

Adaptive Database Optimisation of Industrial IoT Workloads Using Machine Learning and Local Generative AI

Lakmali Shashika Karunarathne^{1*}, Nalinda Somasiri²

¹Lecturer, ²Associate Professor, Department of Computer and Data Science, York St John University London Campus, United Kingdom.

E-mail: ^{1*}l.karunarathne@yorksj.ac.uk, ²n.somasiri@yorksj.ac.uk

Abstract

Industrial Internet of Things (IIoT) systems generate sensor workloads that require scalable and workload-aware database management. This study evaluates an adaptive PostgreSQL optimisation framework combining conventional indexing, machine-learning (ML) query-latency prediction, local Generative AI (GenAI), and Hybrid ML–GenAI reasoning. A High-Rack Storage System dataset with 19,634 observations and 18 sensor variables was normalised into 353,412 atomic sensor readings and replayed at four scales up to 10,602,360 records. Eight SQL workload families were benchmarked under no-secondary-index, single-index, and composite-index configurations, producing 480 measured executions. A leakage-aware ML pipeline predicted candidate latency, while local Qwen3:4B generated indexing recommendations from SQL structure, database scale, and pre-execution plan information. Across eight held-out workloads, all 24 GenAI requests completed successfully with 100% recommendation stability. Conventional composite indexing, ML-only, GenAI-only, and Hybrid ML–GenAI each achieved a 20.044% mean improvement over the unindexed baseline, 50% exact agreement with the empirical oracle, and 7.499 ms mean regret. The results show that AI can support database optimisation, but added decision complexity did not outperform a strong conventional strategy within the constrained action space.

Keywords: Industrial Internet of Things, Database Optimisation, Machine Learning,

* Corresponding Author

Journal of IoT in Social, Mobile, Analytics, and Cloud, December 2026, Volume 8, Issue 4, Pages 324-344
DOI: <https://doi.org/10.36548/jismac.2026.4.001>

Received: 20.07.2026, received in revised form: 25.08.2026, accepted: 17.09.2026, published: 28.09.2026

© 2026 Inventive Research Organization. This is an open access article under the Creative Commons Attribution-NonCommercial International (CC BY-NC 4.0) License

Generative AI, Query Performance Prediction, Self-Tuning Databases.

1. Introduction

Heterogeneous measurements pertaining to the status of machinery, power, voltage, positions, and various other aspects are produced in IIoT environments constantly. This large amount of data coming from sensors supports analysis as well as monitoring and leads to the increased need for efficient processing and storing of information [1].

The efficiency of a database is highly dependent on the nature of the workloads and its design. Suitable indexing may decrease the response time while extra indexing will take up space and other resources; the effectiveness of automated indexing at scale proves both the potential as well as the difficulties of proper index choice [2].

Such problems led to the development of self-driving databases where systems monitor and adapt their configuration without extensive human interference. The architecture of self-driving DBMS encapsulates this concept by combining workload observation, prediction, and automation of tuning [3].

Machine learning extends this idea further by making use of past behaviors within the database itself. Learnable databases have been studied for configuration tuning, indexing, cardinality estimation, query optimization, and storage issues; demonstrating the power of predictions to supplement cost-based solutions [8].

Large Language Models (LLMs) and Generative Artificial Intelligence (GenAI), on the other hand, provide another useful capability - that of interpreting SQL syntaxes, schemas, execution plans and possible optimizations at the same time. Recent research shows that query rewriting, index design, parameter tuning and performance evaluation are some of the new LLM database use cases [9].

However, GenAI suggestions need to be evaluated empirically since although a logical reasoning might be correct, it does not mean it will yield the minimum latency; besides, inference with LLMs can be much more costly than relational execution [10]. This motivates testing whether quantitative ML guidance and contextual GenAI reasoning provide value beyond strong conventional database design.

The study evaluates this question using a PostgreSQL implementation of a High-Rack Storage System (HRSS) sensor workload. The source data are normalised and replayed at four controlled database scales; eight SQL families are benchmarked under three indexing strategies; the resulting measurements support leakage-aware ML prediction, local Qwen3:4B optimisation, and a Hybrid ML–GenAI decision stage.

The study addresses four research questions:

1. RQ1: How do conventional PostgreSQL indexing strategies affect query execution performance across different IIoT database scales and workload patterns?
2. RQ2: To what extent can machine learning predict database query execution performance for previously unseen SQL workload templates?
3. RQ3: Can a locally deployed Generative AI model generate effective and stable indexing recommendations using SQL workload and pre-execution PostgreSQL query-plan information?
4. RQ4: Does integrating ML-based performance prediction with GenAI-based contextual reasoning provide additional database-optimisation benefits compared with conventional indexing, ML-only and GenAI-only approaches?

Methodologically, RQ1 is addressed through 480 PostgreSQL benchmark executions spanning four database scales, three indexing strategies, and eight SQL workload families. RQ2 is addressed through leakage-aware latency modelling in which complete SQL templates are held out from training. RQ3 is addressed through 24 repeated local Qwen3:4B recommendations across eight held-out workloads. RQ4 is addressed by comparing conventional composite indexing, ML-only, GenAI-only, and Hybrid ML–GenAI selections against measured PostgreSQL outcomes and an empirical oracle. This design tests whether additional decision-model complexity creates measurable database benefit rather than assuming that it does.

2. Related Works

2.1 IIoT Data Management and Scalable Sensor Workloads

IIoT sensor repositories combine high-frequency generation with heterogeneous analytical demands. The HRSS data used here include normal, optimised, and anomalous operational runs and provide a realistic industrial basis for database workload construction [12]. Unlike prior work that primarily models production behaviour or anomaly states, this study transforms the HRSS measurements into a relational workload for controlled database scalability and physical-design experiments.

