Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 01 , January - 2026

A Comparative Performance Evaluation of Join Strategies: Sensitivity to Modern Hardware Properties

Nimisha Modi
Department of Computer Science, Veer Narmad South Gujarat University, Surat, Gujarat, India

***Abstract* - Join processing is one of the most performance-critical components of query execution in relational database systems. While classical join algorithms have been extensively studied, their evaluation has traditionally been guided by asymptotic complexity and disk-oriented cost models. Recent advances in processor, memory, and storage architectures—characterized by multi-core processors and deep cache hierarchies—have introduced new dimensions that reshape the performance behavior of join strategies.**

A comparative, hardware-aware analysis is presented to examine how nested loop, index nested loop, hash join, sort-merge join, and accelerator-based joins interact with underlying hardware properties. The study synthesizes insights from foundational and recent research to analyze join sensitivity to cache locality, memory bandwidth, NUMA effects, parallel scalability, accelerator utilization, and storage characteristics. A structured comparison and hardware-layer mapping illustrate why theoretically efficient join strategies may underperform on contemporary platforms.

The analysis indicates that join performance is now dominated more by architectural constraints than by asymptotic algorithmic complexity, and that no single join strategy is universally optimal across heterogeneous hardware environments. The paper concludes by identifying key research challenges as hardware-aware cost modeling, adaptive join execution, and cross-layer optimization - that are critical for the design of efficient and scalable database systems.

Keywords—Join Algorithms; Query Processing; Hardware-Aware Databases; Hash Join; Sort-Merge Join; Multi-Core Systems; GPU Acceleration; Memory Hierarchy;

I. INTRODUCTION

Join processing constitutes a central component of query execution in relational database management systems and has a direct impact on overall system performance. Most transactional and analytical queries involve joining multiple tables, and the efficiency of these join operations largely determines query execution time. Consequently, join algorithms have remained a persistent focus of research within the database systems community.

Traditional join algorithms such as Nested Loop Join, Sort-Merge Join, and Hash Join were originally designed for disk-based database systems. In such environments, disk I/O was the primary performance bottleneck, and optimization techniques mainly aimed at reducing the number of disk accesses. However, current database systems run on hardware platforms that differ significantly from earlier designs.

Modern hardware provides multi-core CPUs, large main memory, deep cache hierarchies, NUMA architectures, and fast storage devices such as SSDs. These advancements have shifted performance bottlenecks from disk I/O to memory access latency, cache efficiency, synchronization overhead, and parallel execution. As a result, join performance is increasingly influenced by hardware characteristics such as cache locality, memory bandwidth, and available CPU parallelism.

In addition, modern workloads often process large datasets with high concurrency and data skew, especially in analytical and decision-support applications. Join algorithms must therefore efficiently utilize available hardware resources while handling diverse data distributions. This has led to increased interest in hardware-aware and parallel join strategies.

This paper presents a unified, hardware-aware comparison of classical and modern join algorithms, examining how join strategies interact with CPU caches, memory systems, accelerators, and storage technologies. The analysis identifies the architectural factors that dominate join performance and leads to the selection of suitable join techniques for contemporary database systems.

II. JOIN ALGORITHMS IN DATABASE SYSTEMS – REVIEW

Join processing is a core operation in relational query execution and has been extensively studied for several decades. Classical join algorithms were originally designed for disk-based systems, but the emergence of multi-core CPUs, large main memories, GPUs, and high-performance storage devices has significantly reshaped their performance characteristics. This section reviews fundamental join algorithms and discusses their sensitivity to modern hardware properties.

A. Nested Loop Join (NLJ)

The Nested Loop Join (NLJ) is the most basic join algorithm, in which each tuple of the outer relation is compared against all tuples of the inner relation. Its computational complexity is $O(N \times M)$, where N and M are the sizes of the outer and inner relations, respectively, making it impractical for large tables under naive implementations.

Despite its simplicity, NLJ remains relevant in query optimizers due to its predictable behaviour and suitability for small relations or highly selective joins. Graefe's classical

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 01 , January - 2026

survey highlights NLJ as a foundational operator from which more optimized join techniques evolved [7].

