Benchmarking Embedded Databases for IoT Applications

Ethan Rogge, ...

December 2025

1 Introduction to the Technology: Embedded Databases

Embedded databases constitute a class of data management systems designed to run *inside* an application rather than as an external server process. The DBMS executes within the same address space as the host program and is invoked via direct API calls instead of network protocols. This architectural choice profoundly shapes both the capabilities and limitations of embedded database systems.

1.1 Definition of Embedded Databases

An embedded database is a DBMS that:

- operates as a software library linked to an application,
- does not require a separate server or daemon,
- stores data locally on the device's filesystem or in memory,
- handles queries or operations through a direct programming interface.

This contrasts with traditional server-based DBMS—such as PostgreSQL or MySQL—where applications connect via network protocols to an external process that manages concurrency, storage, and query execution.

Embedded databases offer several key advantages:

- **Low latency:** no network round-trip is required.
- **Minimal overhead:** no inter-process communication.
- **Portability:** the database engine is distributed with the application.
- **Small footprint:** suitable for embedded devices, IoT nodes, or mobile apps.
- **Tight control:** the application can directly manage caching and storage.

However, these systems also come with inherent limitations:

- reduced concurrency and multi-user capabilities,
- limited or absent SQL support in many engines,
- no access control or advanced security features,
- responsibility for durability and schema evolution often shifted to the application.

These trade-offs make embedded databases particularly suitable for resource-constrained environments such as IoT nodes, sensors, robotics, or standalone desktop analytics applications.

1.2 Architectural Models of Embedded Databases

Embedded database technologies vary significantly in their internal design. Several architectural models are commonly used:

1.2.1 Page-oriented storage engines

These systems manage their own page cache, buffer pools, and on-disk B-tree structures. Examples include BerkeleyDB, SQLite, and LMDB. Their properties:

- efficient key-based lookups,
- transactional semantics,
- predictable read/write performance,
- mature caching and recovery mechanisms.

1.2.2 Log-structured merge trees (LSM)

Engines such as LevelDB or RocksDB use LSM trees to support:

- extremely fast writes through append-only logs,
- background compaction processes,
- excellent performance for streaming or time-series ingestion.

These engines are widely used in mobile apps, IoT gateways, and distributed storage systems requiring sustained write throughput.

1.2.3 In-process analytical engines

Modern embedded engines such as DuckDB embed a complete analytical SQL engine directly inside the application. Their distinguishing features include:

- columnar storage,
- vectorised execution and query pipelining,
- efficient analytical queries (aggregations, joins, window functions),
- no need for a separate database server.

Such systems illustrate the evolution of embedded technology: beyond simple key–value stores, they support full SQL analytics locally within the application.

1.3 Query Capabilities in Embedded Systems

Because embedded engines target environments with constrained resources or specific workloads, their query capabilities vary widely.

Record-oriented APIs. Many engines expose operations such as:

```
get(key),  put(key,value),  delete(key)
```

These APIs offer very fast local access and are ideal for caching or metadata storage.

Full SQL support. Some embedded DBMS such as SQLite or DuckDB support full SQL semantics. This enables local execution of analytical queries, joins, and aggregations without a remote server. Such systems are attractive for standalone desktop analytics and local data processing pipelines.

Hybrid architectures. Certain applications combine multiple embedded technologies:

- a lightweight key–value store for fast caching or ingestion,
- a more expressive embedded engine for analytical queries.

This hybrid model is precisely the one adopted by our application.

1.4 Embedded Databases in IoT and Edge Computing

IoT devices frequently require real-time data processing close to where data is generated. Embedded databases are central to these architectures because they allow devices to:

- store recent sensor readings locally,
- perform preliminary filtering or anomaly detection,
- operate autonomously during network outages,
- reduce bandwidth by transmitting only summarised data,
- respect privacy constraints by keeping raw data on device.

Edge computing architectures often include:

- microcontrollers or single-board computers running a lightweight embedded store,
- intermediate nodes performing local analytics,
- cloud or server components performing long-term storage or heavy computations.

Thus, embedded databases form the backbone of many modern distributed sensing and monitoring systems.

1.5 Relevance to Our Application

The dataset used in our project (Intel Lab sensor data) represents a typical IoT scenario involving time-series measurements collected by multiple devices. The application requires:

- fast access to recent readings,
- local computation of sensor statistics,
- detection of anomalies or outages,
- global analytics over large volumes of historical data.

