On Practical Considerations for the Adoption of Worst-Case Optimal Joins in RDBMSs

Ayoub Berdai*, Kawtar Younsi Dahbi, Dalila Chiadmi, Slimane Bah
SIP Research Team, Mohammed V University in Rabat, Ecole Mohammadia d'Ingénieurs, Morocco

***Abstract*—Worst-case optimal join (WCOJ) algorithms have attracted significant interest in both academia and industry due to their strong asymptotic performance guarantees. However, their integration into mature relational database management systems (RDBMSs) remains severely limited. Unlike prior theoretical surveys, this study analyzes the practical considerations for WCOJ adoption across three core query engine layers: interpretation, optimization, and execution, before extending to distributed environments. The findings indicate that the primary bottlenecks delaying widespread adoption are architectural rather than algorithmic. Specifically, WCOJ integration necessitates a fundamental shift in the optimization layer toward hybrid planning and new objective functions based on intersection costs. Furthermore, at the execution layer, the unpredictable pointer-chasing inherent to hierarchical index traversals creates severe hardware inefficiencies. By synthesizing ongoing research efforts, this survey provides a structured roadmap of engineering trade-offs to guide practitioners in successfully integrating WCOJs into conventional database architectures.**

***Keywords*—Worst-case optimal join; AGM-bound; join algorithms; query processing**

I. Introduction

Since their inception, conventional relational database management systems (RDBMSs), such as PostgreSQL, Oracle, and SQL Server have relied almost exclusively on binary joins. This is primarily driven by the cost-based, Selinger-style query optimizers inherited from System R [1], and further solidified by the SQL standard itself, where `JOIN` is a binary operation.

While this approach performs well for acyclic queries and star-schema analytics, it struggles to scale when processing complex and cyclic queries [2], [3]. The primary bottleneck is the massive intermediate results generated, that are often orders of magnitude larger than the final output, leading to redundant processing. Historically, this observed overhead was overlooked due to the long-standing consensus that binary joins were sufficient to match the join size in the worst case.

However, Atserias, Grohe, and Marx proved a tighter bound on the output size of joins in the worst case, now widely known in the literature as the AGM-bound [4], [5]. The AGM-bound demonstrates that binary joins are suboptimal for a specific class of complex and cyclic queries. For instance, to list all triangles of friends in a social network, binary join plans require $O(N^2)$, where N is the cardinality of the *friendships* table, whereas the actual worst-case output size is proved to be $O(N^{1.5})$.

Few years after this breakthrough, join algorithms that match the AGM-bound have emerged. These algorithms share a similar paradigm [6], [7]. They evaluate all relations simultaneously, one attribute at a time. This approach avoids redundant intermediate results and only materializes the final output. These algorithms are known in the literature as worst-case optimal joins (WCOJs).

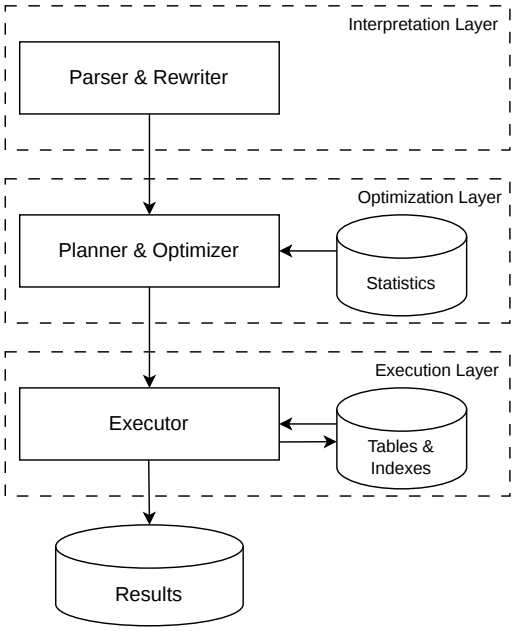


Fig. 1. Abstract architecture of a modern query engine. This survey analyzes the integration of WCOJ algorithms across the three highlighted layers: interpretation, optimization, and execution.

Empirical benchmarking demonstrates that while WCOJs can outperform binary joins by orders of magnitude on densely cyclic graph patterns (e.g., finding cliques) by avoiding intermediate result explosion, they frequently underperform traditional binary join plans on standard acyclic analytical workloads, such as the TPC-H [8] and Join Order Benchmarks [9].

Driven by these practical constraints, WCOJs have seen limited adoption in mature RDBMSs. Their use has largely been restricted to specialized and academic systems, while traditional relational engines continue to rely almost exclusively on binary joins. Table I presents examples of engines adopting WCOJs and how it is integrated.

This is especially relevant due to the recent developments in the SQL standard. With the introduction of SQL/PGQ in SQL:2023 [22], RDBMSs are expected to support graph

*Corresponding author

TABLE I. EXAMPLES OF THE ADOPTION OF WORST-CASE OPTIMAL JOINS IN DATABASE MANAGEMENT SYSTEMS

System	Worst-Case Optimal Join Adoption
LogicBlox [10], [6]	LogicBlox is a commercial Datalog engine that employs the Leapfrog Triejoin (LFTJ) as its *workhorse*. In fact, the algorithm was developed and implemented as an engineering solution even before its theoretical optimality guarantees were discovered [7]. This was a natural fit, as non-recursive Datalog rules provide a perfect structural description of the query's hypergraph. However, LFTJ relies on persistent indices, which theoretically requires maintaining every permutation of the relation's attributes, resulting in an $O(d!)$ storage overhead, where d is the relation arity. Their workaround comes from LogiQL, their query language, which encourages the sixth normal form (6NF), where a relation only holds a primary key and one attribute. These low-arity relations (binary or ternary) cap the permutations at 2! or 3!, making index maintenance manageable. For wider n-ary relations where materializing all permutations is impossible, the engine avoids building them upfront to save space, though this can force the optimizer into suboptimal plans. To mitigate this, Veldhuizen suggested loading indexes lazily as needed by the LFTJ algorithm, a practical compromise that subsequent research acknowledges as a severe bottleneck [11].
EmptyHeaded [12]	EmptyHeaded is a relational system for graph processing that supports a Datalog-like query language. It expands upon the classical approach of relational algebra by using generalized hypertree decompositions (GHDs) for logical query plans. The engine processes the decomposed hypertree much like a traditional engine processes an abstract syntax tree of a relational algebra expression, using binary joins for acyclic nodes. However, for the densely cyclic GHD nodes, they use a variant of Generic Join [7]. For attribute ordering, EmptyHeaded uses a combination of static heuristics, prioritizing the highest-degree attributes among the joined relations ordered by the minimum active domain (i.e. estimated cardinality with filters applied). Finally, for physical WCOJ execution, the engine first encodes the data into different contiguous memory structures (e.g. bitsets or sparse arrays) to leverage SIMD instructions during intersections, dynamically selecting the algorithm depending on the sparsity of the data.
LevelHeaded [13]	Building upon the hardware-accelerated WCOJ foundation of its predecessor EmptyHeaded, LevelHeaded extends from pure graph analytics to a unified engine designed for Business Intelligence (BI) and Linear Algebra (LA) workloads. While it keeps the core mechanics of multi-way set intersections and SIMD execution, it introduces two major shifts. First, to handle standard SQL aggregates and matrix mathematics without materializing massive intermediate results, it introduces a technique that pushes aggregations directly down into the WCOJ traversal (suitable for BI and LA workloads). This is done, by leaving the aggregation key attribute at the end of the join attribute order effectively using the Trie structure as the aggregator while keeping track of the final results in temporary arrays. However, this only works if the aggregations can be done iteratively (e.g. `COUNT`, `SUM`, `MAX`) and would fail with aggregations such as `MEDIAN`. Second, it completely abandons the static heuristics for attribute ordering, introducing the literature's very first cost-based optimizer for WCOJs, which dynamically selects the global variable ordering by estimating intermediate intersection cardinalities.
Umbra [8]	Umbra is a modern, general-purpose hybrid transactional/analytical processing (HTAP) database system. It introduces worst-case optimality directly into its mutable SQL environment. For query optimization, the engine does not extract a hypergraph from scratch, but instead, generates a traditional binary join plan, then employs cost-based heuristics and cardinality estimates to dynamically gather multiple binary joins into a single multi-way WCOJ operator where it is deemed optimal. To support heavy-write workloads while avoiding the expensive persistent index maintenance overhead, Umbra relies on Hash-Tries built Just-In-Time (JIT) during query execution. These JIT Hash-Tries are constructed lazily to minimize memory consumption and build time, where it only materializes the deeper levels of the trie for tuples that successfully match the join conditions at the upper levels. Ultimately, this design represents arguably the most pragmatic blueprint for conventional RDBMSs to adopt WCOJs. Because it reuses existing engine statistics, refines pre-existing binary plans, and implements WCOJs as an isolated physical operator reliant entirely on JIT indices, it can serve as an example for the integration of WCOJs into traditional relational architectures with minimal disruption.
Kùzu [14]	Kùzu is an embeddable property graph database and the evolution of GraphflowDB [15], an earlier academic prototype built to explore continuous subgraph matching and early WCOJ implementations. To integrate WCOJs efficiently at scale, Kùzu directly leverages its native graph storage rather than relying on the heavy relational index workarounds seen in prior systems. For its primary indices, Kùzu utilizes persistent adjacency lists stored natively on disk in a columnar Compressed Sparse Row (CSR) format, which serves as the foundation for evaluating multi-way set intersections. During physical execution, the WCOJ operators directly traverse and intersect these persistent CSR lists to resolve cyclic query patterns. However, to optimize complex query patterns, Kùzu supplements these persistent structures with dynamically built filters via its accumulate-semijoin-probe (ASP-Join) operator. This mechanism relies on sideways information passing (SIP) to construct JIT semi-join hash tables from the probe tuples, eagerly pruning the persistent build-side CSR lists before the full multi-way intersection phase executes. Finally, to overcome the memory explosion inherent for evaluating densely connected graphs via WCOJs, Kùzu completely abandons flat-tuple materialization, and all intermediate multi-way join results yielded from these indices are strictly maintained and passed through the pipeline in compressed, factorized representations.
DuckPGQ [16]	DuckPGQ is an extension for DuckDB [17], a popular embeddable, vectorized analytical RDBMS, that introduces standard SQL/PGQ support to a purely relational environment. Unlike native graph databases such as Kùzu that rely on persistent adjacency lists, DuckPGQ translates graph pattern matching directly into standard relational query plan. To handle the severe intermediate result explosion of these cyclic queries without using heavy trie indices, it adopts Linear-Chained hash tables [18]. This novel data structure acts as the engine's primary join index, where it uses linear probing to resolve distinct key collisions but strictly isolates duplicate keys into collision-free chains. During physical execution, instead of materializing the full combinatorial cross-product of a multi-way join, the engine evaluates the join by returning lightweight *hit-lists* (pointers to these collision-free chains), natively achieving factorized query processing. By intersecting these factorized hit-lists, DuckDB effectively evaluates WCOJs and factorized aggregations entirely within standard relational hash-join operators. Furthermore, DuckDB diverges from Umbra and LevelHeaded by making WCOJ execution adaptive at runtime. Rather than forcing the query optimizer to statically commit to a multi-way WCOJ plan based on fragile cardinality estimates, the engine begins execution normally and dynamically switches to factorized hit-list intersections only if it observes the intermediate tuple count exploding during the execution.
MillenniumDB [19], [20], [11], [21]	MillenniumDB is a graph oriented database management system that unifies diverse data models (e.g., property graphs, RDF) via their foundational *"Domain Graph"* abstraction. While other engines focus on WCOJs speed, MillenniumDB specifically tackles the storage bottleneck: the $O(d!)$ index overhead traditionally required to maintain every permutation of relational attributes in persistent tries [11]. To seamlessly adopt WCOJs without this spatial explosion, the engine uses compact data structures normally employed in compressed text indexing, most notably *the Ring* (and related extended Burrows–Wheeler transforms) [11]. Rather than materializing massive B-trees for every permutation, the Ring treats graph edges as cyclic strings of length three (source, label, target). It sorts their rotations lexicographically and encodes the resulting columns using highly compressed *Wavelet Trees* [11]. During physical execution, these wavelet trees allow the Leapfrog Triejoin (LTJ) iterators to navigate forward and backward between columns without requiring physical memory pointers. As the WCOJ operator performs multi-way intersections, it natively probes these wavelet trees, successfully achieving worst-case optimal time bounds while using *almost no extra space* beyond the raw graph data itself. Finally, because these compressed indices are so lightweight to probe, MillenniumDB diverges from static WCOJ planners by employing adaptive variable elimination orders, allowing the engine to dynamically recompute and shift the global WCOJ trie-traversal path mid-execution based on exact intermediate cardinalities discovered during the join.

