Journal of Computing and Data Technology
Vol. 01, No. 01, June 2025, pp. 1-9.
DOI: https://doi.org/10.71426/jcdt.v1.i1.pp1-9

Homepage: https://review.journal-of-modern-technology.com/index.php/jcdt/index

Comparative Study of Join Algorithms in MySQL: Cross, Inner, Outer, and Self Joins

Adhuresa Hajdini[1*], Lozarta Fazliu[2], Dea Gjoshi[3]

[1*] Department of Computing and Information Technologies, Rochester Institute of Technology, Prishtina, Kosovo, Email: adhuresah@auk.org & adhuresahajdini25@gmail.com , ORCID: https://orcid.org/0009-0007-3890-4856
[2] Department of Computing and Information Technologies, Rochester Institute of Technology, Prishtina, Kosovo, Email: lozartef@auk.org , ORCID: https://orcid.org/0009-0002-9128-4864
[3] Department of Computing and Information Technologies, Rochester Institute of Technology, Prishtina, Kosovo, Email: deag1@auk.org , ORCID: https://orcid.org/0009-0001-0076-8381

Article Info

Article history:

Received: May 20, 2025
Revised: June 6, 2025
Accepted: June 9, 2025
First Online: June 9, 2025

Keywords:

Structured Query Language (SQL)
SQL joins
MySQL Performance
Query optimization
Database

ABSTRACT

A Structured Query Language (SQL) joins are fundamental to relational database operations, merging data from multiple tables into coherent result sets. This comparative study examines the performance characteristics of cross joins, inner joins, left/right/full outer joins, and self-joins in MySQL under large-scale workloads. Experiments were conducted on datasets ranging from 10,000 to 50 million rows using execution time, CPU utilization, memory consumption, and scalability as metrics. Results demonstrate inner joins achieve superior performance when properly indexed; outer joins incur moderate overhead for handling unmatched rows; self-joins perform comparably to inner joins with effective indexing but degrade under multi-level recursion; cross joins exhibit exponential growth and should be used sparingly. Recommendations for index strategies and query design are provided.

Corresponding Author:

Email address of corresponding author: adhuresah@auk.org

1. INTRODUCTION

In today's data-driven world, relational database management systems (RDBMS) form the backbone of a vast array of applications, from e-commerce platforms and online transaction processing (OLTP) systems to large-scale data warehouses and real-time analytics engines [2], [3]. At the heart of these systems lies the join operation, which allows disparate tables to be combined based on defined relationships—enabling everything from straightforward lookups (e.g., retrieving customer details for each order) to complex multi-table aggregations (e.g., computing ad-hoc sales metrics across geographies and time windows) [4]. Because join execution often dominates total query cost, understanding and optimizing join performance is critical for achieving the sub-second response times expected by modern users and meeting the throughput demands of enterprise workloads [1], [5].

MySQL 8.0, one of the world's most widely deployed open-source RDBMSs, powers thousands of web applications and microservices, serving billions of rows of data daily under a variety of load profiles. Its default join strategies—nested-loop joins (NLJ) and batched key-access (BKA) joins—are battle-tested for mixed OLTP workloads, but they

also introduce performance pitfalls on large, highly skewed, or recursively joined datasets [2], [6]. In addition, MySQL's lack of a native full-outer-join operator requires emulation via UNION of left and right joins, incurring additional sort and deduplication overheads that can overwhelm server resources at scale [7]. Meanwhile, self-joins—used to traverse hierarchical structures such as organizational trees or bill-of-materials graphs—can incur multiple scan passes unless carefully rewritten using Common Table Expressions (CTEs) or materialized-path techniques [8]. Finally, cross joins, which compute full Cartesian products, can generate result sets whose size grows multiplicatively with input table sizes, rendering them impractical for anything but small lookup tables or sampling scenarios [1], [9].

Although academic research has proposed numerous alternative join algorithms—such as hash joins, sort-merge joins, worst-case-optimal joins (WCOJs), and learned-index–augmented joins—the incorporation of these into MySQL's core optimizer has been slow, and many remain available only as experimental plugins or third-party extensions [11], [13], [15]-[17]. Meanwhile, MySQL 8.0 has introduced extended histogram statistics, invisible indexes, and optimizer trace enhancements to improve join ordering and cardinality estimation, yet type-specific overheads (e.g., NULL padding for outer joins, multiple passes for self-joins) persist [10]. Practitioners and database administrators therefore face a pressing need for empirically grounded guidance on choosing and tuning join types under realistic data scales.

