Succinct Structure Representations for Efficient Query Optimization

ZHEKAI JIANG*, EPFL, Switzerland
QICHEN WANG*, Nanyang Technological University, Singapore
CHRISTOPH KOCH, EPFL, Switzerland

Structural decomposition methods offer powerful theoretical guarantees for join evaluation, yet they are rarely used in real-world query optimizers. A major reason is the difficulty of combining cost-based plan search and structure-based evaluation. In this work, we bridge this gap by introducing meta-decompositions for acyclic queries, a novel representation that succinctly represents all possible join trees and enables their efficient enumeration. Meta-decompositions can be constructed in polynomial time and have sizes linear in the query size. We design an efficient polynomial-time cost-based optimizer based directly on the meta-decomposition, without the need to explicitly enumerate all possible join trees. We characterize plans found by this approach using a novel notion of width, which effectively implies the theoretical worst-case asymptotic bounds of intermediate result sizes and running time of any query plan. Experimental results demonstrate that, in practice, the plans in our class are consistently comparable to—even in many cases better than—the optimal ones found by the state-of-the-art dynamic programming approach, especially on large and complex queries, while our planning process runs by orders of magnitude faster, comparable to the time taken by common heuristic methods.

CCS Concepts: • **Theory of computation → Database query processing and optimization (theory)**; • **Information systems → Query optimization**; **Query planning**; **Join algorithms**.

Additional Key Words and Phrases: Conjunctive queries; Plan enumeration; Cost-based optimization

ACM Reference Format:
Zhekai Jiang, Qichen Wang, and Christoph Koch. 2026. Succinct Structure Representations for Efficient Query Optimization. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 240 (June 2026), 27 pages. https://doi.org/10.1145/3802117

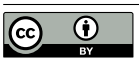
1 Introduction

Joins are the cornerstone of relational database queries. However, query optimizers face extreme difficulty when planning large multi-way join queries, as the search space of join orders grows exponentially as the number of relations increases. In fact, finding the optimal join order of a query is well-known to be NP-hard in general [13, 44]. For this reason, beyond a small number of joins, current query optimizers must abandon exhaustive dynamic programming and resort to heuristic or greedy algorithms. As a consequence, real-world optimizers often miss the best plans and produce suboptimal ones that can be by orders of magnitude slower.

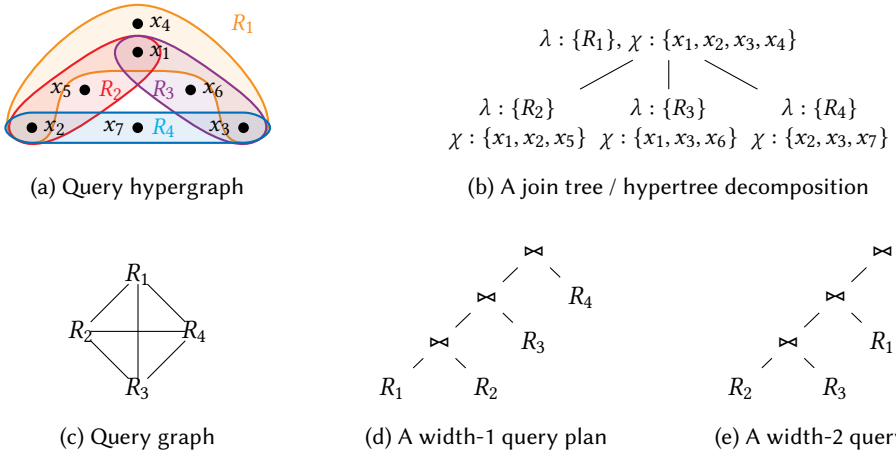
The database theory community has long been providing tools that exploit query structures to tackle this complexity. These approaches usually *decompose* queries into specific tree structures,

*Both authors contributed equally to this research.

Authors' Contact Information: Zhekai Jiang, EPFL, Lausanne, Vaud, Switzerland, zhekai.jiang@epfl.ch; Qichen Wang, Nanyang Technological University, Singapore, qichen.wang@ntu.edu.sg; Christoph Koch, EPFL, Lausanne, Vaud, Switzerland, christoph.koch@epfl.ch.



ACM 2836-6573/2026/6-ART240
https://doi.org/10.1145/3802117



(a) Query hypergraph

(b) A join tree / hypertree decomposition

(c) Query graph

(d) A width-1 query plan

(e) A width-2 query plan

Fig. 1. The hypergraph, a join tree, the query graph, and two query plans of the query $Q_{1.1}$ in Example 1.1

such as join trees [55] using the GYO algorithm [25, 56], which were later generalized to hypertree decompositions [21]. The resulting decompositions effectively capture the hardness of queries and, through specific requirements on connectivity, allow one to evaluate queries with theoretically-guaranteed instance-optimal complexity using the Yannakakis algorithm [7, 55] or its variants [21, 50]. Over the years, these structure-based methods have been generalized to evaluate joins with projections [4, 15, 27], aggregations [2, 31], differences [28, 59], top-k [51], comparisons [52], etc., all with strong theoretical performance guarantees. These results suggest that leveraging a query's structural properties can be of great help for query optimization and can dramatically improve query evaluation, with robust worst-case bounds. Yet, in practice, despite limited attempts by, e.g., Scarcello et al. [43], RelationalAI [3], and Gottlob et al. [20], real-world systems rarely utilize such structure-based approaches. Therefore, they continue to struggle with large, complex queries.

Why are structure-based methods rarely adopted in mainstream systems? A core obstacle is the dichotomy between cost-based plan search and structure-based evaluation [33]. Current cost-based optimizers usually search for query plans with optimal or near-optimal cost under some cost model using (1) **exact algorithms**, mostly based on dynamic programming, such as [39, 45, 46, 48, 49], which have to run in exponential time, and thus do not scale to very large numbers of relations; (2) **heuristic strategies** such as greedy operator ordering (GOO) [17], iterative dynamic programming [32], UnionDP [37], and adaptive approaches [9, 40], which have neither optimality guarantees nor a reasonable approximation factor unless under some assumption, such as independence of predicate selectivities; (3) **transformation-based techniques**, such as the Volcano/Cascades framework [23, 24] and genetic algorithms [6]; and (4) more recently, **machine learning–based tuning and rewriting** [12, 35, 38, 54, 61, 62]. Although certain approaches leverage, e.g., *query graphs* [9, 37, 39, 40], they only show the existence of join attributes between relations but do not have a clear connection with the requirements of the aforementioned structural decompositions that help reason about asymptotic complexities.

Example 1.1. Consider the query

$$Q_{1.1} \leftarrow \pi_{\emptyset}(R_1[x_1, x_2, x_3, x_4] \bowtie R_2[x_1, x_2, x_5] \bowtie R_3[x_1, x_3, x_6] \bowtie R_4[x_2, x_3, x_7]),$$

which is a Boolean acyclic query. Its hypergraph is shown in Fig. 1a, and a join tree, which is also a hypertree decomposition, in Fig. 1b. Its query graph, as shown in Fig. 1c, is a clique with the four relations as vertices, as each pair shares some join attribute. Fig. 1d is a query plan that follows the

structure of the join tree (which we formally define later). Fig. 1e does not, but it is still within the search space of traditional algorithms according to the query graph.

Theoretical research typically focuses on worst-case asymptotic data complexity, assuming that cost differences among decompositions are negligible. In practice, however, different decompositions may lead to significantly different concrete cost values. Therefore, using only structural properties, such as hypertree widths, is not sufficient [20]. However, selecting the best decomposition is challenging, as there can be a super-exponential number of candidates. Without an efficient cost-based exploration of alternatives, explicitly scanning through all of them would be computationally prohibitive. Furthermore, evaluation plans for structure-based methods are not natively supported by current systems that use standard binary query plans [20, 33]. As a result, deploying them requires a dedicated optimization and execution pipeline, which limits their practical adoption.

1.1 Our Contributions

In this work, we take a major step towards bridging structural decomposition techniques with cost-based query optimization. We start by introducing **a novel *width* measurement for query plans** and show that, for query plans of width 1, we can effectively embed the query's tree structure within a traditional query plan to guide query planning even in the absence of cost information. In particular, we show that acyclic Boolean queries and relation-dominated queries can be evaluated in *instance-optimal time* using width-1 plans. We also present a method which, for any given join tree of a query, constructs the optimal width-1 query plan that adheres to the join tree in $O(|Q|)$, *linear in the query size* $|Q|$.

By searching for plans of minimal width, we restrict the search space to only those that follow the query's optimal tree decompositions. To make this practical, we introduce *meta-decompositions* for acyclic conjunctive queries that **universally encode, in one succinct structure, the entire family of join trees of the query**. Despite the possibly super-exponential number of join trees of a given query, this representation **can be constructed in polynomial time and has size linear in the query size**. We give an efficient procedure that enumerates all join trees in polynomial amortized delay from a meta-decomposition, as a potential tool for other structure-based methods requiring explicit join tree enumeration.

Nevertheless, such an enumeration still incurs an exponential overhead. Therefore, we develop an efficient ***structure-guided cost-based optimization* framework to find the optimal width-1 query plan among all join trees** without naïvely scanning all candidates. For queries whose meta-decompositions have bounded fan-out, *our algorithm finds the optimal width-1 plan in linear time*. For general queries, we also provide efficient heuristics with the same linear complexity and a low constant factor.

Although we currently restrict our focus to acyclic queries only, such queries in fact account for the overwhelming majority of real workloads [10, 36]. As shown in Table 1, among 8,125 queries across several popular benchmarks, 7,929 (97.59%) are acyclic. Structure-based methods for optimizing acyclic queries have therefore been of sustained interest [5, 8, 28, 36, 50–53, 55, 60].

We then empirically demonstrate that our approach can indeed be game-changing for query optimization for large joins. We **implement an optimizer metaDecomp** and evaluate it on queries from 4 benchmarks, which include many large, complex queries. We compare the results against several state-of-the-art approaches. The results show that width-1 plans consistently match or even outperform the plans found by these baselines, while taking very small amounts of time for optimization. We also demonstrate that join tree enumeration can be by orders of magnitude faster than existing approaches based on naïve GYO reductions, and that selecting join trees

Benchmark	# Queries	# Acyclic	% Acyclic
TPC-H [47]	22	21	95.45 %
JOB [34]	113	113	100 %
STATS-CEB [26]	2603	2603	100 %
Spider-NLP [57]	4712	4709	99.94 %
DSB [16] (SPJ queries only)	300	240	80 %
Musicbrainz [37]	375	243	64.8 %
Total	8125	7929	**97.59 %**

Table 1. Number of acyclic queries in major benchmarks. Part of the data comes from [36].

using our cost-based optimization framework significantly improves the effectiveness of existing structure-based methods.

Paper organization. The rest of the paper is organized as follows: After laying out the necessary background in Section 2, we first define the *width* notion of query plans and discuss the connection between query structures and the widths of query plans in Section 3. In Section 4, we define *meta-decompositions*, the algorithm to construct them, as well as our way of enumerating all join trees based on them. In Section 5, we demonstrate how to perform cost-based optimization using meta-decompositions. In Section 6, we present experimental results to show the efficiency of query optimization and execution using this approach. We conclude and discuss possible future work in Section 7. Due to space constraints, missing proofs can be found in the full version of the paper [30].

2 Preliminaries

We write 2^S for the set of all subsets, i.e., the power set, of a set S. If a function $f : A \to B$ and $S \subseteq A$, we let $f(S) = \{f(a) : a \in S\}$. For a set of sets S, we write $\cup S = \bigcup_{x \in S} x$. If $f : A \to T$, where T is a set, and $S \subseteq A$, we let $f(S) = \bigcup_{x \in S} f(x)$.

2.1 Conjunctive Queries and Hypergraphs

Conjunctive queries can be represented by hypergraphs, and our terminology follows the seminal references on this topic [18, 19, 21, 22, 50]. Let $\mathcal{D} = \{R_1[\bar{x}_1], \cdots, R_n[\bar{x}_n]\}$ be a database consisting of n relations, where $\bar{x}_i$ is the set of attributes of relation R_i. A conjunctive query (with self-predicates) is defined as

$$Q \leftarrow \pi_{\bar{y}} \left(\sigma_1(R_1[\bar{x}_1]) \bowtie \cdots \bowtie \sigma_n(R_n[\bar{x}_n])\right),$$

where σ_i is the set of selection predicates involving only attributes in $\bar{x}_i$, $\bar{y}$ is the set of output attributes, and $\bowtie$ denotes natural join (with attribute renaming to avoid ambiguity). If $\bar{y} = \bigcup_{i \in [n]} \bar{x}_i$, the query is called a full join query and we omit $\bar{y}$ for simplicity; if $\bar{y} = \emptyset$, the query is a Boolean query that returns true if there exists a tuple satisfying all specified joins and predicates, or false otherwise. We also require the query to be *self-join-free* at the logical level: If multiple relations correspond to the same physical relation, they are treated as distinct logical relations with unique renamings.