pattern matching for a standard-compliant SQL engine. Such workloads commonly contain highly-connected and cyclic patterns where binary join plans are now known to be suboptimal. Thus, efficient support of the emerging graph-relational workloads requires the adoption of the more suitable WCOJs, making it not merely a theoretical consideration, but a requirement.

This leads us to ask: ***What are the reasons for the limited adoption of WCOJs in mature RDBMSs, despite the attractive theoretical promises?***

To answer this question, WCOJs are examined through a query engine abstraction inspired from PostgreSQL. This abstraction divides the query engine into three layers, as illustrated in Fig. 1. The *interpretation layer* covers query languages, functionalities, and query rewriting. The *optimization layer* consists of generating query plans, and choosing the optimal one using cost models and statistics. The *execution layer* focuses on WCOJ implementation and interaction with the hardware. Moreover, beyond the three layers, WCOJs is studied in distributed environments. Such framework is sufficient for the evaluation of WCOJs as it encapsulates the entire query lifecycle.

This study affirms previous notes from Ngo [23], a seminal figure in the field. While WCOJ algorithms are well understood in isolation, their adoption as first class operators within mature RDBMSs remains largely undefined. Practical deployment demands coordinated changes across multiple layers of the query engine, including cardinality estimation, physical plan enumeration, attribute ordering and memory management, which have been specifically designed around binary join processing over several decades.

The aim is to map the literature on a practical framework, highlight the trade-offs for engineers, and unveil possible research opportunities for researchers.

The rest of this study is structured as follows: Section II, Section III and Section IV analyze the three layers of the query engine abstraction (i.e., interpretation, optimization and execution) detailing integration challenges and the current state of adoption at each layer. Section V discusses distributed WCOJs, while Section VI reviews related work. The study is concluded in Section VII.

II. The Interpretation Layer

The interpretation layer serves as the primary interface between the user and the query engine. Its role is to parse and normalize the incoming user query, representing it as an abstract syntax tree (AST) before translating it into a relational algebra tree that is understandable by the subsequent layers.

Moreover, this layer applies logical rewriting rules to simplify the query [24]. These heuristics stretch from semantic query optimization (e.g, removing `DISTINCT` clauses when duplicates are mathematically impossible) and query flattening (e.g, subquery unnesting and `OR` expansion into `UNION`) to rearranging operations in an attempt to reduce the data volume early in the execution pipeline (e.g, predicate pushdown and eager aggregations).

A. Query Representation

Conventional RDBMSs normalize the AST of a query into a relational algebra tree [Fig. 2(e) represents a possible relational algebra tree for the SQL query in Fig. 2(b)], in an attempt to reduce the optimization task to finding the most efficient sequence of binary join operators. Although this approach is structure-independent, these engines often scan for specific patterns, such as star or snowflake schemas. These hard-coded detections allow these systems to trigger specialized transformations to optimize for those specific, pre-defined query shapes.

Despite these targeted heuristics, this model suffers from two major limitations. First, practitioners must manually hard-code detection procedures for every new pattern that emerges and implement unique transformations for each. Second, the fundamental assumption that joins are binary operations remains a bottleneck, where even the successful detection of triangles or cliques in the query structure would still yield a suboptimal query plan. Thus, the primary difficulties in query representation are not inherent from the query language semantics [e.g., it is possible to express complex cyclic patterns in SQL such as Fig. 2(b)], but rather from the internal query representations of RDBMSs, which map these semantics into binary relational algebra trees.

To resolve these limitations, systems like EmptyHeaded [12] move towards using the generalized hypertree decompositions (GHDs) extracted from the hypergraph of the query. For instance, Fig. 2(d) illustrates the hypertree decomposition for the hypergraph in Fig. 2(c) with the cyclic part isolated in a single bag, proving that they must be processed simultaneously. This approach allows the engines to partition the query into a tree of *bags* based on its structure rather than only cardinalities.

In this approach, bags containing cyclic patterns (more than two relations) are evaluated using multi-way WCOJ algorithms, while the rest of bags and intermediate results are joined using traditional binary operators. This ensures structural optimality regardless of whether the query follows a standard star or complex graph patterns.

B. Query Rewriting and Eager Aggregations

Once the relational algebra tree of the query is built, traditional engines rewrite the query using transformations that are known to improve the performance of the query without altering it. Such transformations involve pushing down aggregations and predicates towards the leaf relations to reduce input sizes before joins occur [25].

While early filtering can be highly beneficial [26], as it reduces the active domain for WCOJ iterators [6], the eager pushdown of aggregations may break the pure conjunctive nature of the query on which WCOJs have been formalized [4], forcing the optimizer to fall back to a binary execution plan.

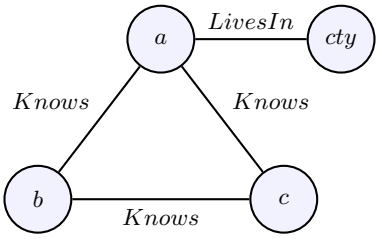
Theoretical frameworks such as AJAR [27], later integrated into LevelHeaded [13], have explored the evaluation of these aggregate-join queries without breaking worst-case optimal bounds. Their approach is to compute partial aggregations on the fly as the engine traverses the GHD of the query. However, this method is strictly constrained. First, the aggregated

```
SELECT a.id, b.id, c.id, cty.name
FROM GRAPH social_graph
MATCH (a:Person)-[:KNOWS]->(b:Person)-[:KNOWS]->(c:Person)-[:KNOWS]->(a),
      (a)-[:LIVES_IN]->(cty:City)
WHERE cty.name = 'Rabat';
```

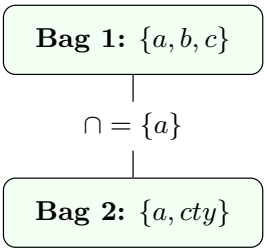
(a)

```
SELECT k1.src, k2.src, k3.src, cty.name
FROM Knows k1
JOIN Knows k2 ON k1.dst = k2.src
JOIN Knows k3 ON k2.dst = k3.src AND k3.dst = k1.src
JOIN LivesIn li ON k1.src = li.person_id
JOIN City cty ON li.city_id = cty.id
WHERE cty.name = 'Rabat';
```

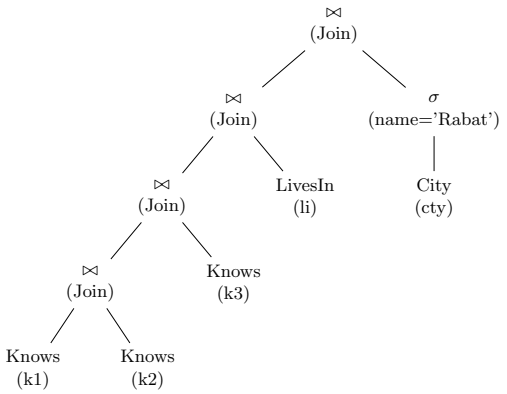
(b)



(c)



(d)



(e)

Fig. 2. (a) A SQL/PGQ query seeking a triangle of friends where one of them lives in Rabat. (b) The equivalent SQL query requiring multiple binary joins. (c) The hypergraph representation of the query. (d) The generalized hypertree decomposition of the hypergraph, isolating the cyclic pattern into the first bag and the acyclic relation into the second bag. (e) An example of the relational algebra tree to process the query.

attribute must be pushed to the end of the global join attribute order. And second, the engine is limited to commutative and iterative functions such as `COUNT` and `SUM`.

C. Functional Limitations

Traditional RDBMSs support multiple join variants, whereas WCOJ research has primarily focused on equi-joins. Supporting complex non-equi predicates (e.g., inequalities, spatial overlaps) typically requires reverting to binary joins or employing specialized geometric adaptations [28], [29]. However, these adaptations cannot be generalized to support all possible use-cases, which are inherited naturally from binary joins.