These requirements naturally motivate a hybrid architecture:

- a lightweight embedded store for fast lookups and caching,
- an embedded analytical engine for SQL-based analysis,
- a traditional server-based DBMS for baseline comparison.

2 Introduction to the Tools

In this section, we examine the two database management systems chosen to illustrate embedded database technology: **BerkeleyDB** and **DuckDB**. Although both systems belong to the family of embedded DBMS, they represent two fundamentally different models: BerkeleyDB is a classical key–value engine, while DuckDB is a modern analytical SQL engine designed to run in-process. For completeness and for comparison with a traditional relational DBMS, we also consider **PostgreSQL** as a baseline server-based engine.

Each subsection explains the internal architecture of the tool, its execution model, and its strengths and limitations with respect to the requirements of our IoT application.

2.1 BerkeleyDB

BerkeleyDB (BDB) is a mature, high-performance embedded storage engine that provides key–value access through a simple API. It operates as a library linked directly into the application, without any external server process. As one of the earliest and most widely deployed embedded databases, it has been used in operating systems, networking equipment, web browsers, and mobile devices.

2.1.1 Architecture

BerkeleyDB is built around a **page-oriented storage engine** and uses B-trees as its primary on-disk structure. Data pages are cached in an internal buffer pool, and the engine manages its own journaling, logging, and recovery mechanisms. It exposes a simple interface:

`put(key, value)`, `get(key)`, `delete(key)`,

with optional transactional semantics.

Since the DBMS runs within the same process as the application, all operations bypass the overhead of inter-process communication. This leads to extremely low latency for key lookups and updates.

2.1.2 Core Features

The main features of BerkeleyDB include:

- **High performance** for point reads, inserts and updates.
- **Configurable storage models**, including B-tree, hash tables, and queues.
- **Transaction support** with ACID guarantees.
- **Crash recovery** through write-ahead logging.
- **Small footprint**, making it suitable for embedded systems and IoT devices.

2.1.3 Transaction Model and Concurrency

BerkeleyDB supports multiversion concurrency control (MVCC) and fine-grained locking. However, concurrency is generally limited to threads within the same process. Unlike server-based DBMS, BerkeleyDB does not natively serve multiple external clients simultaneously.

2.1.4 Limitations

Despite its performance advantages, BerkeleyDB lacks:

- SQL query capabilities,
- built-in analytical operators (aggregations, joins, windows),
- multi-user concurrency architecture,
- ad-hoc querying capabilities.

Any form of analytics must be implemented manually within the application. For this reason, BerkeleyDB is best viewed as a **local cache or storage component** rather than a general-purpose DBMS.

2.1.5 Relevance to Embedded Applications

BerkeleyDB is extremely well suited for applications that require:

- storing recent data locally,
- fast retrieval of time-critical information,
- operation in resource-constrained devices,
- offline operation or intermittent connectivity.

In our IoT scenario, these characteristics make BerkeleyDB the ideal engine for storing and retrieving the most recent sensor readings at very low latency.

2.2 DuckDB

DuckDB is a modern embedded database designed specifically for analytical workloads. It provides a full SQL interface and implements many of the optimisations found in columnar analytical systems such as MonetDB or Analytical Processing Systems, but packaged as a lightweight in-process library.

2.2.1 Architecture

DuckDB follows a **columnar storage model** and uses vectorised execution. Instead of processing one row at a time, DuckDB processes data in fixed-size vectors (e.g., 1024 rows), greatly improving CPU efficiency and cache locality. It employs late materialisation, compressed storage, and pipelined execution to optimise analytical queries.

The engine runs entirely within the host process, eliminating the need for a database server and reducing overhead for local analytical tasks.

2.2.2 Vectorized Execution Model

Vectorised execution allows DuckDB to:

- evaluate expressions over batches of rows,
- exploit SIMD instructions,
- reduce interpretation overhead,

- pipeline operations efficiently.

This model is particularly advantageous for scans, aggregations, joins, and window functions, making DuckDB well suited for analytical workloads.

2.2.3 Columnar Storage and Query Processing

DuckDB stores tables in columnar format, meaning that values of the same column are stored contiguously. This enables:

- faster sequential scans for analytical queries,
- data compression,
- better I/O efficiency,
- reduced memory footprint.

Unlike server DBMS such as PostgreSQL, DuckDB does not maintain row-oriented storage optimised for transactional workloads. This explains its poor performance on UPDATE, DELETE, and frequent point-lookups.