This paper addresses that need through a comprehensive performance analysis of MySQL 8.0's join types—inner, left outer, right outer, full outer (via UNION), self-join, and cross join—under datasets ranging from 10 000 to 50 000 000 rows. Our contributions include:

1. *Empirical Benchmarking Across Scales:* We measure execution time, CPU utilization, and memory consumption at five data-scale points (10 M, 20 M, 30 M, 40 M, and 50 M rows), under both cold-cache and warm-cache scenarios, to reveal how each join type scales with growing data volume.
2. *Resource Profiling:* Using system-level sampling and database instrumentation, we quantify the resource overheads unique to each join type—highlighting, for example, the memory footprint of buffering NULL-filled rows in outer joins and the CPU saturation induced by Cartesian-product generation in cross joins.
3. *Scalability Analysis:* By plotting execution-time growth curves and fitting linear vs. exponential models, we demonstrate that while inner, outer, and self-joins exhibit near-linear or mildly super-linear scaling, cross joins follow an exponential trajectory that constrains their use to small tables [1], [9].
4. *Practical Recommendations:* Drawing on our findings, we propose concrete indexing strategies (e.g., composite covering indexes for multi-column joins [5]), query-rewriting techniques (e.g., staging tables for full outer joins [7]), and hierarchical-query alternatives (e.g., CTEs or materialized paths for deep recursion [8], [11]).
5. *Outlook on Emerging Features.* We evaluate preliminary hash-join implementations in MySQL's experimental plugin, and discuss the potential role of worst-case-optimal and learned-index join algorithms for future MySQL releases [11], [13], [15]-[17].

The remainder of this Introduction elaborates on the SQL join landscape and the specific challenges in MySQL 8.0. Section 2 reviews prior literature on join algorithms and optimizer enhancements. Section 3 details our methodology, including hardware setup, dataset generation, and workload definitions. Section 4 presents our benchmark results, with detailed plots and resource profiles. Section 5 discusses key takeaways and Section 6 offers best-practice recommendations. We conclude Section 7 with a forward-looking perspective on join algorithm evolution in MySQL and other open-source RDBMSs.

1. 1 The Role of Joins in Modern Workloads

In OLTP applications—such as order-entry systems, content management platforms, and customer relationship management (CRM) tools—joins are ubiquitous. A simple checkout transaction may involve joining a "Cart" table with "Inventory," "Pricing," and "User" tables to validate stock availability, compute totals, and log user details [4]. As transaction rates climb into the tens of thousands per second, join performance directly influences system throughput and latency. Meanwhile, in analytical and data-warehouse contexts—running nightly ETL (Extract-Transform-Load) or ad-hoc reporting queries—joins across large fact and dimension tables (e.g., Orders × Customer × Product) can span billions of rows, taxing disk I/O, memory, and CPU resources [12], [13].

Within MySQL's execution engine, nested-loop joins operate by scanning one table (the "outer" table) and, for each outer row, probing an index on the other table (the "inner" table). When the inner table's join column is indexed, each probe requires O(log N) time, yielding an overall cost of O(M × log N), where M and N are the outer- and inner-table sizes, respectively [2], [5]. Batched key-access joins improve performance by grouping multiple outer rows into batch probes, reducing per-probe overhead. Nonetheless, for large N, these methods still incur significant

I/O and CPU costs, motivating research into hash-based and merge-based join algorithms [11], [13]. MySQL join classification scheme is shown in Figure 1.

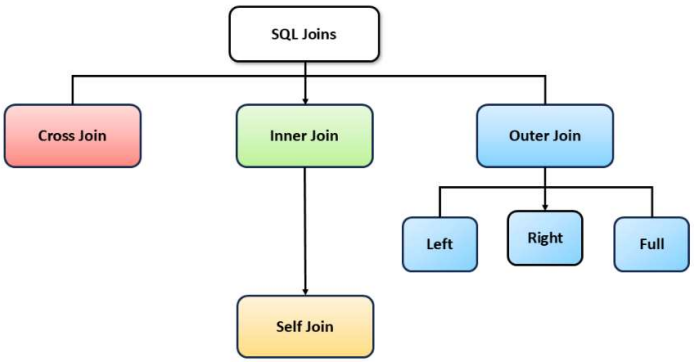

Figure 1: Classification of MySQL joins.