Moreover, while applying operators like `LIMIT` or `TOP-K` is trivial for the binary approach, it presents a challenge for WCOJs. In WCOJs, the global variable ordering might not align with the `ORDER BY` clause, often forcing the engine to materialize the full result before applying the limit.

Research into combinatorial output-sensitive sampling [30] and top-k algorithms [31] highlights the complexity of *"drilling down"* into a Trie to fetch a specific sample without full traversal. These limitations make WCOJs unfit for some of the most used features in conventional databases, such as pagination. As a result, query engines can always rely on traditional binary joins as a safe fallback strategy when these operations are required.

D. Discussion

Looking at how conventional RDBMSs can support emerging graph workloads via SQL/PGQ, two major paths emerge. The first consists of converting SQL/PGQ queries into standard SQL using a combination of equi-joins, unions, and filters, similar to DuckPGQ [16]. For example, Fig. 2(b) is a possible equivalent SQL query for the SQL/PGQ query in Fig. 2(a). This strategy is suitable for quick syntax support over native integration.

As Large Language Models are increasingly employed to generate queries, how can this translation layer leverage the database schema to contextualize and optimize such queries into efficient standard SQL?

The second path, which aligns with the focus of this survey, requires a generalized query representation. Manually hard-coding detection routines for every new query shape fails to scale. The current consensus leans towards the use of GHDs. However, finding a tree decomposition with the minimum width is a known NP-hard problem.

A critical open challenge revolves around the development of adaptive algorithms, or lightweight learned heuristics, that can approximate near-optimal GHDs at runtime without introducing unacceptable overhead.

Beyond representation, the interpretation layer must bridge the gap between SQL functionalities and the need for

conjunctive queries for WCOJs. Relational heuristics like eager aggregations break this pure conjunctive structure. Similarly, non-equi joins or pagination (`LIMIT`/`TOP-K`) force the engine to revert to binary execution or materialize massive intermediate results.

A few workarounds have already been proposed, such as limiting partial aggregations to commutative functions pushed to the end of the trie. However, the question still stands: *How can WCOJs natively support complex non-equi predicates, and how can engines execute output-sensitive limits without sacrificing early-pruning?*

III. The Optimization layer

The optimization layer identifies the most efficient execution plan from a wide search space of possible plans. In conventional RDBMSs, this is typically achieved through a cost-based approach, which estimates the cost of multiple plans using statistics and data summaries to select the least expensive option.

While binary join optimization has matured from dynamic programming [1] to genetic algorithms [32], WCOJ algorithms require fundamental changes. The cost metric must shift from minimizing intermediate result sizes to minimizing intersection costs, a complex task that would require access to new statistics, a way to define the optimal attribute order, and the ability of hybrid planning (i.e., using both binary and WCOJs in a query plan).

A. The Objective Function

The primary challenge for WCOJ adoption lies in the objective function of the optimizer. While the goal of a conventional optimizer is to arrange joins in a tree structure while minimizing the cardinalities of intermediate results, one that integrates WCOJs must focus on finding the optimal permutation of the join attributes. This ordering should minimize the *width* of the value domains, ensuring that the most selective intersections are performed first to avoid invalid paths earlier.

This divergence in the optimizer goals, from minimizing the data volume (focusing on memory) to minimizing search steps (focusing on computation) creates a conflict. Thus, plans that appear expensive to a traditional optimizer might actually be the most efficient for an optimizer leveraging WCOJs.

To resolve the discrepancy between the two paradigms, the query planner must be refined on all fronts, from the addition of more suitable statistics, the adoption of newer heuristics, to enabling the ability to yield hybrid plans, and with it new unified cost models. These aspects are covered in the rest of this section.

B. Statistics and Cost Estimation

For a cost-based optimizer to make informed decisions, accurate estimations are essential. Traditional engines rely on value histograms and lists of the most common values. However, these scalar (i.e., flat, one-dimensional) statistics fail significantly for cyclic queries because they assume attribute independence, leading to severe underestimations of join sizes.

In contrast, WCOJ performance is heavily tied to the structure of the join query, making existing scalar statistics unreliable. WCOJ optimizers would require *graph statistics*, such as the degrees distributions [33], correlation information, fractional edge covers [7], [4], or graph-specific summaries [34], to map how data points are interconnected. Conventional RDBMSs do not natively support these expensive graph statistics, forcing WCOJ optimizers to operate with low-fidelity estimates.

Other potential techniques for accurate estimation include synopsis-based methods, which build summary structures for cardinality estimations, and sampling methods, which use data samples to estimate cardinalities without execution [34]. However, these approaches are especially hard to adapt for WCOJ, since the probability of capturing cyclic join results in a subset of data is smaller, which may lead to catastrophic underestimations.

C. Hybrid Planning

WCOJs are not substitutes for binary joins. Real-world workloads rarely consist of purely cyclic or acyclic patterns, necessitating query planners capable of yielding hybrid plans that mix between binary and WCOJs [35].

While establishing a unified cost metric remains an active challenge [35], the GHD provides a robust theoretical framework [12]. By decomposing a query into a tree of *bags*, the planner can isolate cyclic components to be resolved via WCOJs, while utilizing traditional binary joins to assemble the final result from these acyclic interconnections. For example, considering the GHD in Fig. 2(d), an optimizer would evaluate Bag 1 (the cyclic triangle) using a multi-way WCOJ, then use a binary join on the shared attribute, a, to attach the acyclic relation in Bag 2.

Free Join [9] pushes this hybridization further to the execution layer. Rather than choosing between distinct operator types, it utilizes lazy hash-based data structures to maintain the high-throughput, scan-centric benefits of traditional binary hash joins. Free Join coordinates these hash probes through a global variable-ordering logic. The algorithm satisfies the AGM-bound while reducing the pointer-heavy memory overhead of traditional WCOJs through its *column-oriented lazy trie*, a unified data structure that builds inner sub-tries as needed.

D. Cost Modeling

To compute the cost of hybrid plans the optimizer needs a unified metric to compare with. In traditional RDBMSs, costs are typically modeled as a function of input and output sizes. This is a result of the assumption that cost is monopolized by intermediate results materialization.

In contrast, WCOJ cost relies heavily on the structure of the query, since its complexity is dictated by the fractional edge cover of the query hypergraph. However, using the AGM-bound as a cost model can be unreliable, since it will often overestimate the actual runtime. This approach can cause the optimizer to reject valid WCOJ plans, as it computes the worst-case cost, whereas in real life workloads, datasets have high skew or sparsity making it orders of magnitude

smaller than estimated. Moreover, if the optimizer relies on a pessimistic estimation, it will frequently perceive WCOJs as prohibitively expensive, defaulting to a binary join tree plan.

To address this, researchers have proposed the *intersection cost* (*i-cost*) as a unified metric for hybrid plans [35]. Unlike the AGM-bound, *i-cost* focuses on the computational effort of the execution rather than the theoretical maximum size of the output. *i-cost* represents the sum of sizes from all adjacency lists accessed during a multi-way intersection. By using degree distributions to estimate the size of these adjacency lists, the optimizer can move away from global pessimistic bounds and toward a local, data-aware estimation.

E. Attribute Ordering

Once the optimizer chooses to use a WCOJ algorithm within a plan, it must determine a suitable attribute order. It is well understood that a poor choice in the determination of the order in which attributes must be processed can severely degrade performance, as the algorithm might keep searching in empty sub-trees rather than pruning early [6], [3]. To this end, several approaches can be employed.

1) Static heuristics: A join can be seen as a constraint satisfaction problem, thus standard techniques may apply. The query optimizer can base its decision on inexpensive heuristics to avoid expensive computations.

For instance, the *"max-degree"* approach prioritizes the most constrained attributes (i.e., attributes shared by most relations). Since each relation limits further the value domain of the attribute, ordering attributes by descending degree often result in early pruning [36], [6].

Moreover, because such heuristics are computationally inexpensive, they can also serve as a fallback strategy for the more advanced strategies discussed below.

2) Cost-based ordering: Static heuristics are fast because they focus solely on query structure, however, they are *instance independent* (i.e., blind to the actual data). Consequently, they may prove inadequate. For instance, a high-degree attribute might have low selectivity (e.g. a boolean value), making it a suboptimal starting attribute.

Cost-based ordering attempts to resolve this by estimating the actual runtime cost of each attribute permutation using statistics. Thus, instead of merely counting the hyperedges to calculate degrees, the optimizer computes an estimated cost for each permutation [35], [12], [37].

This process may leverage existing statistics, such as histograms and information on unique values, to estimate the intersection cost. However, the optimizer must account for search cost. For instance, when the data is dense, sequential scans may be cheaper than leapfrog searches, conversely, when the data is sparse, leapfrog searches are more efficient.

Finally, since cost estimation can be expensive, it cannot be calculated for all possible permutations which is factorial to the number of attributes, the cost-based ordering can use a greedy approach, where it seeks the next cheapest attribute using dynamic programming similar to the ones used by traditional planners but adapted for linear ordering, rather than trees [35].

3) Adaptive and learned optimization: The integration of adaptivity and machine learning into database management systems, particularly at the query optimization layer, has emerged as an effective paradigm [38]. When applying these techniques, research generally divides into two distinct approaches: workload-level learning and query-level adaptivity.

The first approach applies learned optimization at the workload level (*inter-query*), continuously refining the decision-making of the optimizer across successive executions. ReJOOSp [39] and AlphaJoin [40], for example, model optimization as a reinforcement learning problem. By exploring the massive combinatorial space of join orders over time, these agents learn to identify optimal execution paths and structural boundaries that static, rule-based heuristics miss.

The second approach pushes adaptivity directly into the physical execution of a single query (*intra-query*). Rather than committing to a single static plan, the engine initiates execution by running a multitude of distinct plans simultaneously for a brief exploratory period. By leveraging immediate runtime feedback from these short, parallel executions, the optimizer dynamically identifies and transitions to the most efficient plan used for the rest of the query execution. For instance, ADOPT [41] uses this adaptivity to rapidly converge on near-optimal attribute orders for plans using WCOJs.

F. Discussion

When integrating WCOJs within the optimization layer, two primary challenges emerge. First, the query engine requires new generalized cost models. Traditional intermediate cardinality estimations are unsuitable for WCOJs. The current consensus relies on instance-dependent metrics, such as the intersection cost (*i-cost*), and degree distributions to measure the exact computational effort [35].

