


# The Quest for Faster Join Algorithms

Paraschos Koutris  

University of Wisconsin-Madison, WI, USA

Shaleen Deep 

Microsoft, Madison, WI, USA

Austen Fan 

University of Wisconsin-Madison, WI, USA

Hangdong Zhao 

University of Wisconsin-Madison, WI, USA

---

## Abstract

Joins are the cornerstone of relational databases. Surprisingly, even after several decades of research in the systems and theory database community, we still lack an understanding of how to design the fastest possible join algorithm. In this talk, we will present the exciting progress the database theory community has achieved in join algorithms over the last two decades. The talk will revolve around five key ideas fundamentally shaping this research area: tree decompositions, data partitioning, leveraging statistical information, enumeration, and algebraic techniques.

**2012 ACM Subject Classification** Theory of computation → Database theory

**Keywords and phrases** Conjunctive Queries, Joins, Tree Decompositions, Enumeration, Semirings

**Digital Object Identifier** 10.4230/LIPIcs.ICDT.2025.1

**Category** Invited Talk

## 1 Introduction

Joins are the cornerstone of relational databases. First introduced by Codd [8] as a core operator in relational algebra, they are now a key component of any database engine.

In any relational database engine, a join query can be executed via a plan of binary joins ( $\bowtie$ ). The problem at hand is to optimize the order of these binary joins, as well as the specific algorithm that computes each join. There has been a huge amount of literature on this topic, and it is still a very active area of research. A key difficulty is that the join optimization problem is computationally hard, and is also based on gathering accurate statistical information, a process that is often erroneous. Query optimizers will notoriously choose suboptimal join plans. Nevertheless, the database systems architecture of how to do joins has remained virtually the same for the last three decades.

From a theoretical perspective, the story of join algorithms is quite different. Over the past couple of decades, there has been significant and exciting progress in our understanding of how to evaluate joins more efficiently. Novel techniques and new algorithmic frameworks have produced faster and faster join algorithms. Several ideas produced in the database theory community are also slowly moving into practice and are incorporated into real-world database systems. In this talk, we will tell this story and then talk about the research problems that are still open.

**Organization.** We will start with presenting some basic notions and algorithmic ideas about joins. Then, we will discuss the progress on join algorithms by focusing on five fundamental ideas that have driven the research community. Finally, we will close with the many open questions that still remain.



© Paraschos Koutris, Shaleen Deep, Austen Fan, and Hangdong Zhao;  
licensed under Creative Commons License CC-BY 4.0

28th International Conference on Database Theory (ICDT 2025).

Editors: Sudeepa Roy and Ahmet Kara; Article No. 1; pp. 1:1–1:12

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 2 The Basics

To keep the presentation simple, we will focus on relational queries with natural joins ( $\bowtie$ ) and projections ( $\pi$ ), of the following form:

$$Q = \pi_U(\bowtie_{e \in E} R_e(X_e))$$

Here, the attributes are indexed by  $[n] = \{1, 2, \dots, n\}$ , and the join is described by a hypergraph  $H = ([n], E)$ .  $X_e$  denotes a vector of attributes indexed by the nodes in  $e \in E$ . The projection attributes are  $X_U$ , where  $U \subseteq [n]$ . When  $U = \emptyset$ , we have a *Boolean join* and when  $U = [n]$  we have a *full join* – no projections. For example, consider the following two full joins (triangle join and square join) which we will use as running examples throughout:

$$\begin{aligned} Q_{\Delta} &= R(X_1, X_2) \bowtie S(X_2, X_3) \bowtie T(X_3, X_1) \\ Q_{\square} &= R(X_1, X_2) \bowtie S(X_2, X_3) \bowtie T(X_3, X_4) \bowtie U(X_4, X_1) \end{aligned}$$

The key task is to compute the output  $Q(D)$  of join query  $Q$  over a relational instance  $D$ . We will denote by  $N = |D|$  the input size (total number of tuples in the instance) and by  $OUT = |Q(D)|$  the output size (total number of tuples in the output). To measure runtime, we will use *data complexity*, which means that we consider the query to be fixed. We can now pose the central question precisely.

*Given a join query  $Q$  and an instance  $D$ , what is the fastest algorithm that computes  $Q(D)$  w.r.t. the input and output size?*

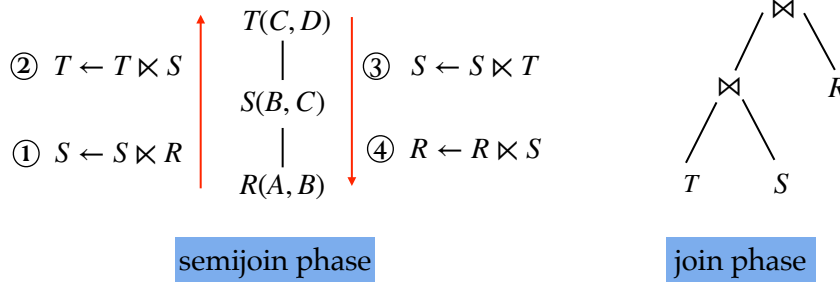
**Yannakakis' Algorithm and Acyclicity.** The first effort to design fast join algorithms focused on a class of joins called *acyclic joins*<sup>1</sup>. A join is acyclic if we can construct a tree  $T$  with nodes the relations  $R_e$  such that for every attribute  $X$ , the subgraph of  $T$  that contains the relations with the attribute  $X$  is connected:  $T$  is then called the *join tree* of  $Q$ . Figure 1 shows the join tree for the acyclic join  $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$ . An acyclic join satisfies some very nice properties and admits algorithms that are linear in the input and output size when there are no projections.