Modern systems often employ NLJ when join cardinalities are small or when other join strategies incur higher setup costs. However, NLJ exhibits poor cache locality and limited parallelism, making it less suitable for multi-core and main-memory systems without further optimization.

B. Block Nested Loop Join

Block Nested Loop Join (BNLJ) improves upon NLJ by processing multiple tuples of the outer relation in blocks, thereby reducing repeated scans of the inner relation. By exploiting available memory buffers, BNLJ significantly reduces I/O overhead and improves cache reuse. In contemporary main-memory databases, block-oriented processing aligns better with CPU cache hierarchies and memory bandwidth constraints.

Graefe [7] and Graefe [8] note that block-based join execution forms the basis for vectorized and cache-conscious query engines. Nevertheless, BNLJ remains limited in scalability compared to hash- and sort-based joins when faced with large datasets and high degrees of parallelism.

C. Index Nested Loop Join

Index Nested Loop Join (INLJ) accelerates NLJ by leveraging an index on the join attribute of the inner relation. For each tuple in the outer relation, indexed lookups are performed using a B-tree index on the inner relation, reducing the expected complexity to $O(N \log M)$, where N is the size of the outer relation and M is the size of the inner relation.

INLJ is particularly effective when the outer relation is small and the index is highly selective. However, its performance is highly sensitive to memory access latency and index traversal costs. On modern hardware, pointer-heavy index structures may suffer from cache misses and branch mispredictions, limiting scalability on multi-core systems [3],[4]. As a result, INLJ is often outperformed by hash joins in main-memory environments.

D. Hash Join

Hash join has become one of the most widely used join strategies for equi-join predicates in many modern database systems. It operates by building a hash table on the smaller relation and probing it with tuples from the larger relation. Extensive research has shown that hash joins benefit significantly from main-memory execution and multi-core parallelism.

Balkesen et al. [3],[4] provide a detailed comparison of hash join variants, demonstrating how cache-aware partitioning, SIMD processing, and thread synchronization strategies can dramatically affect performance. Albutiu et al. [2] and Barthels et al. [5] further extend hash join designs to many-core and distributed environments.

The efficiency of hash joins is tightly coupled to memory bandwidth availability, cache capacity, and data skew. Recent studies highlight the need for careful tuning to hardware characteristics such as NUMA architectures and memory contention [4].

E. Sort-Merge Join

Sort-Merge Join (SMJ) performs joins by sorting both input relations on the join key and then merging them. While traditionally considered expensive due to sorting costs, SMJ has regained importance in modern systems where inputs are already sorted or indexed.

Albutiu et al. [2] and Balkesen et al. [3] demonstrate that parallel sort-merge joins scale efficiently on multi-core processors and can outperform hash joins under certain workload and hardware conditions. SMJ exhibits sequential memory access patterns, making it more cache-friendly and predictable compared to hash-based methods.

Additionally, SMJ is well-suited for range joins and streaming scenarios, where ordered data can be processed incrementally with minimal synchronization overhead.

F. Hardware-Aware and Accelerated Join Algorithms

Recent research has explored join acceleration on specialized hardware such as GPUs and high-speed storage devices. GPU-based joins exploit massive parallelism and high memory bandwidth but require careful management of data transfer and workload balance [12], [13]. Wu et al. [6] demonstrate efficient join and aggregation processing on GPUs, highlighting performance gains for analytical workloads.

Similarly, emerging storage technologies such as NVMe SSDs have shifted the I/O bottleneck, motivating new join execution models that overlap computation and data access [9],[10]. These studies reveal performance mismatches between traditional join algorithms and modern hardware, emphasizing the need for configuration-aware and adaptive execution strategies.

Recent work has also revisited core database structures from a hardware-aware perspective, emphasizing the need for parallelism- and accelerator-conscious designs across indexing and query processing components, further reinforcing the motivation for hardware-sensitive join evaluation [1].

G. Summary and Research Gap

While classical join algorithms such as NLJ, hash join, and sort-merge join are well understood, their performance is increasingly influenced by hardware properties including cache hierarchies, memory bandwidth, parallelism, and accelerator devices.

Recent studies show a shift toward adaptive, hardware-aware data access, where architectural factors increasingly influence core query operator performance beyond indexing [11].