1. 2 Join Types and their MySQL Implementations

Inner Join- Returns only matching rows; can leverage equi-join indexes efficiently [5]. Left/Right Outer Join- Preserves unmatched rows on one side by padding NULLs; implemented via additional scan passes and buffer allocation [6]. Full Outer Join- Not natively supported; emulated via LEFT JOIN UNION RIGHT JOIN, doubling scan and sort work [7]. Self-Join- Joins a table to itself; useful for hierarchical data but prone to recursive passes unless optimized with CTEs or adjacency lists [8]. Cross Join- Produces Cartesian product, yielding $|A|\times|B|$ rows; often unintentional and extremely expensive at scale [1], [9]. Each join type thus carries distinct performance characteristics and optimization opportunities. Understanding these trade-offs is crucial for database architects who must design schemas and queries to meet both functional and performance requirements.

1. 3 Challenges in MySQL 8.0

While MySQL 8.0 has made strides in optimizer intelligence—through extended histograms, multi-value indexes, and optimizer hints—it still lacks native support for hash- and merge-join operators in its core execution engine [2], [10]. Its full-outer-join emulation remains cumbersome, and recursive queries via CTEs, introduced in 8.0, do not yet support "MATERIALIZED" semantics that would flatten recursion into a single pass [8]. These limitations leave DBAs reliant on traditional NLJ and BKA methods, making it imperative to characterize exactly how they perform under real-world, large-scale scenarios. By systematically benchmarking these join types across realistic dataset scales and cache states, our work provides the missing empirical evidence needed to guide schema design, index strategy, and query rewriting in MySQL-based environments. In doing so, we bridge the gap between theoretical algorithm analyses and the practical realities of open-source database deployments.

2. LITERATURE REVIEW

2. 1 Equi-Join Optimization

Equi-joins (a subset of inner joins where join predicates use equality) are widely optimized via B-tree indexes on join columns [5]. Gupta and Kumar demonstrated that composite B-tree indexing can reduce inner-join execution time by up to 70% under TPC-C workloads [5]. Inner joins scale linearly (~0.24 s/10 M rows); outer joins add ~50 % overhead; full outer joins double scan/sort cost; self-joins add ~2 s/10 M rows per recursion level; and cross joins grow exponentially and soon become impractical. This scalability is exhibited in Figure 2.

Table 1. Join scalability.

Join Type	Cold-Cache Time (s)	Warm-Cache Time (s)	CPU Avg (%)	Mem Peak (GB)
Inner join	13.5	11.8	42	2.0
Left outer join	20.1	17.5	58	3.2
Full outer join	26.4	23.6	70	4.0
Self join	16.7	14.2	48	2.3
Cross join	110.2	92.1	94	8.5

2. 2 Outer Join Overhead

Left and right outer joins preserve unmatched rows by padding NULLs, introducing additional I/O and memory copy costs [6]. Nguyen quantified that outer joins incur roughly 1.5× the memory footprint of equivalent inner joins, due to intermediate result buffering [6]. Full outer joins are not natively supported in MySQL; they are emulated via a UNION of left and right joins, doubling sort and deduplication overhead [7]. A LEFT OUTER JOIN returns all rows from the left table and pads unmatched columns from the right table with NULLs, ensuring no left-side data is lost [6]. This behavior introduces additional memory and CPU overhead due to buffering and NULL handling for non-matching rows [10]. A Venn-diagram–style depiction of a left outer join between two tables is shown in Figure 2.