We define the associated *hypergraph* for the conjunctive query Q as $H = (V(H), E(H))$, where the vertices set $V(H)$ consists of all attributes appearing in the query, i.e., $V(H) = \bigcup_{i \in [n]} \bar{x}_i$, and the *hyperedges* set $E(H) \subseteq 2^{V(H)} \setminus \{\emptyset\}$ contains one hyperedge for each relation, i.e., $E(H) = \{e_1, \cdots, e_n\}$, where each $e_i = \bar{x}_i$.

2.2 Acyclicity and Join Trees

A conjunctive query is acyclic if its associated hypergraph is acyclic, and we define acyclicity in the standard way in database theory, e.g., [1, Section 6.4]. Namely, a hypergraph H is acyclic if and only if the output of the GYO algorithm [25, 56] on H is empty [1, Theorem 6.4.5]. Equivalently, H is acyclic if and only if there exists a *join tree* [1, Theorem 6.4.5], or, a width-1 hypertree decomposition [21].

Slightly differently from the usual definition of join trees [1], where each vertex is simply a relation, in this paper, we define them using notations for hypertree decompositions [21], to make it easier to generalize to the meta-decompositions that we will define later.

Let Q be an acyclic query with associated hypergraph H. A *join tree* of Q is a labeled tree $T = (V(T), r(T), E(T), \chi, \lambda)$ with vertices $V(T)$, root $r(T)$, and edges $E(T)$. $\chi : V(T) \to 2^{V(H)}$ is a function s.t. $\bigcup_{p \in V(T)} \chi(p) = V(H)$, and $\lambda : V(T) \to 2^{E(H)}$.[1] And T needs to satisfy the following conditions:

(C_1) **Coverage condition.** For each $e \in E(H)$, there exists exactly one $p \in V(T)$ with $\lambda(p) = \{e\}$.
(C_2) **Connectedness condition.** For each $v \in V(H)$, the set of vertices $\{p \in V(T) : v \in \chi(p)\}$ induces a connected subtree of T. Or, equivalently, for all pairs of vertices $p, q \in V(T)$ and all s on the (unique) path between p and q on T, we have $\chi(p) \cap \chi(q) \subseteq \chi(s)$.
(C_3) **Width-1.** For all $p \in V(T)$, $|\lambda(p)| = 1$ and, without loss of generality, $\chi(p) = V(\lambda(p))$.

Example 2.1. We revisit the query $Q_{1.1}$ in Example 1.1. Its hypergraph is illustrated in Fig. 1a. It is easy to verify that the join tree in Fig. 1b satisfies conditions (C_1) and (C_3). Regarding condition (C_2), consider, e.g., attribute x_1. The vertices in the join tree that include x_1 are the ones with R_1, R_2, and R_3, which induce a connected subtree. Similar arguments apply for all other attributes as well.

We assume all trees are rooted and consider rerootings as distinct trees. We write T_p as the subtree rooted at p, i.e., with p and all its descendants. For every neighbor (child or parent) q of vertex p in T, we write $T_{p \to q}$ for the subtree with

$$V(T_{p \to q}) = \{s \in V(T) \setminus \{p\} : \text{the path from } p \text{ to } s \text{ in } T \text{ contains } q\}.$$

Intuitively, if q is a child of p, it is the same as T_q. If q is a parent of p, it denotes the subtree of T with all vertices except those in T_p.

We define the *fan-out* $f(p)$ of any $p \in V(T)$ as the number of children of p on T. The *fan-out* of the entire tree T, denoted by $f(T)$, is the maximum fan-out among all nodes of T.

Given a join tree T, for every $p \in V(T)$, we define the *induced query* $Q(p)$ as the join of all relations in the subtree rooted at p.

One important observation is that *an acyclic conjunctive query may have a super-exponential number of distinct join trees.*

Theorem 2.2. *Consider* star queries[2] *of the form*

$$Q_{\text{star}} \leftarrow R_1[\bar{x}_1] \bowtie \cdots \bowtie R_n[\bar{x}_n],$$

where for any $i, j \in [n]$, $\bar{x}_i \cap \bar{x}_j = \{x\}$, for some attribute x. There are n^{n-1} possible join trees for such a star query with n relations.

Example 2.3. The query $Q_{2.3} \leftarrow R_1[x_1, x_2] \bowtie R_2[x_1, x_3] \bowtie R_3[x_1, x_4] \bowtie R_4[x_1, x_5]$ is such a star query. Fig. 2a shows its associated hypergraph, and Fig. 2b shows two of its valid join trees.

In fact, the possible join trees of such queries in Theorem 2.2 and Example 2.3 are simply *all possible trees with n vertices* (or, equivalently, all spanning trees of a complete graph K_n), each labeled by one relation. Any such tree satisfies the connectedness condition (C_2). There are a total of n^{n-1} such trees (n^{n-2} distinct non-rooted trees [41], each of which has n possible root nodes).

2.3 Query Plans and Join Ordering

A *query plan* $\mathcal{P} = (V(\mathcal{P}), r(\mathcal{P}), E(\mathcal{P}))$ for a query Q is a tree rooted at $r(\mathcal{P})$, where each non-leaf node is a relational operator (e.g., join, projection, selection), and each leaf node corresponds to a

[1] For simplicity, we may use hyperedges (e.g., R_1) to label and identify the vertices of join trees instead of writing the full λ- and χ-labels when the context is clear.

[2] We note that, in some literature, e.g., [9], such queries are called "*clique*" queries, because their *query graphs* are cliques.

(a) Hypergraph of the query (b) Two valid join trees

Fig. 2. Hypergraph and two valid join trees of the star query $Q_{2.3}$ in Example 2.3

base relation. A plan specifies an order of operations to evaluate the query, where the outputs of child nodes serve as inputs to the parent. In this work, we focus on query optimization for binary joins, where each join operation in the query plan has exactly two child nodes. For the purpose of join ordering, *we use only join operators in internal nodes and assume selections and projections are inherently integrated in the join or scan operation at the lowest possible level.* Namely, in each step of join or scan, we apply a filter condition if all relevant attributes already appear in the subtree, and we project out attributes that neither appear in the output nor act as a join key with some remaining relation.

Given a query plan $\mathcal{P}$, for every $p \in V(\mathcal{P})$, we define the *induced query* $Q(p)$ as the join of all relations in the subtree rooted at p (with all possible projection and filters inherently applied).

We exclude Cartesian products: for every join node, the outputs of its children must share some common attribute as the join key.

A query plan is called *left-deep* (or *right-deep*) if all join operations have their right (left) child node corresponding to a base relation; otherwise, the query plan is referred to as *bushy*.

We apply cost functions C on query plans, in the following form, to measure the optimality of query plans.

$$C(\mathcal{P}) = \begin{cases} c(r(\mathcal{P})), & \text{if } r(\mathcal{P}) \text{ is single relation} \\ c(r(\mathcal{P})) \oplus C(\mathcal{P}_1) \oplus C(\mathcal{P}_2), & \text{if } r(\mathcal{P}) \text{ is join operation} \end{cases}$$

where $c(r(\mathcal{P}))$ is the cardinality of the output of $r(\mathcal{P})$; $\mathcal{P}_1$ and $\mathcal{P}_2$ (if applicable) are the left and right subtrees of $r(\mathcal{P})$, respectively; and $\oplus$ is some operator to aggregate $c(r)$, $C(\mathcal{P}_1)$, and $C(\mathcal{P}_2)$. In this work, we adopt the cost function C_{out} to minimize the total intermediate result size by substituting the $\oplus$ operator simply with arithmetic +, as is standard in, e.g., [13, 17, 40, 46].

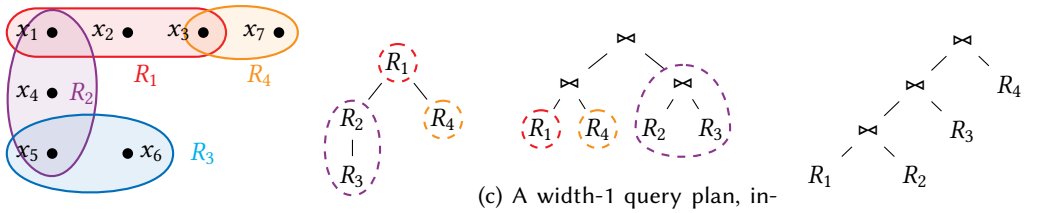
The typical approach to find the optimal plan is to recursively find the optimal split of a set of relations into two disjoint subsets [39, 45, 46, 48, 49]. With a given cost function C, the problem can be solved by the dynamic programming (DP) algorithm DPccp [39] in $O(3^n)$. The recently proposed algorithm DPconv [46] reduces the complexity for the cost model C_{out} to $O(2^n n^3 W \log(Wn))$, where W is the largest join cardinality.

3 Widths of Query Plans and Width-1 Query Plans

In this section, we begin by introducing our notion of widths for query plans, which connects a plan's structure to the potential sizes of its intermediate results. We consider the "interface" of each intermediate step in the query plan, i.e., the set of attributes that serve as join keys with the remaining relations. Intuitively, the width of a query plan shows the number of relations needed to cover the interface of each step in the query plan. This measure helps quantify the sizes of intermediate results and the complexity of a query plan before looking at the specific database instance.

Example 3.1. Consider the query

$$Q_{3.1} \leftarrow \pi_{\emptyset}(R_1[x_1, x_2, x_3] \bowtie R_2[x_1, x_4, x_5] \bowtie R_3[x_5, x_6] \bowtie R_4[x_3, x_7]).$$



(a) Hypergraph of the query (b) A join tree (c) A width-1 query plan, induced by the join tree in Fig. 3b (d) A width-2 query plan

Fig. 3. The hypergraph, a join tree, a width-1 query plan, and a width-2 query plan of the query $Q_{3.1}$

Its hypergraph is shown in Fig. 3a and a join tree in Fig. 3b. Given this join tree, Fig. 3c is an example of a valid width-1 query plan. After joining R_1 and R_4, x_1 is the only attribute in the intersection with the remaining relations R_2 and R_3, and it can be covered by only one relation R_1. Similarly, at the step that joins R_2 and R_3, x_1 is the only attribute in the interface. A width of 1 implies that, if the size of any one single relation is bounded by N, then the size of each intermediate step can be bounded by N, as it is at most the size of the relation that covers the interface.

On the contrary, Fig. 3d shows a query plan that is width-2: When we join R_1 and R_2, the interface contains x_5 and x_3, which are needed to join R_3 and R_4, respectively. But there is no single relation that can cover both x_5 and x_3, so the intermediate sizes can be as high as N^2, instead of N as in the width-1 plan in Fig. 3c.

More formally, given a query plan $\mathcal{P} = (V(\mathcal{P}), r(\mathcal{P}), E(\mathcal{P}))$, for any node $p \in V(\mathcal{P})$, let $H_p = (V(H_p), E(H_p))$ be the hypergraph induced by the relations being joined in the subtree rooted at p, and let the induced query $Q(p) = \bowtie_{e \in E(H_p)} e$. Let $\overline{H}_p = (V(\overline{H}_p), E(\overline{H}_p))$ be the *residual hypergraph* with relations that are not yet joined in the subtree rooted at p, i.e., $E(\overline{H}_p) = E(H) \setminus E(H_p)$, and $V(\overline{H}_p) = \cup(E(\overline{H}_p))$. We define the *interface* $I(p) = V(H_p) \cap V(\overline{H}_p)$, i.e., the set of attributes that will serve as join keys with the remaining relations. The *width of node* p is the smallest number of relations needed to cover this interface, i.e.,

$$w(p) = \min \left\{ |S| : S \subseteq E(H_p) \text{ and } I(p) \subseteq \cup S \right\}.$$

The width of the query plan $\mathcal{P}$ is the maximum width over all nodes.

The width of a query plan indicates an upper bound on the minimal intermediate results over all nodes p during evaluation.

Theorem 3.2. *Given a Boolean conjunctive query Q and a query plan $\mathcal{P}$ for the query, the maximum intermediate result size, i.e., $\max_{p \in V(H_p)} |\pi_{I(p)} Q(p)|$, is upper-bounded by $O(N^{w(\mathcal{P})})$ in the worst case, where N is the maximum cardinality of any base relation.*

This notation of width has a direct connection with join trees:

Example 3.3. We revisit the query $Q_{3.1}$ from Example 3.1. Take p as the node of the join tree with R_2, as an example. The induced query $Q(p) = R_2 \bowtie R_3$. In the width-1 query plan in Fig. 3c, the right child of the root node, which we denote by q, induces exactly $Q(p) = R_2 \bowtie R_3$. In the other direction, for the node q of the query plan, we consider its children u and v (which correspond to the leaf nodes with R_2 and R_3 respectively). u has interface $I(u) = \{x_1, x_5\}$, which can be covered by a single relation R_2. And v has interface $I(v) = \{x_5\}$, which can be covered by R_3. However, for the width-2 query plan in Fig. 3d, there is no such q of the query plan such that $Q(p) = Q(q) = R_2 \bowtie R_3$.