2.2.4 Strengths and Limitations

Strengths.

- State-of-the-art analytical performance.
- Fully embedded, no server required.
- Full SQL support including window functions.
- Highly portable and easy to integrate into applications.

Limitations.

- Very slow for write-heavy workloads (UPDATE, DELETE).
- Poor performance for random point-lookups.
- Not a multi-user or concurrent transactional system.

2.2.5 Role in Analytical Embedded Workloads

DuckDB is ideal for applications requiring:

- local execution of analytical SQL queries,
- processing of large datasets in batch,
- interactive analytics without deploying a database server,
- embedded ETL or data science pipelines.

In our application, DuckDB is responsible for executing analytics such as anomaly detection, aggregations, top-k selection, and temporal analyses.

3 Common Assessment

In this section, we evaluate how the three database systems — BerkeleyDB, DuckDB, and PostgreSQL — behave when applied to the requirements of our IoT monitoring application. We first describe the dataset and the application workload, then present our benchmarking methodology. The benchmark results themselves are provided in Section **??**.

3.1 Application Description

The objective of the application is to process real-world IoT sensor data collected from a wireless sensor network. The application must support both **local, low-latency operations** (e.g., retrieving the most recent reading of a sensor) and **global analytical tasks** (e.g., computing averages, detecting anomalies, finding outages). This dual requirement motivates the use of embedded databases and provides a meaningful context in which to compare the three database engines.

3.1.1 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.

3.1.2 Application Requirements

The IoT application processes both *recent* and *historical* data. Its main requirements include:

- **rapid access to the most recent sensor values**, possibly in real-time,
- **local statistical computations** on one or more sensors,
- **global analytical queries** over large subsets of historical data,
- **detection of anomalies** in temperature patterns,
- **detection of outages**, i.e., time gaps in the readings,

- **comparisons between sensors**, such as correlation analysis.

These requirements naturally lead to distinct workload patterns, some of which are latency-sensitive (e.g., LAST_READINGS) while others are throughput- or scan-intensive (e.g., OUTAGES or DAILY_SUMMARY).

3.1.3 Implemented Functionalities

The application provides eleven functionalities, each corresponding to one or more query types evaluated in the benchmark. The most relevant operations are:

- **Ingestion (LOAD)**: insert a batch of readings.
- **Point lookup (LAST_READINGS, READ)**: retrieve recent sensor values.
- **Updates & deletes**: modify or remove existing readings.
- **Aggregations**: average temperature per sensor (AVG_SENSOR), daily summaries (DAILY_SUMMA or top-K sensors (TOPK_SENSORS).
- **Anomaly detection**: identify values that deviate from the mean.
- **Outage detection**: find time gaps using window functions.
- **Sensor comparison**: compute cross-sensor correlations.
- **Full profile**: a combined workload summarising multiple operations.

Each functionality maps cleanly to one or more benchmark phases. This ensures that the benchmark reflects realistic requirements rather than synthetic tasks.

3.1.4 Hybrid Architecture

The application adopts a hybrid architecture reflecting the strengths and limitations of each database system:

- **BerkeleyDB** is used as a *local embedded cache* storing the most recent sensor readings. Its extremely fast key–value access makes it ideal for point lookups and small-scale local statistics.
- **DuckDB** is used as an *embedded analytical engine* providing full SQL capabilities. It executes most analytical queries, such as aggregations, window functions, and correlation analysis.
- **PostgreSQL** serves as a *baseline reference* for evaluating the strengths and weaknesses of embedded DBMS compared to a traditional server-based relational engine.

This architecture mirrors typical IoT and edge-computing deployments, where:

- the **device** stores recent readings locally,
- an **embedded engine** performs analytics on device or gateway,
- a **server** stores or aggregates data at scale.

The hybrid model also highlights how embedded databases complement rather than replace traditional relational systems.

3.2 Benchmark Methodology

To evaluate the performance of the three systems, we designed a benchmark based directly on the application's requirements. The benchmark includes ingestion, point lookups, updates, deletions, aggregations, anomaly detection, and temporal analysis operations. 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.