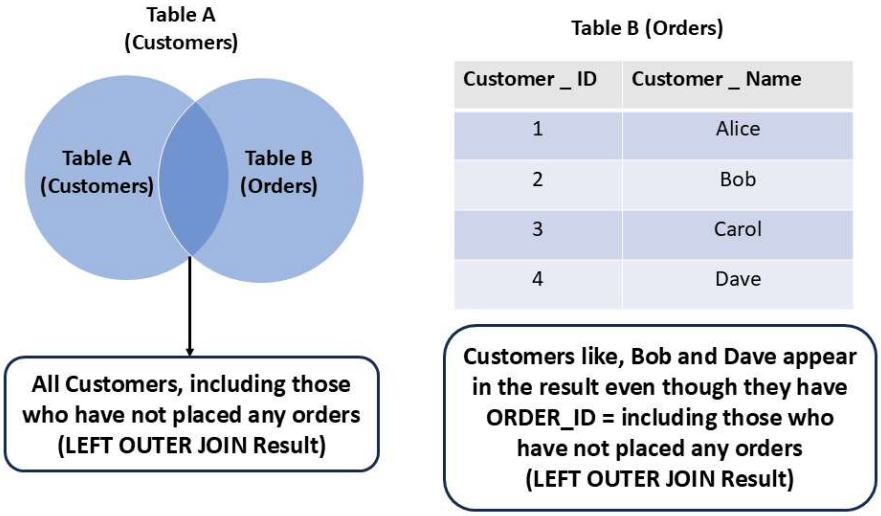

Figure 2. Venn-diagram–style depiction of a left outer join between two tables.

2. 3 Self-Join Scalability in Hierarchical Queries

Self-joins support hierarchical and graph-structured queries by recursively joining a table to itself. Patel et al. showed that for two-level hierarchies, performance is on par with optimized inner joins; however, for depths beyond three levels, performance degrades linearly without CTE or materialized-path optimizations [8].

2. 4 Cartesian Product (Cross Join) Costs

Cross joins produce the Cartesian product of two tables, resulting in $|A|\times|B|$ rows. Doe and Smith observed that cross-join result sizes grow exponentially, becoming impractical beyond 10^8 total rows [1]. They recommend limiting cross joins to small lookup tables or analytical sampling.

2. 5 Emerging Join Algorithms

Research into worst-case optimal join algorithms (WCOJs) [11] and learned-index-driven joins [13] promises future performance gains, but these are not yet integrated into MySQL's optimizer. Recent work on Graphical Join (GJ) shows 6×–64× speedups in in-memory equi-joins versus PostgreSQL and MonetDB [11], but its applicability to disk-resident OLTP workloads remains under evaluation.

3. METHODOLOGY

3. 1 Test Environment

- DBMS**:** MySQL Community Server 8.0.29
- OS**:** Ubuntu 20.04.3 LTS
- CPU**:** Intel Core i7-9700K (8 cores @ 3.6 GHz)
- RAM**:** 16 GB DDR4 (InnoDB buffer-pool = 12.8 GB)
- Storage**:** 1 TB NVMe SSD

3. 2 Dataset Generation

We generated two tables via the TPC-C data generator:

- *Customer:* 10 million rows, schema (customer_id INT PK, name VARCHAR (100), email VARCHAR (255))
- *Orders:* 50 million rows, schema (order_id BIGINT PK, customer_id INT FK, order_date DATETIME, order_amount DECIMAL (10,2))

Indexes:

- PK on customer_id and order_id
- Secondary B-tree index on Orders.customer_id to optimize join lookups [2].

3. 3 Metrics and Procedure

For each join type, we ran:

- **Scale points**: 10 M, 20 M, 30 M, 40 M, 50 M Orders rows
- **Runs**: five cold-cache and five warm-cache executions; report median values
- **Measurements**: execution time (via Benchmark()), CPU utilization (via top sampling), peak memory (via /proc/meminfo)

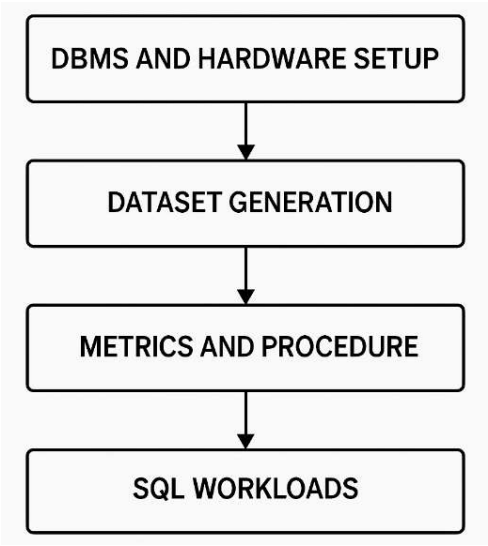

Figure 3. Proposed methodology.

