INFO-H-415: Advanced Databases

Benchmarking Embedded Databases on IoT Sensor Data

ROGGE Ethan () – ethan.rogge@ulb.be
DONNAY Basile (619065) – basile.donnay@ulb.be
CHEOUIRFA Anas (514347) – anas.cheouirfa@ulb.be

M. Esteban Zimányi (ezimanyi@ulb.ac.be)

Academic Year 2025-2026

Glossary

1 What Is an Embedded Database?

1.1 Definition of an Embedded Database

An embedded database is a database management system (DBMS) that is integrated directly into an application as a software library, rather than operating as a standalone server process. It executes within the same memory address space as the host application and requires no external runtime environment, configuration, or network communication. By design, the database is self-contained: data is typically stored in one or more local files on disk or in memory, and all operations are performed through direct function calls or language-native APIs. This tight coupling renders the database transparent to the end user and invisible in the deployment architecture. In contrast, client–server database systems—such as PostgreSQL or MySQL—reside in a separate process or machine and communicate with applications over a network protocol, introducing latency and operational complexity.

1.2 Typical Use Cases

Embedded databases are well suited to scenarios where deploying a full database server is impractical, unnecessary, or impossible. They are widely used in desktop and mobile applications—such as email clients, media players, and productivity suites—where local data persistence is required without user-facing database management. In embedded systems and Internet of Things (IoT) environments, they provide reliable storage on resource-constrained hardware, including sensors, industrial controllers, and network gateways, often operating without persistent network connectivity. Command-line utilities, scientific workflows, and data processing pipelines also leverage embedded databases to cache intermediate results or manage configuration state. Furthermore, they serve as the persistence layer in offline-first applications, edge computing nodes, and serverless functions, where low latency, autonomy from external services, and minimal operational overhead are essential.

1.3 Advantages of Embedded Databases

Embedded databases offer several compelling benefits that stem from their architecture. Because they run as a linked library within the application process, they eliminate the need for a separate database server, drastically reducing deployment complexity and operational overhead. Data access incurs near-zero latency, as there is no network round-trip or inter-process communication. Their self-contained nature—often storing an entire dataset in a single local file—ensures portability, simplifies backup and distribution, and enables robust operation in disconnected or offline environments. Additionally, embedded databases typically exhibit a small memory and CPU footprint, making them ideal for mobile devices and embedded hardware. With no external dependencies or configuration required, they streamline development, testing, and distribution, allowing developers to ship applications that "just work" out of the box.

1.4 Limitations and Trade-offs

Despite these advantages, embedded databases entail inherent trade-offs. Because they share the application's memory space and lack a dedicated server process, they generally provide limited concurrency control—often supporting only single-writer or coarse-grained multi-threaded access—rendering them unsuitable for high-throughput, multi-user environments. The absence of a centralized management interface complicates tasks such as remote monitoring, automated backups, or diagnostics. Moreover, while some embedded systems offer rich query languages (e.g., full SQL), others expose only basic data primitives (e.g., key–value operations), shifting the burden of complex logic—such as filtering, indexing, or joins—onto the application layer. This can increase development effort and reduce maintainability. Finally, because the database and application coexist in the same process, bugs or crashes in one component may compromise the integrity of the other. These constraints imply that embedded databases excel in local, single-process workloads but are not intended to replace traditional DBMS in distributed or highly concurrent systems.

1.5 Diversity of Data Models

The term "embedded database" encompasses a broad spectrum of data models, reflecting the varied requirements of local persistence. At one end, relational embedded databases (e.g., SQLite, DuckDB) support structured schemas, ACID transactions, and full SQL, enabling complex analytical queries. At the other, key–value stores (e.g., Berkeley DB, LevelDB) treat data as an associative array, offering minimal structure but extremely fast access by identifier. Between these extremes lie document-oriented, graph-based, and columnar embedded systems, each optimized for specific access patterns. This diversity means that selecting an embedded database is not merely a question of "server vs. no server," but also of choosing the right data model for the workload: a relational engine may be ideal for aggregations and joins, while a key–value store may be preferable for caching or configuration.

1.6 Purpose of This Benchmark

The goal of this benchmark is not to crown a "best" embedded database, but to clarify in which contexts one technology is more appropriate than another. DuckDB and Berkeley DB were designed for fundamentally different purposes: DuckDB as an in-process analytical SQL engine optimized for vectorized columnar processing, and Berkeley DB as a low-level, schema-less key–value store focused on simplicity and reliability. Neither is universally superior; each excels within its intended domain. By evaluating them under workloads inspired by real-world IoT and edge scenarios—featuring both high-volume time-series ingestion and complex per-sensor analytics—this benchmark aims to provide developers with evidence-based guidance for architectural decision-making. The central insight is not raw performance, but *fitness for purpose*. In a landscape where "embedded database" can denote anything from a full SQL engine to a raw hash table, understanding these distinctions is essential to selecting the right tool for the job.

2 Introduction to the Evaluated Systems

To explore the spectrum of embedded database designs, this benchmark evaluates two representative embedded systems **Berkeley DB** and **DuckDB** alongside **PostgreSQL** as a traditional client–server baseline. These choices reflect distinct points on the embedded database continuum: Berkeley DB embodies the classical, low-level key–value model optimized for transactional reliability; DuckDB represents the modern analytical embedded engine designed for in-process SQL over columnar data; and PostgreSQL provides a reference for full-featured relational systems with external deployment.

Each system is described below in terms of its architectural foundations, core capabilities, and alignment with IoT-inspired sensor workloads involving time-series ingestion, point access, and analytical querying.