2.2 Autonomous and Self-Tuning Database Systems

Autonomous DBMS research treats configuration selection as a learning problem in which previous executions help identify influential settings and recommend configurations for new workloads [4]. Physical index selection is a particularly important component because gains depend on workload predicates and access paths. Prior adaptive-indexing research likewise shows that low-latency analytics benefit from workload-aware rather than indiscriminate indexing [11].

2.3 Machine Learning for Database Optimisation

Learned index research reframes key-to-location access as a predictive mapping, demonstrating that machine learning can augment traditional physical data structures [5]. Learned query optimisers extend this principle to plan selection by modelling relationships among queries, data distributions, and candidate plans [6]. Practical systems increasingly combine learned components with established optimisers rather than replacing them entirely; Bao exemplifies this approach by using learned guidance while retaining DBMS robustness [7].

2.4 Generative AI for Database Optimisation

Optimisation of DBMS using LLMs is a relatively new approach that encompasses SQL transformations, index design, parameter tweaking, and diagnostics. The flexible nature of this approach is appealing in that the SQL, schema, size of data, and plans can be considered within the same reasoning framework [9]. Flexibility, however, does not make the need for rigorous validation irrelevant. LLM-based optimisation consumes significant computing resources,

which is why this work considers local GenAI as an adviser to PostgreSQL execution [10].

3. Proposed Work

Figure 1 summarises the complete experimental workflow, linking source-data preparation and relational replay to PostgreSQL benchmarking, ML/GenAI recommendation, and final measured database evaluation.

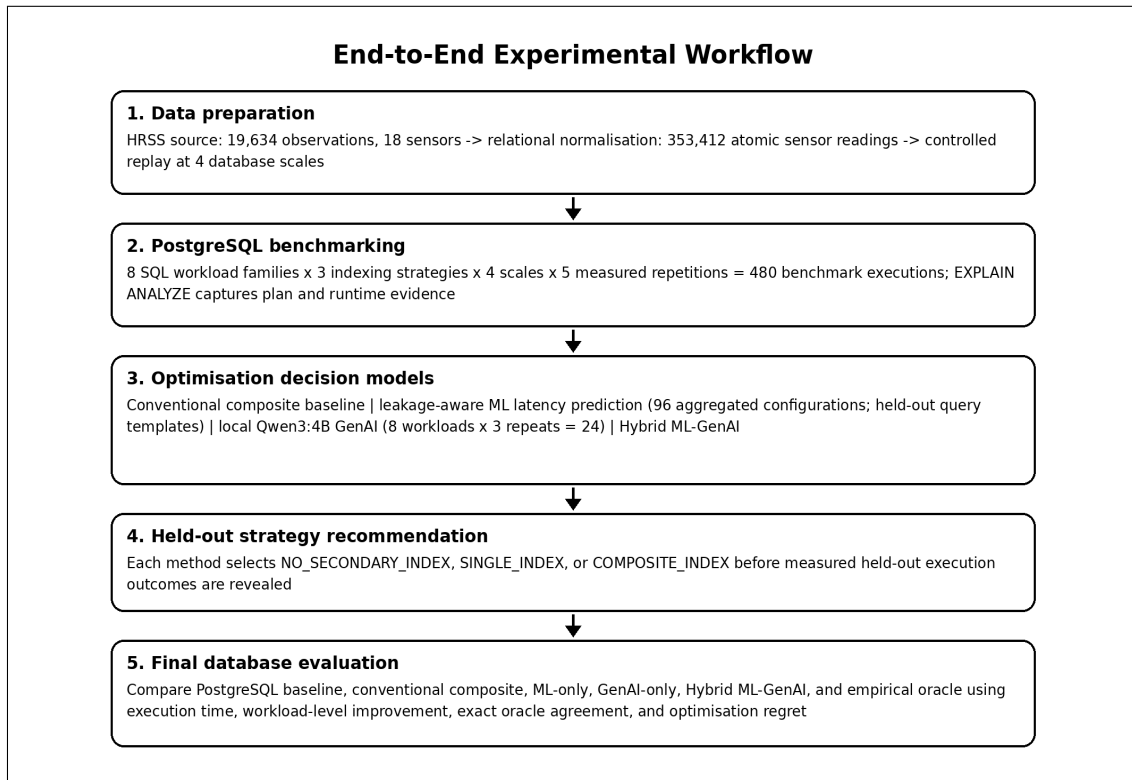


Figure 1. End-to-end workflow from HRSS data preparation to PostgreSQL benchmarking, ML/GenAI strategy recommendation, and final database evaluation

3.1 Dataset and Experimental Workload

The experiment employs HRSS_anomalous_optimized.csv found in the public High Storage System Data for Energy Optimization database; anomalous observations are marked using the provided label attribute [12]. This dataset comprises 19,634 rows, 20 attributes, and 18 sensor attributes associated with distance, power, and voltage of six elements. There are 15,117 normal observations and 4,517 anomalous observations (23.01%) with no missing values and 111 operational cycles. Table 1 provides an overview of these source data attributes.

Table 1. Characteristics of the HRSS IIoT dataset used in the experiment

Characteristic	Value
Source observations	19,634
Total source columns	20
Sensor variables	18
Operational components	6
Operational cycles	111
Normal observations	15,117
Anomalous observations	4,517
Anomaly proportion	23.01%
Missing values	0

The anomaly label is retained as operational context rather than treated as the primary research outcome.

3.2 Relational Normalisation

The source data use a wide format with 18 sensor measurements per timestamp. For database experimentation, each measurement was converted into an atomic long-format row, so 19,634 source observations produced 353,412 sensor-reading records as defined in Equation (1).

$$N_{readings} = N_{observations} \times N_{sensors}. \quad (1)$$

Therefore,

$$19,634 \times 18 = 353,412$$

sensor-reading rows were created.

The relational schema separates components, sensor definitions, cycles, and readings. Each fact row stores sensor identifier, cycle identifier, relative timestamp, value, and anomaly state; 353,412 therefore denotes database rows rather than independent statistical observations.