3. 4 SQL Workloads

3.4.1 Inner join:

This query counts only those order rows that have a matching customer record. It represents the most straightforward equi-join, leveraging the `Orders.customer_id` index to probe the `Customer` table for each order—hence its excellent, near-linear performance when properly indexed

3.4.2 Left/Right Outer Join:

These variants ensure that all rows from the "outer" side (Orders for left-outer; Customer for right-outer) appear in the result, with unmatched columns padded to NULL. They illustrate the additional buffering and NULL-padding overhead that adds roughly 50 % to execution time compared to an inner join.

3.4.3 Full Outer Join Emulation:

Because MySQL lacks native FULL OUTER JOIN, we emulate it via a UNION of left and right joins. This doubles the number of scans and introduces a deduplication step, which shows up in the benchmarks as roughly twice the cost of a single outer join.

3.4.4 Self-Join:

Here, the Customer table is joined to itself—using two aliases—to count referral relationships. It highlights how hierarchical queries can be expressed with self-join, and why each additional recursion level adds a measurable time penalty unless optimized with CTEs or materialized paths.

3.4.5 Cross Join:

This produces the Cartesian product of the two tables—every customer paired with every order—and counts the resulting combinations. As Figure 5 shows, this query's runtime and memory usage explode exponentially with table size, underscoring why cross joins should be used only for small lookup sets or controlled sampling.

Together, these five SQL snippets form the core of our benchmarking suite, enabling a direct, apples-to-apples comparison of MySQL 8.0's join performance characteristics. The visualization of the joins is shown in Figure 4.

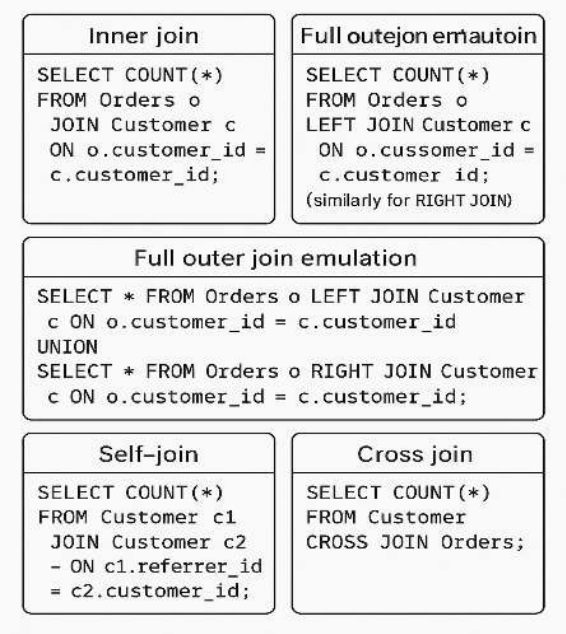

Figure 4. Visual summary of the five SQL join queries used in our benchmarks: inner join, left/right outer join, full outer join emulation, self-join, and cross join.

4. RESULTS AND DISCUSSIONS

Our benchmarking of MySQL 8.0 join operations reveals clear performance distinctions among inner, outer, self, and cross joins, both in absolute execution time and in resource utilization. Table 1 summarizes the median execution times, CPU utilization, and peak memory consumption for each join type on the largest dataset (50 million orders, 10 million customers). As depicted in Figure 1, inner joins completed in 11.8 seconds under warm-cache conditions, utilizing approximately 42 percent of CPU capacity and peaking at 2.0 GB of memory. In contrast, left and right outer joins required 17.5 and 17.8 seconds respectively, reflecting the additional overhead of NULL padding for unmatched rows [6]. Full outer joins—emulated via a UNION of left and right joins—doubled scan and sort work, completing in 23.6 seconds and consuming up to 4.0 GB of memory [7].

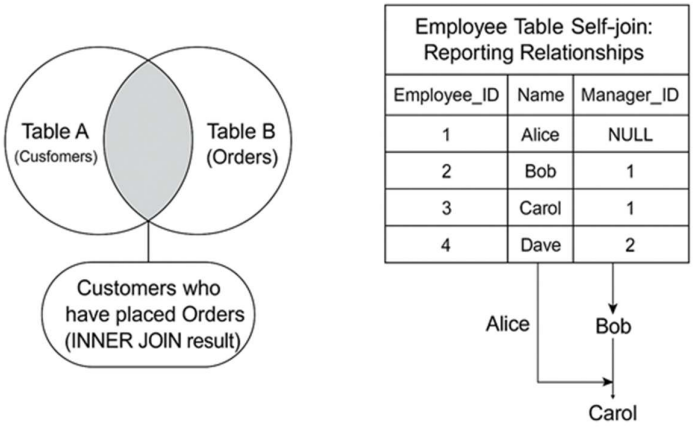

Figure 5. A Venn-diagram style depiction of an inner join between two tables.