We formally define this correspondence as follows:

Definition 3.4. We say a query plan $\mathcal{P}$ is *induced by* a join tree T if

(1) for each node $p \in V(T)$, there exists $q \in V(\mathcal{P})$ with $Q(p) = Q(q)$, i.e., the induced query of node p in the join tree is exactly the induced query of node q in the query plan; and
(2) for each non-leaf node $q \in V(\mathcal{P})$ with two children u and v, there exists some $s \in E(H_u)$ and $t \in E(H_v)$ such that $I(u) \subseteq s$, $I(v) \subseteq t$, and there exists an edge between s and t in $E(T)$.

Under this definition, we have that

Theorem 3.5. *Given a query plan $\mathcal{P}$ for a query Q, the width $w(\mathcal{P}) = 1$ if and only if there exists a join tree T of Q that induces $\mathcal{P}$.*

Theorem 3.5 directly leads to the relationship between width-1 query plans and acyclic queries:

Theorem 3.6. *A conjunctive query Q has width-1 query plans if and only if Q is acyclic.*

Width-1 query plans are closely related to structure-based query evaluation methods. Such methods follow a similar pattern: given a join tree, they apply a series of reductions from the leaves upwards until only a single node remains. Width-1 query plans exploit the join tree structure in a similar way, evaluating the query bottom-up: each node $p \in V(T)$ completes all its internal joins with its child subtrees before joining with its parent. Such a strategy provides several benefits.

Limiting intermediate result sizes. We have seen that, by projecting out attributes that are neither part of the final output nor in the interface at each step, we limit the sizes of intermediate results, as shown by Theorem 3.2. Although this approach excludes some alternative plans with higher widths, the excluded ones do not preserve such a theoretical guarantee. Taking again Example 3.1, if we choose the width-2 query plan in Fig. 3d, x_3 has to be kept in all intermediate results before R_4 is joined at the very end, whereas, in the width-1 query plan in Fig. 3c, it can already be projected out after completing $R_1 \bowtie R_4$.

Local optimization. Instead of searching in a vast space of arbitrary join orders for all relations, we can locally optimize the join order at each node of the join tree. For each node $p \in V(T)$ with children $c_1, \cdots, c_{f(p)}$ in the join tree, we only need to determine the optimal join order between p and $Q(c_1), \ldots, Q(c_{f(p)})$. The order in which these joins are performed can be optimized independently at each node, without considering other parts of the query. Such an approach, therefore, significantly prunes the search space for optimization, making plan selection much more tractable, and can yield near-optimal performance for most practical queries. In the experiments (Section 6), we show in Table 2 that the fan-out $f(T)$ tends to be very low in real benchmarks.

Example 3.7. Consider again the join tree in Fig. 3b for the query $Q_{3.1}$. For width-1 query plans induced by this join tree, there must exist a subtree with $R_2 \bowtie R_3$, as required by Definition 3.4(1). It therefore remains only to optimize the order to join R_1, $(R_2 \bowtie R_3)$, and R_4, which is the local optimization problem at the root node R_1 of the join tree. If the cardinalities $|R_1 \bowtie R_4| < |(R_2 \bowtie R_3) \bowtie R_1|$, then the optimal plan in our cost model is the one in Fig. 3c.

With only local search needed, assuming a low fan-out, finding the optimal width-1 query plans induced by a join tree can be done more efficiently than through a global search:

Theorem 3.8. *Given a join tree T, the optimal width-1 query plan induced by T can be found in time $O(f(T)2^{f(T)}|Q|)$.*

Structure-guided query evaluation. An additional advantage of width-1 query plans is that they naturally accommodate structure-guided query evaluation methods. For example, we can simulate the Yannakakis algorithm for evaluating *relation-dominated* conjunctive queries [50] by following a width-1 query plan. A conjunctive query is relation-dominated if it is acyclic and there exists a

relation $R_i[\bar{x}_i]$ that contains all the output attributes, i.e., $\bar{y} \subseteq \bar{x}_i$. It shall be noted that an acyclic Boolean query is also a relation-dominated query, since $\bar{y} = \emptyset$. For this type of query, we have the following guarantee of evaluation time:

Theorem 3.9. *A relation-dominated conjunctive query can be evaluated in $O(|db|)$ time using a width-1 query plan, where $|db|$ represents the size of the database.*

To create a linear-time width-1 query plan for evaluating a relation-dominated query, let R_i denote the relation containing all output attributes. One can construct the query plan using any join tree with R_i as the root. The width-1 query plan will then evaluate the query bottom-up. Because all output attributes are located at the root of the join tree and at the end of the query plan, when the query plan is evaluating node p, all attributes in T_p but not in p are removed from the query, guaranteeing linear running time.

To conclude this section, we highlight some important distinctions between our definition of width-1 query plans and Yannakakis-style query plans [50, 55]. Although they both evaluate queries by traversing join trees, width-1 query plans do not use full reductions via semi-joins as in Yannakakis-style algorithms. We have shown, nevertheless, that these plans, by appropriate projections of irrelevant attributes, still preserve favorable theoretical guarantees of asymptotic complexity for Boolean conjunctive queries (Theorem 3.2) and relation-dominated queries (Theorem 3.9). In Section 6, we will experimentally show that these plans, in practice, are indeed highly efficient. Furthermore, without the need for full reductions, width-1 query plans are much easier to integrate into the query execution frameworks of current database systems, enabling more efficient cost-based optimization, as we will discuss in the remainder of the paper.

4 Meta-Decompositions

Even though we have shown a connection between join trees and width-1 query plans in Theorem 3.2 and Theorem 3.5, it is impractical to find the optimal width-1 query plan by enumerating every possible join tree—As noted earlier, a query can have a super-exponential number of possible join trees. Therefore, in this section, we introduce and explore a generalized representation that efficiently captures all join trees of an acyclic query. This enables us to optimize queries directly on this representation, eliminating the need to enumerate all possible join trees.

We highlight an observation that *the exponential number of possible join trees often arises when multiple relations share common join attributes*, as is the case for star queries discussed in Theorem 2.2 and illustrated by Example 2.3. In such a query, any pair of relations has valid join predicates, so in join trees of such queries, these relations can be connected in any arbitrary tree structure. On the contrary, in the case of a query like in Example 3.1, where the join attributes are unique for all possible pairs of relations to be joined, the query demonstrates a more "tree-like" structure. There is a unique tree structure for such a query, with n distinct, structurally isomorphic join trees, differing only by the choice of the root relation.

In light of this, we introduce *meta-decompositions*. In the case of acyclic conjunctive queries, they can capture all possible join trees but require only linear space. One of the major differences from standard join trees is the addition of "minor nodes" to handle cases when more than two relations share the same join key. These nodes have empty λ-labels, meaning they are not associated with a single relation, but their χ-labels contain the shared join keys. Additionally, we add a κ-label to each node to denote the "interface" with the remaining relations of the join tree that are not in the subtree rooted at the node.

Example 4.1. The meta-decomposition of the query $Q_{2.3}$ in Example 2.3 is shown in Fig. 4. The root of this tree is a minor node, showing that the four children share precisely the attribute x_1. Note also that the κ-label of all 4 non-root nodes is exactly $\{x_1\}$.

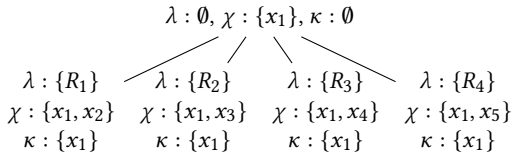


Fig. 4. Meta-decomposition of the query $Q_{2.3}$

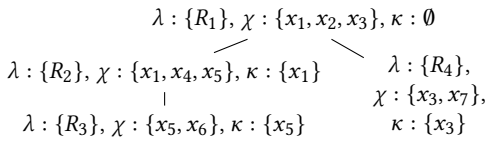


Fig. 5. Meta-decomposition of the query $Q_{3.1}$

Example 4.2. The meta-decomposition of the query $Q_{3.1}$ in Example 3.1 is shown in Fig. 5. It has the exact same structure as the join tree shown in Fig. 3b and has no minor node, as all pairs of relations have different intersections of attributes.

The formal definition is given as follows.

Definition 4.3. Given a conjunctive query Q and the associated hyergraph H, its *meta-decomposition* $M = (V(M), r(M), E(M), \lambda, \chi, \kappa)$ is a labeled tree, where

- $V(M)$ is the set of vertices, $r(M) \in V(M)$ is the root, and $E(M)$ is the set of edges,
- $\lambda : V(M) \to 2^{E(H)}$ maps each vertex of M to a set of hyperedges of H (possibly empty), and
- $\chi : V(M) \to 2^{V(H)}$ and $\kappa : V(M) \to 2^{V(H)}$ each maps each vertex of M to a set of vertices of H,

such that it satisfies (C_1), (C_2), and

(C_3') For all $p \in V(M)$, $\chi(p) \subseteq V(\lambda(p))$ if $\lambda(p) \neq \emptyset$.

(C_4) **Interface condition.** For every non-root $p \in V(M)$, let q be p's parent node. $\kappa(p)$ satisfies
  (a) $\kappa(p) = \chi(p) \cap \chi(V(M) \setminus V(M_p)) \subseteq \chi(q)$, where M_p is the subtree of M rooted at p; and
  (b) $\kappa(p) \not\subseteq \chi(s)$ for all $s \in V(M) \setminus V(M_q)$ if $\kappa(p) \neq \chi(q)$.
  For the root node $r(M)$, we set $\kappa(r(M)) = \emptyset$.

(C_5) **Minor nodes and uniqueness condition.** For every maximal subset of nodes $S = \{p_1, \ldots, p_n\} \subseteq V(M)$ $(n \geq 2)$ such that $\kappa(p_1) = \cdots = \kappa(p_n) = \kappa$ for some value of κ, there exists one unique node $m \in V(M)$ with $\lambda(m) = \emptyset$ and $\chi(m) = \kappa$.

In this paper, we consider acyclic queries and their "minimal-width" ("width-1") meta-decompositions, which satisfy

(C_3'') For all $p \in V(M)$ such that $\lambda(p) \neq \emptyset$, we have $|\lambda(p)| = 1$ and, w.l.o.g, $\chi(p) = V(\lambda(p))$.

Besides introducing minor nodes in condition (C_5) and defining κ as the interface in condition (C_4)(a), we also include condition (C_4)(b) that limits the shape of the meta-decomposition M: There can be multiple nodes that can satisfy $\kappa(p) \subseteq \chi(q)$, while condition ($C_4$)(b) limits q to be the highest node with $\kappa(p) \subseteq \chi(q)$. We illustrate this with the following example:

Example 4.4. Consider the query

$$Q_{4.4} \leftarrow R_1[x_1, x_2, x_6] \bowtie R_2[x_1, x_2, x_3, x_7] \bowtie R_3[x_1, x_3, x_4, x_8] \bowtie R_4[x_1, x_4, x_9] \bowtie R_5[x_1, x_5]$$

Its hypergraph and meta-decomposition are shown in Fig. 6a and Fig. 6b, respectively. Fig. 6c and Fig. 6d show two valid join trees of this hypergraph. We highlight that R_5 could in fact be the child of any other node of the join tree, because its only interface with the rest of the hypergraph is $\{x_1\}$. For example, Fig. 6e is also a valid join tree. But, for the meta-decomposition in Fig. 6b, according to condition (C_4)(b), we make R_5 the child of the highest possible node that contains x_1, which in this case is the root node, i.e., the minor node with χ-label $\{x_1, x_3\}$.

4.1 Constructing Meta-Decompositions

Similar to the original GYO algorithm [25, 56], a meta-decomposition can be constructed by a reduction-based algorithm, as shown in Algorithm 1. In this algorithm, $o(e, H) = e \cap (\cup(E(H) \setminus \{e\}))$

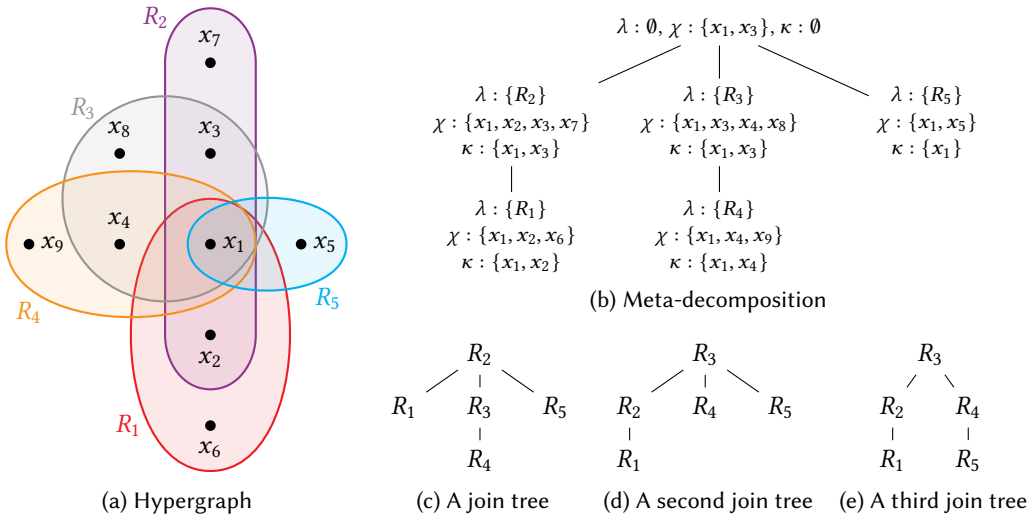


Fig. 6. Hypergraph, meta-decomposition, and three join trees of the query $Q_{4.4}$ in Example 4.4

denotes the set of overlapping vertices of hyperedge e and the remaining hyperedges $E(H) \setminus \{e\}$. As in the original GYO algorithm [25, 56], a hyperedge e is *reducible* (also called an *ear*), if there exists another single hyperedge $e' \in E(H) \setminus \{e\}$ such that $o(e, H) \subseteq e'$. The subsuming hyperedge e' is called the *witness*. And, exceptionally, if there is only one hyperedge e in the hypergraph, it is also considered an ear. We note that the GYO algorithm removes exactly one ear (among possibly many) in each round, thereby establishing a total order of all hyperedge removal. In contrast, Algorithm 1 *simultaneously removes all ears in every round* until only a single hyperedge remains in the hypergraph. This process introduces only a partial order for all hyperedges, which *maintains the flexibility to represent all possible ordering of ear removal, and hence all possible join trees*, through the resulting meta-decomposition.