3.3 Controlled Database Replay and Scalability

The normalised table was replayed at four controlled scales to evaluate how increasing physical database size changes query and indexing behaviour. Table 2 lists the replay factors and resulting sensor-reading record counts.

Table 2. Controlled PostgreSQL database replay scales

Scale	Replay factor	Sensor-reading records
S1	1×	353,412
S2	3×	1,060,236
S3	10×	3,534,120
S4	30×	10,602,360

The larger tables are replay-based scalability workloads, not additional independent IIoT samples.

3.4 PostgreSQL Environment

Experiments used PostgreSQL 18.3 (64-bit) on Microsoft Windows x86-64 with the same schema and query workload across all scales. The experimental minor release was verified against the official PostgreSQL 18.3 release notes [13].

PostgreSQL EXPLAIN exposes estimated plans, while EXPLAIN ANALYZE executes the statement and reports actual runtime behaviour; the experiment follows the PostgreSQL 18 major-version documentation applicable to the 18.3 server build [14].

Table 3. PostgreSQL indexing strategies evaluated in the experiment

Strategy	Physical design
NO_SECONDARY_INDEX	Baseline configuration without additional secondary indexes
SINGLE_INDEX	Separate B-tree indexes on sensor_id, cycle_id, anomaly and relative_timestamp

COMPOSITE_INDEX	Workload-aware multicolumn indexes including (sensor_id, cycle_id, relative_timestamp), (anomaly, sensor_id) and (cycle_id, anomaly)
-----------------	--

Table 3 defines the three indexing strategies selected to create a controlled progression in physical-design complexity. NO_SECONDARY_INDEX provides the native PostgreSQL control condition without additional secondary structures. SINGLE_INDEX represents a conventional low-complexity design using independent B-tree indexes on the principal filter attributes. COMPOSITE_INDEX represents a stronger workload-aware conventional baseline whose column order is aligned with recurring multi-attribute predicates. The bounded three-strategy action space makes the effect of each physical-design level directly measurable while providing identical candidate actions to the ML and GenAI decision layers [15].

The principal composite index, (sensor_id, cycle_id, relative_timestamp), was aligned with common access predicates. PostgreSQL multicolumn B-tree indexes are most effective when restrictions correspond to leading columns, so the order was chosen from the observed query structures rather than arbitrarily [16]. These three strategies are controlled alternatives, not an exhaustive index-design search space.

3.5 SQL Workload Design

Eight SQL families were designed to cover temporal retrieval, anomaly and cycle aggregation, threshold filtering, joins, recent-reading retrieval, and combined predicates. Table 4 links each query identifier to its principal database operations.

Table 4. Representative IIoT SQL workload families

Query ID	Query type	Principal operations
Q1	Temporal Range	Sensor, cycle and temporal predicates; ordering
Q2	Anomaly Aggregation	Anomaly filtering; grouping by sensor
Q3	Cycle Aggregation	Cycle filtering; aggregate functions
Q4	Threshold Filtering	Sensor filtering and measurement threshold

Q5	Sensor Metadata Join	Fact-to-dimension join
Q6	Cycle Anomaly Summary	Cycle and anomaly aggregation
Q7	Latest Sensor Readings	Ordered recent-reading retrieval
Q8	Combined Filtering	Sensor and anomaly predicates

Together they expose filtering, selectivity, aggregation, grouping, ordering, and join behaviour across non-identical access patterns.

3.6 Benchmarking Procedure

Each of eight queries was evaluated at four scales under three indexing configurations, with five measured executions after warm-up, giving 480 benchmark executions as shown in Equation (2).

$$N_{benchmark} = 4 \times 3 \times 8 \times 5 = 480. \quad (2)$$

EXPLAIN ANALYZE supplied measured execution information and planner output. Repetitions were then aggregated by scale, query, and strategy to produce the 96 configuration-level observations in Equation (3).

$$N_{configuration} = 4 \times 8 \times 3 = 96 \quad (3)$$

Aggregation prevents repeated measurements of the same database configuration from being treated as independent ML samples.

3.7 Machine-Learning Query-Latency Modelling

The ML stage used only pre-execution information: record count, structural query flags, predicate count, table and index size, estimated plan cost, estimated rows, intended index use, strategy, query type, complexity, and plan-node information.

Post-execution latency and runtime buffer statistics were excluded to reduce leakage. Candidate regressors were Dummy, Ridge, Random Forest, and XGBoost; the latter was included for its scalable gradient-boosted modelling of nonlinear feature interactions [17].

Categorical variables were one-hot encoded and the target was log1p-transformed. Evaluation held out complete Query_ID templates, preventing scale or index variants of the

same SQL family from appearing in both training and testing. The lowest predicted candidate latency determined the ML-only recommendation.

3.8 Local Generative-AI Optimisation

Local GenAI used Qwen3:4B through Ollama with thinking disabled, temperature 0, a 300-token response limit, and a 4,096-token context window [18].

For each held-out workload, the prompt contained schema, scale, SQL, parameters, workload characteristics, pre-execution plan information, and the three candidate strategies. True measured latency, empirical optimum, and ML predictions were withheld. Eight workloads were evaluated three times each, producing the 24 recommendations in Equation (4).

$$N_{GenAI} = 8 \times 3 = 24. \quad (4)$$

A workload-level consensus strategy was derived from the three repeated recommendations.

3.9 Hybrid ML–GenAI Optimisation

The Hybrid stage added ML-predicted candidate latencies and rankings to the GenAI context while still withholding measured held-out outcomes. The same three-repeat protocol was applied to the eight workloads.