These new cost models must eventually align with the ongoing shift toward learned query optimization. While machine learning has already proven its ability to capture complex data correlations compared to traditional cardinality estimation, as highlighted in the Cambridge Report [42], extending this approach to reliably predict the sizes of cyclic queries without introducing unacceptable compilation overhead remains an open challenge.

Furthermore, the objective of the optimizer must fundamentally shift from reordering a binary join tree to finding the optimal join attribute order for WCOJ processing. *While static and cost-based heuristics offer practical starting points, dynamically discovering the optimal attribute order at runtime is still an active [41], open research question.*

Second, because WCOJs are not substitutes for binary joins, optimizers must support hybrid planning. The query engine must be capable of yielding hybrid plans that deploy WCOJs when beneficial. To achieve this without relying on complex hypergraph decompositions, modern engines explore alternative heuristics. For instance, systems like Umbra [8] generate a standard binary join plan and combines specific sub-trees into multi-way WCOJ operators based on cardinality estimates. Other approaches, such as Free Join [9], avoid separating operators entirely and deploys a unified data

structure to integrate binary hash-join efficiency and multi-way optimality into a single physical operator.

In this regard, a major open challenge is the development of new approaches for query planners to accurately measure cost and evaluate these hybrid plans, or adaptively trigger multi-way operator combinations at runtime, without suffering from the massive combinatorial search space explosion.

IV. The Execution Layer

The execution layer is where the query engine translates the logical plan from the optimizer into concrete physical operations. Here, the theoretical advantages of WCOJs confront the physical constraints of modern hardware.

While the asymptotic complexity of the algorithm is determined by its logic, the actual runtime performance depends heavily on its implementation and interactions with CPU pipelines, memory hierarchies and storage. This section studies the gap between the mature binary joins, which have been optimized for modern hardware over decades, and the emerging WCOJs.

Traditionally, binary joins in relational engines rely on a scan-centric execution model, either sequentially scanning sorted arrays (sort-merge join), or performing high-speed calculations to probe hash tables (hash join). However, WCOJs rely heavily on a search-centric approach, jumping over large sections of data that cannot participate in the join.

This paradigm is shared by the two primary implementations: the *sort-based* approach, spearheaded by the Leapfrog Triejoin [6], which relies on the global variable order, where instead of scanning every tuple sequentially, its iterators *leapfrog* over non-shared values. And on the other hand, the *hash-based* approach, characterized mainly by Generic Join [43], which avoids global sorting by resolving variable bindings using nested hash lookups. While theoretically superior for cyclic queries, both approaches disrupt the sequential flow that the hardware expects.

A. Computation

The structural shift from scanning to searching in WCOJs creates inefficiencies at the processor level. Modern general-purpose CPUs rely heavily on instruction pipelining and branch prediction to maintain throughput, a design that favors the predictable control flow and sequential access patterns of traditional binary joins [44].

WCOJs are inherently branch-heavy. For example, Leapfrog Triejoin requires a loop of conditional jumps, since it constantly needs to check if any of the involved iterators is less than the current maximum value [`if key < max_key then seek()`]. Because the branch direction depends entirely on the data distribution, this unpredictability triggers frequent pipeline flushes, stalling execution cycles and significantly reducing the instructions-per-cycle compared to binary joins.

Furthermore, the hierarchical nature of WCOJs significantly complicates vectorization. While binary hash joins can leverage flat data structures and SIMD to probe hash tables in batches [45], [46], Trie traversal is data-dependent and difficult to parallelize through a single instruction stream [47]. Although recent work attempts to bridge this gap by mapping joins to matrix multiplication [48], the lack of *vectorized search* primitives remains a significant blocker for adopting WCOJs in vectorized engines. To address this, systems like EmptyHeaded [12] adaptively encode Trie nodes based on the data sparsity, translating complex pointer traversals into parallelizable SIMD set intersections.

B. Primary Memory

The performance of in-memory query processing is often dominated by memory latency rather than CPU throughput [49]. Thus, even if the CPU pipeline is perfectly saturated, the runtime performance would likely suffer waiting for data to be fetched. In this regard, the result of the structural differences between binary and WCOJs becomes more severe.

Typically, binary joins process sequential memory pages, a pattern that aligns with hardware prefetchers loading data into the L1 and L2 caches before it is requested. In contrast, WCOJ requires navigating Tries, which involves pointer chasing, following references from parent to child nodes that are scattered over different memory pages and only known after the parent is retrieved. This serial dependency defeats hardware prefetching, leading to a higher rate of cache misses. As a result, the WCOJ algorithms often spend more time waiting for main memory than performing intersections [50].

C. Hierarchical Indexing

WCOJ algorithms require prefix trees (e.g. Tries) during the intersections processing. Such uncommon indices are not supported in conventional RDBMSs. This creates a trade-off between building indices Just-In-Time (JIT) or maintaining persistent indices.

In the case of JIT index building, as implemented in DuckDB [18] and Umbra [8], the benefit of worst-case optimality must outweigh the overhead of building the index. For *one-time* analytical queries, the construction cost can dominate the total execution time, potentially making a binary plan faster despite its theoretical inferiority.

On the other hand, LogicBlox [10], resolve to maintaining persistent Trie indices as their primary storage. While this eliminates the JIT build cost, it shifts the burden to data ingestion and updates, but is totally acceptable for read-only engines or with limited updates, or systems employing a sixth normal form (6NF) data architecture [6].

Middle ground between these two approaches may be found by maintaining light indices (mostly flat) and building the more expensive indices during query execution [51]. The border between these two approaches is a gray area, for instance, Umbra maintains flat data structure for a faster building stage of their Tries [8]. Recent efforts work on making Tries, more update-friendly [52].

D. Secondary Memory

Traditional RDBMS are required to be able to handle datasets, even when their sizes exceeds main memory capacity. For binary joins, this problem has been solved via GRACE

Hash Join [53] for instance, which partitions inputs into disk-resident buckets that are processed sequentially.

However, spilling to the disk is far more penalized in WCOJs. As discussed previously, they rely on random accesses, which is orders of magnitude slower than sequential reads in disk, even on NVMe SSDs. Moreover, theoretical I/O-optimal algorithms for cyclic joins remain an open problem [54].

Partial solutions to these challenges, however, have seen the day. Approaches like Quadtrees [55] attempt to decouple the topology from the data values, storing the structure in compact bitmaps. More advanced systems, such as the Ring [11] and CompactLTJ [21], go further by enabling execution directly on the compressed representation. This "Computation over Compression" approach effectively decreases memory footprint, partially mitigating the main memory issues described above, and attempts to avoid the need to spill to disk.

Umbra [8] uses LeanStore's [56] pointer *swizzling* technique, and leverages its architecture designed around flash-based storage, effectively using disk as main memory while introducing minimal overhead.

E. Workload-Aware Execution

In real-world scenarios, queries rarely run in isolation. Workload-aware techniques exploit overlapping patterns across such queries, which can improve the overall workload runtime performance dramatically.

During a workload, parts of the joins may be shared across multiple queries, the engine might choose to cache the results of those common parts for future use, which would cut the total execution time depending on their occurrence and size. However, WCOJs avoid to materialize intermediate results, which prevents them from being cached.

Attempts in this regard include Flexible Caching [57], where instead of materializing full intermediate results, it caches partial results at carefully chosen bags from the tree decomposition of the query's hypergraph. This enables reusing repeated sub-joins when beneficial while still retaining the advantages of WCOJs.

F. Discussion

As explored in this section, while WCOJs successfully bound the asymptotic complexity of cyclic queries, they confront the physical limitations of modern hardware. The execution of traditional binary joins aligns with CPU pipelining and memory prefetching. In contrast, WCOJs reliance on unpredictable pointer-chasing during Trie traversals breaks this alignment. This mismatch between algorithmic design and hardware constraints paves the way for a multitude of open challenges and research questions.

One of the main research paths within the execution layer is hardware acceleration. While the majority of existing literature focuses on multi-core CPUs, leveraging the low latency to manage complex and branch-heavy workloads, there is a critical need to explore GPU offloading to leverage massive data parallelism and high throughput during join processing.

Furthermore, FPGAs present another interesting alternative, offering custom, high-throughput data paths alongside superior energy efficiency. FPGAs can especially be versatile when handling WCOJs. They can be deployed in-line between the CPU and disk to accelerate decompression and data filtering, placed on the same socket as the CPU to offload specific computational tasks, or utilized as external, high-throughput units similar to GPUs.

Beyond designing WCOJ variants tailored to these specific hardware topologies, a critical research trajectory is the development of a heterogeneous WCOJ algorithm, capable of dynamically routing specific sub-tasks to the most suitable compute unit.

The optimal use of these compute units, and particularly their vector processing capabilities (SIMD/SIMT), is dependent on the underlying index structures, which must be redesigned to be vector-friendly, such as sparsity-aware layouts that map intersections directly to bitwise instructions as proposed by EmptyHeaded [12]. However, while transitioning to such layouts unlocks hardware throughput, it introduces new complexities regarding mutability and state management across devices.

These structures are optimized for read-heavy workloads where data density is predetermined. In dynamic database environments with frequent insertions, continuously monitoring data sparsity and rebuilding or transferring these index layouts between the CPU, GPU, and FPGA imposes massive synchronization and computational overheads, becoming a bottleneck to the very execution pipelines they are meant to accelerate.

One of the possible solutions is to map these joins into matrix multiplications, for which GPUs are optimized. However, this approach is heavily affected by data sparsity. Thus, it is essential to define thresholds at which mutable relational data should transition between a sparse array and a dense SIMD-friendly bitset at runtime. Another interesting research path in this regard, is determining cost models to dynamically route WCOJ parts among available compute units (e.g., dispatching dense, matrix-mapped joins to GPUs while keeping sparse, pointer-heavy intersections on CPU).

Beyond compute orchestration, multi-way intersections suffer from severe limitations across the memory and storage hierarchy. In main memory, the massive bandwidth required to continuously feed the accelerators creates a *"memory wall"* bottleneck. Furthermore, as massive workloads are inevitably pushed out-of-core, WCOJs face critical I/O challenges. Unlike traditional binary joins that adopt well-established, sequential disk-spilling mechanisms, WCOJs depend on traversing deeply nested index structures. Spilling these complex structures to secondary storage (NVMe/SSDs) results in severe I/O thrashing.