Example 4.5. For the star query $Q_{2.3}$ in Example 2.3, the four hyperedges can be removed in any order in the standard GYO algorithm. For example:

(1) With the order R_4 (witness R_3) – R_3 (witness R_2) – R_2 (witness R_1) – R_1, we obtain the first join tree in Fig. 2b.
(2) With the order R_4 (witness R_3) – R_2 (witness R_1) – R_3 (witness R_1) – R_1, we obtain the second join tree in Fig. 2b.

Furthermore, we note that we can also obtain the second join tree by keeping the order in (1) but only changing witnesses, namely R_4 (witness R_3) – R_3 (witness R_1) – R_2 (witness R_1) – R_1.

To address this source of variation, in every round, our algorithm first handles such cases where some edges $e_1, \ldots, e_n$ are *mutually reducible*, i.e., $o(e_1, H) = \cdots = o(e_n, H) = o$ (Lines 4–8). (For example, in the case of $Q_{2.3}$, all pairs among R_1, R_2, R_3, and R_4 are mutually reducible.) The algorithm creates a minor node p with $\chi(p) = o$ if such a node does not yet exist, and then removes $e_1, \ldots, e_n$ from the hypergraph. We then add a special hyperedge $e_m = o$, which conceptually corresponds to the minor node, back to the hypergraph, so that the acyclicity of H is preserved. The rationale for adding such an e_m can be illustrated by the following example:

Example 4.6. Consider the query

$$Q_{4.6} \leftarrow \pi_{\emptyset}(R_1[x_1, x_2, x_3, x_4] \bowtie R_2[x_1, x_2, x_5] \bowtie R_3[x_1, x_3, x_6] \bowtie R_4[x_2, x_3, x_7] \bowtie R_5[x_1, x_2, x_3, x_8]),$$

Algorithm 1: Constructing the meta-decomposition of an acyclic hypergraph

input : An acyclic hypergraph H
output : The meta-decomposition M of H

1 $H' \leftarrow$ a copy of the input hypergraph H with $V(H') = V(H)$ and $E(H') = E(H)$
2 $V(M) \leftarrow \emptyset, E(M) \leftarrow \emptyset$
// Reduction, creating the vertices in the meta-decomposition at the same time
3 **while** $E(H')$ *is not empty* **do**
// First check if it is possible to create minor nodes for mutually reducible hyperedges
4 **foreach** *maximal* $S \subseteq E(H')$ *such that* $|S| > 1$, *and, for all* $e \in S$, $o(e, H')$ *has the same value* o **do**
5 **foreach** $e \in S$ **do**
6 Add a node p to $V(M)$ with $\lambda(p) = \{e\}$ if $e \in E(H)$, or $\lambda(p) = \emptyset$ if it is special, $\chi(p) = e$, and $\kappa(p) = o(e, H')$, if such p does not exist yet
7 Remove e from $E(H')$
8 Add a special hyperedge $e_m = o$ to $E(H')$
9 $\mathcal{E} \leftarrow$ the set of all ears in $E(H')$
10 **foreach** $e \in \mathcal{E}$ **do**
11 Add a new node p to $V(M)$ with $\lambda(p) = \{e\}$ if $e \in E(H)$, or $\lambda(p) = \emptyset$ if it is special, $\chi(p) = e$, and $\kappa(p) = o(e, H')$, if such p does not exist yet
12 Remove e from $E(H')$
13 **if** *there exists at least two vertices* $p' \in V(M) \cup \mathcal{E}$ *such that* $\kappa(p') = \kappa(p)$ **then**
14 Add a (minor) node v to $V(M)$ with $\lambda(v) = \emptyset$, $\chi(v) = \kappa(p)$, and $\kappa(v) = \kappa(p)$, if such v does not exist yet
15 $r(M) \leftarrow$ the node $p \in V(M)$ such that $\kappa(p) = \emptyset$
// Generate the edges in the meta-decomposition
16 **foreach** *non-root node* $p \in V(M)$ **do**
17 **if** *there exists a (minor) node* $q \in V(M)$ *such that* $\lambda(q) = \emptyset$ *and* $\chi(q) = \kappa(p)$ **then**
18 Add an edge (q, p) in $E(M)$
19 **else**
20 Find the node $q \in V(M)$ such that $\kappa(p) \subseteq \chi(q)$ and $\kappa(p) \nsubseteq \kappa(q)$ // "highest" possible parent
21 Add an edge (q, p) in $E(M)$
22 **return** $M = (V(M), r(M), E(M), \lambda, \chi, \kappa)$

which is similar to Example 1.1 but contains an additional relation R_5. In this case, at the beginning, R_1 and R_5 are both ears and mutually reducible, as $o(R_1, H) = o(R_5, H) = \{x_1, x_2, x_3\}$. However, if we simply remove both R_1 and R_5 at the same time, *the remaining hyperedges, R_2, R_3, and R_4, form a cyclic hypergraph.* But, if we add $e_m = o(R_1, H) = o(R_5, H) = \{x_1, x_2, x_3\}$, then e_m, R_2, R_3, and R_4 still form an acyclic query, and the algorithm can proceed to pick R_2, R_3, and R_4 as ears, before e_m is picked and removed at the very end.

We formalize this idea as follows:

Proposition 4.7. *Given an acyclic hypergraph H, if there exists a set of distinct ears $S = \{e_1, \ldots, e_n\} \subseteq E(H)$ $(n \geq 2)$ such that $o(e_1, H) = \cdots = o(e_n, H) = o$, then, the hypergraph H', with $E(H') = (E(H) \setminus S) \cup \{o\}$ and $V(H') = \cup E(H')$, is acyclic.*

In each round, the algorithm then finds and removes the set $\mathcal{E}$ of all ears of H'. For each $e \in \mathcal{E}$, we create a node p with set $\kappa(p) = o(e, H')$. p may be either physical or minor, depending on whether e is a hyperedge in H or a special hyperedge created while removing mutually reducible hyperedges. During the procedure, the algorithm creates minor nodes whenever necessary (Lines 6, 11, and 13–14).

This algorithm has the following guarantee of asymptotic complexity:

Theorem 4.8. *Algorithm 1 terminates in time* $O(|E(H)|^3)$.

Proof sketch. The key observations are (1) that the size $|E(H')|$ decreases by at least one in each iteration of the while-loop (Line 3), and (2) that, in each iteration of the while loop, all sets S (Line 4) can be enumerated in $O(|E(H')|) = O(|E(H)|)$ time, if we maintain a hash map that maps each $o \subseteq V(H')$ to all hyperedges $e \in E(H')$ such that $o(e, H') = o$, which is updated whenever a hyperedge e is added or removed from $E(H')$. □

We show the correctness of the algorithm:

Theorem 4.9. *Given an acyclic query with an acyclic hypergraph H, Algorithm 1 returns a valid meta-decomposition of H.*

Finally, we note that meta-decompositions are highly space-efficient. They only require space linear to the size of the hypergraph.

Theorem 4.10. *In the meta-decomposition M of an acyclic hypergraph H as constructed by Algorithm 1, the number of vertices $|V(M)|$ and the number of edges $|E(M)|$ are $O(|E(H)|)$.*

Proof sketch. Vertices in M are either physical or minor. We create one physical node per hyperedge in $E(H)$. For minor nodes, we note that each minor node has fan-out at least 2, meaning there must be at least two edges incident to each minor node. □

4.2 Join Tree Enumeration

The meta-decomposition allows one to efficiently enumerate all possible join trees of an acyclic query, as we show with an algorithm in this section. This not only serves as proof of the universality of this representation but also provides a powerful approach to enumerating join trees, which could be of great help to other structure-based methods that require doing so.

As an intuition, we can observe that the distinct join trees of the same hypergraph may differ in the following ways:

(1) **"Re-branching".** If a vertex p in a join tree has children c_1 and c_2 such that $\chi(c_1) \cap \chi(p) \subseteq \chi(c_2)$, then c_1 could have been a child of c_2 without violating any condition. In the meta-decomposition, condition (C_4) ensures that c_1 is positioned at the highest possible node. When enumerating join trees, we can make c_1 a child of c_2 if $\kappa(c_1) \subseteq \chi(c_2)$, without violating any condition. The join tree in Fig. 6e based on the meta-decomposition in Fig. 6b from Example 4.4 demonstrates such a case, where R_5 could be a child of any one of R_1, R_2, R_3, and R_4.
(2) **Rerooting.** A rerooting of a join tree is still a valid join tree.
(3) **Mutually reducible relations.** If multiple relations share some exact same set of attributes, they can be connected in arbitrary tree structures. This situation is represented by minor nodes in meta-decompositions. An example is the root node with χ-label $\{x_1\}$ in Fig. 4 for Example 2.3, based upon which we should be able to enumerate, among others, the join trees in Fig. 2b.

These alternatives should be considered to enumerate all join trees, and the meta-decomposition retains all the necessary information to facilitate this. Algorithm 2 is an algorithm to enumerate all join trees. For each node in the meta-decomposition starting from the root, it recursively enumerates all possible join trees given by the subtree rooted at each of its children (Lines 3–4), and then enumerates all possible ways to combine these join trees into one. For minor nodes r, we find its neighbors that share the common set of attributes given by $\chi(r)$ (Line 6) and generate all possible distinct structures of join trees, where each vertex will be replaced by the join tree for a subtree given by a neighbor. (At this point, we consider all rerootings of the same tree as identical.) At the

very end of this algorithm, we return all combined join trees we obtain, and it remains only to collect all rerootings. Due to space constraints, we omit some lower-level details, e.g., the exact steps for combining sub-join trees and algorithms to enumerate all trees and rerootings without repetition. The full details are given in the full version of the paper [30].

Algorithm 2: Enumerating all join trees based on a meta-decomposition

input : A meta-decomposition $M = (V(M), r(M), E(M), \lambda, \chi, \kappa)$
output : All join trees of the hypergraph induced by M

1 **Function** enumRec(v: *a node on* M)**:**
2 $C \leftarrow$ set of children c of v on M, sorted by partial order $\supseteq$ on $\kappa(c)$
3 **foreach** $c \in C$ **do**
4 $\mathcal{T}_c \leftarrow$ all rerootings T_c' of all trees T_c returned by enumRec(c) such that $\kappa(c) \subseteq \chi(r(T_c'))$
5 **if** $\lambda(v) = \emptyset$ **then** // minor node
6 $C_{\text{origin}} \leftarrow \{c \in C : \kappa(c) = \chi(v)\}$ // They can be found at the head of the sorted C
7 **if** $\kappa(v) = \chi(v)$ **then**
8 $\mathcal{T} \leftarrow$ all non-isomorphic trees with vertices $C_{\text{origin}} \cup \{v\}$, all rerooted to v
// The minor node v will eventually be replaced on Line 18 by the parent of r
9 **else** // $\kappa(v) \neq \chi(v)$, the parent should not participate
10 $\mathcal{T} \leftarrow$ all non-isomorphic trees with vertices C_{origin}
11 $\mathcal{T} \leftarrow$ set of trees T' that can be obtained from some $T \in \mathcal{T}$, where each $c \in V(T)$, in bottom-up order, is replaced by some $T_c \in \mathcal{T}_c$, and, for each non-root $c \in V(T)$ with parent p, T_c could be attached as a child of any $u \in V(T_p)$ such that $\kappa(c) \subseteq \chi(u)$
12 $C_{\text{proper}} \leftarrow C \setminus C_{\text{origin}}$, preserving the order // The set of children c such that $\kappa(c) \subsetneq \chi(v)$
13 **else**
14 $\mathcal{T} \leftarrow$ a set of one tree with one vertex v only
15 $C_{\text{proper}} \leftarrow C$, preserving the order
16 **foreach** *child* $c \in C_{\text{proper}}$ *in order,* $T_c \in \mathcal{T}_c$ **do**
17 **if** $\lambda(c) = \emptyset$ *and* $\kappa(c) = \chi(c)$ **then**
// The root of T_c would be a minor node from Line 8. We find a parent of each of its children.
18 $\mathcal{T} \leftarrow$ set of trees T' that can be obtained from some $T \in \mathcal{T}$, where for each child d of $r(T_c)$, we attach the subtree rooted at d as a child of some $u \in V(T)$ such that $\kappa(d) \subseteq \chi(u)$
19 **else**
20 $\mathcal{T} \leftarrow$ set of trees T' that can be obtained from some $T \in \mathcal{T}$, where T_c is attached as a child of some $u \in V(T)$ such that $\kappa(c) \subseteq \chi(u)$
21 **return** $\mathcal{T}$
22 **return** *all rerootings of all trees returned by* enumRec($r(M)$)