Existing studies often focus on optimizing individual join strategies for specific platforms. However, a unified comparative analysis that explicitly evaluates join sensitivity to heterogeneous hardware remains limited—motivating the focus of this paper.

Table I. provides a qualitative comparison of join strategies with respect to their sensitivity to key hardware characteristics. Hash and sort-merge joins exhibit the highest sensitivity to memory hierarchy and bandwidth, while NLJ-based methods are more affected by storage latency and lack of parallelism.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 01 , January - 2026

Accelerated joins shift the bottleneck toward data movement and configuration tuning.

TABLE I. **Comparison of join strategies based on their sensitivity to hardware properties**

Join Strategy	CPU Cache Sensitivity	Memory Bandwidth Dependence	Multi-core Scalability	GPU Suitability	Storage (NVMe) Sensitivity	Key Observations
Nested Loop Join (NLJ)	Low	Low	Poor	Not suitable	High	Simple but inefficient; dominated by comparison cost and poor locality [7]
Block Nested Loop Join (BNLJ)	Moderate	Moderate	Limited	Not suitable	Moderate	Improved cache reuse via blocking; still limited parallelism [7], [8]
Index Nested Loop Join (INLJ)	High	Moderate	Limited	Not suitable	Low	Performance dominated by cache misses and pointer chasing [3], [4]
Hash Join	High	Very High	Excellent	Moderate	Low	Sensitive to cache size, NUMA effects, and memory contention [3], [4], [5]
Sort-Merge Join (SMJ)	Moderate	High	Excellent	Moderate	Moderate	Sequential access patterns favor cache and prefetching [2], [3]
GPU-based Hash/Sort Joins	Low (CPU)	Very High (GPU)	Massively parallel	Excellent	Low	Requires careful data transfer and load balancing [6], [12], [13]
Distributed / Many-core Joins	High	Very High	Excellent	Moderate	Moderate	Network and synchronization overheads dominate [5]

III. CONTEMPORARY HARDWARE PROPERTIES AFFECTING JOIN PERFORMANCE

The evolutions of hardware architectures have fundamentally altered the performance of join algorithms. Traditional cost models based on disk I/O and tuple comparisons are increasingly insufficient. Instead, join performance is now shaped by processor microarchitecture, memory hierarchy, parallel execution capabilities, accelerator devices, and storage technologies.

A. CPU Microarchitecture and Cache Hierarchies

Modern CPUs feature deep cache hierarchies (L1–L3) and wide SIMD units designed to maximize instruction-level parallelism. Join algorithms with predictable memory access patterns—such as sort-merge joins—benefit from improved cache prefetching and reduced cache misses [2],[3]. In contrast, pointer-intensive operations, common in index nested loop joins, often incur frequent cache misses and branch mispredictions, limiting scalability despite logarithmic complexity [4]. As shown in architecture-aware studies, cache-conscious partitioning and vectorized probing are critical for optimizing hash joins on modern CPUs [3],[4].

B. Memory Bandwidth and NUMA Effects

As databases increasingly operate entirely in main memory, memory bandwidth has become a dominant performance bottleneck. Hash joins, in particular, are highly bandwidth-intensive due to random memory accesses during hash table probing [3]. On NUMA (Non-Uniform Memory Access) systems, improper memory placement can severely degrade join performance. Research demonstrates that NUMA-aware scheduling and data partitioning are essential to avoid remote memory access penalties, especially for parallel hash joins [4],[5].

C. Multi-core and Many-core Parallelism

Modern processors expose substantial thread-level parallelism, often spanning dozens to hundreds of cores. Parallel join algorithms must efficiently synchronize threads and minimize contention. Sort-merge joins often scale well due to their naturally parallel sorting and merging phases [2].

Hash joins, while highly parallelizable, are sensitive to synchronization overheads and data skew. Distributed and many-core environments further amplify these challenges, introducing communication and coordination costs that can dominate execution time [5].