The primary open challenge here consists of bridging between byte-addressable main memory and block-addressable secondary storage during multi-way joins. While state-of-the-art systems like Umbra [8] have proposed low-overhead out-of-core execution using their pointer swizzling technique, where they translate between memory addresses and disk offsets, this only reduces translation costs. They do not resolve the fundamental I/O bottleneck caused by

WCOJ access patterns. Because attribute-at-a-time evaluation requires traversing hierarchical index structures across all participating relations, the unpredictable pointer-chasing disrupts the block-oriented access required for efficient secondary storage. Designing an architecture capable of prefetching massive, sparse index sub-trees without major overhead remains a critical unsolved challenge.

V. The Distributed Layer

While distributed RDBMSs deserve a survey of their own, they share the issue of limited WCOJ adoption. Concepts within the interpretation layer can be shared among shared-memory and distributed RDBMSs. However the difference shows up when it comes to the optimization and execution layers. This is the result of the core shift in objectives.

While shared-memory RDBMSs focus on minimizing the memory latency, the distributed ones focus on the network bandwidth. A much stricter constraint, since data movement through the network can be slower by multiple orders of magnitude. Thus, distributed RDBMSs require a complete rethinking of how WCOJs are processed at scale [3].

Traditionally, distributed binary joins rely on hash or range partitioning. For example, in the case for the hash approach, to join $R(a, b)$ and $S(b, c)$, the engine hashes on the join key b on both relations, and sends each tuple to the adequate node (e.g. `hash(b) % #nodes`), which guarantees that all matching tuples will land on the same node, and proceeds with local joins independently [58].

However, for WCOJs, this hashing technique would fail. For instance, in the triangle query, $Q = R(a, b) \bowtie S(b, c) \bowtie T(a, c)$, hashing on attribute b would only bring relations R and S together but not T (since it has no attribute b). To resolve this issue without losing worst-case optimality, the engine needs to adopt a higher-dimensional distribution approach.

A. The HyperCube Shuffle

The standard solution for distributed multi-way joins is *HyperCube Partitioning* [58]. The cluster is organized into a virtual high-dimensional grid, as illustrated in Fig. 3. For the triangle query involving variables a, b, c, the P processors are arranged into a cube of dimensions such as $P_a \times P_b \times P_c = P$.

Each relation is broadcast along the dimensions it misses. For example, a tuple $R(a, b)$ is hashed to coordinates $(hash(a), hash(b))$. Since it lacks variable c, it must be sent to all nodes corresponding to $(hash(a), hash(b), *)$. Effectively, it is replicated P_c times.

This approach guarantees that for any possible triangle (a, b, c), all participating tuples from R, S, T will meet at exactly one unique intersection point in the grid. However, this introduces a massive *replication cost*. Since unlike binary shuffles which move data once ($O(N)$), HyperCube shuffles multiply the data volume ($O(N \times P^{1-1/\rho^*})$) [59]. Thus, HyperCube shuffling would only work if the gains from the WCOJ outweigh the losses incurred by shuffling through the network.

B. Handling Data Skew

One of the major challenges that distributed RDBMSs have to face is data skew, and in WCOJs, skew can be much harder to handle since it is multi-dimensional. The complexity is no longer because of frequent key values, but also of the structure, resulting in large replications.

Multiple unique issues can arise from data skew in distributed WCOJs. For instance, in HyperCube shuffling, if a single node is assigned a dense region of the hypergraph, it might run out of memory, or have the bulk of the work to process (more than all other nodes), thus delaying the entire query.

To resolve these issues, hybrid processing is normally relied on, where high-degree vertices are treated differently, often being broadcast to all nodes to parallelize their intersection workload, while low-degree vertices are hashed normally. This results in a hybrid approach that dynamically adapts to the structure of the query and the frequency of values [60].

While Myria [61], [3] successfully proved that the communication bounds of hypercube partitioning are optimal (it transfers less data across the cluster than traditional binary hash joins), it was frequently bottlenecked by memory due to the heavy replication. To resolve this, subsequent approaches such as BiGJoin [62] leverage the Timely Dataflow computation model [63] to stream-process Generic Join [7].

By pairing the continuous execution with on-the-fly skew handling, the system maintains a low memory footprint throughout the entire query execution. Furthermore, modern implementations like EdgeFrame [64] abandon the hypercube shuffle entirely. Instead, they broadcast heavily compressed graphs directly to worker nodes, thus trading main memory capacity for a WCOJ execution without the need for shuffling.

C. Network Optimization

Given the high cost of replication, modern efforts focus on reducing the payload before the communication happens. One of such approaches develop on similar techniques from distributed binary joins and use Bloom filters [65]. Before the massive shuffle, nodes only exchange tuples that *might* be in the final result. Tuples that do not make the cut are thus discarded locally, preventing them from consuming valuable bandwidth. This is especially effective when the joins are highly filtered which should be the case for cyclic queries more often than not.

Other approaches can focus on the data format itself. For example, *Adaptive Factorization* [18] would be suitable for such an environment, sending data in a compressed form, where instead of sending redundant tuples $(a, b_1), (a, b_2), \ldots$, the system sends a grouped representation $a \rightarrow \{b_1, b_2, \ldots\}$. This *structural compression* can significantly reduce the serialization overhead during the HyperCube shuffle, directly addressing the replication penalty.

D. The Multi-Round Shuffle

While the HyperCube shuffle minimizes latency by completing the join in a single communication phase, it

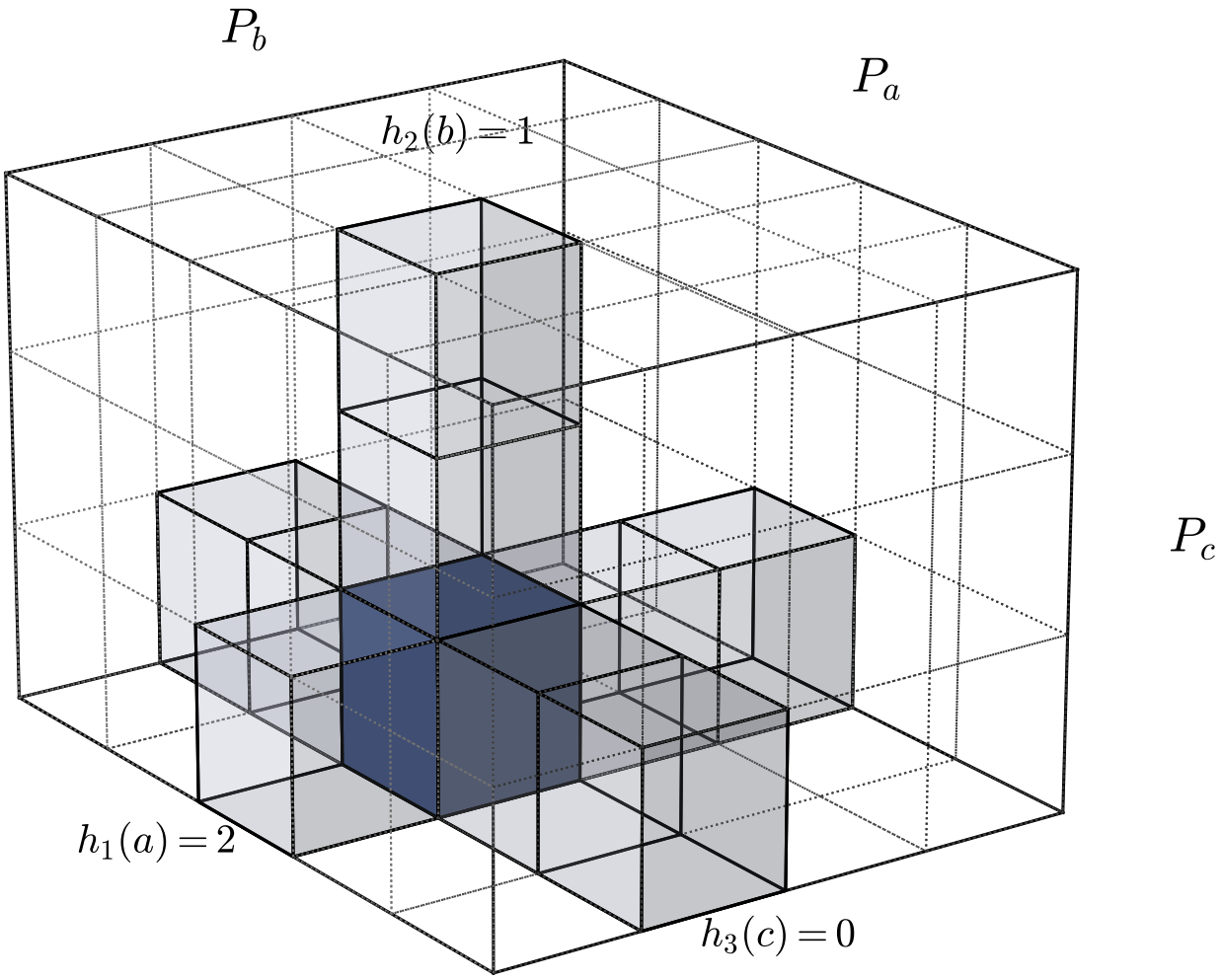


Fig. 3. An illustration of the HyperCube shuffle organizing a cluster into a virtual 3D grid with dimensions $P_a \times P_b \times P_c$ for a triangle query. A tuple lacking a variable is hashed to its known coordinates, such as $h_1(a) = 2$ and $h_2(b) = 1$, and replicated across the missing P_c dimension. This approach guarantees that all participating tuples from the queried relations will meet at exactly one unique intersection point in the grid to evaluate the join.

requires a high bandwidth cost due to the massive data movement and replication. To resolve this issue, especially for environments with limited bandwidth or even for complex queries for which HyperCube shuffling would require excessive duplication, multi-round approaches offer a possible alternative.

As formalized by Ketsman-Suciu [66], single-round algorithms are often bounded by the fractional vertex cover of the query hypergraph. By allowing the computation to proceed in $O(\log P)$ rounds, the maximum load per worker can be reduced to $O(m/P^{1/\rho^*})$, where m is the size of the largest input relation. This bound is asymptotically superior for many cyclic queries, thus trading rounds for memory and bandwidth efficiency.