3.2.1 Workload Definition

Each benchmark phase corresponds to a specific application-level task:

- **LOAD**: insert N rows.
- **READ** and **LAST_READINGS**: retrieve recent sensor values.
- **UPDATE / DELETE**: modify or remove records.
- **AVG_SENSOR**, **TOPK_SENSORS**: compute aggregated statistics.
- **ANOMALIES**: detect outliers relative to average temperature.
- **OUTAGES**: detect time gaps using window functions.
- **COMPARE_TWO**: sensor correlation analysis.
- **FULL_PROFILE**: combined analytical workload.

These phases collectively model typical IoT monitoring tasks involving both transactional and analytical operations.

3.2.2 Data Scales

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

$$N \in \{1\,000,\ 10\,000,\ 100\,000\}.$$

For each size, a stratified sampling algorithm selects an 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.

3.2.3 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.

4 Benchmark Results

This section presents the performance obtained for the full set of benchmark phases executed on the three database engines: *BerkeleyDB*, *DuckDB*, and *PostgreSQL*. All experiments were executed for $N \in \{10^3, 10^4, 10^5\}$ rows. Each phase was run six times; the first execution was discarded to avoid cache warm-up effects, and the execution times reported in the figures correspond to the average of the remaining five runs.

We distinguish between (i) OLTP-like operations (LOAD, READ, UPDATE, DELETE, LAST_READINGS) and (ii) analytical SQL operations (AVG_SENSOR, HOT_SENSORS, ANOMALIES, DAILY_SUMMARY, OUTAGES, TOPK_SENSORS, COMPARE_TWO, FULL_PROFILE).

4.1 OLTP-like operations

LOAD. Figure 1 shows the time to load N rows into each DBMS. DuckDB exhibits very poor scalability, increasing from a few seconds at 10^3 rows to more than 330 s for 10^5 rows. This linear but steep growth is due to its immutable storage model and internal checkpointing during bulk inserts. PostgreSQL remains efficient, staying below 1.5 s even for 10^5 rows thanks to optimized WAL handling and bulk insert mechanisms. BerkeleyDB is by far the fastest (below 0.1 s for all N), illustrating the efficiency of a key–value store for append-heavy workloads.

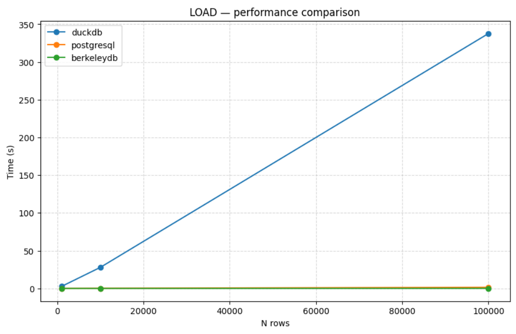

Figure 1: LOAD — average duration vs. number of rows.

READ. The READ benchmark retrieves records by key (sensor identifier). As shown in Figure 2, BerkeleyDB dominates with sub-millisecond response times. DuckDB and PostgreSQL both scale linearly, but DuckDB becomes extremely slow at 10^5 rows (more than 70 s), confirming that it is not optimized for repeated point lookups. PostgreSQL remains significantly faster (below 26 s), relying on index-based lookups.

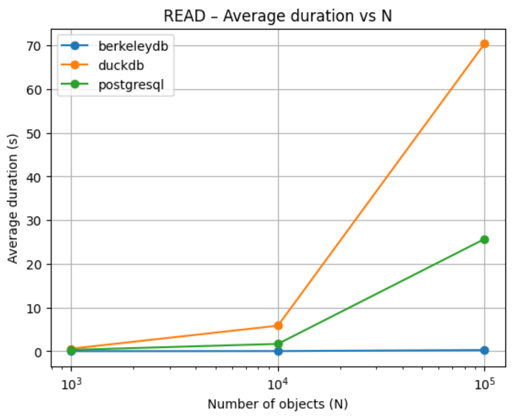

Figure 2: READ — average duration vs. number of rows.

UPDATE and DELETE. Figures 3 and 4 show that BerkeleyDB again provides the best performance with almost constant-time updates and deletions. PostgreSQL is slower but scales predictably, reaching around 17 s for UPDATE and 3 s for DELETE at 10^5 rows. DuckDB is by far the slowest (over 240 s for UPDATE and 11 s for DELETE at 10^5), due to table rewrites induced by its append-only columnar storage.