2.1 Berkeley DB: A Transactional Key–Value Embedded Engine

Berkeley DB (BDB) is a mature, embeddable key–value database engine originally developed at the University of California, Berkeley, and later maintained by Sleepycat Software (now Oracle). It operates as a library linked directly into the host application, requiring no external process or network communication. This design delivers minimal latency and a small footprint qualities that have made it a staple in operating systems, network appliances, and embedded devices for decades.

2.1.1 Architecture and Data Model

Berkeley DB stores data as opaque key–value pairs, where both keys and values are arbitrary byte arrays. To accommodate diverse access patterns, it supports multiple access methods: B-tree for ordered keys and efficient range scans via cursors, hash tables for constant-time exact-match lookups, and Queue/Recno structures optimized for sequential, append-heavy workloads such as log buffering. All storage structures are managed through a page-oriented engine that features a configurable buffer pool, write-ahead logging (WAL), and periodic checkpointing to ensure durability.

2.1.2 Transactional Guarantees and Concurrency

The system provides full ACID semantics through a WAL-based recovery mechanism. It supports concurrent access from multiple threads or even processes within a shared environment (`DB_ENV`), relying on fine-grained page and record-level locking. Nevertheless, its concurrency model is tuned for moderate write loads within a single application context and is not designed for high-throughput, multi-client OLTP scenarios.

2.1.3 Strengths for Edge and IoT Workloads

Berkeley DB excels in resource-constrained environments where data persistence must survive sudden crashes such as sensor buffers during power loss and where low memory and CPU usage are critical, for instance on microcontrollers or edge gateways. It is particularly well-suited for workloads dominated by fast point inserts and retrievals, and its simplicity and portability make it attractive when query expressiveness is secondary to reliability and footprint. Its decades-long adoption in routers, industrial controllers, and mobile applications underscores its robustness in such settings.

2.1.4 Limitations in Analytical Contexts

A key limitation of Berkeley DB is the absence of any built-in query language. All application logic including filtering, aggregation, and schema enforcement—must be implemented manually in the host code. Although range scans are possible via cursors, complex analytical operations require explicit traversal and in-memory processing, which increases development effort, maintenance burden, and limits scalability for large-scale analytics.

2.1.5 Role in This Benchmark

In our evaluation, Berkeley DB serves as the archetype of a *storage-focused embedded database*. We use it to measure the cost of implementing sensor workload primitives—such as "store reading for sensor X at time T" or "retrieve the latest value for sensor Y"—using only low-level key–value operations, thereby reflecting how many real-world IoT systems manage local state without a query layer.

2.2 DuckDB: An Analytical SQL Engine for In-Process Data

DuckDB is a modern, open-source embedded database designed specifically for analytical workloads. Unlike traditional embedded systems that prioritize efficient storage, DuckDB brings the power of columnar query engines directly into the application process, enabling rich SQL analytics over local data without requiring a separate server.

2.2.1 Columnar Architecture and Execution Model

DuckDB stores tables in a columnar layout, where values of the same attribute are stored contiguously. This design enables efficient compression, sequential I/O, and CPU cache-friendly access—essential traits for analytical scans. Queries are executed using a vectorized model: operations are evaluated over fixed-size batches of tuples (typically 1,024 rows), leveraging SIMD instructions and minimizing interpretation overhead. The engine is fully pipelined, allowing intermediate results to flow between operators without full materialization. Moreover, DuckDB natively supports modern analytical file formats such as Parquet and includes highly optimized CSV parsing, enabling SQL queries to run directly over raw files without prior loading into memory.

2.2.2 SQL Capabilities and Language Ecosystem

DuckDB implements a rich SQL dialect that includes window functions, complex aggregations, common table expressions, and subqueries. It offers official client bindings for Python, R, Java (via JDBC), Node.js, C/C++, and even WebAssembly, making it a popular choice for data science notebooks, local ETL pipelines, and embedded analytics in desktop or server applications.

2.2.3 Performance Profile

Thanks to its columnar storage and vectorized execution, DuckDB achieves state-of-the-art performance on OLAP-style queries—such as wide scans, joins, and groupings—and consistently outperforms row-oriented embedded engines in analytical benchmarks. However, it is not optimized for high-frequency point updates or deletes, indexed key lookups without covering columns, or write-heavy transactional patterns. Its performance scales well with the number of CPU cores and benefits from fast local storage (e.g., SSDs), though memory consumption can increase significantly during complex queries due to the materialization of vectorized intermediates.

2.2.4 Concurrency and Durability

DuckDB employs a single-writer, multi-reader concurrency model based on multiversion concurrency control (MVCC) and WAL-based durability. Multiple readers can query the database simultaneously, but all write operations are serialized. While this model is sufficient for typical analytical workflows—such as one process appending data while others run read-only queries—it is not suited for high-concurrency OLTP scenarios involving many independent writers.

2.2.5 Role in This Benchmark

DuckDB represents the *analytical embedded database* paradigm. In our tests, it executes the same sensor-derived queries as Berkeley DB—but declaratively, using SQL. This allows us to quantify the trade-off between hand-coded key–value logic and optimized in-process analytical execution, particularly for tasks such as aggregations, anomaly detection, and cross-sensor correlations.

2.3 PostgreSQL: A Traditional Client–Server Baseline

PostgreSQL is a powerful, open-source relational database system that follows the classic client–server model. It runs as a standalone daemon, supports full SQL, ACID transactions, advanced indexing, and a rich extension ecosystem (e.g., TimescaleDB for time-series data). Although not embedded, it serves as a critical reference point in our benchmark.