Multi-round algorithms typically decompose the query execution into multiple iterative steps [67]. Instead of resolving the entire cyclic structure in one massive shuffle, the system may perform a sequence of semi-join reductions or compute partial intersections (e.g., joining subsets of relations to limit the search space) before reshuffling the reduced intermediate results for the final join. This removes non-joining tuples early, and prevents them from being replicated across the cluster needlessly.

While this approach might significantly reduce the total volume of data moved across the network, minimizing the risk of congestion, it introduces multiple synchronization barriers, which in the distributed setting, it requires the entire cluster to wait for the slowest node to complete the current round before proceeding to the next one. For short-running or interactive queries, the accumulated tail latency of multiple barriers often outweighs the bandwidth savings, making single-round approaches preferable despite their higher data cost.

E. Discussion

Within distributed environments, the primary performance constraint shifts from main memory latency to network bandwidth. Because one-dimensional hash partitioning fails to colocate cyclic query structures, the current consensus

relies on HyperCube shuffling to route multi-dimensional data. To prevent the resulting data replication from entirely exhausting network bandwidth, distributed engines heavily rely on payload reduction techniques prior to communication. This includes applying pre-shuffle Bloom filters to discard unneeded tuples [65] and using adaptive factorization to compress the remaining payload during transit [18].

As the database industry increasingly adopts cloud environments with on-demand node allocation, a major research opportunity is the design of elastic execution engines. In such systems, cluster expansion (or shrinkage) could be triggered during runtime. For instance, the query optimizer might choose to scale resources mid-query, or this decision could be pushed directly to the execution layer, where early data scans can provide instance-dependent information to achieve an optimal balance of performance and cost.

This paradigm shift towards adaptive systems extends far beyond dynamic cluster sizing. This broader scope of runtime adaptivity presents multiple interesting research opportunities, particularly regarding how query engines manage data preparation for transit and algorithmic choice.

For instance, adaptivity can be integrated directly into the filtering and compression stages. Rather than universally applying these reduction techniques, a system could dynamically determine their necessity based on runtime data characteristics. For example, a small dataset might be shuffled in its entirety, intentionally avoiding the computational overhead associated with semi-join filtering or structural compression.

Furthermore, another critical research direction involves enabling the optimizer to dynamically choose between single-round and multi-round execution. Implementing this effectively would require the development of robust cost models and runtime heuristics capable of accurately comparing the bandwidth costs of a single-round HyperCube shuffle against the accumulated synchronization latencies found in the multi-round approach.

VI. Related Work

Worst-case optimal joins are indeed a *hot topic* in database research. However, the broader shift to handle the ever evolving volume and complexity of modern data workloads [42] must be recognized.

While this survey specifically focuses on the practical aspects of integrating WCOJs into mature RDBMSs, it builds upon a large foundation of prior literature. An extensive body of surveys, authored by seminal figures in the field, covers the underlying formal theory and the proof frameworks for worst-case optimality in detail [23], [7]. Furthermore, this analysis complements a previous work [5] that studies the algorithmic aspects of this class of multi-way joins.

Beyond the theoretical and algorithmic foundations, the integration of artificial intelligence and machine learning has emerged as a transformative paradigm across computer science, and database research is no exception.

At the interpretation layer, this AI-driven evolution is most manifested in Natural Language Interfaces to Databases (NLIDB). Specifically, Text-to-SQL generation has transitioned from rule-based semantic parsing to leveraging deep learning and Large Language Models (LLMs). Comprehensive surveys [68] have studied the different models, and the upcoming challenges.

Moreover, modern benchmarks like Spider [69] and BIRD [70] demonstrate the increasing capacity of LLMs to autonomously navigate complex, cross-domain database schemas [71]. Such advancements are deeply connected with the necessity for WCOJs, as AI lowers the barrier for end-users to generate highly complex, multi-way join queries.

The optimization layer is undergoing a similar transformation. Traditional cost-based optimizers are increasingly being improved or replaced by learned cost models [42]. A critical driver of this shift is the rapid advancement in cardinality estimation [72], [73], leveraging deep neural networks and advanced statistical models capable of capturing high-dimensional correlations within data distributions. This is directly relevant to the adoption of WCOJs, as the multi-dimensional skew and structural correlations inherent to cyclic graph queries have historically caused severe underestimations in traditional approaches.

Beyond estimations, adaptive techniques and reinforcement learning are being deployed to refine statistics or dynamically adjust execution plans during running queries. While these methods do not always outperform mature deterministic algorithms across all scenarios, they can offer huge performance improvements in complex, highly correlated workloads [74], [75], [42].

The efficacy of these adaptive and learned optimizers is even more necessary because of the modern cloud environments [76]. The database industry has widely adopted cloud-native architectures that are decoupled from storage and compute, enabling an unprecedented degree of scalability and flexibility [77]. For the evaluation of WCOJs, this elasticity can be transformative, since such systems can adopt optimizers that anticipate the explosive cyclic queries that will induce processing spikes, and provision compute resources on demand. Previous work [78] addressed handling data skew in a graph query within such an elastic environment, which showcases the little overhead paid for the possible large gains.

A prominent research area in modern database systems is hardware acceleration. To overcome the physical execution bottlenecks of increasingly complex workloads, the database community is actively exploring hardware-software co-design. As detailed in comprehensive surveys across the domain, this research typically focuses on three paradigms: the design of special-purpose CPUs and advanced vectorization techniques, the utilization of Graphics Processing Units (GPUs) to exploit massive thread-level concurrency [79], [80], and the development of custom data pipelines using Field-Programmable Gate Arrays (FPGAs) to accelerate all or distinct parts of the query execution engine [81].

VII. Conclusion

This survey explored the practical considerations for the adoption of worst-case optimal joins, specifically focusing on conventional relational database management systems. This

TABLE II. SUMMARY OF INTEGRATION CHALLENGES AND PROPOSED SOLUTIONS FOR WCOJS ACROSS DBMS LAYERS

Layers	Integration Challenges	Proposed Solutions
Interpretation	Representing queries in relational algebra trees requires hard-coded procedures to detect cyclic query patterns.	Using generalized hypertree decompositions (GHDs) of the query hypergraph to isolate cyclic bags [12], [35].
	Eager aggregation breaks the pure conjunctive query structure required by the AGM-bound.	Evaluating aggregate-join queries via on-the-fly partial aggregations restricted to iterative functions [27], [13].
	WCOJs do not natively support common functionalities such as non-equi joins and `TOP-K` clauses.	Employing geometric adaptations for non-equi predicates [28], [29] and combinatorial sampling for output limits [30], [31].
Optimization	Relying on scalar statistics that assume attribute independence leads to cardinality underestimations for cyclic joins.	Introducing graph statistics such as degree distributions [33] and synopsis-based estimations [34].
	Inability of query planners to yield hybrid execution plans that combine traditional binary joins with WCOJs.	Utilizing GHDs to isolate multi-way bags [12], or deploying unified data structures that fuse binary and multi-way execution [9].
	Using the theoretical AGM-bound as a cost metric, which overestimates actual costs and causes optimizers to reject valid WCOJ plans.	Adopting data-aware metrics like the intersection cost (i-cost) to measure computational effort rather than theoretical output size [35].
	Suboptimal join attribute ordering degrades WCOJ performance.	Applying max-degree static heuristics [6], cost-based greedy ordering [35], or reinforcement learning [39], [41].
Execution	Branch-heavy, search-centric Trie traversals causes unpredictable control flows that stall CPU pipelines and hinder vectorization.	Translating pointer traversals into parallelizable SIMD set intersections [12] and mapping joins to matrix multiplication [48].
	The severe trade-off between expensive Just-In-Time (JIT) index construction overhead and the massive storage footprint of persistent Tries.	Employing 6NF data architectures [6], maintaining lightweight flat indices for fast JIT construction [8], [51], or utilizing update-friendly Tries [52].
	Random access patterns of WCOJ iterators severely penalizing performance when datasets exceed main memory and spill to disk.	Operating directly on compressed structures like Quadtrees [55] and Wavelet Trees [11], [21], or leveraging pointer swizzling [8], [56].
	The inherent avoidance of intermediate result materialization in WCOJs preventing traditional workload-aware caching mechanisms.	Implementing flexible caching of partial results at specific GHD bags to reuse repeated sub-joins [57].
Distributed	Traditional one-dimensional hash partitioning fails to colocate the complete structure of cyclic graph queries.	Employing HyperCube partitioning to enable multi-dimensional data distribution [58].
	Severe multi-dimensional data skew exhausting individual node memory and delaying query execution.	Implementing hybrid vertex broadcasts [60], stream-processing via Timely Dataflow [62], [63], or broadcasting compressed graphs [64].
	Massive data replication inherent to HyperCube shuffles bottlenecking network bandwidth.	Filtering unneeded tuples via pre-shuffle Bloom filters [65] and applying Adaptive Factorization for structural compression [18].
	Accumulated tail latency from synchronization barriers in multi-round execution models.	Bounding computation to logarithmic rounds [66] or performing sequences of semi-join reductions prior to the final shuffle [67].

analysis is organized using a simplified query engine, inspired by PostgreSQL, to evaluate the integration attempts and their limitations at each layer. And with the growing shift to cloud computing, the adoption of WCOJs within distributed environments is also examined.

This study affirms that despite the theoretical guarantees, WCOJs integration is constrained by decades of optimization designed around binary joins. Nevertheless, their practical value is undeniable, as demonstrated by the emergence of modern, specialized systems built natively around them.

DBMS development is a complicated and slow process where correctness, completeness and stability generally take precedence over performance optimization. Often, new workloads can be accommodated via translation layers rather than a complete engine rewrite. For example, DuckPGQ successfully maps SQL/PGQ graph queries into SQL, thus removing the need for native execution. On the other hand, a rapid adoption of vector databases has been observed in recent years, because vector similarity workloads cannot be substituted by traditional relational operations, the industry was forced to adopt native solutions almost instantly.

Historically, theoretical superiority has not guaranteed practical adoption, as happened with Yannakakis algorithm for instance. However, the trajectory of WCOJs appears more promising. The sustained volume of research published across major database venues over the past decade, coupled with proven practical performance in specific workloads, suggests a slow but steady shift toward broader mainstream acceptance.

Ultimately, this survey aims to successfully cover this interesting research path, and act as an accessible entry point for newcomers and for researchers that are looking for new challenges. Table II provides a clear summary of engineering trade-offs for practitioners attempting to integrate WCOJs in their systems.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers and the editors of the journal for their valuable feedback and constructive suggestions, which significantly contributed to improve the clarity and overall quality of this manuscript.