3.10 Evaluation Metrics

Execution-time improvement relative to the no-secondary-index baseline was calculated as for the adaptive comparisons, Equation (5) is applied separately to each of the eight held-out workloads and the resulting percentages are then averaged. Accordingly, the reported 20.044% is a mean of workload-level percentage improvements; it is not calculated from the two grand mean execution times.

$$Improvement(\%) = \frac{T_b - T_s}{T_b} \times 100, \quad (5)$$

where T_b is baseline execution time and T_s is execution time under the selected strategy.

Exact strategy agreement with the empirical oracle was calculated as

$$Agreement(\%) = \frac{N_{matching}}{N_{workloads}} \times 100, \quad (6)$$

where $N_{matching}$ is the number of recommendations matching the empirically fastest

strategy.

Absolute optimisation regret was defined as

$$Regret_{ms} = T_s - T_o, \quad (7)$$

where T_o is execution time under the empirical optimum. Percentage regret was calculated as

$$Regret(\%) = \frac{T_s - T_o}{T_o} \times 100. \quad (8)$$

Recommendation stability was calculated as

$$Stability(\%) = \frac{N_{consensus}}{N_{successful}} \times 100, \quad (9)$$

where $N_{consensus}$ is the number of repeated recommendations agreeing with the workload consensus.

4. Results and Discussion

4.1 Conventional PostgreSQL Indexing Performance

Indexing effectiveness varied substantially by query semantics and database scale: selective temporal and aggregation workloads benefited strongly, while join-oriented and broader workloads showed smaller or occasionally negative gains.

4.1.1 Temporal-Range Scalability

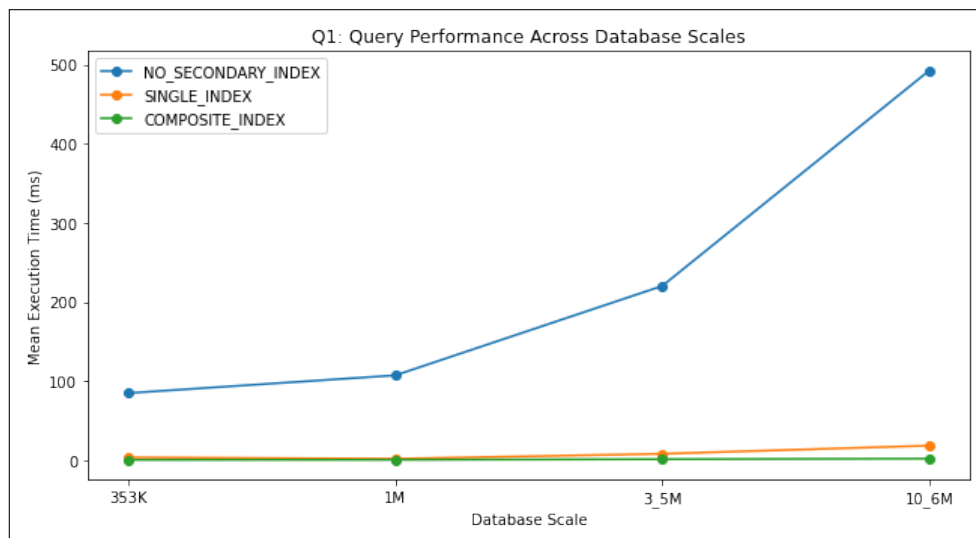


Figure 2. Execution-time scalability of the Q1 temporal-range workload under no-secondary-index, single-index and composite-index configurations

Figure 2 shows that Q1 was highly sensitive to physical access path. At 10,602,360 records, mean execution time fell from 471.479 ms without secondary indexes to 2.267 ms with the composite configuration, a 99.519% improvement. The result reflects close alignment between the query predicates and the leading columns of the multicolumn index [16].

4.1.2 Anomaly-Aggregation Scalability

Figure 3 shows that for Q2, composite indexing reduced mean execution time at the largest scale from 608.714 to 294.549 ms, a 51.611% improvement. The (anomaly, sensor_id) structure supports filtering and grouping, although substantial aggregation work remains after row selection.

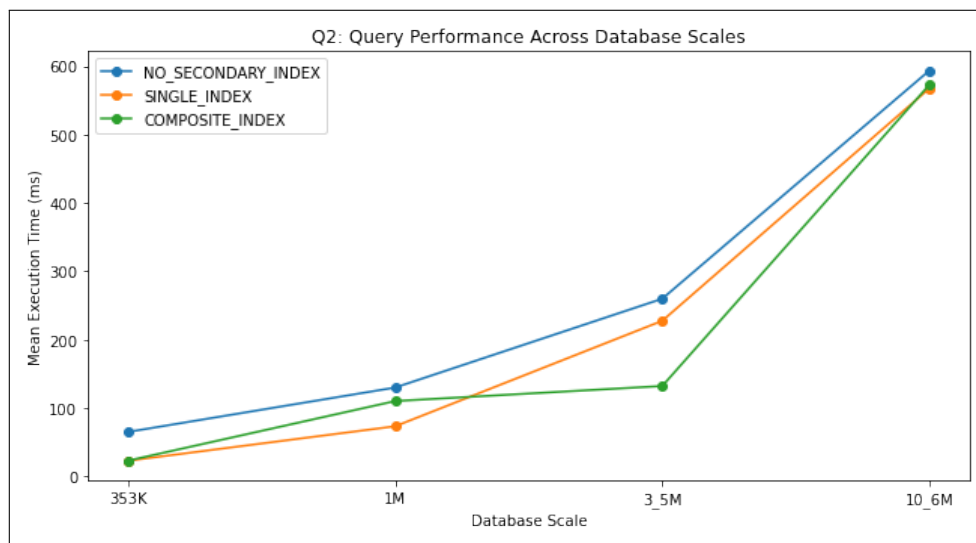


Figure 3. Execution-time scalability of the Q2 anomaly-aggregation workload across PostgreSQL indexing strategies

4.1.3 Cycle Aggregation and Threshold Filtering

At 10,602,360 records, Q3 decreased from 470.582 to 36.255 ms (92.296%), while Q4 decreased from 474.537 to 142.511 ms (69.968%). The results confirm that workload-aligned filtering can greatly reduce downstream processing.