2.3.1 Why Include a Non-Embedded System?

We include PostgreSQL to establish an upper bound on analytical expressiveness and correctness, to quantify the performance cost of network-based or inter-process access compared to in-process execution—even when running locally and to highlight the operational overhead (installation, configuration, monitoring) that is absent in embedded alternatives. In all tests, PostgreSQL runs on the same machine as the benchmark client, connected via Unix-domain sockets to minimize network effects and isolate architectural differences.

2.3.2 Architecture and Execution Model

PostgreSQL follows a classic client–server architecture: the database engine runs as a dedicated daemon process, while applications connect via a network protocol—even over localhost. It uses a row-oriented storage layout, optimized for mixed OLTP and light OLAP workloads, and relies on a cost-based query optimizer to generate execution plans. Key architectural features include multiversion concurrency control (MVCC) for high-concurrency transaction processing, write-ahead logging (WAL) for durability and point-in-time recovery, a variety of index types (including B-tree and specialized structures like BRIN and GIN), parallel query execution for CPU-intensive operations, and comprehensive SQL support including window functions and common table expressions.

2.3.3 Strengths for General-Purpose Workloads

PostgreSQL excels in environments that demand strong ACID guarantees across complex transactions, concurrent access from many independent clients, rich data modeling (with schemas, constraints, and foreign keys), and extensibility through extensions such as PostGIS or TimescaleDB. Its robustness, standards compliance, and active ecosystem make it a default choice for server-side applications, web backends, and enterprise data systems.

2.3.4 Limitations in Embedded and Edge Contexts

Nevertheless, PostgreSQL's architecture introduces overhead that is often unnecessary or even prohibitive in embedded scenarios. Process isolation and inter-process communication add latency, its memory and CPU footprint are significantly larger than those of embedded alternatives, its row oriented storage is less efficient for wide analytical scans compared to columnar engines, and its operational complexity (installation, configuration, monitoring) conflicts with the "zero-admin" ethos of embedded systems. These characteristics render it ill-suited for deployment on resource constrained devices or in applications requiring self-contained, serverless binaries.

2.3.5 Role in This Benchmark

In our evaluation, PostgreSQL runs locally on the same machine as the benchmark client and serves three key purposes: it establishes an upper bound on query expressiveness and correctness for analytical tasks; it quantifies the performance gap between in process execution (DuckDB, Berkeley DB) and external server access; and it helps highlight architectural trade offs specifically, when the added power of a full DBMS justifies its operational and performance overhead in edge-like workloads. By comparing PostgreSQL's behavior to that of the embedded systems, we gain insight into the practical boundaries of embedded analytics, answering not just "how fast?" but "when is embedded enough?"

3 Benchmark Design and Experimental Setup

In this section, we describe the dataset, the workload scenario, and the benchmark methodology used to evaluate BerkeleyDB, DuckDB, and PostgreSQL. The focus is exclusively on the benchmark; no full end-user application is implemented. Instead, the benchmark scripts execute a family of queries and updates derived from a realistic IoT monitoring scenario.

3.1 Dataset Description

We use the publicly available Intel Berkeley Research Lab dataset, a widely studied real-world IoT dataset. It contains:

- over 2.3 million timestamped readings,
- collected from 54 sensors deployed indoors,
- covering temperature, humidity, light intensity, and voltage,
- timestamped both as `epoch` and as human-readable date/time.

Each row corresponds to the measurement of a single sensor at a specific time. This dataset is particularly suited to evaluating embedded databases because:

- it resembles typical IoT monitoring workloads,
- it is large enough to stress test analytical engines,
- it requires both fast point lookups and global computations.

For the benchmark, the raw `data.txt` file is converted to CSV and loaded into the different database engines. To simulate deployments of various scales, subsamples of size 1 000, 10 000, and 100 000 rows are created using a stratified sampling procedure that preserves the chronological order within each sensor.

3.2 Workload Scenario and Requirements

The benchmark workload is inspired by IoT monitoring use cases in which a sensor platform must:

- ingest new measurements,
- retrieve recent values for specific sensors,
- update or delete measurements,
- compute statistics per sensor and per day,
- detect anomalous measurements,
- detect outages (periods with missing data),
- compare the behaviour of different sensors.

These requirements lead to a set of benchmark phases that jointly cover both OLTP-like and OLAP-like operations on the same dataset.

3.3 Benchmark Methodology

To evaluate the performance of the three systems, we designed a benchmark whose phases are directly derived from the workload scenario above. Our objective is to study:

- how each system scales with increasing dataset size,
- whether the behaviour is linear or exponential,
- how embedded systems compare to a traditional RDBMS.

3.3.1 Workload Definition

Each benchmark phase corresponds to a well-defined operation on the sensor dataset. Below we describe in detail what each phase does:

- **LOAD**: bulk insertion of N rows from the CSV file into the database. For BerkeleyDB, each row is stored as a value keyed by a composite key (sensor identifier and timestamp). For DuckDB and PostgreSQL, the rows are inserted into a relational table with columns for timestamp, sensor id, temperature, humidity, light and voltage. This phase models initial ingestion or batch loading.

- **READ**: a series of point lookups of individual rows. The benchmark repeatedly selects specific sensor/timestamp keys and retrieves the corresponding measurement. This models access to individual measurements, as might occur for debugging or detailed inspection of specific events.

- **LAST_READINGS**: for a set of sensor identifiers, retrieve the most recent reading of each sensor. In SQL systems, this is implemented as a grouped query using `MAX(timestamp)` or window functions; with BerkeleyDB, it corresponds to accessing the last key for each sensor. This models dashboards or monitoring tools that show the latest state of each sensor.

- **UPDATE**: modify specific fields (e.g. temperature or humidity) of existing rows selected by sensor id and time range. In SQL systems, this is performed with `UPDATE` statements; in BerkeleyDB, the corresponding values are overwritten. This phase simulates corrections to recorded measurements.