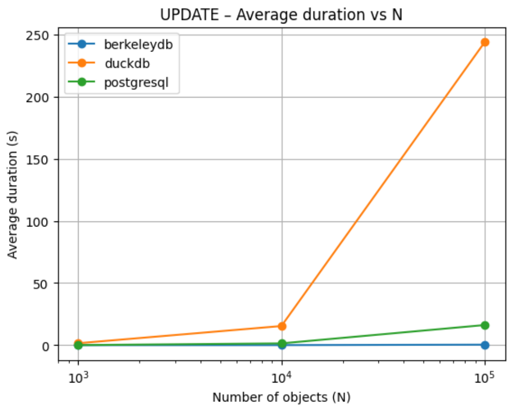

Figure 3: UPDATE — average duration vs. number of rows.

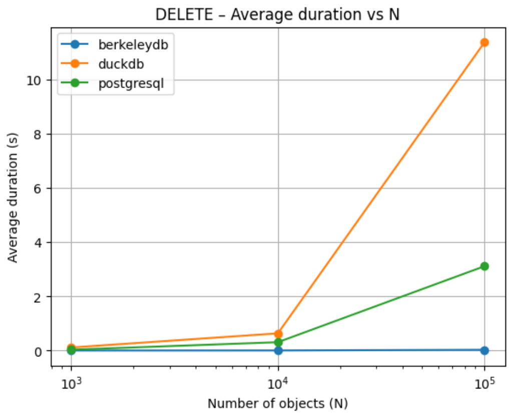

Figure 4: DELETE — average duration vs. number of rows.

LAST_READINGS. This phase simulates an IoT device requesting the most recent readings for a set of sensors. Figure 5 shows that BerkeleyDB is the fastest (fractions of milliseconds even at 10^5 rows). DuckDB performs moderately well (≈ 21 ms at 10^5), while PostgreSQL is the slowest (≈ 31 ms at 10^5), though still within acceptable limits.

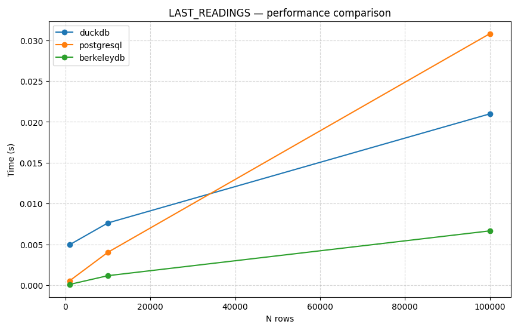

Figure 5: LAST_READINGS — average duration vs. number of rows.

4.2 Analytical SQL operations

ANOMALIES. The ANOMALIES phase computes deviations from the mean temperature for each sensor using SQL aggregations and window logic. As shown in Figure 6, DuckDB is consistently faster (from 0.006 s to 0.022 s) than PostgreSQL (from 0.002 s to 0.070 s), thanks to vectorized columnar execution. BerkeleyDB cannot execute this query.

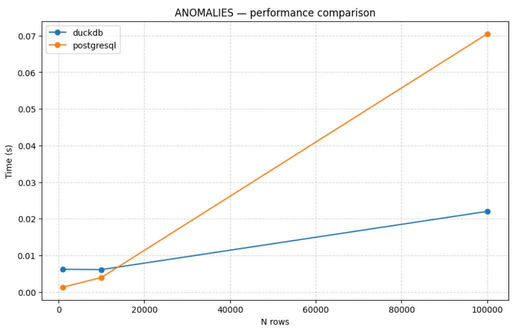

Figure 6: ANOMALIES — performance comparison between DuckDB and PostgreSQL.

AVG_SENSOR. Figure 7 presents the average execution time to compute the average temperature per sensor. DuckDB clearly outperforms PostgreSQL, achieving about 0.01 s at 10^5 rows, versus 0.027 s for PostgreSQL.

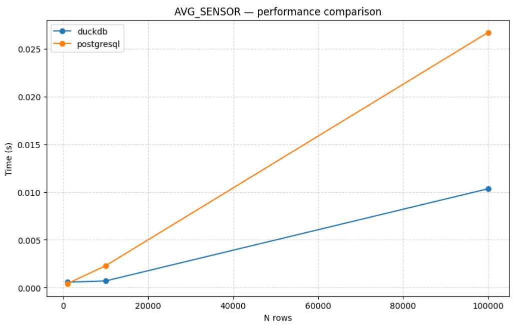

Figure 7: AVG_SENSOR — performance comparison.