Example 4.11. Consider again the query $Q_{4.4}$ in Example 4.4. Its meta-decomposition is shown in Fig. 6b. Let the root node be r, and we use $v_1, \ldots v_5$ to denote the nodes such that $\lambda(v_i) = \{R_i\}$. The root node r is a minor node, with $\chi(r) = \{x_1, x_3\}$. We start by enumerating all possible join trees induced by the subtrees of the meta-decomposition, rooted at each child of r. For the subtree with v_2 and v_1, there is only one valid join tree, rooted at v_2, which has one child v_1. It is not possible to reroot to v_1 because $\kappa(v_2) = \{x_1, x_3\} \not\subseteq \chi(v_1)$. The case is similar for the subtree with v_3 and v_4. And the subtree with v_5 induces only one join tree with one node v_5. Since v_2 and v_3 (and not v_5) have $\kappa(v_2) = \kappa(v_3) = \chi(r) = \{x_1, x_3\}$, we enumerate all possible skeleton tree structures with vertices v_2 and v_3. There are only two such trees, each of which contains one edge connecting v_2 and v_3, and they are rerootings of each other. For each tree, we replace v_2 and v_3 by their induced join tree. In the end, we have one child v_5 with $\kappa(v_5) \subsetneq \chi(r)$. It can be a child of any node v such

that $\kappa(v_5) \subseteq \chi(v)$. In this example, it can, in fact, be a child of any other node. Fig. 6c is a join tree where the skeleton tree structure is rooted at R_2, and R_5 is attached as a child of R_2. Fig. 6d is a similar example where the skeleton is rooted at R_3, and R_5 is attached as a child of R_3. And Fig. 6e is based on the same skeleton as for Fig. 6d, but R_5 is attached as a child of R_4 instead of R_3.

We claim that this algorithm is both correct (producing valid join trees) and complete (enumerating all possible join trees).

Theorem 4.12 (Correctness). *Given a meta-decomposition M of an acyclic hypergraph H, all trees T output by Algorithm 2 are valid join trees of H.*

Theorem 4.13 (Completeness). *Given a meta-decomposition M of an acyclic hypergraph H, Algorithm 2 enumerates all possible join trees of H.*

As is detailed in the proof of Theorem 4.13, for each join tree, we can pinpoint exactly one way in which it is enumerated by the algorithm based on the meta-decomposition. Noting that the number of children of each node on a meta-decomposition is bounded by the fan-out, we get the following guarantee of enumeration delay.

Theorem 4.14 (Time complexity). *Given a meta-decomposition M of an acyclic hypergraph H, Algorithm 2 enumerates all possible join trees of H in an amortized delay of $O(f(M) \cdot |V(H)|)$.*

5 Cost-Based Optimization from Meta-Decompositions

Even though we have presented an algorithm, enumerating all possible join trees is infeasible, as their number grows super-exponentially with the number of relations, as shown by Theorem 2.2. Instead, we propose a cost-based optimization algorithm that *operates directly on the meta-decomposition* to find an optimal width-1 query plan. Similar to the case for join tree enumeration discussed in Section 4.2, the search space for such plans is defined by four key factors: (1) selecting a root of the join tree to induce the width-1 query plan; (2) unnesting minor nodes; (3) handling possible re-branching, and (4) determining the local join order for each node and its neighbors. In this section, we detail our procedure to address them.

5.1 Re-Branching

As previously discussed, re-branching can happen when a child node c of a node p in the meta-decomposition could also be a child of another descendant node d of p in a join tree, if $\kappa(c) \subsetneq \kappa(d)$. This scenario, however, is in fact very rare in practice: we found no query in the JOB benchmark that exhibited this scenario, and only one query in the TPC-H benchmark can involve re-branching. Therefore, for simplicity, we do not consider re-branching when describing the algorithm in this section. As we show in the full version of the paper [30], accommodating a possible re-branching adds only a constant factor to the overall complexity.

5.2 The Overall Algorithm

Given a meta-decomposition M, our algorithm to produce a width-1 query plan is shown in Algorithm 3. This algorithm first makes two traversals of M, one bottom-up and one top-down.

In the bottom-up traversal, for each non-root node q, we find the optimal plan to join the relation in q and all the plans given by the children of q, which gives the optimal plan to evaluate $Q(T_q)$, the query induced by the subtree of M rooted at q.

Example 5.1. For the query $Q_{3.1}$ in Example 3.1, with meta-decomposition shown in Fig. 5, this bottom-up traversal visits the nodes in the following order. (For simplicity, we simply use R_i to refer to the node with λ-label $\{R_i\}$.)

Algorithm 3: Finding the optimal width-1 query plan to evaluate the query induced by a meta-decomposition

input : A meta-decomposition $M = (V(M), r(M), E(M), \lambda, \chi, \kappa)$
output : The optimal query plan
// Bottom-up traversal
foreach *non-root node* $q \in V(M)$, *in bottom-up order* **do**
  $p \leftarrow q$'s parent node on M
  $\mathsf{plan}(T_{p\to q}) \leftarrow \mathsf{optimizeLocal}(q, \{\mathsf{plan}(T_{q\to c}) : c \in \mathsf{children}(q)\})$
$\mathsf{plan}(M) \leftarrow \mathsf{optimizeLocal}(r, \{\mathsf{plan}(T_{r\to c}) : c \in \mathsf{children}(r)\})$
// Top-down traversal
foreach $c \in \mathsf{children}(r)$ **do**
  $\mathsf{plan}(T_{c\to r}) \leftarrow \mathsf{optimizeLocal}(r, \{\mathsf{plan}(T_{r\to c'}) : c' \in \mathsf{children}(r) \setminus \{c\}\})$
foreach *non-root node* $q \in V(M)$, *in top-down order* **do**
  $p \leftarrow q$'s parent node on M
  foreach *child* $c \in C(q)$ **do**
    $\mathsf{plan}(T_{c\to q}) \leftarrow \mathsf{optimizeLocal}(q, \{\mathsf{plan}(T_{q\to p})\} \cup \{\mathsf{plan}(T_{q\to c'}) : c' \in \mathsf{children}(r) \setminus \{c\}\})$
return *the plan* $\mathsf{plan}(T_{p\to q}) \bowtie \mathsf{plan}(T_{q\to p})$ *with the minimum cost among all* $(p, q) \in E(M)$

(1) R_3. It generates the plan for the subtree T_{R_3}, or $T_{R_2\to R_3}$, containing R_3 only. The plan is simply a scan of R_3.
(2) R_2. It generates the plan for T_{R_2}, or $T_{R_1\to R_2}$, containing R_2 and R_3. It simply needs to join R_2 with the plan for T_{R_3} in the previous step, i.e., $R_2 \bowtie R_3$.
(3) R_4. It generates the plan for the subtree T_{R_4}, or $T_{R_1\to R_4}$, containing R_4 only. The plan is simply a scan of R_4.
(4) R_1. This is the root node. The traversal terminates here.

Then, in the top-down traversal, for each node q and each child c of q on the meta-decomposition, we find the optimal plan to join q and all plans given by the neighbors (parent and children) of q except c, which gives the optimal plan to join all relations in $T_{c\to q}$.

Example 5.2. Again for $Q_{3.1}$, the top-down traversal visits the nodes in the following order.

(1) $R_1 \to$ child R_2. It generates the plan for the subtree $T_{R_2\to R_1}$, containing R_1 and R_4. The plan is simply $R_1 \bowtie R_4$.
(2) $R_2 \to$ child R_3. It generates the plan for the subtree $T_{R_3\to R_2}$, containing R_1, R_2, and R_4. Similarly, we join R_2 with the result of $R_1 \bowtie R_4$, which is the plan for the subtree $T_{R_2\to R_1}$.
(3) $R_1 \to$ child R_4. It generates the plan for the subtree $T_{R_4\to R_1}$, containing R_1, R_2, and R_3. Here, R_2 and R_3 are in the same subtree $T_{R_1\to R_2}$ of R_1. So, we first take the plan generated for this subtree, which is $R_2 \bowtie R_3$, and then join this result with R_1.

At the end, we choose the root of the query plan, which is the minimum-cost plan of the form $\mathsf{plan}(T_{p\to q}) \bowtie \mathsf{plan}(T_{q\to p})$ among all $(p, q) \in E(M)$. For example, the width-1 plan shown in Fig. 3c corresponds to $\mathsf{plan}(T_{R_2\to R_1}) \bowtie \mathsf{plan}(T_{R_1\to R_2})$.

In the example we just considered, the fan-out is 2. For higher fan-outs, at each node in the meta-decomposition, we need to decide on an order to join the results of multiple subtrees. To do so, the algorithm calls the function optimizeLocal, which optimizes the local join for each node and its neighbors. We will elaborate on this in Section 5.3.

5.3 Optimizing the Local Join with Neighbors

One essential component of this algorithm is to, for a vertex q in the meta-decomposition, find the optimal plan to join the relation in this vertex and the optimal query plans given by (a subset of) its

neighbors (optimizeLocal). This can be viewed as a classic star join optimization problem, where R_q is the hub relation, and the result of the induced query $Q(T_{c_i})$ from each of its neighbors c_i is a satellite relation, denoted by R_i for $i = 1, \ldots, f(q)$, where $\chi(c_i) \cap \chi(q) \neq \emptyset$ and $\chi(c_i) \cap \chi(c_j) \subseteq \chi(q)$ for all $i \neq j$. Unfortunately, even in this restricted setting, this problem of optimizing the join of R_q and all R_i's is still NP-hard, because this can be reduced from the problem of optimizing the ordering of a Cartesian product $S_1 \times S_2 \times \cdots \times S_n$—which is already shown to be NP-hard [44]—by adding a shared attribute to each S_i such that all tuples have the same value for this attribute. By using dynamic programming algorithms [39, 46], the problem can be solved in $\widetilde{O}(2^{f(q)})$, where $\widetilde{O}$ suppresses poly-logarithmic factors and the fan-out $f(q)$ is the number of children of q on the meta-decomposition. However, as mentioned in Section 3, *the fan-out of real-world queries is typically very small*. This makes the exponential-time exact algorithm practical for the vast majority of cases. In the rare case that a node has a high fan-out, making the exact DP algorithm too slow, we can substitute it with a heuristic. An effective choice is Greedy Operator Ordering (GOO) [17], which iteratively joins the neighbor that results in the smallest intermediate result. This method is outlined in the full version of the paper [30].

5.4 Case Study: Optimizing Relation-Dominated Queries with Meta-Decompositions

Our framework naturally incorporates projection pushdown for early cardinality reduction. When building a plan for a subquery, such as $\mathsf{plan}(T_{p \to q})$ in Algorithm 3, we can immediately project the result to only the attributes needed for subsequent operations. Specifically, after computing the joins within the subtree T_q, we only need to retain (1) attributes that are part of the final output, and (2) attributes required for joining with the remaining relations, which is $\kappa(q)$ by definition.

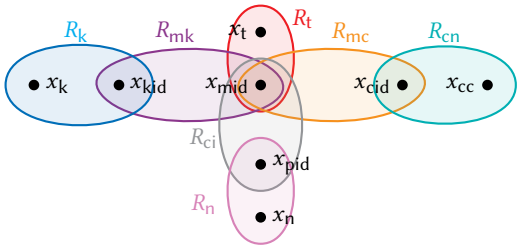
This strategy is especially effective for relation-dominated queries, where a single relation contains all output attributes. If a subtree contains no output attributes, we can project its result to only $\kappa(q)$, i.e., the join keys with the remaining relations, which can significantly reduce the size of intermediate results. However, traditional dynamic-programming-based optimizers lack such a comprehensive structural view. Therefore, they typically do not identify such opportunities and must consider the entire search space of query plans, relying solely on cost estimation.