- **DELETE**: remove rows corresponding to a subset of sensors or a given time window. In SQL systems, this uses `DELETE` with predicates; in BerkeleyDB, it deletes keys. This models the purging of obsolete or corrupted data.

- **AVG_SENSOR**: compute the average temperature per sensor over all measurements in the dataset. In SQL, this is a `GROUP BY` query: for each sensor id, compute `AVG(temperature)`. This phase quantifies how fast each system can perform basic aggregation over the full dataset.
- **HOT_SENSORS**: identify sensors whose average temperature is above a threshold and that have at least a minimum number of readings. This combines aggregation (`AVG`) with a `HAVING` filter and optionally a `LIMIT k` to obtain a top-k list of the hottest sensors. It models alerting scenarios such as "which sensors are overheating".
- **ANOMALIES**: detect potentially anomalous measurements for each sensor. Concretely, for each sensor the benchmark computes its mean temperature and either standard deviation or interquartile range, and then selects measurements whose temperature deviates from the mean by more than a fixed threshold (e.g. more than two standard deviations). This is implemented as a combination of aggregations and joins or window functions in SQL systems. It models anomaly detection on individual time-series.
- **DAILY_SUMMARY**: compute daily statistics per sensor, such as the minimum, maximum, and average temperature per day. In SQL, this corresponds to grouping by sensor id and date (extracted from the timestamp) and applying `MIN`, `MAX` and `AVG` aggregations. This models daily reports or summary dashboards.
- **OUTAGES**: detect periods where a sensor did not send any data for an unusually long time. In SQL systems, this is implemented using a window function such as `LAG(timestamp)` partitioned by sensor id and ordered by time; the query computes the difference between consecutive timestamps and filters those where the gap exceeds a given threshold (e.g. one hour). This phase models outage detection in monitoring systems.
- **TOPK_SENSORS**: compute the k sensors with the highest average temperature or with the largest number of readings. This is a `GROUP BY` followed by an `ORDER BY` and `LIMIT`. It models ranking tasks, such as finding the busiest or hottest sensors.
- **COMPARE_TWO**: compare the behaviour of two sensors over time, e.g. by computing the correlation between their temperature time-series. In SQL, this is implemented by joining the measurements of two sensors on aligned timestamps (or time buckets) and then computing correlation coefficients or summary statistics. This models cross-sensor analysis.
- **FULL_PROFILE**: a combined workload that chains several of the analytical queries above. For DuckDB and PostgreSQL, this phase approximates the cost of computing per-sensor averages, daily summaries, outages and top-k sensors in sequence. For BerkeleyDB, which lacks SQL, the benchmark measures a simple local statistical summary over the most recent cached values. This phase illustrates the cumulative cost of a typical analytic session on the dataset.

Together, these phases cover ingestion, point lookups, transactional updates, aggregations, anomaly detection, temporal analysis, and cross-sensor comparisons.

3.3.2 Data Scales

To assess scalability, we evaluate the benchmark using four dataset sizes:

$$N \in \{1000,\ 10000,\ 100000,\ 1000000\}.$$

For each size, a stratified sampling algorithm selects an approximately equal number of rows per sensor whenever possible and preserves chronological order. This ensures that the benchmark is representative of the full dataset and preserves time-series structure.

3.3.3 Execution Protocol

Each benchmark phase is executed six times consecutively. The first execution is discarded because it typically includes caching, compilation, or buffer pool initialisation effects. The average of the remaining five executions is used as the reported performance metric.

This procedure follows best practices in benchmarking, including:

- warming up caches to avoid cold-start bias,
- averaging over multiple runs to reduce variance,
- ensuring reproducibility across systems.

3.3.4 Hardware and Environment

All experiments were executed on the same machine to guarantee fairness. The benchmark environment includes:

- Python 3.x for orchestrating the benchmark scripts,
- BerkeleyDB with Python bindings,
- DuckDB 1.x,
- a local PostgreSQL server instance,
- identical CSV input files and sampling procedure for all engines.

Details such as CPU model, RAM, operating system, and software versions can be reported to further support reproducibility.

4 Benchmark Results

This section reports the benchmark results for the three engines: *BerkeleyDB* (embedded key–value), *DuckDB* (embedded analytical SQL), and *PostgreSQL* (server-based baseline). We evaluate four scales $N \in \{10^3, 10^4, 10^5, 10^6\}$ rows. Each phase is executed multiple times with cache warm-up and we report the average duration (in seconds). The figures below plot the average duration versus N (log-scale on the y axis).

We separate (i) **OLTP-like operations** (LOAD/READ/UPDATE/DELETE and LAST_READINGS) from (ii) **analytical SQL operations** (AVG_SENSOR, HOT_SENSORS, ANOMALIES, DAILY_SUMMARY, OUTAGES, TOPK_SENSORS, COMPARE_TWO, and the analytical bundle FULL_PROFILE). BerkeleyDB does not provide SQL, therefore it only participates in phases that can be expressed with key–value access and local statistics.

4.1 OLTP-like operations

4.1.0.1 LOAD. Figure 1 shows the cost of inserting N rows. DuckDB is extremely slow for ingestion, increasing from 2.918 s at 10^3 rows to 3135.635 s at 10^6 rows. PostgreSQL scales much better, from 0.013 s at 10^3 to 21.665 s at 10^6. BerkeleyDB is consistently the fastest for ingestion, from 0.001 s (10^3) to 0.733 s (10^6).