► **Theorem 1** (Yannakakis [47]). *Let  $Q$  be an acyclic join query and  $D$  an instance. Then, if  $Q$  is Boolean or full, we can compute it in time  $O(N + OUT)$ . In general, we can compute it in time  $O(N \cdot OUT + OUT)$ .*

To explain Yannakakis' algorithm, consider the case of a full join. Then, Yannakakis' algorithm guarantees that each one of the intermediate relations can never be bigger than the input and output size. To achieve this strong guarantee, it needs two phases (see Figure 1). In the first phase, we take the join tree, root it from an arbitrary node, and then perform a bottom-up and top-down semi-join pass. After the semijoin pass, we are guaranteed that every remaining tuple contributes in some tuple in the output result. In the second phase, we compute the join in a bottom-up manner.

Yannakakis' algorithm is simple enough that it can be written as a query plan with only two operators: projections and joins. It will serve as the fundamental block for what we present in the next sections.

<sup>1</sup> In this article, we use acyclicity to mean  $\alpha$ -acyclicity. However, there are other notions of acyclicity, such as  $\beta$ -acyclicity,  $\gamma$ -acyclicity, and Berge-acyclicity.



■ **Figure 1** A depiction of Yannakakis' algorithm for the acyclic join  $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$ .

► **In Practice.** The semijoin operators in Yannakakis' algorithm can be expensive and increase the cost compared to a one-pass join-project plan. Hence, a direct implementation of the algorithm will incur a large overhead. However, several recent research papers have shown that we can still apply the core idea of Yannakakis' algorithm – the filtering of tuples that will not contribute to the final output – while maintaining good practical performance. One idea in this direction is to replace the semijoins with an approximate filtering operator, specifically a Bloom filter [46]. Bloom filters are much more efficient compared to a semijoin and thus incur a small overhead while maintaining most of the filtering power. In another direction, [5] produces the same guarantees as Yannakakis' algorithm by introducing plans with new relational operators.

### 3 Beyond Acyclicity: Tree Decompositions

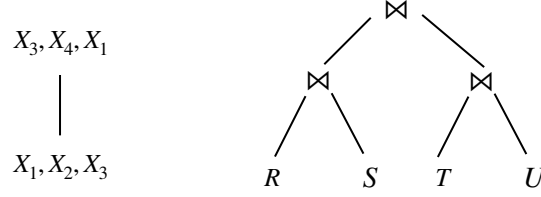
What can we do when the join is not acyclic? The idea is simple: we will attempt to make the join acyclic by computing larger and larger intermediate relations, which can then be used as “input relations” in the join. Of course we have to be careful, so that the intermediate relations do not blow up in size. The task then becomes: *what are the smallest possible intermediate relations we can construct such that we end up with an acyclic join?* This question leads naturally to the idea of *tree decompositions*.

A tree decomposition of a join  $Q$  can be viewed as a join tree where each node is not an input relation, but an intermediate relation. In more technical terms, each node in the tree (called typically a *bag*) is assigned a subset of the attributes  $X_1, \dots, X_n$ . We still want the connectedness property to hold. Additionally, we want the attributes of each relation  $R_e$  to appear in some bag of the tree; this is necessary to guarantee correctness.

The algorithm is then: compute the relation at each bag; then use Yannakakis' algorithm to compute the final join output. To compute a bag  $B$  with attributes  $X_B$ , we can use a simple algorithm: find the smallest subset of relations that “cover” all the attributes and join them. If we have  $w$  such relations, this costs  $O(N^w)$ . The quantity  $w$  is called the *integral edge cover* for  $X_B$ , denoted  $w(X_B)$ , and the largest  $w$  over all bags is called the *generalized hypertree width (ghw)* of the tree decomposition [20]. Clearly, we want to find the tree decomposition with the smallest ghw overall, and this is called the ghw of the join.

$$\text{ghw}(Q) := \min_{\text{decomp}} \max_{T \text{ bag } B} w(X_B)$$

$$Q_{\square} = R(X_1, X_2) \bowtie S(X_2, X_3) \bowtie T(X_3, X_4) \bowtie U(X_4, X_1)$$



■ **Figure 2** A tree decomposition for the cyclic join  $Q_{\square}$  along with the corresponding query plan with runtime  $O(N^2 + OUT)$ .