D. GPU Acceleration

GPUs offer massive parallelism and high memory bandwidth, making them attractive for join processing. GPU-based join algorithms typically employ hash-based or sort-based strategies optimized for SIMD-style execution [12],[13]. Recent work demonstrates significant performance gains for analytical workloads when joins and grouped aggregations are offloaded to GPUs [6]. However, data transfer overheads between CPU and GPU memory, as well as workload imbalance, remain key limitations. Consequently, GPU acceleration is most effective for large, compute-intensive joins.

E. Storage Technologies and NVMe Devices

Emerging storage technologies such as NVMe SSDs have reduced I/O latency and increased throughput, narrowing the gap between storage and memory speeds. This shift challenges traditional assumptions that joins are purely CPU- or memory-bound [9]. Studies reveal performance mismatches between database configurations and modern storage devices, indicating that join execution strategies must adapt to exploit asynchronous I/O and high parallelism at the storage level [10]. As a result, join algorithms increasingly overlap computation with data access, particularly in hybrid memory–storage systems.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 01 , January - 2026

F. *Selection of Join Strategy*

The effectiveness of a join strategy is no longer determined solely by algorithmic complexity. Instead, it depends on a complex interaction between data characteristics and hardware properties. Hash joins dominate in memory-rich, multi-core systems, while sort-merge joins excel when data is ordered or bandwidth is constrained. Accelerator-based joins offer substantial gains but require careful hardware–software co-design [3],[6],[9].

IV. COMPARATIVE ANALYSIS OF JOIN STRATEGIES UNDER MODERN HARDWARE CONSTRAINTS

This section presents a comparative analysis of classical and modern join strategies with respect to their sensitivity to contemporary hardware properties. Rather than evaluating absolute performance, the analysis focuses on relative behaviour under varying architectural conditions, synthesizing insights from prior studies. The hardware-aware discussion in Section 3 provides a consolidated view of how join algorithms interact with storage, memory, CPU, and accelerator layers.

A. *Computational Complexity vs. Hardware Efficiency*

Traditional join algorithm selection has largely relied on asymptotic computational complexity. While nested loop joins exhibit quadratic complexity and hash or sort-merge joins offer near-linear behaviour, complexity alone is no longer a reliable predictor of performance. For example, index nested loop joins theoretically reduce complexity through logarithmic index access, yet empirical studies show that pointer chasing and branch misprediction significantly degrade cache efficiency on modern CPUs [3],[4]. In contrast, hash joins—despite higher memory demands—often outperform index-based joins due to superior cache utilization and predictable access patterns when properly tuned [3]. This observation reflects a broader trend in which hardware efficiency increasingly outweighs algorithmic complexity in determining join performance.

B. *Cache Locality and Memory Access Patterns*

Cache behaviour is another dominant factor influencing join performance. As shown in Table 1, sort-merge joins benefit from sequential memory access, resulting in high cache line utilization and effective hardware prefetching [2],[3]. Hash joins, while efficient in equi-join scenarios, suffer from random memory accesses that can overwhelm cache hierarchies if partitioning is not cache-conscious [4]. Nested loop–based joins generally exhibit poor locality unless enhanced through blocking. Even then, their reuse potential is limited compared to hash and sort-merge joins.

C. *Memory Bandwidth and NUMA Sensitivity*

As main-memory database systems become prevalent, memory bandwidth has emerged as a primary bottleneck. Hash joins are particularly sensitive to bandwidth constraints due to frequent memory accesses during probing phases [3]. On NUMA architectures, suboptimal data placement can lead to remote memory access, significantly increasing latency [4],[5]. Sort-merge joins tend to be more robust under NUMA conditions, as their access patterns are more sequential and partitionable. Distributed and many-core join algorithms further magnify memory-related challenges by introducing synchronization and interconnect overheads, as observed in large-scale parallel systems [5].

D. *Parallelism and Scalability*

Parallel scalability is a critical differentiator among join strategies. Hash joins and sort-merge joins exhibit strong scalability on multi-core processors, provided that workload skew and synchronization overheads are controlled [2],[3]. Sort-merge joins often scale more predictably due to natural task decomposition during sorting and merging phases. Nested loop and index nested loop joins, by contrast, offer limited opportunities for fine-grained parallelism and often become bottlenecked by single-thread performance or shared index structures.