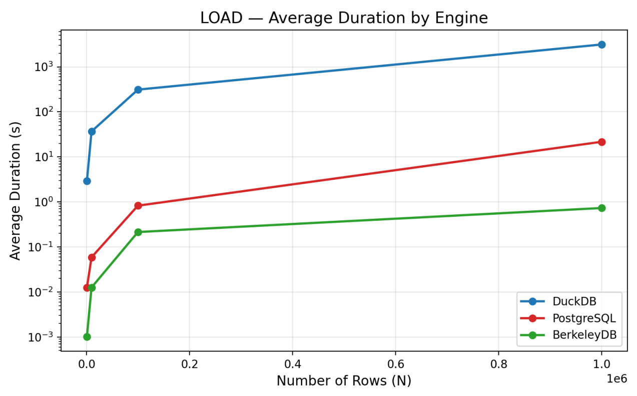

Figure 1: LOAD — average duration vs. number of rows.

4.1.0.2 READ. The READ phase performs repeated point reads by key (ops=N). Figure 2 shows BerkeleyDB dominating across all scales: 0.000696 s at 10^3 and 2.461 s at 10^6. PostgreSQL increases from 0.294 s (10^3) to 213.115 s (10^6), while DuckDB increases from 0.568 s (10^3) to 729.178 s (10^6), confirming that DuckDB is not designed for high-frequency random point lookups.

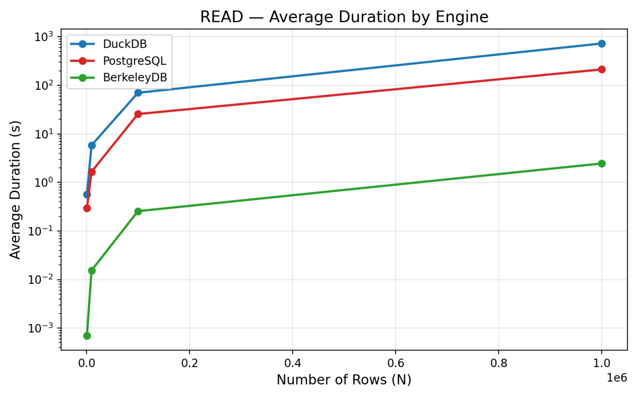

Figure 2: READ — average duration vs. number of rows.

4.1.0.3 UPDATE. UPDATE performs point updates (ops=$N/2$). Figure 3 shows BerkeleyDB remaining the fastest: 0.000682 s at 10^3 and 3.976 s at 10^6. PostgreSQL grows from 0.039 s (10^3) to 113.582 s (10^6). DuckDB is by far the slowest, from 1.525 s (10^3) to 1610.980 s (10^6), consistent with its columnar/append-oriented storage behaviour for updates.

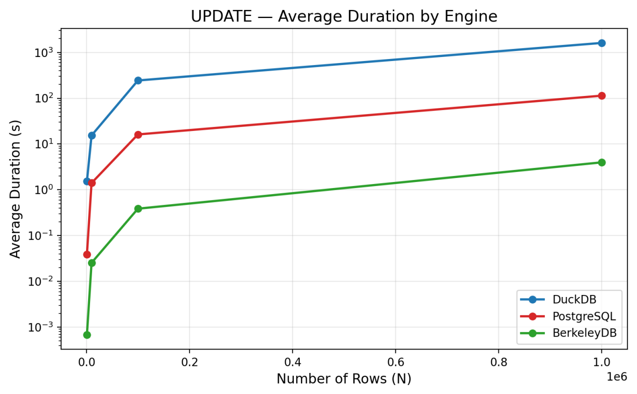

Figure 3: UPDATE — average duration vs. number of rows.

4.1.0.4 DELETE. DELETE performs point deletions (ops=$N/10$). Figure 4 shows BerkeleyDB again dominating: 8.7×10^{-5} s (10^3) to 0.303 s (10^6). PostgreSQL goes from 0.031 s (10^3) to 22.431 s (10^6), and DuckDB from 0.111 s (10^3) to 98.637 s (10^6). DuckDB remains substantially slower than PostgreSQL for deletions.

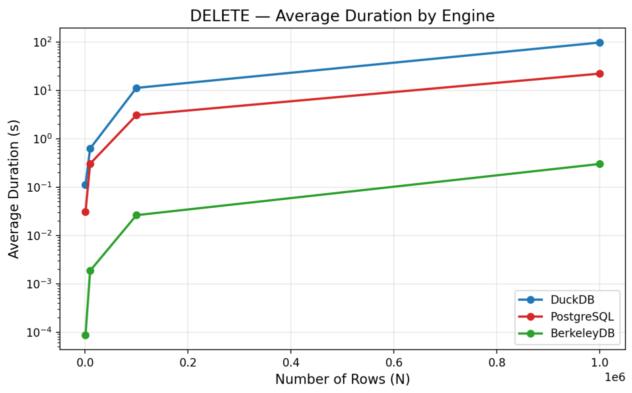

Figure 4: DELETE — average duration vs. number of rows.

4.1.0.5 LAST_READINGS. This phase retrieves the most recent readings for a set of sensors (1 sensor at 10^3 and 10^4, 3 sensors at 10^5, 25 sensors at 10^6). Figure 5 shows BerkeleyDB providing very low latency: 9.3×10^{-5} s (10^3) and 0.001137 s (10^6). DuckDB increases from 0.004961 s (10^3) to 0.157615 s (10^6), while PostgreSQL increases from 0.000547 s (10^3) to 1.456066 s (10^6). In this workload, BerkeleyDB is the clear choice for low-latency "latest value" retrieval at the edge.