Example 5.3. Take query 17f in the join order benchmark (JOB) as an example. Omitting irrelevant attributes in each relation, this query can be abstractly represented as

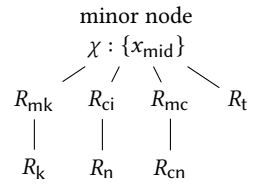
$$Q_{5.3} \leftarrow \pi_{x_{\mathsf{n}}}(R_{\mathsf{ci}}[x_{\mathsf{mid}}, x_{\mathsf{pid}}] \bowtie R_{\mathsf{cn}}[x_{\mathsf{cid}}, x_{\mathsf{cc}}] \bowtie \sigma_{x_{\mathsf{k}}=\cdots}(R_{\mathsf{k}}[x_{\mathsf{kid}}, x_{\mathsf{k}}]) \bowtie R_{\mathsf{mc}}[x_{\mathsf{mid}}, x_{\mathsf{cid}}] \\ \bowtie R_{\mathsf{mk}}[x_{\mathsf{mid}}, x_{\mathsf{kid}}] \bowtie \sigma_{x_{\mathsf{n}}=\cdots}(R_{\mathsf{n}}[x_{\mathsf{pid}}, x_{\mathsf{n}}]) \bowtie R_{\mathsf{t}}[x_{\mathsf{mid}}, x_{\mathsf{t}}]).$$

This is a relation-dominated query, as it has only one output attribute x_{n}, which is an attribute of R_{n}. Fig. 7 shows the hypergraph, the optimal width-1 query plan (found by metaDecomp given cardinalities of the real data set), along with the join tree inducing it, and a width-2 query plan (which is the plan given by dynamic programming). In the width-1 plan in Fig. 7d, *all intermediate join results can be projected to a single attribute*, since the interface is always only one attribute, and there is no output attribute before the final join with R_{n}. But, with the width-2 query plan in Fig. 7e, *the result of* $((R_{\mathsf{mk}} \bowtie R_{\mathsf{k}}) \bowtie R_{\mathsf{t}}) \bowtie R_{\mathsf{ci}}$ *has to be projected to two attributes* x_{mid} *and* x_{pid}, which are join keys with R_{mc} and R_{n}, respectively. Executing these plans, we indeed observe that *the width-1 plan has a* 1.72x *speedup over the width-2 plan.*

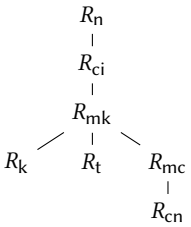
As with evaluating select-project-join queries, one can easily modify the cost model and use a join tree–based query evaluation algorithm, including any Yannakakis-style algorithm, that incorporates cost-based optimization via meta-decompositions. We hope this work inspires further research in related areas and encourages the integration of these theoretically desirable query evaluation techniques into practical systems.



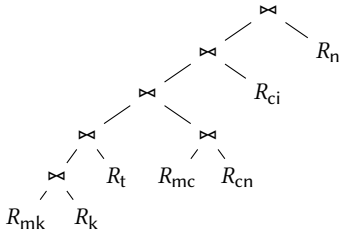
(a) Hypergraph



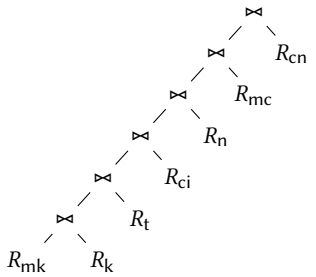
(b) Meta-decomposition (simplified)



(c) Join tree inducing the width-1 plan in Fig. 7d



(d) Optimal width-1 query plan, induced by the join tree in Fig. 7c



(e) Optimal plan by dynamic programming, of width 2

Fig. 7. The hypergraph, the meta-decomposition, the optimal width-1 query plan with the inducing join tree, and the optimal query plan (of width 2) found by dynamic programming, of the query JOB 17f in Example 5.3

6 Experimental Evaluation and Analysis

In this section, we present experimental results to answer the following research questions:

(RQ1) How fast is query optimization using meta-decompositions?
(RQ2) How good are width-1 query plans, compared to the globally optimal ones?
(RQ3) To what extent is our proposed meta-decompositions-based query optimization sensitive to errors in cardinality estimation?
(RQ4) To what extent can our proposed meta-decompositions-based join tree enumeration technique help structure-based query evaluation?

6.1 Experimental Setup

6.1.1 Experimental Environment. Experiments are conducted on an Apple M4 Pro machine with 14 cores and 48 GB RAM, running macOS Tahoe 16.0. We use DuckDB as our test database because of its excellent performance and simple cost model. The experiments use DuckDB 1.2.2, Scala 3.6.3, GraalVM 17.0.12, and LLVM 20.1.3. The full source code for experiments is available at [29].

We implement metaDecomp in Scala. For each query, it first computes the meta-decomposition (Section 4) and then applies our cost-based optimization algorithm (Section 5) to select the best width-1 query plan. In the experiments, we compare the following query optimization approaches:

(1) metaDecomp, using our implementation.
(2) DPconv [46], the state-of-the-art dynamic programming optimizer.
(3) DuckDB's native optimizer [42], which uses DPccp [39] for queries with up to 12 relations and Greedy Operator Ordering (GOO) [17] otherwise.
(4) UnionDP, a heuristic approach proposed in [37, Algorithm 4] which (i) partitions the query graph into subgraphs, each with size up to a threshold (which we set to 12, consistent with DuckDB),

Benchmarks →		DSB	JOB	Musicbrainz	JOBLarge	Overall
Number of relations	Min	5	4	2	8	2
	Median	6	8	10	18	8
	Max	9	17	26	34	34
Fan-out	Min	2	2	1	2	1
	Median	3	3	4	4	3
	Max	4	5	9	5	9
Number of join trees	Min	5	12	2	72	2
	Median	7	144	22500	40000	126
	Max	54	352512	$> 10^8$	$> 10^8$	$> 10^8$

Table 2. Structural properties of queries in each benchmark

(ii) runs dynamic programming on each subgraph, and (iii) runs dynamic programming on the entire graph, treating each subgraph as a composite node.

(5) Yannakakis$^+$ [50], a state-of-the-art structure-guided query evaluation approach.
(6) LearnedRewrite [61, 62], a machine learning–based query rewriter.
(7) LLM-R^2 [35], a query rewriter based on machine learning and large language models (LLMs).

For Yannakakis$^+$, we use the implemented system as a black box and run the rewritten queries in DuckDB. For LearnedRewrite and LLM-R^2, we similarly evaluate them using their pre-trained models as-is and run the rewritten queries in DuckDB. For metaDecomp, DPconv, and UnionDP, the query plans are enforced in DuckDB by rewriting queries into a sequence of subqueries that create a temporary view at each step, as is standard in, e.g., [20, 50]. For all three methods, we apply the same strategies of selection and projection pushdown as explained in Section 2.3. Namely, we project out all irrelevant attributes (those that are neither in the output nor in subsequent joins) at every step, and we integrate each selection (filter) condition in the join or scan operation at the lowest possible level, i.e., as soon as all relevant attributes exist.

Each query plan is executed 10 times, and we report the median optimization and execution time. To ensure accuracy, we exclude I/O time from time measurements. The full dataset is preloaded into the DuckDB database, and the estimated cardinalities are preloaded into memory, with all loading time excluded from timing. We enforce a 5-minute timeout on execution time. Queries that none of the optimizers can complete within this time limit are excluded from the results.

6.1.2 Queries, Benchmarks, and Cardinalities. We experiment with the following four benchmarks:

(1) The SPJ (select-project-join) queries in the **Decision Support Benchmark (DSB)** [16]. We use the implemented system to generate a 10 GB database instance and generate queries using the default settings. We only consider acyclic queries (240 out of 300).
(2) The **Join Order Benchmark (JOB)** [34] over the IMDB dataset. All queries in this benchmark are already acyclic.
(3) **Musicbrainz**, a benchmark proposed by [37] with significantly larger queries that join up to 26 relations. We use the implemented query generator with default settings, except that we project the output to at most 5 randomly-selected attributes. We retain only the acyclic queries (243 out of 375).
(4) **JOBLarge**, a new benchmark we introduce to extend JOB to simulate practical but larger analytical scenarios, where each query is generated by selecting two queries from the JOB that share at least one common output attribute and merging them into a single query, with a join on a shared attribute, such that the query remains acyclic. We only retain generated queries that take DuckDB at least 50 ms to evaluate.

In Table 2, we show detailed statistics of some structural properties of queries in each benchmark.

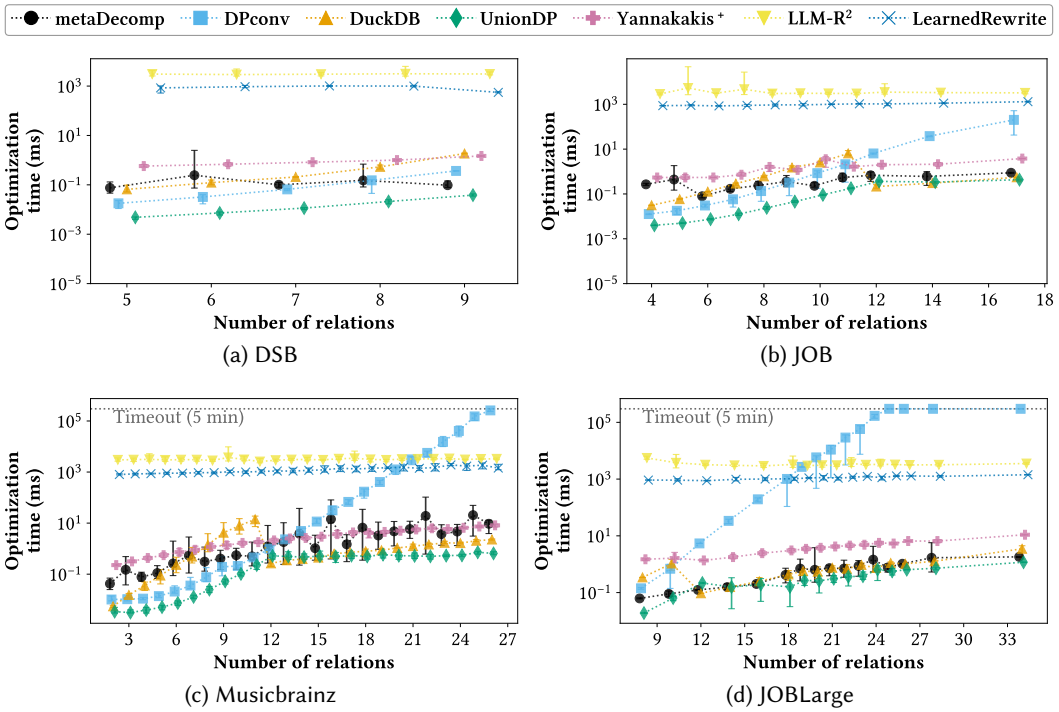


Fig. 8. Optimization time for queries in each benchmark. For each number of relations, the dot shows the average, and the whiskers show the minimum and maximum optimization time.

For metaDecomp, DPconv, and UnionDP, which all follow the same cost model, we precompute (by running appropriate `COUNT(*)` queries) true cardinalities of all possible intermediate join results for the relatively small benchmarks DSB and JOB, in the approach and format consistent with the implementation of DPconv [46]. For the larger benchmarks Musicbrainz and JOBLarge, computing exact cardinalities for all possible intermediate joins is infeasible due to the much larger query sizes. Therefore, we use estimated cardinalities instead, via DuckDB's `EXPLAIN` command. Note, however, that these estimates may sometimes be highly inaccurate and may not reflect the true relative sizes of intermediate results. DuckDB, Yannakakis$^+$, Learned Rewrite, and LLM-R^2 are used as black boxes with their own cost models.

6.2 Results

In this section, we present the results of our experiments.

6.2.1 (RQ1) Optimization Time. We first compare the optimization time. The results are shown in Fig. 8. The figures clearly show that the optimization time of metaDecomp remains almost constant as the number of relations increases, typically below 10 ms, as is also the case for DuckDB, UnionDP, and Yannakakis$^+$. The two machine learning–based approaches, LLM-R^2 and LearnedRewrite, exhibit stable but higher optimization time, in the order of seconds, due to the overhead of the complex models and, for LLM-R^2, the interaction with LLMs. DPconv, however, has an exponential growth, and its optimization time surpasses that of metaDecomp starting at approximately 9 relations. For many queries in Musicbrainz and JOBLarge (starting from approximately 25 relations), DPconv fails to find a valid plan within the 5-minute limit, again highlighting that traditional exponential-time dynamic programming–based optimizers do not scale to such large queries.