DECLARATION ON GENERATIVE AI

During the preparation of this work the author(s) used Google Gemini in order to refine and reformulate the original

drafts, reorganize the text and improve clarity and readability of the content. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication.

REFERENCES

[1] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '79. New York, NY, USA: Association for Computing Machinery, 1979, p. 23–34. [Online]. Available: https://doi.org/10.1145/582095.582099

[2] P. Fuchs, P. Boncz, and B. Ghit, "Edgeframe: Worst-case optimal joins for graph-pattern matching in spark," in *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, ser. GRADES-NDA'20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3398682.3399162

[3] S. Chu, M. Balazinska, and D. Suciu, "From theory to practice: Efficient join query evaluation in a parallel database system," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 63–78. [Online]. Available: https://doi.org/10.1145/2723372.2750545

[4] A. Atserias, M. Grohe, and D. Marx, "Size bounds and query plans for relational joins," in *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, 2008, pp. 739–748.

[5] A. Berdai and D. Chiadmi, "Attempts in worst-case optimal joins on relational data systems: A literature survey," in *2023 IEEE 6th International Conference on Cloud Computing and Artificial Intelligence: Technologies and Applications (CloudTech)*, 2023, pp. 01–08.

[6] T. L. Veldhuizen, "Leapfrog triejoin: a worst-case optimal join algorithm," 2013. [Online]. Available: https://arxiv.org/abs/1210.0481

[7] H. Q. Ngo, C. Ré, and A. Rudra, "Skew strikes back: new developments in the theory of join algorithms," *SIGMOD Rec.*, vol. 42, no. 4, p. 5–16, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2590989.2590991

[8] M. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann, "Adopting worst-case optimal joins in relational database systems," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 1891–1904, Jul. 2020. [Online]. Available: https://doi.org/10.14778/3407790.3407797

[9] Y. R. Wang, M. Willsey, and D. Suciu, "Free join: Unifying worst-case optimal and traditional joins," *Proc. ACM Manag. Data*, vol. 1, no. 2, Jun. 2023. [Online]. Available: https://doi.org/10.1145/3589295

[10] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the logicblox system," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1371–1382. [Online]. Available: https://doi.org/10.1145/2723372.2742796

[11] D. Arroyuelo, A. Gómez-Brandón, A. Hogan, G. Navarro, J. Reutter, J. Rojas-Ledesma, and A. Soto, "The ring: Worst-case optimal joins in graph databases using (almost) no extra space," *ACM Trans. Database Syst.*, vol. 49, no. 2, Mar. 2024. [Online]. Available: https://doi.org/10.1145/3644824

[12] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, "Emptyheaded: A relational engine for graph processing," *ACM Trans. Database Syst.*, vol. 42, no. 4, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3129246

[13] C. Aberger, A. Lamb, K. Olukotun, and C. Re, "Levelheaded: A unified engine for business intelligence and linear algebra querying," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 449–460.

[14] X. Feng, G. Jin, Z. Chen, C. Liu, and S. Salihoglu, "KÙZU graph database management system," in *Proceedings of the 13th Conference on Innovative Data Systems Research (CIDR)*, Amsterdam, The Netherlands, 2023. [Online]. Available: https://www.vldb.org/cidrdb/2023/p29-feng.pdf

[15] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu, "Graphflow: An active graph database," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1695–1698. [Online]. Available: https://doi.org/10.1145/3035918.3056445

[16] D. ten Wolde, T. Singh, G. Szárnyas, and P. Boncz, "Duckpgq: Efficient property graph queries in an analytical RDBMS," in *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org, 2023. [Online]. Available: https://vldb.org/cidrdb/2023/duckpgq-efficient-property-graph-queries-in-an-analytical-rdbms.html

[17] M. Raasveldt and H. Mühleisen, "Duckdb: an embeddable analytical database," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1981–1984. [Online]. Available: https://doi.org/10.1145/3299869.3320212

[18] P. Groß, D. ten Wolde, and P. Boncz, "Adaptive factorization using linear-chained hash tables," in *Proceedings of the 15th Annual Conference on Innovative Data Systems Research (CIDR 2025)*, Amsterdam, The Netherlands, Jan. 2025, january 19–22, 2025. [Online]. Available: https://vldb.org/cidrdb/papers/2025/p21-gro.pdf

[19] D. Vrgoc, C. Rojas, R. Angles, M. Arenas, D. Arroyuelo, C. B. Aranda, A. Hogan, G. Navarro, C. Riveros, and J. Romero, "Millenniumdb: A persistent, open-source, graph database," 2021. [Online]. Available: https://arxiv.org/abs/2111.01540

[20] D. Arroyuelo, D. Campos, A. Gómez-Brandón, Y. Linker, G. Navarro, C. Rojas, and D. Vrgoč, "Compactltj: Space & time efficient leapfrog triejoin on graph databases: Compactltj: Space & time efficient leapfrog triejoin ..." *The VLDB Journal*, vol. 34, no. 6, Sep. 2025. [Online]. Available: https://doi.org/10.1007/s00778-025-00945-5

[21] D. Arroyuelo, D. Campos, A. Gómez-Brandón, G. Navarro, C. Rojas, and D. Vrgoč, "Space & time efficient leapfrog triejoin," in *Proceedings of the 7th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, ser. GRADES-NDA '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3661304.3661898

[22] "Information technology — Database languages — SQL — Part 16: Property Graph Queries (SQL/PGQ)," International Organization for Standardization, Geneva, CH, Standard, June 2023.

[23] H. Q. Ngo, "Worst-case optimal join algorithms: Techniques, results, and open problems," 2018. [Online]. Available: https://arxiv.org/abs/1803.09930

[24] H. Pirahesh, J. M. Hellerstein, and W. Hasan, "Extensible/rule based query rewrite optimization in starburst," in *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '92. New York, NY, USA: Association for Computing Machinery, 1992, p. 39–48. [Online]. Available: https://doi.org/10.1145/130283.130294

[25] W. P. Yan and P.-r. Larson, "Eager aggregation and lazy aggregation," in *Proceedings of the 21th International Conference on Very Large Data Bases*, ser. VLDB '95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, p. 345–357.

[26] Y. Yang, H. Zhao, X. Yu, and P. Koutris, "Predicate transfer: Efficient pre-filtering on multi-join queries," in *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org, 2024. [Online]. Available: https://vldb.org/cidrdb/2024/predicate-transfer-efficient-pre-filtering-on-multi-join-queries.html

[27] M. R. Joglekar, R. Puttagunta, and C. Ré, "Ajar: Aggregations and joins over annotated relations," in *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 91–106. [Online]. Available: https://doi.org/10.1145/2902251.2902293

[28] M. A. Khamis, H. Q. Ngo, C. Ré, and A. Rudra, "Joins via geometric resolutions: Worst case and beyond," *ACM Trans. Database Syst.*, vol. 41, no. 4, Nov. 2016. [Online]. Available: https://doi.org/10.1145/2967101

[29] M. A. Khamis, H. Q. Ngo, and A. Rudra, "FAQ: questions asked frequently," *CoRR*, vol. abs/1504.04044, 2015. [Online]. Available: http://arxiv.org/abs/1504.04044

[30] S. Deng, S. Lu, and Y. Tao, "On join sampling and the hardness of combinatorial output-sensitive join algorithms," in *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 99–111. [Online]. Available: https://doi.org/10.1145/3584372.3588666

[31] N. Tziavelis, W. Gatterbauer, and M. Riedewald, "Optimal join algorithms meet top-k," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 2659–2665. [Online]. Available: https://doi.org/10.1145/3318464.3383132

[32] The PostgreSQL Global Development Group, *Genetic Query Optimization (GEQO) in PostgreSQL*, PostgreSQL Global Development Group, 2025, section 61.3 of PostgreSQL 18 Documentation. [Online]. Available: https://www.postgresql.org/docs/18/geqo-pg-intro.html

[33] M. Joglekar and C. Ré, "It's all a matter of degree: Using degree information to optimize multiway joins," *CoRR*, vol. abs/1508.01239, 2015. [Online]. Available: http://arxiv.org/abs/1508.01239

[34] K. Kim, H. Kim, G. Fletcher, and W.-S. Han, "Combining sampling and synopses with worst-case optimal runtime and quality guarantees for graph pattern cardinality estimation," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 964–976. [Online]. Available: https://doi.org/10.1145/3448016.3457246

[35] A. Mhedhbi and S. Salihoglu, "Optimizing subgraph queries by combining binary and worst-case optimal joins," *Proc. VLDB Endow.*, vol. 12, no. 11, p. 1692–1704, Jul. 2019. [Online]. Available: https://doi.org/10.14778/3342263.3342643

[36] R. Dechter and I. Meiri, "Experimental evaluation of preprocessing techniques in constraint satisfaction problems," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI'89. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, p. 271–277.