E. *Accelerator-Based Join Processing*

GPU-accelerated join algorithms represent a significant departure from traditional CPU-centric execution models. By exploiting massive parallelism and high memory bandwidth, GPU-based hash and sort joins achieve substantial throughput improvements for large analytical workloads [6],[12],[13]. However, the comparative advantage of GPU joins is highly workload-dependent. Data transfer overheads between host and device memory can offset computational gains, particularly for small or selective joins. Consequently, accelerator-based joins are most effective when join processing dominates execution cost and data movement can be amortized [6].

F. *Storage-Aware Join Behavior*

Emerging storage technologies such as NVMe SSDs have reduced I/O latency and increased parallelism, challenging traditional assumptions that joins are purely memory- or CPU-bound [9]. Storage-aware join execution increasingly overlaps computation with I/O, blurring the boundary between storage and memory layers. Comparative studies reveal that join strategies optimized for disk-based systems may underutilize modern storage capabilities unless explicitly redesigned [10].

G. *Synthesis of Comparative Insights*

The comparative analysis reveals that no single join strategy dominates across all hardware environments. Instead, join performance emerges from a complex interaction between algorithm design and hardware characteristics: Hash joins excel in memory-rich, multi-core systems but are bandwidth- and NUMA-sensitive. Sort-merge joins provide predictable scalability and cache efficiency, particularly for ordered data. Nested loop variants remain useful for small or selective joins but scale poorly on modern hardware. GPU-based joins offer high throughput for large analytical workloads but introduce data movement overheads. These observations reinforce the need for hardware-aware join selection and motivate adaptive query execution models that dynamically align join strategies with underlying architectural properties.

This section provides a comparative, hardware-centric evaluation of join strategies, bridging classical algorithm analysis with modern system architecture considerations. The insights derived here directly inform the conclusions and future research directions discussed in the subsequent section.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 01 , January - 2026

V. DISCUSSION AND RESEARCH CHALLENGES

The comparative analysis in Section 4 highlights that join performance in modern database systems is no longer governed solely by algorithmic complexity. Instead, it emerges from a complex interaction between join strategies, data characteristics, and evolving hardware architectures. This section discusses the broader implications of these findings and outlines key research challenges that remain open in the design and optimization of join processing.

TABLE II. PERFORMANCE BOTTLENECKS OF JOIN STRATEGIES

Join Strategy	Primary Bottlenecks	Key Hardware Sensitivity
Nested Loop Join	CPU comparisons; storage latency	Poor cache reuse
Block Nested Loop Join	Limited parallelism; memory bandwidth	Moderate cache benefit
Index Nested Loop Join	Cache misses; branch misprediction (pointer chasing)	Cache & branch predictor
Hash Join	Memory bandwidth; NUMA locality	Cache size, synchronization
Sort-Merge Join	Sorting cost; memory bandwidth	Prefetch efficiency
GPU Hash Join	Host–device data transfer; load imbalance	PCIe bandwidth, GPU memory
Distributed Joins	Synchronization overhead; network latency	Interconnect bandwidth

Table II summarizes the primary and secondary performance bottlenecks that limit each join strategy on contemporary hardware platforms. Rather than algorithmic complexity, the table emphasizes hardware-level constraints that most strongly influence execution efficiency.

A. Rethinking Cost Models for Join Selection

Traditional query optimizers rely on cost models primarily based on disk I/O and tuple cardinalities. While such models were effective in disk-oriented systems, they are increasingly inadequate for modern hardware environments where joins execute predominantly in main memory [7],[8]. As demonstrated in Sections 3 and 4, factors such as cache miss rates, memory bandwidth saturation, NUMA locality, and synchronization overheads have a substantial impact on join performance [3],[4]. Accurately modeling these effects remains a significant challenge. Future cost models must incorporate hardware-aware parameters—such as cache line utilization and memory access latency—to make reliable join strategy decisions.

B. Adaptivity and Dynamic Join Strategy Selection

Modern workloads are often heterogeneous and dynamic, exhibiting varying data distributions, skew patterns, and query mixes. A join strategy that performs well under one configuration may degrade significantly under another [5]. Although several studies advocate adaptive and hybrid join techniques, most systems still rely on static plan selection. Developing runtime-adaptive join mechanisms that can switch strategies in response to observed hardware and workload conditions remains an open research problem. Such mechanisms must balance adaptability with low overhead to avoid negating performance gains.