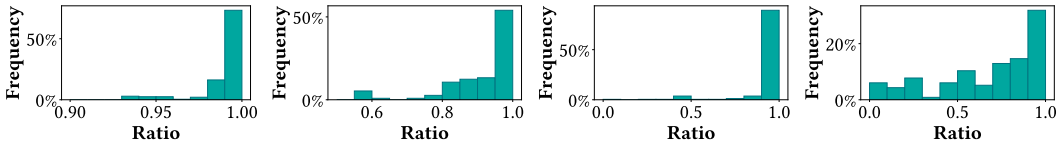


Fig. 9. Ratios between the costs of the globally optimal query plans and the costs of the optimal width-1 plans. A ratio of 1 indicates they have the same cost, and is included in the bar between 0.9 and 1.0.[3]

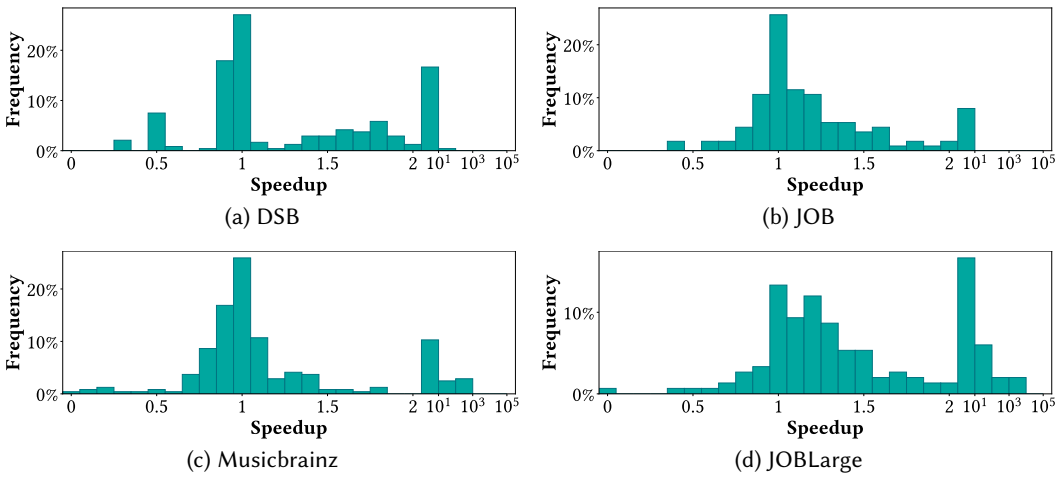


Fig. 10. Histograms of execution speedups of the optimal width-1 plans over the globally optimal plans

6.2.2 (RQ2) Quality of width-1 query plans. Even though metaDecomp restricts its search to width-1 plans only, as shown in Fig. 9, for most queries, it *finds query plans with cost values comparable to those of the optimal plans*. Furthermore, Fig. 10 shows the speedups achieved by executing the optimal width-1 query plans over executing the globally optimal query plans. Note that, even though one would typically not expect speedups since the optimal plans have the minimum possible cost values, in many cases, the optimal width-1 query plans in fact run faster in the actual execution, thanks to the small intermediate result sizes through exploiting queries' structural properties.

6.2.3 (RQ1) & (RQ2) Combined: Overall Performance. Taking both optimization and execution time into account, we observe in Table 3, Fig. 11, and Fig. 12 that metaDecomp shows a clear advantage overall. This is especially the case for queries that are complex, as shown by the 95th and 99th percentiles in Table 3, and in the points towards the right in the scatter plots in Fig. 12. For DuckDB, UnionDP, Yannakakis$^+$, LearnedRewrite, and LLM-R^2, although their optimization is fast, they are also very likely to generate plans that are significantly worse than the optimal width-1 query plans, especially for larger queries. In addition, for Yannakakis$^+$, there is an overhead in the reduction phase, which may be costly for certain queries. For DPconv, the main disadvantage is the exponential optimization time, which becomes the dominant factor in the overall evaluation time for large queries. The overall results demonstrate that finding and executing width-1 plans with metaDecomp is indeed highly efficient in practice. More detailed, separate figures for individual benchmarks can be found in the full version of the paper [30].

[3]This measure is analogous to speedups. Since width-1 plans are a restricted class, their theoretical cost values cannot be better (lower) than the global optimal, and thus the ratios are at most 1.

Metric ↓	Benchmark → / Method ↓	DSB Mean	DSB Med.	DSB 95th	DSB 99th	JOB Mean	JOB Med.	JOB 95th	JOB 99th	Musicbrainz Mean	Musicbrainz Med.	Musicbrainz 95th	Musicbrainz 99th	JOBLarge Mean	JOBLarge Med.	JOBLarge 95th	JOBLarge 99th
Optimization time	metaDecomp	0.15 ms	0.10 ms	0.32 ms	1.01 ms	0.38 ms	0.26 ms	0.94 ms	1.04 ms	2.62 ms	0.37 ms	10.45 ms	45.05 ms	0.59 ms	0.37 ms	1.67 ms	4.17 ms
	DPconv	0.08 ms	0.03 ms	0.36 ms	0.37 ms	8.32 ms	0.16 ms	38.22 ms	45.08 ms	7.37 s	0.30 ms	14.62 s	244.44 s	42.17 s	1.26 s	> 5 min	> 5 min
	DuckDB	0.32 ms	0.12 ms	1.83 ms	1.87 ms	1.13 ms	0.32 ms	5.58 ms	8.43 ms	1.89 ms	0.47 ms	11.34 ms	18.59 ms	0.62 ms	0.55 ms	1.23 ms	3.81 ms
	UnionDP	0.01 ms	0.007 ms	0.04 ms	0.04 ms	0.10 ms	0.03 ms	0.38 ms	0.45 ms	0.25 ms	0.11 ms	0.64 ms	0.77 ms	0.29 ms	0.24 ms	0.77 ms	1.18 ms
	Yannakakis$^+$	0.78 ms	0.70 ms	1.48 ms	1.50 ms	1.44 ms	1.37 ms	3.78 ms	4.59 ms	2.29 ms	1.72 ms	6.43 ms	8.15 ms	3.55 ms	3.27 ms	6.53 ms	10.89 ms
	LearnedRew.	0.89 s	0.95 s	1.03 s	1.06 s	0.96 s	0.94 s	1.13 s	1.30 s	1.11 s	1.00 s	1.78 s	2.14 s	1.09 s	1.10 s	1.29 s	1.42 s
	LLM-R^2	3.03 s	2.92 s	3.66 s	4.71 s	3.75 s	2.99 s	3.98 s	24.96 s	3.11 s	3.03 s	3.57 s	6.10 s	3.40 s	3.11 s	5.72 s	6.84 s
Execution time	metaDecomp	0.01 s	0.01 s	0.04 s	0.04 s	0.04 s	0.03 s	0.08 s	0.11 s	1.08 s	0.52 s	2.56 s	13.75 s	1.02 s	0.17 s	2.31 s	23.84 s
	DPconv	0.02 s	0.01 s	0.09 s	0.10 s	0.04 s	0.04 s	0.09 s	0.11 s	1.42 s	0.50 s	2.65 s	28.27 s	17.50 s	0.24 s	89.05 s	> 5 min
	DuckDB	23.85 s	0.01 s	> 5 min	> 5 min	0.04 s	0.04 s	0.09 s	0.13 s	95.92 s	0.52 s	> 5 min	> 5 min	22.46 s	0.13 s	> 5 min	> 5 min
	UnionDP	0.03 s	0.01 s	0.10 s	0.11 s	0.05 s	0.04 s	0.10 s	0.12 s	58.75 s	0.28 s	> 5 min	> 5 min	25.04 s	0.15 s	> 5 min	> 5 min
	Yannakakis$^+$	0.03 s	0.02 s	0.08 s	0.08 s	0.06 s	0.06 s	0.12 s	0.15 s	13.63 s	0.41 s	34.61 s	> 5 min	5.11 s	0.13 s	1.52 s	232.60 s
	LearnedRew.	23.95 s	0.05 s	> 5 min	> 5 min	26.91 s	0.15 s	> 5 min	> 5 min	42.47 s	0.42 s	> 5 min	> 5 min	42.03 s	0.14 s	> 5 min	> 5 min
	LLM-R^2	23.79 s	0.01 s	> 5 min	> 5 min	0.04 s	0.04 s	0.10 s	0.14 s	96.32 s	0.53 s	> 5 min	> 5 min	29.94 s	0.12 s	> 5 min	> 5 min
Overall evaluation time	metaDecomp	0.01 s	0.01 s	0.04 s	0.04 s	0.04 s	0.03 s	0.08 s	0.11 s	1.09 s	0.52 s	2.65 s	13.75 s	1.02 s	0.17 s	2.31 s	23.84 s
	DPconv	0.02 s	0.01 s	0.09 s	0.10 s	0.05 s	0.04 s	0.10 s	0.12 s	8.78 s	0.51 s	26.36 s	246.63 s	58.26 s	4.32 s	> 5 min	> 5 min
	DuckDB	23.85 s	0.01 s	> 5 min	> 5 min	0.04 s	0.04 s	0.09 s	0.13 s	95.93 s	0.52 s	> 5 min	> 5 min	22.46 s	0.13 s	> 5 min	> 5 min
	UnionDP	0.03 s	0.01 s	0.10 s	0.11 s	0.05 s	0.04 s	0.10 s	0.12 s	58.75 s	0.28 s	> 5 min	> 5 min	25.04 s	0.15 s	> 5 min	> 5 min
	Yannakakis$^+$	0.03 s	0.02 s	0.08 s	0.08 s	0.06 s	0.06 s	0.13 s	0.15 s	13.63 s	0.41 s	34.61 s	> 5 min	5.11 s	0.13 s	1.52 s	232.60 s
	LearnedRew.	24.84 s	1.01 s	> 5 min	> 5 min	27.87 s	1.12 s	> 5 min	> 5 min	54.93 s	1.43 s	> 5 min	> 5 min	43.12 s	1.28 s	> 5 min	> 5 min
	LLM-R^2	26.82 s	3.03 s	> 5 min	> 5 min	3.79 s	3.05 s	4.01 s	24.99 s	99.43 s	4.06 s	> 5 min	> 5 min	33.34 s	3.41 s	> 5 min	> 5 min
Overall speedup of metaDecomp over ...	DPconv	1.22x	0.99x	2.90x	3.29x	1.18x	1.08x	2.77x	3.11x	1.34x	0.99x	15.49x	138.59x	27.61x	21.23x	2308.55x	5014.14x
	DuckDB	2.75x	0.97x	16938.94x	18945.33x	1.11x	1.07x	2.03x	3.53x	6.39x	1.86x	238.77x	901.19x	1.46x	0.82x	118.01x	1872.97x
	UnionDP	1.28x	1.03x	3.38x	6.57x	1.12x	1.04x	2.09x	3.01x	2.67x	0.94x	293.61x	844.52x	1.91x	0.94x	606.52x	2328.21x
	Yannakakis$^+$	1.83x	1.91x	2.77x	7.03x	1.61x	1.50x	3.75x	5.07x	1.07x	0.72x	47.66x	656.33x	0.76x	0.73x	2.13x	22.05x
	LearnedRew.	142.29x	123.70x	16993.51x	19006.30x	59.91x	38.45x	7958.89x	11031.32x	14.49x	11.44x	442.62x	967.06x	13.65x	8.85x	2444.33x	7240.85x
	LLM-R^2	379.88x	282.20x	17095.75x	19117.48x	103.68x	98.29x	484.18x	832.41x	65.93x	78.95x	927.52x	2488.64x	23.98x	21.60x	174.23x	3632.91x

Table 3. Statistics of query evaluation time. The "mean" values are arithmetic mean for time and geometric mean for speedup factors.

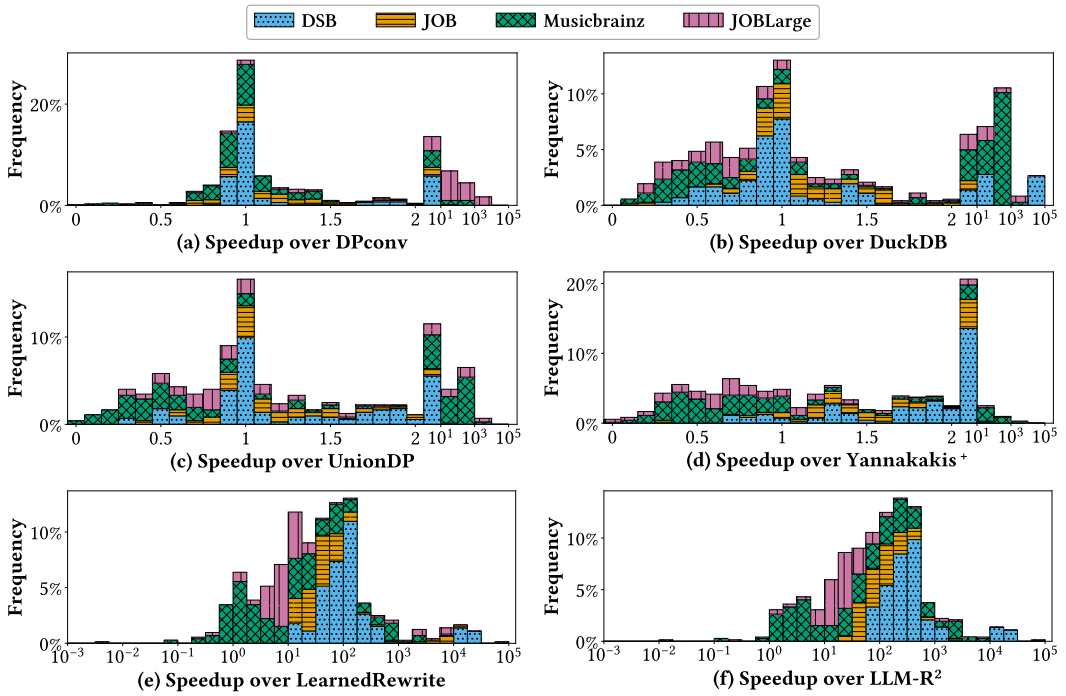


Fig. 11. Histograms of overall speedups of using metaDecomp over using other methods