When the join is acyclic, we have  $\text{ghw} = 1$ . As the join becomes more complex, the generalized hypertree width increases. Using the idea of a tree decomposition produces again an algorithm that can be expressed as a join-project plan, as before (see Figure 2). We should note here that  $\text{ghw}$  is related to the notion of *treewidth* in graph theory [38].

## 4 Data Partitioning

So far we have discussed join algorithms that can be expressed as join-project plans. However, it can be shown that join-project plans are suboptimal [3]. For example, any join-project plan for the triangle join  $Q_{\Delta}$  has cost at least  $N^2$ , but as we will see this join can be computed in time only  $O(N^{3/2})$ . This means that we have to go beyond join-project plans and introduce possibly new operators in the query plan (or maybe new algorithmic frameworks). Surprisingly, to reach state-of-the-art we need to add a single operator: a *partitioning operator* that splits a relation into two disjoint parts which we call *heavy* and *light*.

The key intuition behind data partitioning is that different parts of a relation have different properties (e.g., frequencies of values on each attribute) and behave different with respect to different query plans. This means that it can be often beneficial to split the input into different parts, and apply different query plans to each part.

► **Example 2.** Consider again the triangle join, and take a query plan that attempts to first do the join  $R(X_1, X_2) \bowtie S(X_2, X_3)$ . The problem of this strategy is that in the worst case its output size can be  $N^2$  – this will happen when  $R, S$  have a single value for attribute  $X_2$  that appears  $N$  times. To overcome this problem, we will split  $R$  into two parts: the *light part*  $R_L$  has the tuples where a value of  $X_2$  appears  $\leq \sqrt{N}$  times, and the *heavy part*  $R_H$  all the rest. It turns out that we can join  $R_L$  with  $S$  with a much smaller blowup of the intermediate size:  $|R_L \bowtie S| \leq N \cdot \sqrt{N} = N^{3/2}$ . After computing this join, we can semijoin with  $T$  for the final output. But for  $R_H$  we need to follow a different join plan: we will instead compute  $R_H(X_1, X_2) \bowtie T(X_3, X_1)$  first. The key idea here is that there can be at most  $\sqrt{N}$  values of  $X_2$  that appear in  $R_H$ , hence  $|R_H \bowtie T| \leq N \cdot \sqrt{N} = N^{3/2}$ . After computing this join, we semijoin with  $S$  to obtain the final output. Observe that the power of partitioning and choosing the best of the two query plans brings the runtime cost of the join down to  $O(N^{3/2})$ .

Partitioning helps us in two ways. First, it gives us a faster algorithm to compute each intermediate relation in a tree decomposition. Instead of paying the cost of the integral cover, we now only need to pay the *fractional edge cover*, which we denote by  $f(X_{[n]})$ . The breakthrough work on worst-case-optimal join algorithms [43, 33, 35] showed that any join can be computed in worst-case size time that is equal to  $O(N^{f(X_{[n]})})$ .

This immediately gives us a stronger notion of width, called *fractional hypertree width* (*fhw*). For the triangle query,  $\text{fhw}(Q_\Delta) = 3/2$ , since we can cover each relation with a fractional weight of  $1/2$ . In general:

$$\text{fhw}(Q) := \min_{\text{decomp}} \max_{T \text{ bag } B} f(X_B)$$

But even that is not enough to give us a fast algorithm. Take the example of  $Q_\square$ . If we are guided by single tree decomposition, we only get a quadratic runtime. So, we need to use multiple tree decompositions and partition the data such that each decomposition guides the plan of a different part of the data. Hence, we not only need to decide what plan we want to use for each part of the data, but also into which tree decomposition we will place it. This is a complex optimization problem. To solve this, Khamis et al. proposed the PANDA algorithm [30, 31], which is the current state-of-the-art algorithm for most join queries. We will not describe the details of PANDA, but the key idea is that it prescribes an algorithm that chooses the next operator (between a join, a projection, and a partitioning). PANDA leads to an even tighter notion of width, called *submodular width* [32]:

$$\text{subw}(Q) := \max_{\text{submodular } h} \min_{\text{decomp}} \max_{T \text{ bag } B} h(X_B)$$

The difference in the definition is on the function which specifies the cost of computing each bag. This is now specified by a *submodular function*  $h$ . For the square query,  $\text{subw}(Q_\square) = 3/2$ .

► **Theorem 3** (PANDA [30]). *Let  $Q$  be an join query and  $D$  an instance. Then, if  $Q$  is Boolean or full, we can compute it in time  $\tilde{O}(N^{\text{subw}} + \text{OUT})$ , where  $\tilde{O}$  hides a polylogarithmic factor w.r.t.  $N$ .*

**Attribute-at-a-time joins.** In addition to data partitioning, many worst-case optimal algorithms can be thought as *attribute-at-a-time* joins [43, 34, 45]. In this framework, instead of considering plans that join a relation at a time, we consider more fine-grained plans that grow the intermediate relation an attribute at a time. Such algorithms need different types of data structures (such as hash tries, or sorted tries) to be efficient.