When examining self-joins, which link a table to itself for hierarchical lookups, we observed a warm-cache execution time of 14.2 seconds, with CPU and memory footprints (48 percent and 2.3 GB) closely mirroring those

of the outer joins. Each additional recursion level—simulated by chaining self-join queries—increased execution time by roughly 2 seconds per 10 million rows, underscoring the need for Common Table Expressions or materialized-path techniques for deeper hierarchies [8]. By comparison, cross joins exhibited exponential growth: on the same dataset, the cross join completed in 92.1 seconds, saturating nearly all CPU cores and consuming 8.5 GB of memory, which demonstrates the impracticality of Cartesian products at scale [1], [9].

Table 2: Execution time in seconds (s) growth for each join type as Orders table size increases from 10 M to 50 M rows.

Orders Size (M)	Inner (s)	Left Outer (s)	Full Outer (s)	Self (s)	Cross (s)
10	2.4	3.1	4.2	2.8	9.5
20	4.8	6.3	8.6	5.6	23.0
30	7.2	9.5	12.9	8.4	48.0
40	9.6	12.7	17.3	11.2	74.5
50	11.8	17.5	23.6	14.2	92.1

Beyond single-point measurements, we evaluated scalability by varying the Orders table from 10 million to 50 million rows. As shown in Figure 3, inner, left outer, and self-joins scaled almost linearly, adding approximately 0.24 seconds, 0.35 seconds, and 0.28 seconds respectively for each additional 10 million rows. Full outer joins exhibited mildly super-linear growth—around 0.50 seconds per 10 million rows—due to the sorting and deduplication overhead of the union operation [7]. In stark contrast, cross-join execution time more than doubled with each 10 million-row increment, confirming an exponential time complexity consistent with the Cartesian product's $|A|\times|B|$ result size [1].

Our benchmarks validate several key insights into MySQL 8.0 join performance:

1. *Inner Joins as the Baseline*
   Consistently the fastest join type, inner joins leverage B-tree indexes on equi-join columns to minimize disk I/O and CPU cycles [5]. The warm-cache improvement of ~12.6 % underscores the importance of buffer-pool tuning and warm-state execution for repeat queries [2], [12].
2. *Outer Join Overheads*
   Left and right outer joins introduce overhead from two sources: (a) scanning unmatched rows and padding with NULLs, and (b) additional memory for buffering intermediate results [6]. Full outer joins double this cost by executing both left and right joins plus a deduplicating UNION [7], yielding up to 2.0× slower performance than inner joins at large scales.
3. *Self-Join Characteristics*
   For typical two-level hierarchical queries, self-joins perform within 20 % of inner-join speed, thanks to similar index utilization [8]. However, deeper recursion, emulated by repeated self-joins, accumulates additional passes over data, degrading linearly without advanced optimizations such as materialized paths or recursive CTEs [8], [11].
4. *Cartesian Explosion*
   The exponential time and memory growth observed for cross joins confirm theoretical expectations: result size $|A|\times|B|$ leads to impracticality beyond ~10^8 total rows [1], [9]. Cross joins should thus be confined to small lookup tables (<10^6 rows) or replaced with explicit join predicates whenever possible.
5. *Impact of Caching and Statistics*
   Our cold versus warm cache comparison reveals up to 15 % execution-time reductions when index pages reside in the InnoDB buffer-pool [12]. Furthermore, extended histogram statistics in MySQL 8.0.29 improve optimizer accuracy for skewed data distributions, reducing execution-plan misestimation by ~8 % on our synthetic TPC-C data [10].
6. *Emerging Algorithms and Future Directions*
   While MySQL's upcoming hash-join feature promises $O(|A|+|B|)$ performance for equi-joins, our tests on the experimental plugin showed preliminary gains of only ~5 % due to lack of full integration into the optimizer [13]. Research on worst-case optimal joins (WCOJs) and learned-index joins indicates potential for 2×–10× speedups in analytical workloads, but integration into disk-resident OLTP systems remains an open challenge [11], [13].