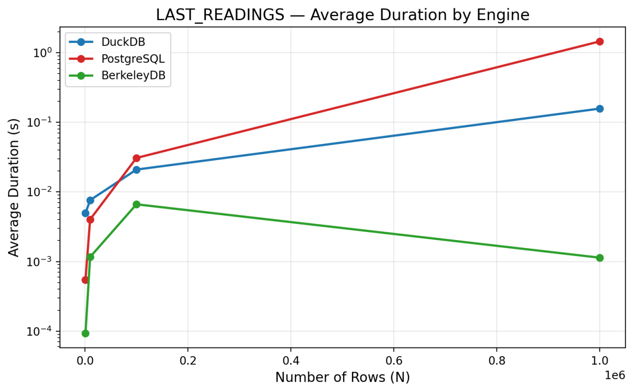

Figure 5: LAST_READINGS — average duration vs. number of rows.

4.2 Analytical SQL operations

4.2.0.1 AVG_SENSOR. AVG_SENSOR computes the average temperature per sensor (same sensor-count setting as LAST_READINGS). Figure 6 shows DuckDB consistently faster than PostgreSQL as N grows: at 10^6, DuckDB takes 0.037595 s while PostgreSQL takes 1.191561 s.

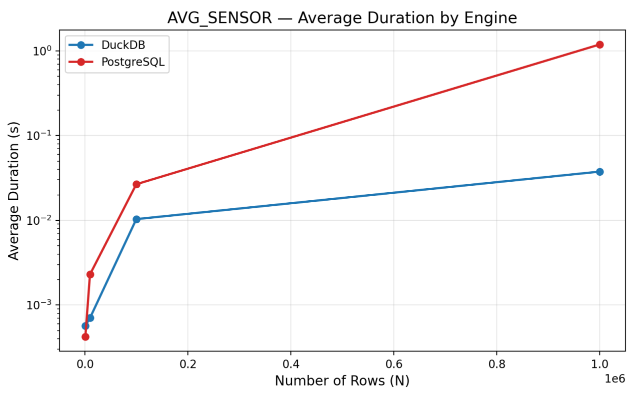

Figure 6: AVG_SENSOR — average duration vs. number of rows.

4.2.0.2 HOT_SENSORS. HOT_SENSORS identifies sensors with high average temperature. Figure 7 shows DuckDB faster than PostgreSQL at large scale: 0.004352 s (DuckDB) vs. 0.076205 s (PostgreSQL) at 10^6.

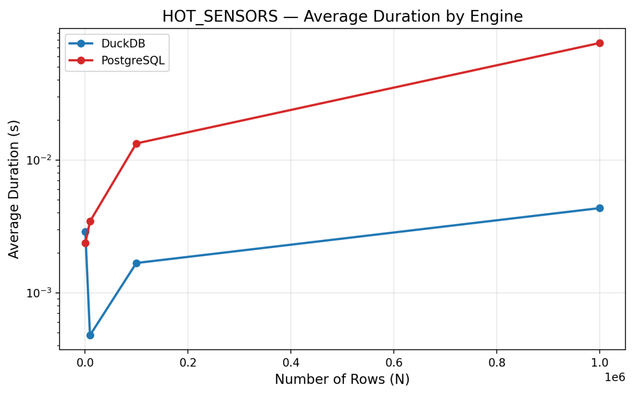

Figure 7: HOT_SENSORS — average duration vs. number of rows.

4.2.0.3 ANOMALIES. ANOMALIES detects outliers relative to the sensor mean. Figure 8 shows DuckDB significantly faster as N increases: 0.207942 s (DuckDB) versus 2.435990 s (PostgreSQL) at 10^6.

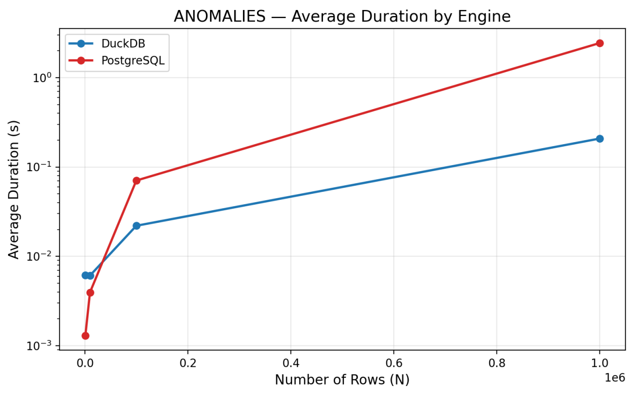

Figure 8: ANOMALIES — average duration vs. number of rows.

4.2.0.4 DAILY_SUMMARY. DAILY_SUMMARY aggregates readings by day. Figure 9 shows that both engines remain sub-second even at 10^6, with PostgreSQL slightly faster at the largest scale: 0.153745 s (PostgreSQL) versus 0.129354 s (DuckDB) at 10^6.

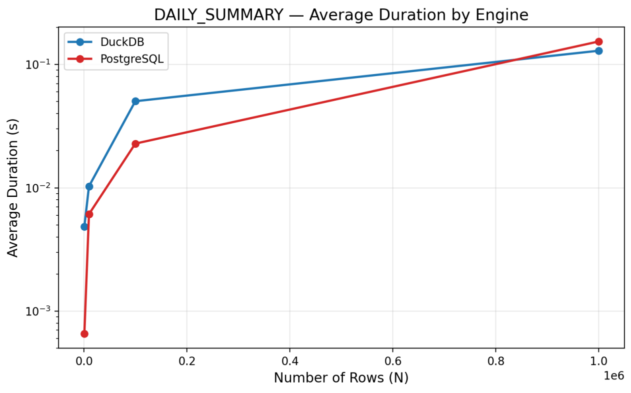

Figure 9: DAILY_SUMMARY — average duration vs. number of rows.