HOT_SENSORS. The HOT_SENSORS query returns sensors whose average temperature exceeds a threshold and have enough readings. As seen in Figure 8, both systems scale linearly, but DuckDB is systematically faster than PostgreSQL.

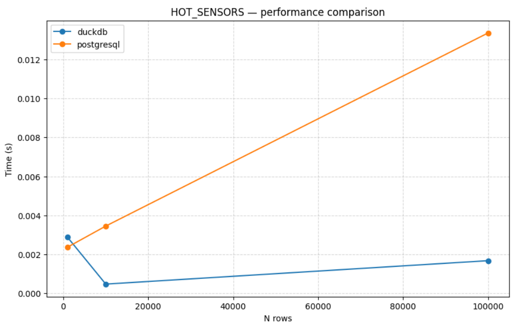

Figure 8: HOT_SENSORS — performance comparison.

OUTAGES. The OUTAGES phase detects large gaps in the time series using a LAG window function. Figure 9 shows that DuckDB scales from roughly 0.003 s to 0.014 s, while PostgreSQL increases from 0.001 s to about 0.040 s. Both are linear, but DuckDB has a smaller slope.

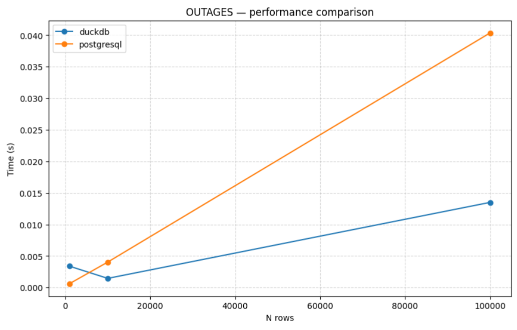

Figure 9: OUTAGES — performance comparison.

TOPK_SENSORS. For the TOPK_SENSORS query (top sensors by average temperature), Figure 10 confirms DuckDB's advantage: its execution time remains around three times lower than PostgreSQL, while both curves grow linearly with N.

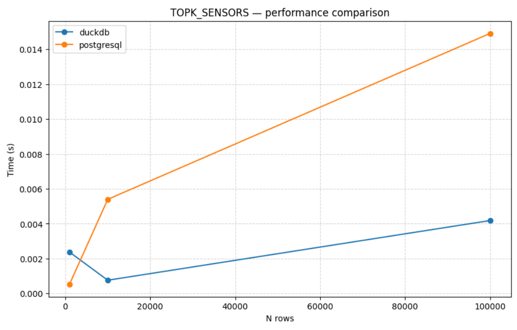

Figure 10: TOPK_SENSORS — performance comparison.

COMPARE_TWO. The COMPARE_TWO query computes the correlation between the temperatures of two sensors. Figure 11 shows that DuckDB remains fast ($\approx$ 0.005 s at 10^5), whereas PostgreSQL is considerably slower ($\approx$ 0.029 s).

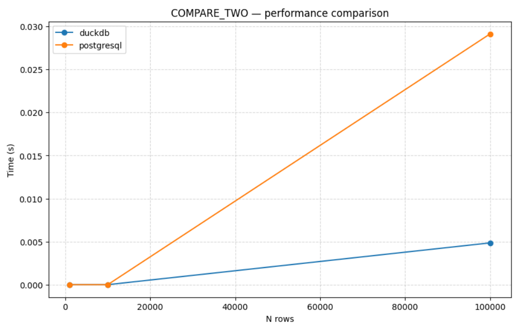

Figure 11: COMPARE_TWO — performance comparison.

DAILY_SUMMARY. Interestingly, Figure 12 shows that PostgreSQL is faster than DuckDB for the DAILY_SUMMARY aggregation. DuckDB reaches about 0.050 s at 10^5 rows, while PostgreSQL stays near 0.023 s. This can be explained by differences in how both engines materialise intermediate grouped results.

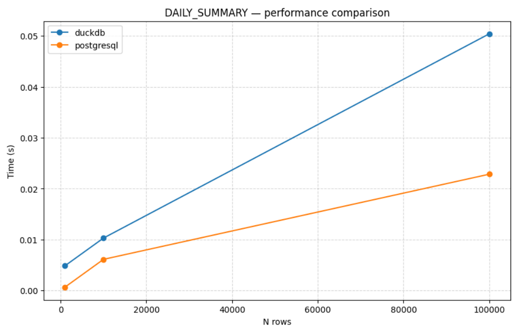

Figure 12: DAILY_SUMMARY — performance comparison.

