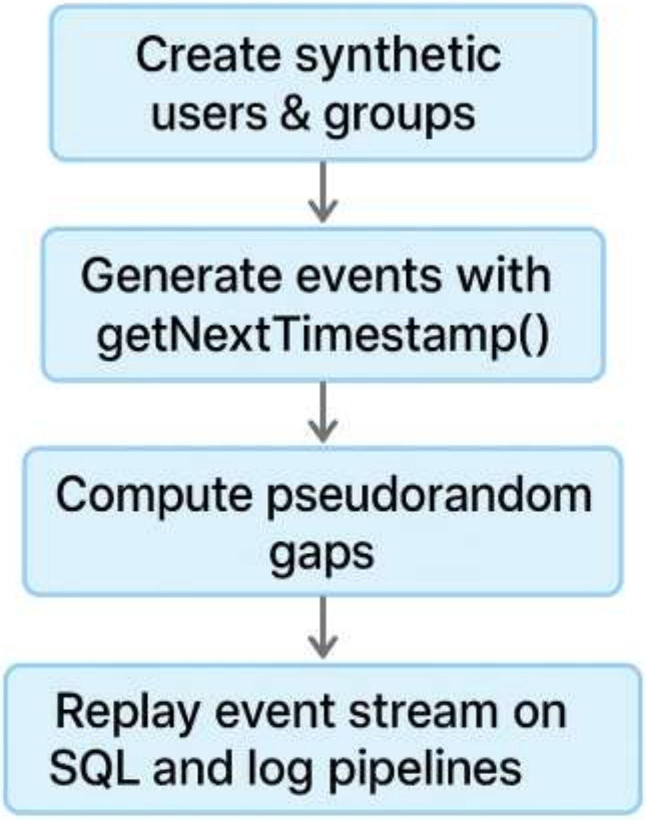
Purpose:

Simulate realistic, reproducible analytics traffic with time-ordered events.

Key Steps:

- Script creation of synthetic users and their personal and public groups.
- Generate multiple events per user, stamping each with a strictly increasing time via getNextTimestamp().
- Compute pseudo random gaps within the configured time window to spread events realistically.
- Replay the entire event stream identically through both SQL-based and log-based ingestion.

Result: A controlled, chronologically distributed workload that exercises both pipelines under identical conditions, enabling fair performance comparison.



Aggregating Impression Data

Purpose:

Roll up raw events from log partitions into queryable summary tables.

Key Steps:

- Calculate boundaries (hourly, daily, monthly, …) for the current period.
- Delete any existing summary rows for the target period.
- For each partition index, unpack new events, filter by timestamp, and tally counts in memory.
- Batch-insert results into the summary table.
- Run a single SQL INSERT … SELECT SUM(...) GROUP BY … over the just-written hourly rows.

Result: Accurate, idempotent summaries for each time bucket.

Comparative Analysis - Storage Footprint

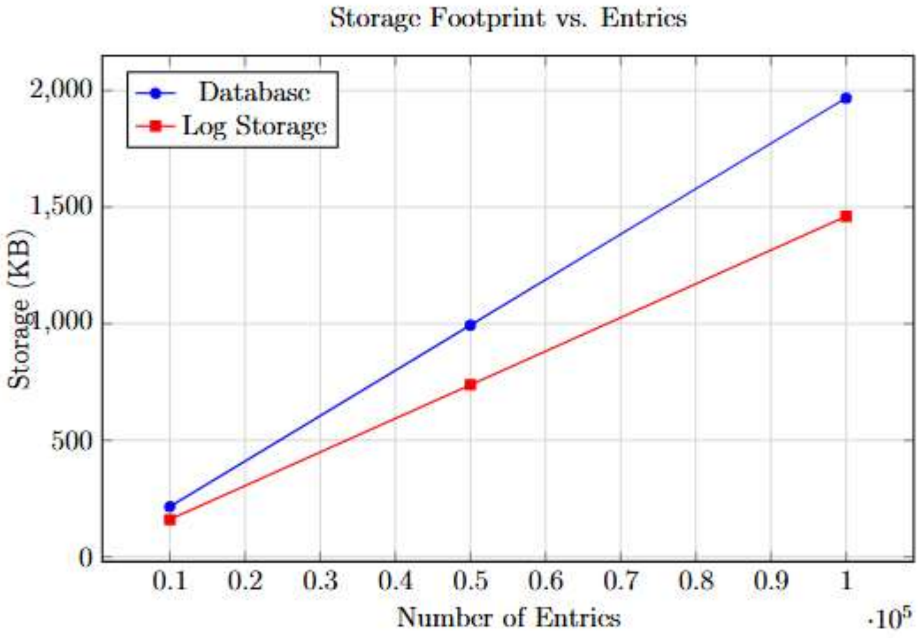
Purpose:

Assess how much disk each pipeline consumes at scale.

Key Steps:

- Load 10.000 to 100,000 events into both the database and log system.
- Measure on-disk size of raw tables versus rotated log files.

Result: Log-based storage uses only ~74 % of the space of the SQL tables at 100,000 events (1,460 KB vs. 1,968 KB).