**Output-sensitive algorithms.** Recent work [16] has made some exciting progress that has led to the first improvement of Yannakakis' algorithm after almost four decades. Recall that when we have projections the runtime of Yannakakis' algorithm for acyclic joins is  $O(N \cdot \text{OUT} + \text{OUT})$ . However, this bound is not optimal. By combining the semijoin pass with data partitioning, we can improve this runtime to  $O(N^{1-\epsilon} \cdot \text{OUT} + \text{OUT})$  where  $\epsilon > 0$  is a query-dependent constant. The particular insight is to use what we call *dynamic partitioning*: this allows to partition intermediate relations and not only input relations.

► **In Practice.** Attribute-at-a-time join algorithms have been implemented in various systems using different variants and data structures. The first implementation is Leapfrog Triejoin [44], a join algorithm which was used in the LogicBlox system. More recently, Freitag et al. [19] implemented such an algorithm in a modern RDBMS, and Freejoin [45] attempts to integrate attribute-at-a-time joins with traditional joins in a single framework. Interestingly, none of these approaches use data partitioning techniques. One reason for this could be that the currently used partitioning algorithms such as PANDA typically require splitting the database into too many pieces, which would incur a huge overhead in practice.

## 5 Leveraging Statistical Information

So far we have analyzed join algorithms with respect to only two parameters: the input size  $N$  and the output size  $OUT$ . Unfortunately, using only these two parameters in the runtime bound gives us join algorithms that have worst-case optimal performance, but in practice may not perform well. The reason behind this gap between theory and practice is that the input and output sizes are not sufficient to accurately describe the database instance. A query optimizer that uses more fine-grained statistical information will thus perform better, even if it uses a suboptimal class of algorithms.

► **Example 4 (Triangles Revisited).** Consider again the case of the triangle join  $Q_\Delta$  and suppose that we know that attribute  $X_1$  is a primary key in the relation  $R(X_1, X_2)$ . In this case, using only the input size as a parameter would produce an algorithm with runtime  $O(N^{3/2})$ . However, an algorithm that joins  $R(X_1, X_2) \bowtie T(X_3, X_1)$  first would produce an intermediate join with output only  $N$ , and thus we could tighten the runtime analysis to  $O(N)$  if we could choose the correct plan.

The above example shows the use of integrity constraints (here, primary keys) as statistical information. Other types of statistical information are cardinality constraints (a bound on the size of each input relation), functional dependencies, key-foreign key constraints, and a more general type of statistical information called *degree constraints* [30]. In their most general form, degree constraints assert that for two disjoint sets of attributes  $X_A, X_B$  in the same relation  $R$ , for any values  $b$  of  $X_B$ , we have that  $|\pi_{X_A}(\sigma_{X_B=b}(R))| \leq d$  for some number  $d > 0$ . Degree constraints generalize both cardinality constraints and functional dependencies, and can nicely fit into the algorithmic framework of join-project-partition plans providing tighter runtime analysis and more optimized query plans [30].

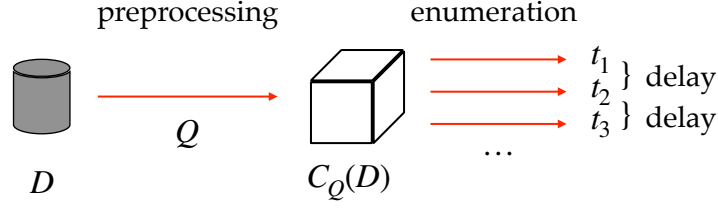
Some very recent work has progressed one step further and has provided algorithms that work for even more fine-grained statistical information. Degree sequences [10] maintain the distribution of frequencies for  $|\pi_{X_A}(\sigma_{X_B=b}(R))|$  instead of a single upper bound. Norm bounds [27] use  $\ell_p$  norms to capture concisely this distribution of frequencies. Finally, partition constraints [9] offer another generalization of degree constraints that also captures the notion of degeneracy in graph theory. All these types of statistical information can be integrated to state-of-the-art join algorithms and can also lead to better cardinality estimation in practice, but of course there is a catch: more fine-grained statistics are more costly to gather and maintain. It is still an open question to find the right balance between the granularity (and type) of statistics and the efficiency of a join algorithm.

## 6 The Enumeration Framework

The efficiency of an algorithm is typically measured by the time it takes to produce the whole output. When the output is a single number (e.g., integer, real) or a Boolean value, this is a natural definition. But in query evaluation, the output is often a relation that can have multiple tuples – in many cases, the join output can be much bigger than the input itself.