C. Managing Memory Bandwidth and NUMA Effects

As main-memory systems scale across multiple sockets, NUMA effects increasingly dominate join performance. Hash joins, in particular, are sensitive to memory placement and bandwidth contention [4]. Despite existing NUMA-aware optimizations, fully eliminating remote memory access penalties is difficult, especially in shared and multi-tenant environments. Future research must explore fine-grained data placement, workload partitioning, and scheduling strategies that can dynamically adapt to changing NUMA conditions without excessive coordination overhead.

D. Exploiting Accelerators without Excessive Data Movement

GPU-accelerated joins demonstrate impressive throughput improvements for large analytical workloads [6],[12],[13]. However, the benefits of accelerator-based execution are often constrained by data transfer costs between host and device memory. A major challenge is minimizing data movement while maximizing computational utilization. Potential research directions include unified memory architectures, tighter CPU–GPU integration, and selective offloading strategies where only performance-critical join phases are accelerated. Achieving seamless integration of accelerators into query execution engines remains an active area of investigation.

E. Storage–Computation Co-design

The emergence of high-performance storage technologies such as NVMe SSDs challenges the long-standing assumption that storage access is orders of magnitude slower than computation [9]. As a result, join algorithms that were optimized for traditional disks may fail to fully exploit modern storage capabilities [10]. Designing join strategies that overlap computation with asynchronous I/O and adapt to storage-level parallelism represents a promising research direction. However, integrating storage-aware execution into existing query engines introduces complexity in scheduling, buffering, and failure handling

F. Toward Unified, Hardware-Aware Join Frameworks

Most existing research focuses on optimizing individual join algorithms for specific hardware platforms. A broader challenge lies in developing unified frameworks that can reason holistically about join execution across heterogeneous environments—including CPUs, GPUs, and emerging storage technologies. Such frameworks must reconcile conflicting optimization goals, such as minimizing latency versus maximizing throughput, while remaining robust to workload variability.

The discussion suggests that future database systems must move beyond static, one-size-fits-all join implementations. Instead, join processing should be adaptive, hardware-aware, and tightly integrated with system architecture. While recent studies provide valuable insights into specific hardware optimizations, a comprehensive approach that unifies algorithm design, cost modeling, and system-level adaptation remains largely unexplored. Addressing these challenges is essential for sustaining join performance as hardware architectures continue to evolve.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 01 , January - 2026

VI. CONCLUSION AND FUTURE WORK

This study presented a comparative, hardware-aware analysis of join strategies in modern database systems, focusing on how architectural properties influence join performance beyond traditional algorithmic complexity considerations. By systematically examining classical and advanced join algorithms through the lens of contemporary hardware—including multi-core CPUs, deep cache hierarchies, main-memory execution, accelerator devices, and high-performance storage—the study highlights the limitations of conventional join evaluation approaches.

The analysis demonstrates that join performance is increasingly dictated by hardware characteristics such as cache locality, memory bandwidth, NUMA effects, and parallel execution capabilities. Hash joins and sort-merge joins, while asymptotically efficient, exhibit distinct sensitivities to memory access patterns and synchronization overheads. Nested loop–based joins, though conceptually simple, scale poorly on modern architectures and remain viable primarily for small or highly selective workloads. Accelerator-based joins offer substantial throughput benefits for analytical queries but introduce new challenges related to data movement and integration complexity.

By integrating prior research with a hardware-layer perspective, the study offers a unified view that connects classical database theory with modern system architecture. The comparative framework developed here clarifies why no single join strategy dominates across all environments and underscores the necessity of hardware-aware decision-making in query execution. Several promising directions emerge from this study.