Comparative Analysis - Query Response Time

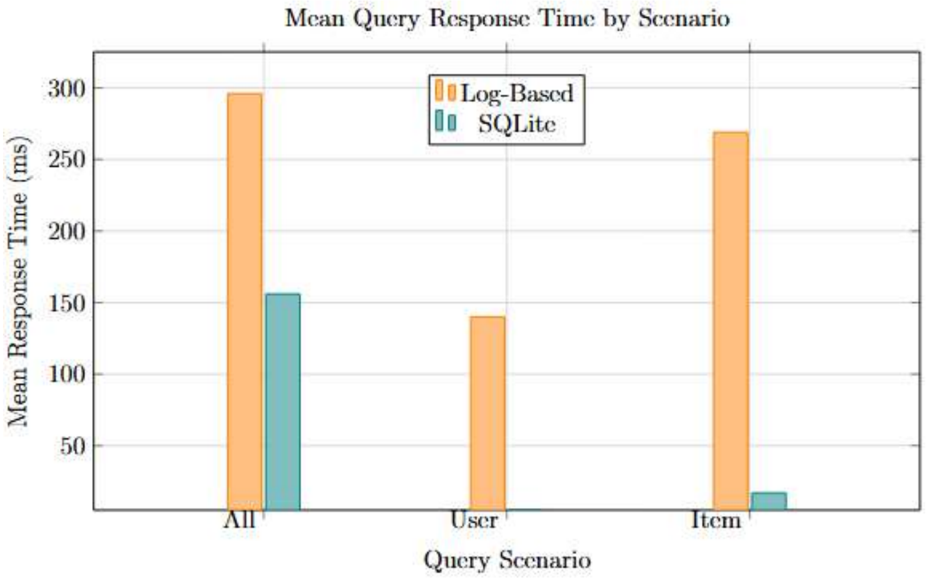
Purpose:

Compare latency across different analytics query scenarios for both pipelines.

Key Steps:

- Execute a broad “all impressions” query over the entire dataset
- Run targeted lookups for a single user and a single item across all partitions
- Perform single-partition scans filtering only the relevant log file
- Measure mean, median, and P95 latencies for each scenario on both systems

Result: Full scans: 296 ms vs. 156 ms; partition scans: 25 ms/48 ms vs. SQL’s 5 ms/17 ms.



Comparative Analysis - Write Throughput - Sequential

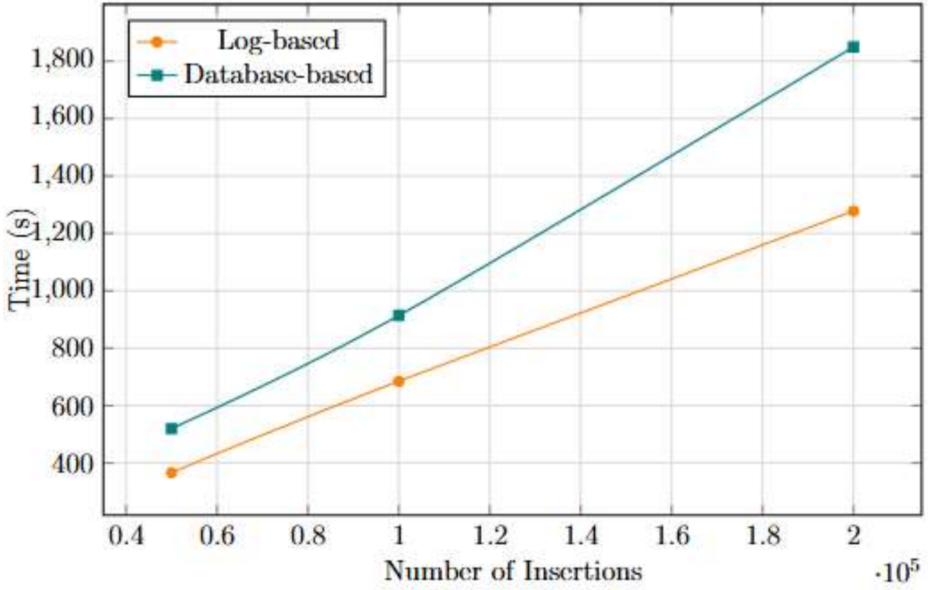
Purpose:

Measure end-to-end Sequential ingestion time for varying data volumes.

Key Steps:

- Sequentially write 50 000, 100 000, and 200 000 events through each pipeline.
- Use the same hardware and script for both log-based and database-based methods.
- Record total execution time for each dataset size.
- Compare raw write costs without parallelism.

Result: Across tested volumes, the log-based pipeline was about 25–30% faster—delivering roughly a 1.3× throughput gain over the database-based approach.