Table 5. Selected composite-index performance at the 10,602,360-record database scale

Query	Workload	Baseline (ms)	Composite (ms)	Improvement (%)
Q1	Temporal Range	471.479	2.267	99.519

Q2	Anomaly Aggregation	608.714	294.549	51.611
Q3	Cycle Aggregation	470.582	36.255	92.296
Q4	Threshold Filtering	474.537	142.511	69.968
Q5	Sensor Metadata Join	903.982	832.798	7.874

Table 5 summarises the wide variation in composite-index benefit at the largest database scale.

4.1.4 Limited and Negative Indexing Benefit

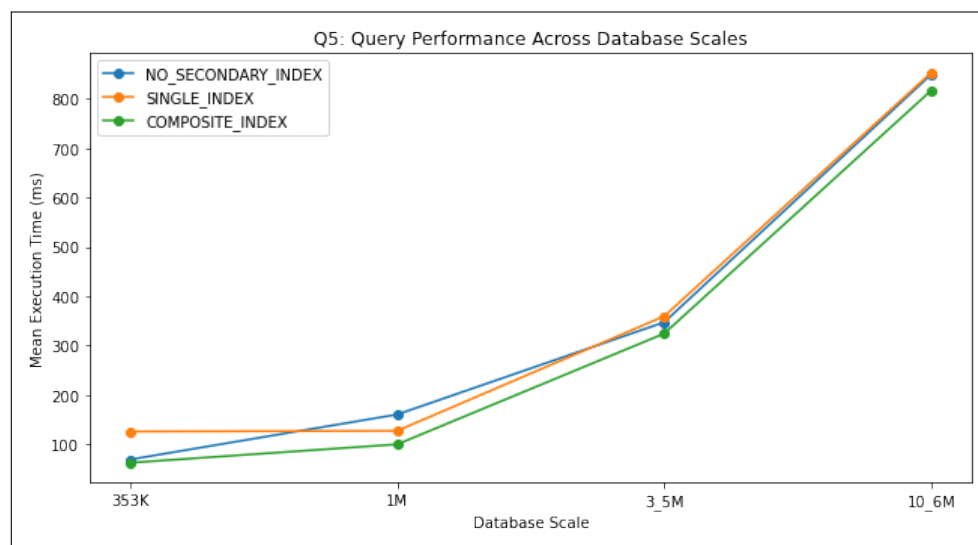


Figure 4. Execution-time scalability of the Q5 sensor-metadata join workload

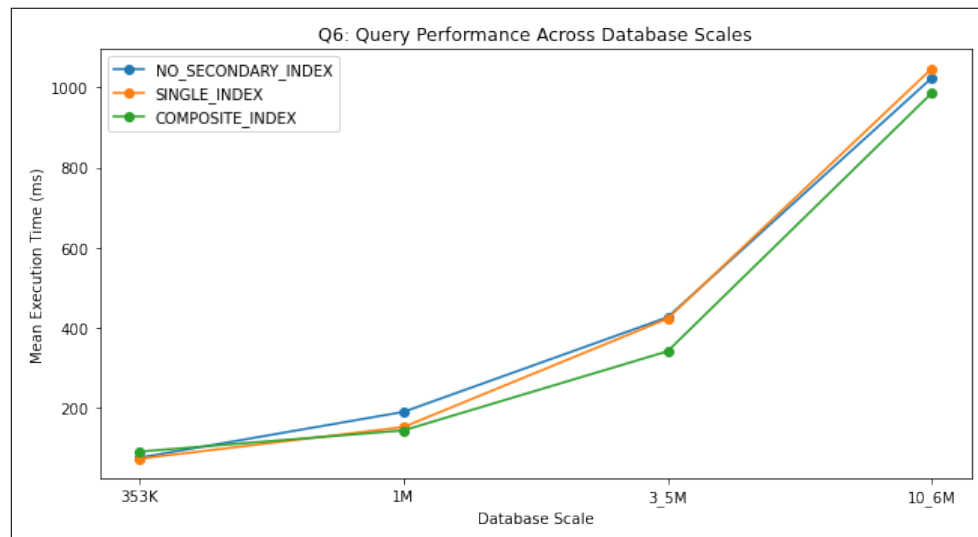


Figure 5. Execution-time scalability of the Q6 cycle-anomaly summary workload

Figures 4 and 5 provide counterexamples. Q5 improved only 7.874% at 10.6 million records (903.982 to 832.798 ms), indicating that join and downstream processing dominated. For Q6 at 3,534,120 records, a single index increased latency from 382.635 to 389.629 ms (-1.828%), showing that additional indexing is not uniformly beneficial.

4.1.5 Index-Storage Overhead

At 10,602,360 records, the composite configuration occupied approximately 869.523 MB. The performance gain therefore carries storage and maintenance cost, reinforcing workload-aware index selection [15].

4.2 ML-Guided Database Optimisation

The ML stage used 96 aggregated scale-query-strategy observations and held out complete SQL templates. For each held-out workload, candidate latencies were predicted for all three indexing strategies before any measured test latency was consulted.

Across eight held-out workloads, ML-only selection produced 300.104 ms mean execution time versus 345.597 ms for the unindexed baseline, a 20.044% mean improvement. Exact oracle agreement was 50.00%, mean regret 7.499 ms, and maximum regret 36.894 ms.

Thus, ML guidance reduced latency but did not consistently identify the fastest strategy. The small configuration-level training set remains an important limitation despite the stricter query-template split.

4.3 Local GenAI Database Optimisation

Local Qwen3:4B was evaluated independently of ML predictions on eight workloads with three repetitions each; all 24 requests completed successfully.