This particular characteristic of join computation allows for the use of runtime measurement for join algorithms in a more fine-grained way. In particular, we can think of a join algorithm for a join query  $Q$  as working in two phases:

**Preprocessing Phase:** in the first phase, the algorithm will read the input instance  $D$  and produce an intermediate data structure  $C_Q(D)$ . For the preprocessing phase, we care to minimize the total preprocessing time.



■ **Figure 3** The enumeration framework.

**Enumeration Phase:** in the second phase, the algorithm will read  $C_Q(D)$  and start producing the output tuples. For this phase, the goal is to minimize the largest time between outputting any two consecutive tuples (including the time to produce the first tuple, and the time from the last tuple to terminating): this time is called the *delay*.

Ideally, we want to achieve constant-delay enumeration, which is the strongest possible delay guarantee [40]. Constant-delay enumeration implies that we need only  $O(1)$  time between seeing two consecutive output tuples. This can be very powerful as an algorithmic framework: if we only want to see  $k$  tuples of the join output, we then need to pay only  $O(N + k)$  time. Additionally, if we know that the output may be reused, we can keep the intermediate data structure only (and not the full join output).

It turns out that we can modify Yannakakis' algorithm such that we can keep linear preprocessing time but also achieve constant-delay enumeration [4].

► **Theorem 5.** *Let  $Q$  be a full acyclic join query and  $D$  an instance. Then, we can enumerate  $Q(D)$  with linear preprocessing time  $O(N)$  and constant delay.*

The algorithm can also be applied to more general acyclic joins with projections, a class called *free-connex acyclic joins*. Combining the above theorem with tree decompositions and data partitioning, we can also obtain for any join query an algorithm with  $\tilde{O}(N^{\text{subw}})$  preprocessing time and constant delay.

The idea of enumeration leads naturally to the idea of a *factorized representation* of an output. There is really no need to keep the join output fully materialized if we can always reconstruct it as fast as possible when needed. This idea first appeared under the name of factorized databases [37, 36]. The additional benefit of a factorized join result is that often we can do post-processing directly on the factorization, without decompressing it and depending on the downstream task [39, 28].

**Enumeration Tradeoffs.** So far we discussed join algorithms that achieve constant-delay enumeration. But we can design join algorithms with different guarantees by trading off a larger delay with a faster preprocessing time (and less space to store the intermediate data structure). This idea can be very useful if there are memory limitations and also gives new points in the design space of join algorithms. Recent work has looked into different types of these tradeoffs for different classes of joins [24, 13, 48, 14].

**Ranked Enumeration.** A practical variant of enumeration asks to produce the output tuples not in any order, but in a specific order determined by a ranking function. For example, we may want to see the output tuples ranked in decreasing order of the sum of some of their attributes, or ranked according to a lexicographic ordering. In this setting of *ranked enumeration*, the target is still to achieve the smallest delay between two consecutive output



tuples. Recent work [15, 41] has shown that for most practical ranking functions and acyclic full joins we need to pay linear preprocessing time and  $O(\log N)$  delay – in other words, the cost of ranking is only an additional logarithmic factor. These join algorithms have been extended to also handle projections [7, 12].

► **In Practice.** Database systems are typically geared to optimize end-to-end performance and not more fine-grained measures such as delay. However, especially in the case of ranked enumeration with limits (top- $k$  processing), by pushing the ranking function inside the join, we can avoid materializing large intermediate relations and thus have big speedups in practice [12, 42]. It is an open research problem how an enumeration-style join framework can be seamlessly incorporated into a join-at-a-time relational engine.

## 7 The Algebraic Lens

So far we have focused on join processing under relational set semantics: an tuple can either occur in the output or not. Many real-world applications require different semantics for join evaluation: SQL uses bag semantics (and so a tuple may appear multiple times in the output), or other times we may want to perform some aggregate computation on the join result (summation, max, min, counting). Many of these semantics can be beautifully captured in one unifying algebraic framework with the use of semirings.

Semirings for joins were first proposed by Green et al. [21]. A semiring  $\mathbb{S} = (\mathbf{D}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$  is an algebraic structure where  $\oplus$  is the addition and  $\otimes$  is the multiplication such that (i)  $(\mathbf{D}, \oplus, \mathbf{0})$  and  $(\mathbf{D}, \otimes, \mathbf{1})$  are commutative monoids, (ii) multiplication is distributive over addition, and (iii)  $x \otimes \mathbf{0} = \mathbf{0}$  for every element  $x \in \mathbf{D}$ . The semiring is also *idempotent* if  $x \oplus x = x$  for every element  $x \in \mathbf{D}$ . An example of an idempotent semiring is the tropical semiring  $(\mathbb{N}, \min, +, +\infty, 0)$  or the Boolean semiring  $(\{0, 1\}, \vee, \wedge, 0, 1)$ .