4.2.0.5 OUTAGES. OUTAGES detects time gaps in the sequence using window logic (e.g., LAG). Figure 10 highlights a clear advantage for DuckDB at scale: 0.025322 s (DuckDB) vs. 0.523254 s (PostgreSQL) at 10^6.

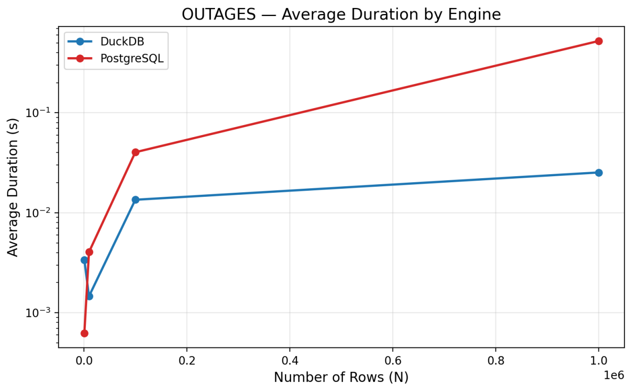

Figure 10: OUTAGES — average duration vs. number of rows.

4.2.0.6 TOPK_SENSORS. TOPK_SENSORS returns the top sensors by average temperature. Figure 11 shows DuckDB consistently faster at larger N: 0.005954 s (DuckDB) vs. 0.091878 s (PostgreSQL) at 10^6.

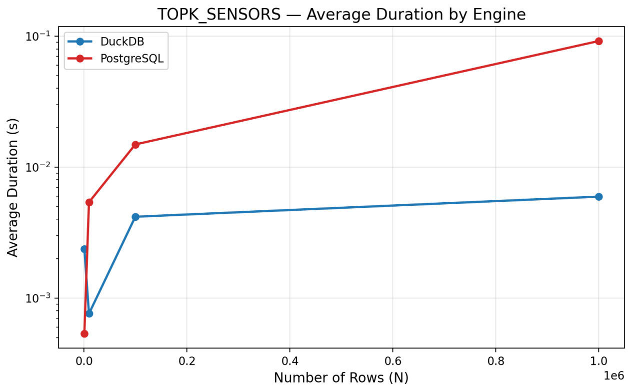

Figure 11: TOPK_SENSORS — average duration vs. number of rows.

4.2.0.7 COMPARE_TWO. COMPARE_TWO computes a correlation-like comparison between two sensors. At small scales the measured average is reported as 0.0 s, which indicates that the runtime is below the timer resolution for those cases. Figure 12 shows a strong advantage for DuckDB at scale: 0.006075 s (DuckDB) vs. 0.114103 s (PostgreSQL) at 10^6.

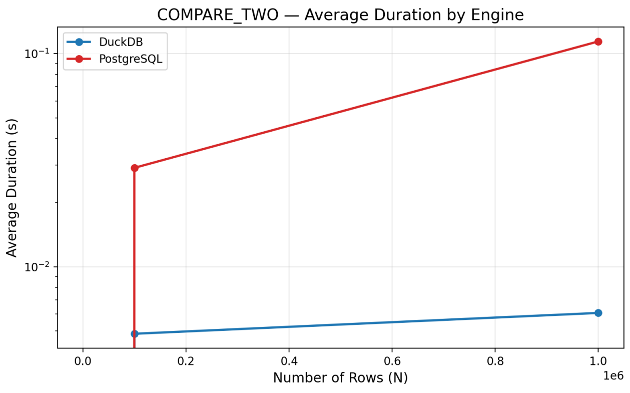

Figure 12: COMPARE_TWO — average duration vs. number of rows.

4.2.0.8 FULL_PROFILE. FULL_PROFILE is a combined workload. For DuckDB and PostgreSQL it corresponds to an *analytical bundle*, while BerkeleyDB runs a *local stats* summary on cached data. Figure 13 shows: at 10^6, DuckDB reaches 0.194521 s, BerkeleyDB 0.280865 s, and PostgreSQL 2.185694 s. Importantly, this does not imply that BerkeleyDB can execute the same analytical SQL; it reflects a different (local) summary computation compatible with a key–value API.

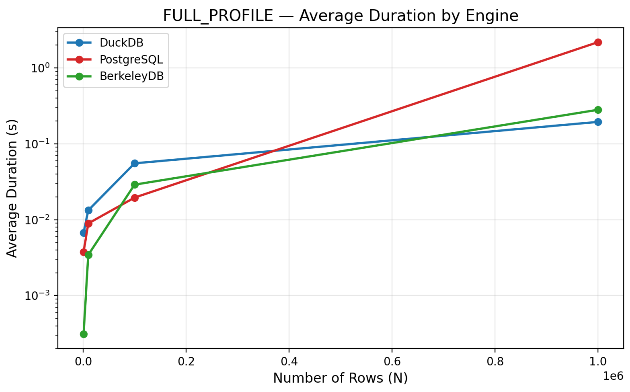

Figure 13: FULL_PROFILE — average duration vs. number of rows.

5 Discussion

This section provides a detailed technical interpretation of the benchmark results. Rather than focusing solely on absolute performance values, we explain *why* each database engine behaves as observed, by relating the results to their internal architecture, storage model, and execution strategies.

5.1 BerkeleyDB

5.1.1 Strengths: Direct Access and Minimal Overhead

The benchmark clearly shows that BerkeleyDB dominates all OLTP-like operations (`LOAD`, `READ`, `UPDATE`, `DELETE`, `LAST_READINGS`) across all dataset sizes.