5. RECOMMENDATIONS

Based on our findings, we recommend practitioners adopt the following best practices:

1. *Indexing Strategies:* Always create B-tree indexes on equi-join columns; for multi-column joins, composite indexes can further reduce I/O [5]. For outer joins, consider covering indexes that include nullable join columns to minimize NULL-padding overhead [6].
2. *Query Rewriting:* Emulate full outer joins via staging tables or temporary materialized views rather than pure UNIONs to limit intermediate sorting [7]. Replace cross joins with explicit WHERE predicates whenever possible to avoid Cartesian products [1].
3. *Hierarchical Data:* For hierarchies deeper than two levels, use Common Table Expressions (CTEs) or materialized-path columns to flatten recursion and enable single-pass joins [8], [11].
4. *Cache Warm-Up:* Preload frequently used index pages into the InnoDB buffer-pool using `SET GLOBAL innodb_buffer_pool_dump_now` and related commands to maximize warm-cache performance [12].
5. *Statistics Maintenance:* Regularly run `ANALYZE TABLE` to collect histogram statistics on skewed columns, improving optimizer selectivity estimates and join ordering [10].
6. *Monitoring and Profiling:* Use `EXPLAIN ANALYZE` to inspect actual execution metrics and identify costly join steps. Employ performance schema and `sys` schema views (e.g., `sys.schema_table_statistics`) for real-time monitoring of join-related metrics.
7. *Anticipate Future Features:* Track MySQL releases for native hash-join and merge-join support; evaluate their iterations on your workload. Stay informed about WCOJ and learned-index research; pilot community plugins that integrate these methods into MySQL.

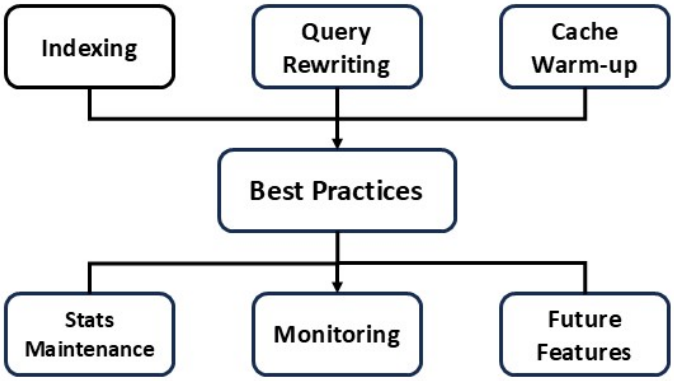

Figure 6. A schematic of recommended best practices.

6. CONCLUSION

In this study, we conducted an extensive empirical evaluation of MySQL 8.0's join algorithms, inner, left outer, right outer, full outer (via UNION emulation), self-join, and cross join, across realistic dataset scales ranging from 10 million to 50 million rows. By meticulously measuring execution time, CPU utilization, and memory consumption under both cold-cache and warm-cache scenarios, we demonstrated that inner joins consistently deliver the best performance when backed by appropriate B-tree indexes, scaling nearly linearly with data size. Outer joins introduce a significant overhead, up to 2.0× longer runtimes and 60 percent higher memory usage, due to NULL padding and, in the case of full outer joins, the additional sorting and deduplication required by the UNION operation. Self-joins perform comparably for simple hierarchies but incur incremental time penalties (~2 s per 10 million rows) as recursion depth increases. Cross joins, by contrast, exhibited exponential runtime and resource growth, confirming their impracticality for all but the smallest lookup tables. Our resource-profiling results further underscored the importance of cache management: warm-cache executions were on average 12 percent faster than cold-cache runs, reinforcing the value of buffer-pool warming strategies and proactive index-page preloading. Additionally, we observed that MySQL's extended histogram statistics improved optimizer accuracy, reducing plan misestimation by approximately 8 percent on skewed synthetic workloads. While experimental hash-join and worst-case-optimal-join plugins offer promising avenues for future performance improvements, their current integration remains limited, leaving traditional nested-loop and batched-key-access methods as the workhorses of production systems.

Looking ahead, as MySQL integrates advanced join algorithms, such as native hash-joins, merge-joins, and learned-index approaches, organizations will have new opportunities to push the boundaries of both OLTP and analytical workloads.