Given an instance  $D$  and a semiring  $\mathbb{S}$ , we can annotate each input tuple  $t$  from relation  $R_e$  with a *value* from the domain  $\mathbf{D}$  of the semiring  $\mathbb{S}$ . For example, if  $\mathbb{S}$  is the Boolean semiring  $(\{0, 1\}, \vee, \wedge, 0, 1)$ , the annotation captures the presence or absence of the tuple  $t$ . If  $\mathbb{S}$  is the counting semiring  $(\mathbb{N}, +, \times, 0, 1)$ , the annotation captures the multiplicity of the tuple  $t$ . The semantics of a join can be naturally lifted to semirings: for a join  $Q = \pi_U(\bowtie_{e \in E} R_e(X_e))$ , we interpret  $\pi_U$  as an  $\oplus$ -sum and  $\bowtie_{e \in E}$  as an  $\otimes$ -product. For example, if we consider the query  $\pi_{\emptyset}(R(X_1, X_2) \bowtie S(X_2, X_3), T(X_3, X_1))$  over the tropical semiring  $(\mathbb{N}, \min, +, +\infty, 0)$ , the output will be the total weight of the minimum-weight triangle. Over the counting semiring, the same query would output the total number of triangles.

Lifting the semantics of a join to semirings may seem that it would create a harder algorithmic problem, but in fact many of the join algorithms we have discussed so far in this paper can work for any semiring with minor modifications. For example, Yannakakis' algorithm can be modified to compute a Boolean acyclic join over any semiring in linear time  $O(N)$  [29]. Joins with negation can also be handled with the same efficiency when we consider semirings [49]. If the semiring is idempotent, then PANDA can also be applied to work for any semiring with the same runtime [29]. Joins over non-idempotent semiring (e.g., counting) are more challenging although we can still apply some of the newer join algorithms [29, 25]. One key takeaway here is that the join algorithms used in practice and in theory can most of the time be lifted to work in this more general algebraic structure of semirings: we get this almost for free!



## 8 Future Directions

In this final part, we present several exciting open problems in join algorithms.

**Towards even faster joins.** PANDA is not an optimal algorithm for join processing. Indeed, one can use Fast Matrix Multiplication (FMM) to speed up some joins. FMM says that one can multiply two  $n \times n$  matrices in time  $O(n^\omega)$ , where  $\omega$  is some constant strictly less than 3 [22]. For example, finding whether a triangle exists in a graph can be done in time faster than  $O(N^{3/2})$  by using fast matrix multiplication along with data partitioning [2]; in fact, FMM improves the runtime of computing any Boolean  $k$ -cycle query. The key idea is that we use FMM for the parts of the data that are dense, and traditional join-project plans for the rest. Some recent work shows how FMM can be incorporated into join plans in a principled way, by treating it as a special operator [23, 11, 26]. However, key questions still remain open. Are there algorithmic techniques other than FMM that we can use to accelerate join processing? Is PANDA optimal if do not use FMM? For example, if we evaluate over a semiring that is not Boolean (e.g., the tropical semiring) then FMM is not applicable anymore.

**Join algorithms for general semirings.** As we mentioned before, many join algorithms can be applied with minor modifications to work for any semiring. However, the state-of-the-art algorithm (PANDA) can not be applied to any non-idempotent semiring (and in particular the counting semiring). Khamis et al. [25] have proposed an algorithm that can count join results, but its performance is bounded by a width measure called *sharp submodular width*,  $\#subw$ , which can be potentially worse than  $subw$ . Hence, our understanding of *what is the fastest algorithm to count the size of a join output* is still incomplete. The core technical difficulty here is that the state-of-the-art join algorithms (such as PANDA) achieve good performance by splitting the data to parts that may have overlapping outputs: two query plans on different parts of the data many produce the same output tuple. It is open whether there is a method to achieve the same runtime performance by creating “output-disjoint” partitions or whether the counting problem for joins is fundamentally harder.

**Towards Lower Bounds.** An important direction in algorithmic research is to show lower bounds that prove the optimality of the algorithm. The progress of the research community on lower bounds for joins has been less significant. One method for proving matching lower bounds is using *fine-grained complexity*. Following this approach, we start with a problem that we believe that is hard, and then attempt to reduce join processing to this problem. An important such starting problem is the *k-clique conjecture*: given a weighted graph with  $n$  vertices, there is no algorithm that computes the minimum-weight  $k$ -clique in time  $O(n^{k-\epsilon})$  for some  $\epsilon > 0$ . Recent work [17] has shown that we can use this conjecture to prove a conditional lower bound for computing joins over the tropical semiring. In particular, it shows a lower bound of  $\Omega(N^{c_{\text{lemb}}})$ , where  $c_{\text{lemb}}$  is a parameter called the *clique embedding power* of the query. It can be show that  $c_{\text{lemb}} \leq subw$ , and in fact it was recently shown that this is tight for joins on binary relations with  $subw < 2$  [6]. However, there are joins for which there is a gap. In these cases, can we find better lower bounds? What are other fundamental hard problems other than  $k$ -clique for join processing?