This behaviour is primarily explained by BerkeleyDB's **embedded key–value architecture**. All operations are executed through direct API calls within the same process as the application. There is no SQL parsing, no query planning, no cost-based optimisation, and no inter-process communication. Consequently, the execution path for a `READ` or `UPDATE` is extremely short: a key lookup in a B-tree followed by a memory or disk access.

For example, even at 10^6 rows, BerkeleyDB completes the `READ` phase in approximately 2.46 seconds for one million point lookups, which corresponds to only a few microseconds per operation. This near-linear scaling is expected from a balanced tree structure combined with an efficient buffer cache.

Similarly, `UPDATE` and `DELETE` operations remain below a few seconds at large scale. BerkeleyDB updates individual records in place, without triggering any global reorganisation of the storage layout. This sharply contrasts with column-oriented systems, where updates often imply rewriting entire data segments.

The `LAST_READINGS` query further highlights BerkeleyDB's suitability for edge and IoT workloads. Retrieving the most recent value for a sensor requires only a small number of key lookups, which explains the sub-millisecond execution times observed even at 10^6 rows.

5.1.2 Weaknesses: Lack of Expressiveness and Analytical Operators

The main limitation of BerkeleyDB is not performance but **expressiveness**. BerkeleyDB does not support SQL, grouping, joins, or window functions. Analytical queries such as `AVG_SENSOR`, `ANOMALIES`, or `OUTAGES` cannot be executed natively.

Any attempt to perform such queries would require full scans implemented in application code, resulting in:

- increased implementation complexity,
- reduced maintainability,
- lack of query optimisation,
- poor scalability for complex analytics.

The `FULL_PROFILE` phase for BerkeleyDB therefore represents a local, simplified statistical summary rather than a true analytical workload. Its low execution time reflects the simplicity of the computation, not an ability to compete with SQL-based analytical engines.

5.2 DuckDB

5.2.1 Strengths: Columnar Storage and Vectorised Execution

DuckDB consistently achieves the best performance for analytical queries such as `AVG_SENSOR`, `ANOMALIES`, `TOPK_SENSORS`, `COMPARE_TWO`, and the analytical `FULL_PROFILE` bundle.

This behaviour is explained by DuckDB's **column-oriented storage model** combined with **vectorised execution**. Data is stored column-wise, meaning that values of the same attribute are contiguous in memory. Analytical queries typically access only a subset of columns and perform the same operation on many rows, which results in:

- excellent cache locality,
- efficient use of SIMD instructions,
- reduced memory bandwidth consumption.

For instance, computing `AVG_SENSOR` at 10^6 rows takes only 0.038 seconds. This corresponds to a full scan of the relevant column executed in a tight vectorised loop, without materialising intermediate row objects.

Similarly, queries such as `ANOMALIES` and `OUTAGES`, which rely on grouping and window-like logic, benefit from DuckDB's pipelined execution model and late materialisation strategy.

5.2.2 Weaknesses: Write Amplification and Poor OLTP Performance

Despite its analytical strengths, DuckDB performs extremely poorly on write-heavy workloads. The benchmark shows that `LOAD` time grows from 2.9 seconds at 10^3 rows to more than 3 100 seconds at 10^6 rows. `UPDATE` and `READ` operations exhibit similarly dramatic growth.

This behaviour is a direct consequence of DuckDB's **append-only, columnar design**. Updates and deletes are not performed in place; instead, they require creating new versions of data segments or rewriting column chunks. As a result, point updates trigger large memory and I/O overhead, known as *write amplification.*

Moreover, DuckDB is optimised for sequential scans rather than random access. Point lookups require scanning column segments, which explains the very poor performance observed for the `READ` phase at large scale.

These design choices are deliberate: DuckDB sacrifices OLTP performance in order to maximise analytical throughput. The benchmark confirms that this trade- off is unsuitable for continuous IoT data ingestion or real-time updates.

5.3 PostgreSQL

5.3.1 Strengths: Balanced Design and Mature Query Processing

PostgreSQL exhibits a more balanced behaviour across workloads. Its **row-oriented storage model** allows efficient point reads and updates, while its mature SQL engine supports a wide range of analytical queries.

For example, at 10^6 rows, `LOAD` completes in 21.7 seconds, which is orders of magnitude faster than DuckDB but slower than BerkeleyDB. This reflects the cost of transactional guarantees, WAL logging, and index maintenance.

Analytical queries such as `AVG_SENSOR` and `ANOMALIES` remain fully supported, although they are slower than DuckDB due to less favourable cache locality and the absence of vectorised execution. Nonetheless, execution times remain within a few seconds even at large scale, demonstrating predictable and robust performance.

5.3.2 Weaknesses: Higher Overhead and Limited Analytical Optimisation

PostgreSQL's main limitation in this benchmark is its higher overhead for analytical queries. The row-based storage layout forces the engine to process tuples one by one, which results in more CPU instructions and poorer cache utilisation compared to DuckDB's columnar execution.

Furthermore, PostgreSQL's client–server architecture introduces additional overhead in terms of memory usage and configuration complexity, making it less attractive for embedded or resource-constrained environments.

5.4 Implications for Embedded IoT Architectures

The benchmark results strongly suggest that no single database engine is optimal for all aspects of an IoT application.

BerkeleyDB is the best choice for the **edge layer**, where low latency, low overhead, and predictable behaviour are critical. DuckDB is ideally suited for **local or offline analytics**, where complex SQL queries must be executed efficiently over large datasets. PostgreSQL remains a solid **general-purpose reference**, offering robustness and versatility at the cost of higher overhead.

These findings naturally motivate a hybrid architecture in which different database technologies are combined according to their strengths, rather than relying on a single engine for all workloads.