Comparative Analysis - Write Throughput - Concurrent

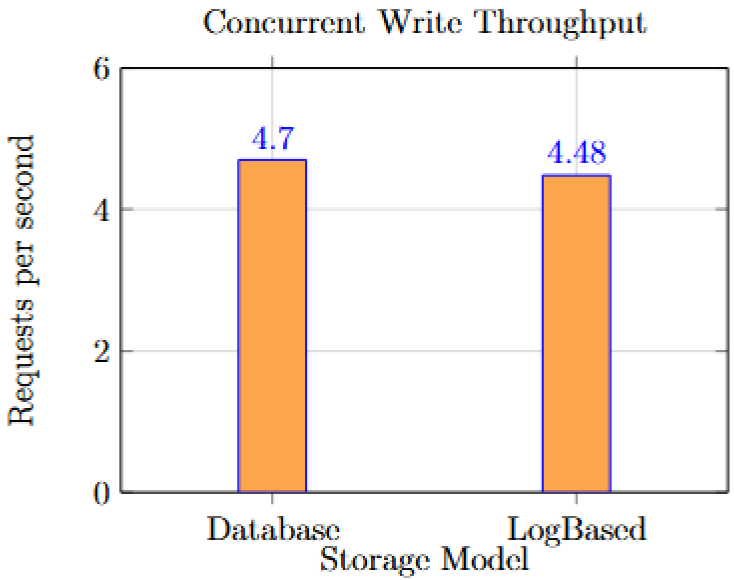
Purpose:

Measure ingestion throughput when multiple clients write simultaneously.

Key Steps:

- Simulate concurrent insert requests against both pipelines.
- Record operations per second and per-request latencies (mean & median).
- Compare how each handles locking, batching, and I/O under load.

Result: Under concurrent load, SQLite achieved about 4.70 req/sec versus 4.48 req/sec for the log-based pipeline, showing nearly identical throughput.



Comparative Analysis - Aggregation Latency

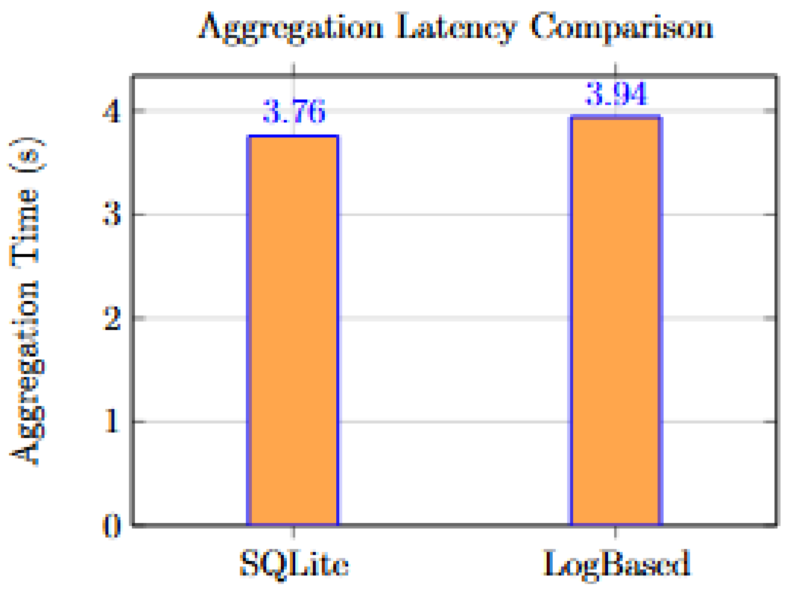
Purpose:

Evaluate end-to-end time to generate analytics summaries from raw events.

Key Steps:

- Run computeStatistics() on a fixed event set.
- Measure wall-clock time from invocation start to completion
- Execute under identical conditions for both log-based and SQL paths.

Result: Log-based aggregation was only about 5% slower than the SQL-only approach, demonstrating near-parity despite the extra decoding step.



Results

- ➔ Write Throughput: Log-based achieved ~1.3× higher throughput than SQL in sequential insertions

  and on par results during concurrency.
- ➔ Storage Footprint: Log files consumed ~74% of the disk space of database tables
- ➔ Query Latency:
  - ◆ Full-dataset scans slower on log (296 ms vs. 156 ms)
  - ◆ Single-partition lookups narrowed to 25 ms–48 ms
- ➔ Aggregation Latency: Log-based roll-ups incurred only ~5% extra time

Future Work

- Partition Metadata
  - Maintain a tiny partition-metadata index mapping each partition to its time range, so the aggregation job can compute the cutoff partition in O(1) without scanning logs.
- Adaptive Partition Compression
  - Apply lightweight or no compression on the most recent “hot” partitions to keep CPU overhead low, then switch to stronger codecs on older, colder partitions, balancing real-time write/read performance.

Conclusion

➔ A hybrid log-and-summary design meets both high-throughput ingest and real-time query demands.

➔ Append-only logs ensure sequential, crash-safe writes with minimal code changes.

➔ Empirical benchmarks validate scalability, efficiency, and robustness.

References

1. M. Rosenblum and J. K. Ousterhout, "The log-structured file system," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*,1992.
2. P. O'Neil, E. O'Neil, and G. Weikum, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, 1996.
3. S. Patro and Others, "Dynamic partition sizing in log-structured storage," in *Proceedings of the 2021 USENIX Annual Technical Conference*, 2021.
4. Y. Zhang and J. Patel, "Hybrid indexing for append-only logs," in *Proc. 2021 IEEE Int. Conf. on Data Engineering (ICDE)*, 2021.

Thank You!
Questions?