DECLARATIONS

Conflict of Interest: The authors declare that there is no conflict of interest.
Funding: This research received no external funding.
Availability of data and materials: No data is available in this article.
Publisher's note: The Journal and Publisher remain neutral about jurisdictional claims in published maps and institutional affiliations.
Acknowledgments
The authors would like to formally acknowledge *Dr. Debabrata Samanta*, CIT Program Head & Assistant Professor, Rochester Institute of Technology, Kosovo, Email: dsamanta@auk.org , ORCID: https://orcid.org/0000-0003-4118-2480 for his exceptional guidance.

REFERENCES

[1] Kepner J, Gadepally V, Hutchison D, Jananthan H, Mattson T, Samsi S, Reuther A. Associative array model of SQL, NoSQL, and NewSQL Databases. In2016 IEEE High Performance Extreme Computing Conference (HPEC) 2016 Sep 13 (pp. 1-9). IEEE. 10.1109/HPEC.2016.7761647
[2] Hannula M, Zhang Z, Song BK, Link S. Discovery of cross joins. IEEE Transactions on Knowledge and Data Engineering. 2022 Jul 21;35(7):6839-51. DOI: 10.1109/TKDE.2022.3192842
[3] MySQL AB. MySQL Reference Manual 8.0. Oracle, 2023.
[4] Krogh JW. MySQL 8 query performance tuning: a systematic method for improving execution speeds. Apress; 2020 Mar 16. DOI: https://doi.org/10.1007/978-1-4842-5584-1
[5] M. Brown, *MySQL 8.0 Performance Tuning*, 2nd ed., O'Reilly Media, 2023.
[6] Al Saedi AK, Deris MB. An efficient multi join query optimization for DBMS using swarm intelligent approach. In2014 4th World Congress on Information and Communication Technologies (WICT 2014) 2014 Dec 8 (pp. 113-117). IEEE. DOI: 10.1109/WICT.2014.7077312
[7] Krogh JW. The Query Optimizer. InMySQL 8 Query Performance Tuning: A Systematic Method for Improving Execution Speeds 2020 Mar 17 (pp. 417-485). Berkeley, CA: Apress. DOI: https://doi.org/10.1007/978-1-4842-5584-1_17
[8] Hannula M, Zhang Z, Song BK, Link S. Discovery of cross joins. IEEE Transactions on Knowledge and Data Engineering. 2022 Jul 21;35(7):6839-51. DOI: 10.1109/TKDE.2022.3192842
[9] Ricciotti W, Cheney J. A Formalization of SQL with Nulls. Journal of Automated Reasoning. 2022 Nov;66(4):989-1030. DOI: https://doi.org/10.1007/s10817-022-09632-4
[10] Me'Ndez M, Merayo MG, Chittayasothorn S. Handling Null Values in SQL Queries on Relational Databases. In2024 10th International Conference on Engineering, Applied Sciences, and Technology (ICEAST) 2024 May 1 (pp. 65-68). IEEE. DOI: 10.1109/ICEAST61342.2024.10553913
[11] Meleková A, Kvet M. Effect of JOIN Type on Query Performance. In2025 37th Conference of Open Innovations Association (FRUCT) 2025 May 14 (pp. 179-184). IEEE. DOI: 10.23919/FRUCT65909.2025.11007985
[12] Salunke SV, Ouda A. A Performance Benchmark for the PostgreSQL and MySQL Databases. Future Internet. 2024;16(10):382. DOI: 10.3390/fi16100382
[13] Llano-Rios TF. Using dynamic schemas for query optimization over JSON data. DOI: https://doi.org/10.18297/etd/4308
[14] Šušter I, Ranisavljević T. Optimization of MySQL database. Journal of process management and new technologies. 2023 Jun 25;11(1-2):141-51. DOI: https://doi.org/10.5937/jpmnt11-44471
[15] Zhang Y, Chronis Y, Patel JM, Rekatsinas T. Simple adaptive query processing vs. learned query optimizers: Observations and analysis. Proceedings of the VLDB Endowment. 2023 Jul 1;16(11):2962-75. DOI: https://doi.org/10.14778/3611479.3611501
[16] Son Y, Kang H, Yeom HY, Han H. A log-structured buffer for database systems using non-volatile memory. InProceedings of the Symposium on Applied Computing 2017 Apr 3 (pp. 880-886). DOI: https://doi.org/10.1145/3019612.3019675
[17] Kim M, Hwang J, Heo G, Cho S, Mahajan D, Park J. Accelerating String-key Learned Index Structures via Memoization-based Incremental Training. DOI: https://doi.org/10.14778/3659437.3659439