A second method of proving lower bounds is by considering simpler computational models. In particular, recent research [18] has looked at *semiring circuits* and *formulas* as simple computational models. For circuits, lower bounds were proved using the notion of *entropic*

*width*,  $\Omega(N^{\text{entw}})$ . This measure is smaller than submodular width,  $\text{entw} \leq \text{subw}$ , but we do not know whether there is a gap. In additional, this lower bound was shown only for tropical semirings. The big open question is whether we can show the same lower bound for the Boolean semiring. Resolving this question would provide a vast generalization on a famous result by Alon and Boppana [1] that shows that monotone Boolean circuits that solve the  $k$ -clique problem over a graph with  $n$  vertices have size  $\Omega(n^k/(\log n)^k)$ .

**From theory to practice.** Finally, an exciting future direction is to bring some of the theoretical algorithmic ideas for join processing to practice. As we already mentioned, several techniques such as worst-case optimal joins, Yannakakis’ algorithm, and ranked enumeration have been implemented in successful prototype systems. But can any of these ideas become part of real-world database systems? For this to be successful, it is important to figure out the minimum changes to existing systems that can support the new join algorithms. What are the applications where the algorithmic innovations will help more? Can some of the current state-of-the-art algorithms such as PANDA become practical? These are all questions that span both theory and systems and require the collaboration of database theoreticians and practitioners.

---

## References

---

- 1 Noga Alon and Ravi B. Boppana. The monotone circuit complexity of boolean functions. *Comb.*, 7(1):1–22, 1987. doi:10.1007/BF02579196.
- 2 Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997. doi:10.1007/BF02523189.
- 3 Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *FOCS*, pages 739–748. IEEE Computer Society, 2008. doi:10.1109/FOCS.2008.43.
- 4 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007. doi:10.1007/978-3-540-74915-8\_18.
- 5 Altan Birlir, Alfons Kemper, and Thomas Neumann. Robust join processing with diamond hardened joins. *Proc. VLDB Endow.*, 17(11):3215–3228, 2024. doi:10.14778/3681954.3681995.
- 6 Karl Bringmann and Egor Gorbachev. A fine-grained classification of subquadratic patterns for subgraph listing and friends. *CoRR*, abs/2404.04369, 2024. doi:10.48550/arXiv.2404.04369.
- 7 Nofar Carmeli, Nikolaos Tziavelis, Wolfgang Gatterbauer, Benny Kimelfeld, and Mirek Riedewald. Tractable orders for direct access to ranked answers of conjunctive queries. In *PODS*, pages 325–341. ACM, 2021. doi:10.1145/3452021.3458331.
- 8 E. F. Codd. A relational model of data for large shared data banks (reprint). In *Software Pioneers*, pages 263–294. Springer Berlin Heidelberg, 2002. doi:10.1007/978-3-642-59412-0\_16.
- 9 Kyle Deeds and Timo Camillo Merkl. Partition constraints for conjunctive queries: Bounds and worst-case optimal joins, 2025. arXiv:2501.04190.
- 10 Kyle Deeds, Dan Suciu, Magda Balazinska, and Walter Cai. Degree sequence bound for join cardinality estimation. In *ICDT*, volume 255 of *LIPICs*, pages 8:1–8:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICs.ICDT.2023.8.
- 11 Shaleen Deep, Xiao Hu, and Paraschos Koutris. Fast join project query evaluation using matrix multiplication. In *SIGMOD Conference*, pages 1213–1223. ACM, 2020. doi:10.1145/3318464.3380607.
- 12 Shaleen Deep, Xiao Hu, and Paraschos Koutris. Ranked enumeration of join queries with projections. *Proc. VLDB Endow.*, 15(5):1024–1037, 2022. doi:10.14778/3510397.3510401.