4.3.1 Recommendation Stability

Figure 6 shows identical recommendations across all three repetitions for every workload, giving 100% mean and minimum stability and 0.950 mean confidence. Exact oracle agreement was nevertheless 50%, demonstrating that consistent LLM recommendations are not equivalent to empirically optimal database decisions [9].

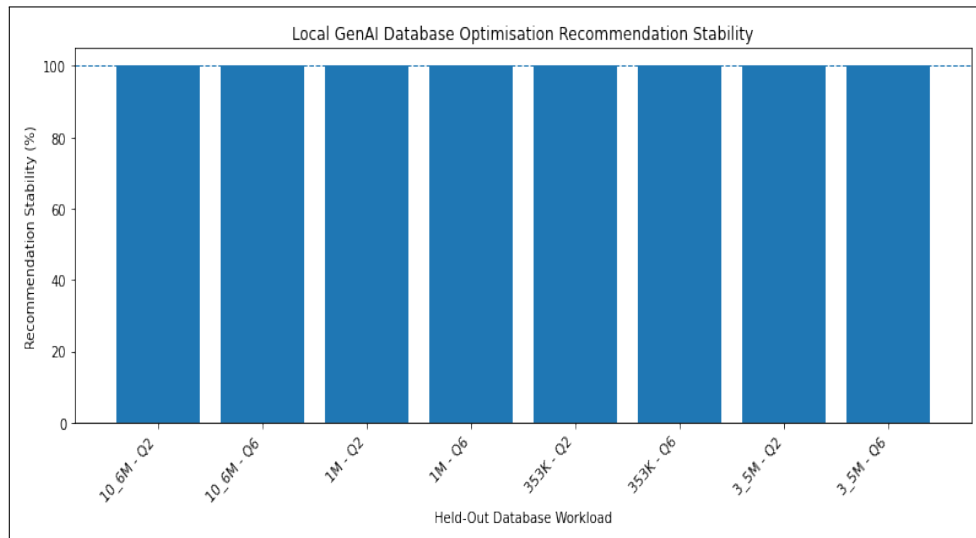


Figure 6. Recommendation stability of the local Qwen3:4B optimiser across the eight held-out IIoT database workloads

4.3.2 Workload-Level GenAI Improvement

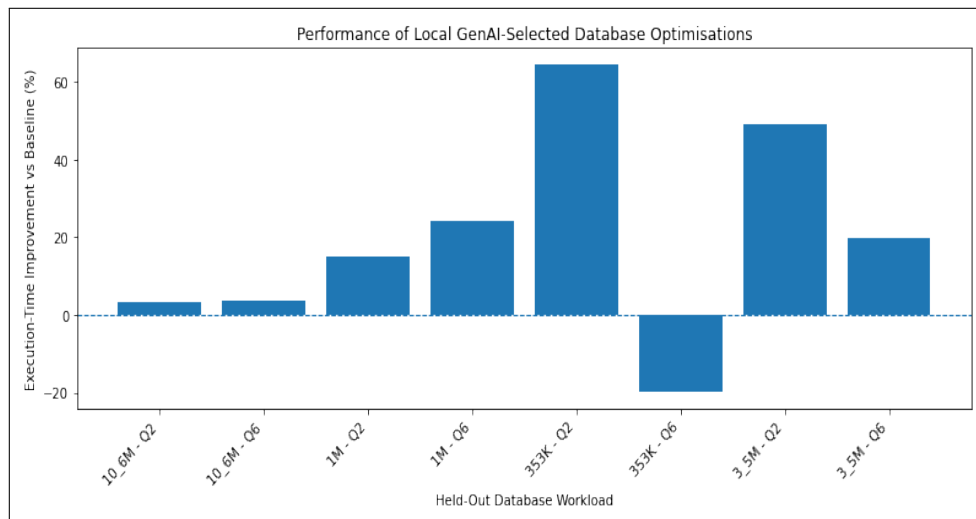


Figure 7. Execution-time improvement achieved by local GenAI-selected database strategies relative to the PostgreSQL baseline

Figure 7 presents the workload-level effects of the GenAI selections. The mean workload-level improvement was 20.044%; the median was 17.563%, the maximum 64.532%, and the minimum -19.911%. The negative case confirms that a syntactically valid recommendation can still degrade measured DBMS performance.

4.3.3 GenAI Optimisation Regret

Figure 8 reports optimisation regret relative to the empirical optimum. Mean GenAI regret was 7.499 ms, median regret 0.066 ms, and maximum regret 36.894 ms. The low median

shows that several non-oracle choices were practically close to the fastest strategy, so regret complements exact strategy agreement.

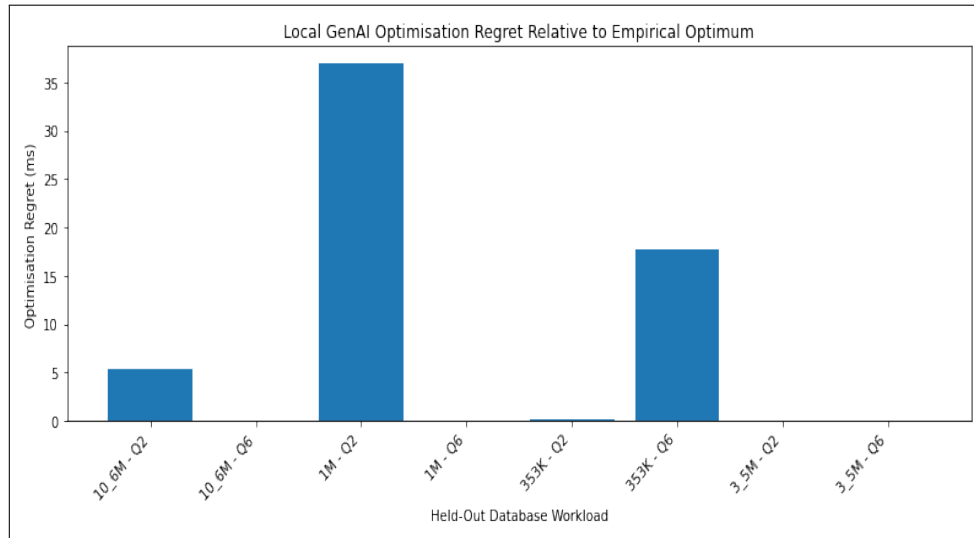


Figure 8. Optimisation regret of local GenAI-selected strategies relative to the empirically fastest indexing configuration

4.3.4 Local GenAI Inference Overhead

Mean local inference time was 21.045 s (median 21.080 s), far exceeding individual SQL execution latency. Local GenAI is therefore more suitable for periodic workload-level tuning than per-query invocation [10].