1. First, there is a clear need for future cost models that explicitly incorporate hardware-level metrics such as cache miss behavior, memory bandwidth utilization, and NUMA locality. Such models would enable query optimizers to make more accurate join strategy selections in heterogeneous environments.
2. Second, adaptive and dynamic join execution remains an open research challenge. Future systems should explore runtime mechanisms capable of monitoring workload and hardware conditions and adjusting join strategies accordingly, while keeping adaptation overhead minimal.
3. Third, tighter integration of accelerator devices into query execution pipelines warrants further investigation. Research into unified memory architectures, selective offloading, and cross-device scheduling could substantially reduce data movement overheads and improve the practicality of GPU-accelerated joins.
4. Finally, the continued evolution of storage technologies calls for deeper exploration of storage–computation co-design. Join algorithms that effectively overlap computation with asynchronous I/O and exploit storage-level parallelism may become increasingly important as the boundary between memory and storage continues to blur.

Overall, the results suggest that effective join processing depends on keeping pace with evolving hardware architectures to ensure both efficiency and adaptability in future database systems.

REFERENCES

[1] Abbasi, M., Bernardo, M. V., Váz, P., Silva, J., & Martins, P. (2024). Revisiting Database Indexing for Parallel and Accelerated Computing: A Comprehensive Study and Novel Approaches. Information, 15(8), Article 429. https://doi.org/10.3390/info15080429

[2] Albutiu, M. C., Kemper, A., & Neumann, T. (2012). Massively parallel sort-merge joins in main memory for multi-core multi-way systems. Proceedings of the VLDB Endowment, 5(10), 1064–1075. https://doi.org/10.14778/2336664.2336678

[3] Balkesen, C., Alonso, G., Teubner, J., & Özsu, M. T. (2013). Multi-core, main-memory joins: Sort vs. Hash revisited. Proceedings of the VLDB Endowment, 7(1), 85–96. https://doi.org/10.14778/2732219.2732227

[4] Balkesen, C., Teubner, J., Alonso, G., & Özsu, M. T. (2013). Main-memory hash joins on modern multi-core CPUs: Tuning to the architecture. 2013 IEEE 29th International Conference on Data Engineering (ICDE), 1327–1330. https://doi.org/10.1109/ICDE.2013.6544839

[5] Barthels, C., Müller, I., Tözün, P., Alonso, G., & Kossmann, D. (2017). Distributed join algorithms on thousands of cores. Proceedings of the VLDB Endowment, 10(5), 517–528. https://doi.org/10.14778/3055540.3055545

[6] Wu, B., Koutsoukos, D., & Alonso, G. (2025). Efficiently Processing Joins and Grouped Aggregations on GPUs. Proc. ACM Manag. Data 3, 1, Article 39. https://doi.org/10.1145/3709689

[7] Graefe, G. (1993). Query evaluation techniques for large databases. ACM Computing Surveys, 25(2), 73–169. https://doi.org/10.1145/152610.152611

[8] Graefe, G. (2012). New algorithms for join and grouping operations. Microsoft Research. https://doi.org/10.1007/s00450-011-0186-9

[9] Haas, G., Leis, V., & Haubenschild, M. (2023). What modern NVMe storage can do, and how to exploit it. Proceedings of the VLDB Endowment, 16(11), 2090–2102. https://doi.org/10.14778/3598581.3598584

[10] Haochen He, Erci Xu, Shanshan Li, Zhouyang Jia, Si Zheng, Yue Yu, Jun Ma, and Xiangke Liao. 2023. When Database Meets New Storage Devices: Understanding and Exposing Performance Mismatches via Configurations. Proc. VLDB Endow. 16, 7 (March 2023), 1712–1725. https://doi.org/10.14778/3587136.3587145

[11] Modi, N. A. (2026). The evolution of tree-based indexing: From static structures to adaptive traversal. International Journal of Innovative Research in Technology (IJIRT), 12(8), 2551–2556. https://doi.org/10.64643/IJIRTV12I8-189997-459

[12] Rui, R., & Tu, Y.-C. (2017). Fast equi-join algorithms on GPUs: Design and implementation. Proceedings of the 29th International Conference on Scientific and Statistical Database Management (SSDBM '17), Article 17, 1–12. Association for Computing Machinery. https://doi.org/10.1145/3085504.3085521

[13] Rui, R., Li, H., & Tu, Y. C. (2020). Efficient Join Algorithms For Large Database Tables in a Multi-GPU Environment. Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases, 14(4), 708–720. https://doi.org/10.14778/3436905.3436927