[37] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proc. VLDB Endow.*, vol. 9, no. 3, p. 204–215, Nov. 2015. [Online]. Available: https://doi.org/10.14778/2850583.2850594

[38] R. Zhu, L. Weng, B. Ding, and J. Zhou, "Learned query optimizer: What is new and what is next," in *Companion of the 2024 International Conference on Management of Data*, ser. SIGMOD '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 561–569. [Online]. Available: https://doi.org/10.1145/3626246.3654692

[39] B. Warnke, K. Martens, T. Winker, S. Groppe, J. Groppe, P. Adhiyaman, S. Srinivasan, and S. Krishnakumar, "Rejoosp: Reinforcement learning for join order optimization in sparql," *Big Data and Cognitive Computing*, vol. 8, no. 7, 2024. [Online]. Available: https://www.mdpi.com/2504-2289/8/7/71

[40] J. Zhang, "Alphajoin: Join order selection à la alphago," in *Proceedings of the VLDB 2020 PhD Workshop co-located with the 46th International Conference on Very Large Databases (VLDB 2020), ONLINE, August 31 - September 4, 2020*, ser. CEUR Workshop Proceedings, Z. Abedjan and K. Hose, Eds., vol. 2652. CEUR-WS.org, 2020. [Online]. Available: https://ceur-ws.org/Vol-2652/paper05.pdf

[41] J. Wang, I. Trummer, A. Kara, and D. Olteanu, "Adopt: Adaptively optimizing attribute orders for worst-case optimal join algorithms via reinforcement learning," 2023. [Online]. Available: https://arxiv.org/abs/2307.16540

[42] A. Ailamaki, S. Madden, D. Abadi, G. Alonso, S. Amer-Yahia, M. Balazinska, P. A. Bernstein, P. Boncz, M. Cafarella, S. Chaudhuri, S. Davidson, D. DeWitt, Y. Diao, X. L. Dong, M. Franklin, J. Freire, J. Gehrke, A. Halevy, J. M. Hellerstein, M. D. Hill, S. Idreos, Y. Ioannidis, C. Koch, D. Kossmann, T. Kraska, A. Kumar, G. Li, V. Markl, R. Miller, C. Mohan, T. Neumann, B. C. Ooi, F. Ozcan, A. Parameswaran, I. Pandis, J. M. Patel, A. Pavlo, D. Porobic, V. Sanca, M. Stonebraker, J. Stoyanovich, D. Suciu, W.-C. Tan, S. Venkataraman, M. Zaharia, and S. B. Zdonik, "The cambridge report on database research," 2025. [Online]. Available: https://arxiv.org/abs/2504.11259

[43] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case optimal join algorithms," *J. ACM*, vol. 65, no. 3, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3180143

[44] A. Birler, T. Schmidt, P. Fent, and T. Neumann, "Simple, efficient, and robust hash tables for join processing," in *Proceedings of the 20th International Workshop on Data Management on New Hardware*, ser. DaMoN '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3662010.3663442

[45] M. Böther, L. Benson, A. Klimovic, and T. Rabl, "Analyzing vectorized hash tables across cpu architectures," *Proc. VLDB Endow.*, vol. 16, no. 11, p. 2755–2768, Jul. 2023. [Online]. Available: https://doi.org/10.14778/3611479.3611485

[46] V. Leis, P. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 743–754. [Online]. Available: https://doi.org/10.1145/2588555.2610507

[47] S. Zeuch, J. Freytag, and F. Huber, "Adapting tree structures for processing with SIMD instructions," in *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*, S. Amer-Yahia, V. Christophides, A. Kementsietsidis, M. N. Garofalakis, S. Idreos, and V. Leroy, Eds. OpenProceedings.org, 2014, pp. 97–108. [Online]. Available: https://doi.org/10.5441/002/edbt.2014.10

[48] S. Deep, X. Hu, and P. Koutris, "Fast join project query evaluation using matrix multiplication," 2020. [Online]. Available: https://arxiv.org/abs/2002.12459

[49] P. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: Hyper-pipelining query execution," in *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 2005, pp. 225–237. [Online]. Available: http://cidrdb.org/cidr2005/papers/P19.pdf

[50] O. Kalinsky, B. Kimelfeld, and Y. Etsion, "The triejax architecture: Accelerating graph operations through relational joins," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1217–1231. [Online]. Available: https://doi.org/10.1145/3373376.3378524

[51] D. Arroyuelo, A. Hogan, G. Navarro, J. L. Reutter, J. Rojas-Ledesma, and A. Soto, "Worst-case optimal graph joins in almost no space," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 102–114. [Online]. Available: https://doi.org/10.1145/3448016.3457256

[52] A. Bigerl, N. Karalis, L. Heidrich, and A.-C. N. Ngomo, "Efficient updates for worst-case optimal join triple stores," in *The Semantic Web – ISWC 2025: 24th International Semantic Web Conference, Nara, Japan, November 2–6, 2025, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2025, pp. 539–555. [Online]. Available: https://doi.org/10.1007/978-3-032-09527-5_29

[53] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Application of hash to data base machine and its architecture," *New Gen. Comput.*, vol. 1, no. 1, p. 63–74, Mar. 1983. [Online]. Available: https://doi.org/10.1007/BF03037022

[54] X. Hu and K. Yi, "Towards a worst-case i/o-optimal algorithm for acyclic joins," in *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 135–150. [Online]. Available: https://doi.org/10.1145/2902251.2902292

[55] D. Arroyuelo, G. Navarro, J. L. Reutter, and J. Rojas-Ledesma, "Optimal joins using compressed quadtrees," *ACM Trans. Database Syst.*, vol. 47, no. 2, May 2022. [Online]. Available: https://doi.org/10.1145/3514231

[56] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann, "Leanstore: In-memory data management beyond main memory," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 185–196.

[57] O. Kalinsky, Y. Etsion, and B. Kimelfeld, "Flexible caching in trie joins," 2016. [Online]. Available: https://arxiv.org/abs/1602.08721

[58] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment," in *Proceedings of the 13th International Conference on Extending Database Technology*, ser. EDBT '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 99–110. [Online]. Available: https://doi.org/10.1145/1739041.1739056

[59] P. Beame, P. Koutris, and D. Suciu, "Communication steps for parallel query processing," *J. ACM*, vol. 64, no. 6, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3125644

[60] ——, "Skew in parallel query processing," in *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 212–223. [Online]. Available: https://doi.org/10.1145/2594538.2594558

[61] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu, "Demonstration of the myria big data management service," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 881–884. [Online]. Available: https://doi.org/10.1145/2588555.2594530

[62] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar, "Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows," *Proc. VLDB Endow.*, vol. 11, no. 6, p. 691–704, Feb. 2018. [Online]. Available: https://doi.org/10.14778/3199517.3199520

[63] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 439–455. [Online]. Available: https://doi.org/10.1145/2517349.2522738

[64] P. Fuchs, P. Boncz, and B. Ghit, "Edgeframe: Worst-case optimal joins for graph-pattern matching in spark," in *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, ser. GRADES-NDA'20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3398682.3399162

[65] R. Li, Q. Ma, X. Shi, and A. Liu, "Sievejoin: Boosting multi-way joins by filtering unneeded intermediate results," *Data Science and Engineering*, vol. 11, no. 1, pp. 143–154, 2026. [Online]. Available: https://doi.org/10.1007/s41019-025-00325-7

[66] B. Ketsman and D. Suciu, "A worst-case optimal multi-round algorithm for parallel computation of conjunctive queries," in *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 417–428. [Online]. Available: https://doi.org/10.1145/3034786.3034788

[67] X. Hu and K. Yi, "Massively parallel join algorithms," *SIGMOD Rec.*, vol. 49, no. 3, p. 6–17, Dec. 2020. [Online]. Available: https://doi.org/10.1145/3444831.3444833

[68] G. Katsogiannis-Meimarakis and G. Koutrika, "A survey on deep learning approaches for text-to-sql," *The VLDB Journal*, vol. 32, no. 4, p. 905–936, Jan. 2023. [Online]. Available: https://doi.org/10.1007/s00778-022-00776-8

[69] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. R. Radev, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task," *CoRR*, vol. abs/1809.08887, 2018. [Online]. Available: http://arxiv.org/abs/1809.08887

[70] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo, X. Zhou, C. Ma, G. Li, K. C. Chang, F. Huang, R. Cheng, and Y. Li, "Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2023.

[71] D. Gao, H. Wang, Y. Li, X. Sun, Y. Qian, B. Ding, and J. Zhou, "Text-to-sql empowered by large language models: A benchmark evaluation," *Proc. VLDB Endow.*, vol. 17, no. 5, p. 1132–1145, Jan. 2024. [Online]. Available: https://doi.org/10.14778/3641204.3641221

[72] K. Kim, J. Jung, I. Seo, W.-S. Han, K. Choi, and J. Chong, "Learned cardinality estimation: An in-depth study," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1214–1227. [Online]. Available: https://doi.org/10.1145/3514221.3526154

[73] J. Sun, J. Zhang, Z. Sun, G. Li, and N. Tang, "Learned cardinality estimation: a design space exploration and a comparative evaluation," *Proc. VLDB Endow.*, vol. 15, no. 1, p. 85–97, Sep. 2021. [Online]. Available: https://doi.org/10.14778/3485450.3485459

[74] S.-J. Qiao, H.-L. Fan, N. Han, L. Du, Y.-H. Peng, R.-M. Tang, and X. Qin, "Learning database optimization techniques: the state-of-the-art and prospects," *Frontiers of Computer Science*, vol. 19, no. 12, p. 1912612, 2025. [Online]. Available: https://doi.org/10.1007/s11704-025-41116-7

[75] T. Faltín, V. Trigonakis, A. Berdai, L. Fusco, C. Iorgulescu, S. Hong, and H. Chafi, "Better distributed graph query planning with scouting queries," in *Proceedings of the 6th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, ser. GRADES-NDA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3594778.3594884

[76] R. Marcus, 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*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1275–1288. [Online]. Available: https://doi.org/10.1145/3448016.3452838

[77] H. Dong, C. Zhang, G. Li, and H. Zhang, "Cloud-native databases: A survey," *IEEE Trans. on Knowl. and Data Eng.*, vol. 36, no. 12, p. 7772–7791, Dec. 2024. [Online]. Available: https://doi.org/10.1109/TKDE.2024.3397508

[78] A. Berdai, A. Soukrat, and D. Chiadmi, "Efficient in-memory rebalancings for skewed results in distributed graph queries," *Journal of King Saud University Computer and Information Sciences*, vol. 37, no. 9, p. 288, 2025. [Online]. Available: https://doi.org/10.1007/s44443-025-00212-1

[79] I. Arefyeva, D. Broneske, G. Campero, M. Pinnecke, and G. Saake, "Memory management strategies in cpu/gpu database systems: A survey," in *Beyond Databases, Architectures and Structures. Facing the Challenges of Data Proliferation and Growing Variety*, S. Kozielski, D. Mrozek, P. Kasprowski, B. Małysiak-Mrozek, and D. Kostrzewa, Eds. Cham: Springer International Publishing, 2018, pp. 128–142.

[80] J. Cao, R. Sen, M. Interlandi, J. Arulraj, and H. Kim, "Gpu database systems characterization and optimization," *Proc. VLDB Endow.*, vol. 17, no. 3, p. 441–454, Nov. 2023. [Online]. Available: https://doi.org/10.14778/3632093.3632107

[81] J. Fang, Y. T. B. Mulder, J. Hidders, J. Lee, and H. P. Hofstee, "In-memory database acceleration on fpgas: a survey," *The VLDB Journal*, vol. 29, no. 1, p. 33–59, Oct. 2019. [Online]. Available: https://doi.org/10.1007/s00778-019-00581-w