- 13 Shaleen Deep, Xiao Hu, and Paraschos Koutris. General space-time tradeoffs via relational queries. In *WADS*, volume 14079 of *Lecture Notes in Computer Science*, pages 309–325. Springer, 2023. doi:10.1007/978-3-031-38906-1\_21.
- 14 Shaleen Deep and Paraschos Koutris. Compressed representations of conjunctive query results. In *PODS*, pages 307–322. ACM, 2018. doi:10.1145/3196959.3196979.
- 15 Shaleen Deep and Paraschos Koutris. Ranked enumeration of conjunctive query results. In *ICDT*, volume 186 of *LIPICs*, pages 5:1–5:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ICDT.2021.5.
- 16 Shaleen Deep, Hangdong Zhao, Austen Z. Fan, and Paraschos Koutris. Output-sensitive conjunctive query evaluation. *Proc. ACM Manag. Data*, 2(5):220:1–220:24, 2024. doi:10.1145/3695838.
- 17 Austen Z. Fan, Paraschos Koutris, and Hangdong Zhao. The fine-grained complexity of boolean conjunctive queries and sum-product problems. In *ICALP*, volume 261 of *LIPICs*, pages 127:1–127:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICs.ICALP.2023.127.
- 18 Austen Z. Fan, Paraschos Koutris, and Hangdong Zhao. Tight bounds of circuits for sum-product queries. *Proc. ACM Manag. Data*, 2(2):87, 2024. doi:10.1145/3651588.
- 19 Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.*, 13(11):1891–1904, 2020. URL: <http://www.vldb.org/pvldb/vol13/p1891-freitag.pdf>.
- 20 Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In *PODS*, pages 21–32. ACM Press, 1999. doi:10.1145/303976.303979.
- 21 Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, pages 31–40. ACM, 2007. doi:10.1145/1265530.1265535.
- 22 Alina Harbuzova, Ce Jin, Virginia Vassilevska Williams, and Zixuan Xu. Improved roundtrip spanners, emulators, and directed girth approximation. In *SODA*, pages 4641–4669. SIAM, 2024. doi:10.1137/1.9781611977912.166.
- 23 Xiao Hu. Fast matrix multiplication for query processing. *Proc. ACM Manag. Data*, 2(2):98, 2024. doi:10.1145/3651599.
- 24 Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. *Log. Methods Comput. Sci.*, 19(3), 2023. doi:10.46298/LMCS-19(3:11)2023.
- 25 Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. On functional aggregate queries with additive inequalities. In *PODS*, pages 414–431. ACM, 2019. doi:10.1145/3294052.3319694.
- 26 Mahmoud Abo Khamis, Xiao Hu, and Dan Suciu. Fast matrix multiplication meets the submodular width. *CoRR*, abs/2412.06189, 2024. doi:10.48550/arXiv.2412.06189.
- 27 Mahmoud Abo Khamis, Vasileios Nakos, Dan Olteanu, and Dan Suciu. Join size bounds using lp-norms on degree sequences. *Proc. ACM Manag. Data*, 2(2):96, 2024. doi:10.1145/3651597.
- 28 Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. In-database learning with sparse tensors. In *PODS*, pages 325–340. ACM, 2018. doi:10.1145/3196959.3196960.
- 29 Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: questions asked frequently. In *PODS*, pages 13–28. ACM, 2016. doi:10.1145/2902251.2902280.
- 30 Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. What do Shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *PODS*, pages 429–444. ACM, 2017. doi:10.1145/3034786.3056105.
- 31 Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. PANDA: query evaluation in submodular width. *CoRR*, abs/2402.02001, 2024. doi:10.48550/arXiv.2402.02001.
- 32 Dániel Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *J. ACM*, 60(6):42:1–42:51, 2013. doi:10.1145/2535926.

- 33 Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In *PODS*, pages 37–48. ACM, 2012. doi:10.1145/2213556.2213565.
- 34 Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16:1–16:40, 2018. doi:10.1145/3180143.
- 35 Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013. doi:10.1145/2590989.2590991.
- 36 Dan Olteanu and Maximilian Schleich. Factorized databases. *SIGMOD Rec.*, 45(2):5–16, 2016. doi:10.1145/3003665.3003667.
- 37 Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2:1–2:44, 2015. doi:10.1145/2656335.
- 38 Neil Robertson and Paul D. Seymour. Graph minors. III. planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984. doi:10.1016/0095-8956(84)90013-3.
- 39 Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *SIGMOD Conference*, pages 3–18. ACM, 2016. doi:10.1145/2882903.2882939.
- 40 Luc Segoufin. Enumerating with constant delay the answers to a query. In *ICDT*, pages 10–20. ACM, 2013. doi:10.1145/2448496.2448498.
- 41 Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *Proc. VLDB Endow.*, 13(9):1582–1597, 2020. doi:10.14778/3397230.3397250.
- 42 Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Ranked enumeration for database queries. *SIGMOD Rec.*, 53(3):6–19, 2024. doi:10.1145/3703922.3703924.
- 43 Todd L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR*, abs/1210.0481, 2012. arXiv:1210.0481.
- 44 Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, pages 96–106. OpenProceedings.org, 2014. doi:10.5441/002/ICDT.2014.13.
- 45 Yisu Remy Wang, Max Willsey, and Dan Suciu. Free join: Unifying worst-case optimal and traditional joins. *Proc. ACM Manag. Data*, 1(2):150:1–150:23, 2023. doi:10.1145/3589295.
- 46 Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. Predicate transfer: Efficient pre-filtering on multi-join queries. In *CIDR*. www.cidrdb.org, 2024. URL: <https://www.cidrdb.org/cidr2024/papers/p22-yang.pdf>.
- 47 Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94. IEEE Computer Society, 1981.
- 48 Hangdong Zhao, Shaleen Deep, and Paraschos Koutris. Space-time tradeoffs for conjunctive queries with access patterns. In *PODS*, pages 59–68. ACM, 2023. doi:10.1145/3584372.3588675.
- 49 Hangdong Zhao, Austen Z. Fan, Xiating Ouyang, and Paraschos Koutris. Conjunctive queries with negation and aggregation: A linear time characterization. *Proc. ACM Manag. Data*, 2(2):75, 2024. doi:10.1145/3651138.