4.4 Hybrid ML–GenAI Optimisation

The Hybrid stage supplied ML-predicted candidate latencies to the GenAI prompt while continuing to hide measured held-out outcomes.

Hybrid ML–GenAI produced 300.104 ms mean execution time, 20.044% mean improvement, 50.00% oracle agreement, 7.499 ms mean regret, and 36.894 ms maximum regret—identical to ML-only and GenAI-only aggregate results.

Hybridisation therefore provided no additional measurable benefit within the evaluated action space; this is a constraint-specific negative finding, not evidence that hybrid optimisation is universally ineffective.

4.5 Comparative Evaluation

Figure 9 and Table 6 show the final held-out comparison. The unindexed baseline has a grand mean execution time of 345.597 ms, whereas conventional composite indexing, ML-only,

GenAI-only, and Hybrid ML–GenAI each have a grand mean of 300.104 ms. The reported 20.044% is the arithmetic mean of the eight workload-specific percentage improvements defined by Equation (5). Because the workloads have unequal baseline latencies, calculating a percentage directly from the two grand means gives 13.164%; this ratio-of-means is a different summary statistic and is not the primary improvement metric used in the study.

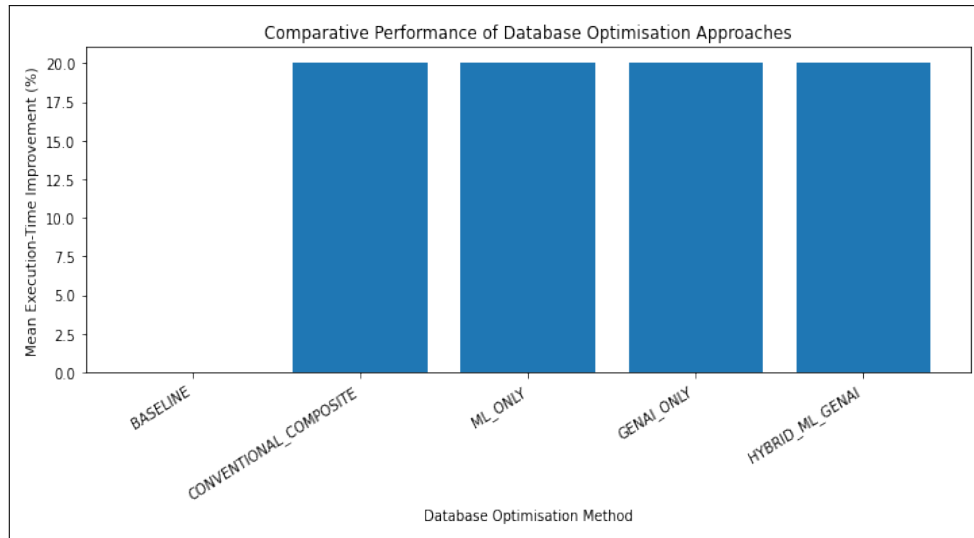


Figure 9. Comparative execution-time improvement of conventional and AI-assisted database-optimisation approaches across the held-out IIoT workloads

Table 6. Comparative performance of database-optimisation approaches across held-out workloads

Optimisation method	Mean execution time (ms)	Mean workload-level improvement (%)	Exact oracle agreement (%)	Mean regret (ms)	Maximum regret (ms)
PostgreSQL baseline	345.597	0.000	0.0	52.992	127.724
Conventional composite	300.104	20.044	50.0	7.499	36.894
ML-only	300.104	20.044	50.0	7.499	36.894
GenAI-only	300.104	20.044	50.0	7.499	36.894

Hybrid ML–GenAI	300.104	20.044	50.0	7.499	36.894
Empirical oracle	292.605	26.639	100.0	0.000	0.000

The oracle therefore retained a further 6.595 percentage points of improvement, but the principal result is the convergence of all four practical optimisation approaches.

Evaluating the AI methods only against the unindexed baseline would incorrectly attribute the full 20.044% gain to AI. The conventional composite baseline demonstrates that the available physical-design action space, not decision-model sophistication alone, drove much of the improvement.

4.6 Discussion of Findings

4.6.1 Workload Dependence of Physical Database Design

The conventional results confirm that physical database design is workload dependent. Q1 benefited strongly because the composite index aligned with selective equality and temporal-range predicates, whereas Q5 gained little and Q6 exhibited a small degradation. These outcomes support evaluating index recommendations against measured workload behaviour rather than assuming that additional access structures are beneficial [2].

4.6.2 ML as a Database-Optimisation Decision Aid

ML results indicate that prior execution information can help design the physical structure, but 50% agreement with the oracle is indicative of a small workload for training as well. While regret complements the exact agreement, it is important to note that the non-oracle approach can still suffer minimal latency loss; the difference is crucial when there are many equivalent database designs in terms of execution cost.

4.6.3 Stable GenAI Reasoning Is Not Equivalent to Optimality

The Qwen3:4B system generated perfectly consistent repeated recommendations yet achieved only 50% exact oracle agreement. Highly confident large language models should not be seen as evidence for optimality of physical database; instead, validation of DBMS is still required [9].