FULL_PROFILE. Finally, Figure 13 presents the FULL_PROFILE workload. For DuckDB and PostgreSQL, this corresponds to the combined cost of several analytical queries (AVG_SENSOR, DAILY_SUMMARY, OUTAGES, TOPK_SENSORS), whereas BerkeleyDB executes a local statistical summary on its cached data. DuckDB provides the best analytical performance (0.012 s to 0.078 s), PostgreSQL is slower (0.018 s to 0.105 s), and BerkeleyDB is extremely fast for its limited local task (0.004 s to 0.030 s).

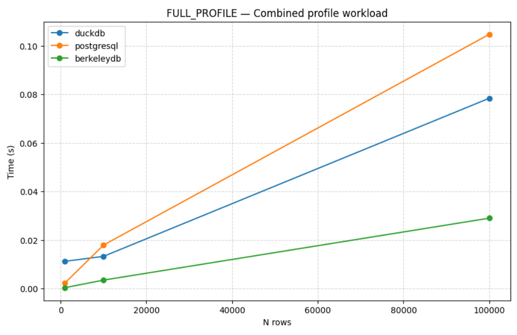

Figure 13: FULL_PROFILE — combined profile workload.

5 Discussion

The experimental results highlight a clear separation of responsibilities between the three engines tested. Each system exhibits strengths that are consistent with its architectural design.

5.1 BerkeleyDB

Strengths. BerkeleyDB offers outstanding performance for key–value access patterns. It provides near-constant-time READ, UPDATE, and DELETE operations, and extremely fast LOAD times even for 10^5 rows. Its footprint is small and it can be embedded directly in applications, making it an excellent candidate for resource-constrained IoT devices.

Weaknesses. BerkeleyDB does not support SQL and cannot execute analytical queries such as aggregations, joins, or window functions. All higher-level logic must be implemented in the application code. As a result, BerkeleyDB is unsuitable as a standalone analytical database and must be complemented by another engine for global analysis.

5.2 DuckDB

Strengths. DuckDB delivers the best performance for all analytical queries in our benchmark (ANOMALIES, AVG_SENSOR, HOT_SENSORS, OUTAGES, TOPK_SENSORS, COMPARE_TWO, and the analytical part of FULL_PROFILE). Its columnar storage and vectorised execution make it particularly well adapted to OLAP workloads on large sensor datasets, while still running as an embedded library without a separate server process.

Weaknesses. On the other hand, DuckDB performs extremely poorly on OLTP-like workloads: LOAD, UPDATE, DELETE and repeated point READs are orders of magnitude slower than with BerkeleyDB or even PostgreSQL. This is a direct consequence of its append-only storage and optimisation for scans rather than random writes. It is therefore not suitable for write-heavy or latency-sensitive transactional workloads.

5.3 PostgreSQL

Strengths. PostgreSQL provides balanced behaviour across workloads. It is much faster than DuckDB for UPDATE and DELETE operations, and its index-based execution plans offer reasonable performance for READ and LAST_READINGS. For analytics, it is systematically slower than DuckDB, but still scales linearly and remains within low tens of milliseconds for 10^5 rows. As a mature relational system, it also offers full transactional guarantees, rich indexing options and a robust client–server architecture.

Weaknesses. PostgreSQL is not optimised for columnar scans and cannot match DuckDB on pure analytical workloads. It also has a higher operational overhead (separate server process, configuration, memory usage) than embedded engines such as DuckDB or BerkeleyDB.

5.4 Overall assessment

Taken together, the results show that no single engine dominates all workloads. Instead, each engine is optimal for a different layer of the IoT architecture:

- **Device / edge layer:** BerkeleyDB is ideal as a local cache of recent readings, thanks to its extremely fast key–value access and tiny footprint.
- **Local analytics layer:** DuckDB is the best choice for complex analytical queries and batch processing over large sensor datasets.
- **Server / baseline layer:** PostgreSQL provides a robust, general-purpose relational backend and serves as a solid reference for comparison.

This validates the architectural choice made in our application: use BerkeleyDB to store the most recent sensor readings for low-latency access, and delegate heavier analytical workloads to DuckDB, while PostgreSQL acts as a traditional relational baseline. The benchmark confirms that embedded key–value technology is extremely well suited for the local IoT cache layer, but must be complemented by an analytical engine to satisfy the full set of application requirements.