4.6.4 Deployment and Hybrid Implications

The time taken for local inference of about 21 seconds means that the system cannot support synchronous per-query inference and needs either periodic or event-triggered tuning. The reason why hybrid ML-GenAI model was unable to beat the simpler approaches is that the various algorithms chose among only three techniques that were pre-defined, one of which, composite indexing, was already very efficient.

4.7 Limitations

Some of the limitations of this experiment include that it is based on eight SQL families, ninety-six machine learning configuration evaluations, eight adaptors, one PostgreSQL setting, and one local large language model. The action space of three strategies is deliberately constrained and may have been a key factor in reaching convergence. Database scaling occurs through controlled replay and not further independent sensor samples. Latency is the key optimization goal; meanwhile, write cost, index build time, memory, energy, and maintenance cost are not all optimized at once.

5. Conclusion

In this research, standard PostgreSQL indexing, leakage-aware ML guidance, local GenAI, and Hybrid ML-GenAI were considered for tuning IIoT sensor workloads with cardinality ranging from 353,412 to 10,602,360. Eight SQL families and three indexing schemes led to 480 benchmark instances and held-out adaptive evaluation. Standard indexing showed highly workload-dependent results: there were improvements for Q1 by 99.519%, but some workloads showed little or negative effect from indexing. In adaptive evaluation, standard composite indexing, ML-only, GenAI-only, and Hybrid ML-GenAI all gave the same 20.044% average workload improvement, 50% oracle accuracy, and 7.499 ms average regret; the empirical oracle achieved 26.639%. The primary finding of the research is that AI can help in database tuning, but increased decision space did not outperform the standard approach within limited actions space. The model Qwen3:4B was always stable for recommendations, but neither stability nor confidence implied empirical optimization. Further research may be focused on larger decision space of index design, other DBMS and deployment settings, and joint latency-storage-maintenance optimization.

References

- [1] ur Rehman, M. H., I. Yaqoob, K. Salah, M. Imran, P. P. Jayaraman, and C. Perera. “The Role of Big Data Analytics in Industrial Internet of Things.” *Future Generation Computer Systems* 99 (2019): 247–259.
- [2] Das, S., M. Grbic, I. Ilic, I. Jovandic, A. Jovanovic, V. R. Narasayya, M. Radulovic, M. Stikic, G. Xu, and S. Chaudhuri. “Automatically Indexing Millions of Databases in Microsoft Azure SQL Database.” In *Proceedings of the 2019 International Conference on Management of Data*, Association for Computing Machinery, 2019, 666–679.
- [3] Pavlo, A., G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. Van Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. “Self-Driving Database Management Systems.” In *8th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2017.
- [4] Van Aken, D., A. Pavlo, G. J. Gordon, and B. Zhang. “Automatic Database Management System Tuning through Large-Scale Machine Learning.” In *Proceedings of the 2017 ACM International Conference on Management of Data*, Association for Computing Machinery, 2017, 1009–1024.
- [5] Kraska, T., A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. “The Case for Learned Index Structures.” In *Proceedings of the 2018 International Conference on Management of Data*, Association for Computing Machinery, 2018, 489–504.
- [6] Marcus, R., P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. “Neo: A Learned Query Optimizer.” *Proceedings of the VLDB Endowment* 12, no. 11 (2019): 1705–1718.
- [7] Marcus, R., P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. “Bao: Making Learned Query Optimization Practical.” In *Proceedings of the 2021 International Conference on Management of Data*, Association for Computing Machinery, 2021, 1275–1288.
- [8] Zou, B., J. You, Q. Wang, X. Wen, and L. Jia. “Survey on Learnable Databases: A Machine Learning Perspective.” *Big Data Research* 27 (2022): 100304.
- [9] Bagui, S., M. Malagutti, R. Morelli, and M. Tamascelli. “Large Language Models in

- Database Management System Optimization: A Survey.” *ACM Transactions on Intelligent Systems and Technology* 17, no. 5 (2026): 1–37.
- [10] Zhang, X., K. Khedri, and R. Rawassizadeh. “Can LLMs Substitute SQL? Comparing Resource Utilization of Querying LLMs versus Traditional Relational Databases.” In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, Volume 4: Student Research Workshop*, Association for Computational Linguistics, 2024, 465–472.
- [11] Karunarathne, L., S. Ganesan, K. Karunarathne, and N. Somasiri. “Database Optimization for Low-Latency Analytics with Adaptive Indexing.” *Journal of Data Science and Intelligent Systems* (2026).
- [12] inIT – Institute Industrial IT and Ostwestfalen-Lippe University of Applied Sciences. High Storage System Data for Energy Optimization. Kaggle dataset, 2018. CC BY-NC-SA 4.0. <https://www.kaggle.com/datasets/inIT-OWL/high-storage-system-data-for-energy-optimization>.
- [13] PostgreSQL Global Development Group. “PostgreSQL 18.3 Release Notes.” 2026. <https://www.postgresql.org/docs/release/18.3/>.
- [14] PostgreSQL Global Development Group. “PostgreSQL 18 Documentation: Using EXPLAIN.” 2026. <https://www.postgresql.org/docs/18/using-explain.html>.
- [15] PostgreSQL Global Development Group. “PostgreSQL 18 Documentation: Chapter 11 – Indexes.” 2026. <https://www.postgresql.org/docs/18/indexes.html>.
- [16] PostgreSQL Global Development Group. “PostgreSQL 18 Documentation: Multicolumn Indexes.” 2026. <https://www.postgresql.org/docs/18/indexes-multicolumn.html>.
- [17] Chen, T., and C. Guestrin. “XGBoost: A Scalable Tree Boosting System.” In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Association for Computing Machinery, 2016, 785–794.
- [18] Yang, A., A. Li, B. Yang, et al. “Qwen3 Technical Report.” arXiv:2505.09388 (2025). <https://arxiv.org/abs/2505.09388>.