6.2.4 (RQ3) Sensitivity to cardinality misestimation. Recent research has shown that structure-based query optimization tends to be more robust even in the presence of errors in cardinality estimation [8, 50, 60]. This is also the case for metaDecomp. To illustrate, we compare metaDecomp

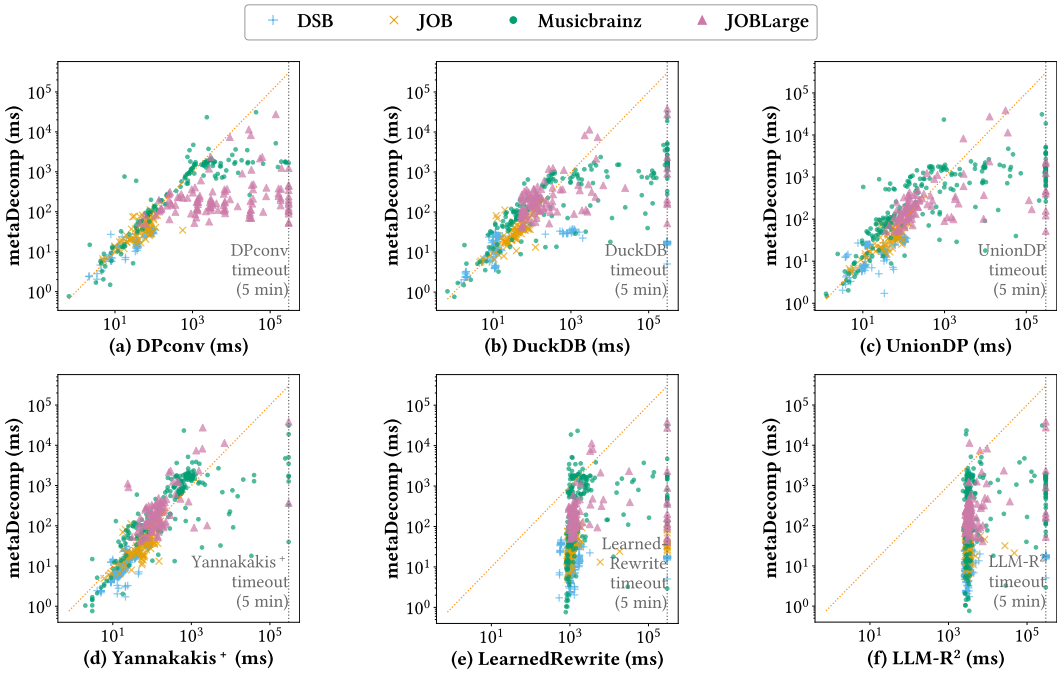


Fig. 12. Scatter plots of overall evaluation time using metaDecomp versus using other methods. Queries for which the difference is less than 10% are omitted in the figures.

	Benchmark →	DSB				JOB			
Method ↓	**Cardinality ↓**	Mean	Median	95th	99th	Mean	Median	95th	99th
metaDecomp	Exact	0.01 s	0.01 s	0.04 s	0.04 s	0.04 s	0.03 s	0.08 s	0.11 s
	Misestimated	0.02 s	0.01 s	0.06 s	0.12 s	0.05 s	0.04 s	0.11 s	0.15 s
DPconv	Exact	0.02 s	0.01 s	0.09 s	0.10 s	0.04 s	0.04 s	0.09 s	0.11 s
	Misestimated	0.03 s	0.01 s	0.11 s	0.14 s	0.46 s	0.05 s	0.23 s	2.07 s
UnionDP	Exact	0.03 s	0.01 s	0.10 s	0.11 s	0.05 s	0.04 s	0.10 s	0.12 s
	Misestimated	0.03 s	0.01 s	0.11 s	0.14 s	0.48 s	0.05 s	0.22 s	1.86 s

Table 4. Query execution time when optimized with exact vs misestimated cardinalities

with DPconv and UnionDP, which use the exact same cost model. We run them on DSB and JOB queries with (1) exact and (2) misestimated cardinalities. The misestimated cardinalities are generated by adding a multiplicative noise to the exact cardinalities using a log-normal distribution. Specifically, for each exact cardinality C, we generate a perturbed estimate $\hat{C} = C \cdot e^{\epsilon}$, where $\epsilon \sim \mathcal{N}(0, \sigma^2)$, i.e., following a normal (Gaussian) distribution, with standard deviation σ set to 10. From the results in Table 4, metaDecomp still has excellent performance despite such errors in cardinality estimation. The advantage is particularly pronounced at the 99th percentile of the JOB benchmark, where DPconv and UnionDP, which rely solely on cardinality values, are significantly misled by the highly inaccurate estimation.

6.2.5 (RQ4) Meta-decompositions applied to other structure-based optimization techniques. Finally, we demonstrate how meta-decompositions can help other recently developed structure-based optimization techniques. A crucial step in those techniques is selecting a good join tree, as it can significantly influence the structures and, consequently, the costs of the generated query plans.

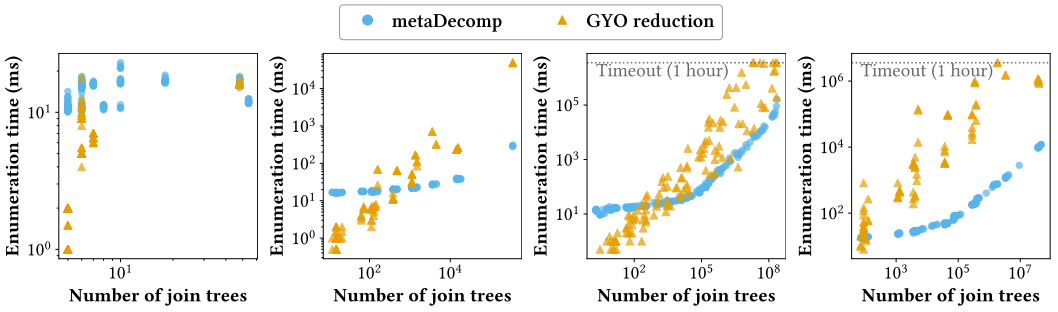


Fig. 13. Scatter plots of join tree enumeration time using metaDecomp versus using naïve GYO reduction

Thus, we first evaluate metaDecomp's efficiency in *enumerating join trees*, compared to the standard GYO reduction–based approach from [14]. As shown in Fig. 13, metaDecomp shows up to *four orders of magnitude of speedups* over the traditional approach. In particular, for many highly complex queries in the Musicbrainz and JOBLarge benchmarks with more than 10^7 join trees, the traditional approach already takes over 1 hour, whereas metaDecomp can complete the enumeration in 2 seconds to 1 minute.

A more effective approach is to apply cost-based optimization directly on meta-decompositions, as has already been done in metaDecomp, since this eliminates the need to enumerate exponentially large numbers of join trees. Therefore, in our second experiment, we use Yannakakis$^+$ [50] to rewrite queries based on join trees selected by metaDecomp that induce the optimal width-1 query plans. *This modification achieves an average speedup of* 1.11x *on JOB queries.* This demonstrates that structure-based evaluation algorithms, such as Yannakakis$^+$, can benefit significantly from our efficient cost-based optimization framework.

7 Conclusions and Future Work

In this paper, we have introduced novel techniques for query optimization based on structural properties via meta-decompositions. They offer promising opportunities to integrate theoretically desirable structure-based optimization into practical database systems to efficiently evaluate large, complex queries. A natural next step is to extend our width notion and meta-decompositions-based framework to cyclic queries. While width-1 query plans cannot exist for cyclic queries (according to Theorem 3.6) we may, e.g., identify minimal-width query plans and represent all minimal-width hypertree decompositions. In addition, to improve our current simple strategy of selection pushdown, it is possible to incorporate existing techniques for evaluating conjunctive queries with comparisons [52]. Similar is the case for many other operators that are so far excluded in this paper, such as aggregations [2, 31], top-k [51], differences [28, 59], etc., for which there already exist theoretically desirable structure-guided approaches. The performance of such methods can also benefit from our cost-based optimization framework. Finally, even though our technique has shown its robustness, it will still benefit from more accurate cardinality estimation. Recent research [11, 58] shows promising techniques that can be adopted to develop efficient, reliable cardinality estimators suitable within our proposed framework.

Acknowledgments

Qichen Wang is supported by the Ministry of Education, Singapore, through the Academic Research Fund Tier 1 grant (RS32/25), and by the Nanyang Technological University Startup Grant. Part of this work was completed while Qichen Wang was at EPFL.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1996. *Foundations of Databases.* Addison-Wesley, Reading, MA, USA.

[2] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (San Francisco, CA, USA) *(PODS '16)*. Association for Computing Machinery, New York, NY, USA, 13–28. https://doi.org/10.1145/2902251.2902280

[3] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2017. What Do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another?. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, IL, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 429–444. https://doi.org/10.1145/3034786.3056105

[4] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *Computer Science Logic*, Jacques Duparc and Thomas A. Henzinger (Eds.). Springer, Berlin, Heidelberg, 208–222. https://doi.org/10.1007/978-3-540-74915-8_18

[5] Liese Bekkers, Frank Neven, Stijn Vansummeren, and Yisu Remy Wang. 2025. Instance-optimal acyclic join processing without regret: Engineering the Yannakakis algorithm in column stores. *Proc. VLDB Endow.* 18, 8 (2025), 2413–2426. https://doi.org/10.14778/3742728.3742737

[6] Kristin Bennett, Michael C. Ferris, and Yannis E. Ioannidis. 1991. *A genetic algorithm for database query optimization.* Technical Report #1004. Center for Parallel Optimization. https://minds.wisconsin.edu/bitstream/handle/1793/59438/TR1004.pdf

[7] Philip A. Bernstein and Dah-Ming W. Chiu. 1981. Using Semi-Joins to Solve Relational Queries. *J. ACM* 28, 1 (Jan. 1981), 25–40. https://doi.org/10.1145/322234.322238

[8] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *Proc. VLDB Endow.* 17, 11 (July 2024), 3215–3228. https://doi.org/10.14778/3681954.3681995

[9] Altan Birler, Mihail Stoian, and Thomas Neumann. 2025. Optimizing Linearized Join Enumeration by Adapting to the Query Structure. *Datenbanksysteme für Business, Technologie und Web (BTW 2025)* (2025), 171–194. https://doi.org/10.18420/BTW2025-09

[10] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *VLDB J.* 29, 2-3 (2020), 655–679. https://doi.org/10.1007/S00778-019-00558-9

[11] Walter Cai, Magdalena Balazinska, and Dan Suciu. 2019. Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, New York, NY, USA, 18–35. https://doi.org/10.1145/3299869.3319894

[12] Jin Chen, Guanyu Ye, Yan Zhao, Shuncheng Liu, Liwei Deng, Xu Chen, Rui Zhou, and Kai Zheng. 2022. Efficient Join Order Selection Learning with Graph-based Representation. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, USA, 97–107. https://doi.org/10.1145/3534678.3539303

[13] Sophie Cluet and Guido Moerkotte. 1995. On the complexity of generating optimal left-deep processing trees with cross products. In *Database Theory — ICDT '95*, Georg Gottlob and Moshe Y. Vardi (Eds.). Springer, Berlin, Heidelberg, 54–67. https://doi.org/10.1007/3-540-58907-4_6

[14] Binyang Dai, Qichen Wang, and Ke Yi. 2023. SparkSQL$^+$: Next-generation Query Planning over Spark. In *Companion of the 2023 International Conference on Management of Data (SIGMOD-Companion '23), June 18–23, 2023, Seattle, WA, USA*. ACM, New York, NY, USA, 115–118. https://doi.org/10.1145/3555041.3589715

[15] Shaleen Deep, Hangdong Zhao, Austen Z. Fan, and Paraschos Koutris. 2024. Output-sensitive Conjunctive Query Evaluation. *Proc. ACM Manag. Data* 2, 5, Article 220 (Nov. 2024), 24 pages. https://doi.org/10.1145/3695838

[16] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: a decision support benchmark for workload-driven and traditional database systems. *Proc. VLDB Endow.* 14, 13 (Sept. 2021), 3376–3388. https://doi.org/10.14778/3484224.3484234

[17] Leonidas Fegaras. 1998. A new heuristic for optimizing large queries. In *Database and Expert Systems Applications*, Gerald Quirchmayr, Erich Schweighofer, and Trevor J.M. Bench-Capon (Eds.). Springer, Berlin, Heidelberg, 726–735. https://doi.org/10.1007/BFb0054528

[18] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. 2016. Hypertree Decompositions: Questions and Answers. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Tova Milo and Wang-Chiew Tan (Eds.). ACM, 57–74. https://doi.org/10.1145/2902251.2902309

[19] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. 2005. Hypertree Decompositions: Structure, Algorithms, and Applications. In *Graph-Theoretic Concepts in Computer Science, 31st International Workshop, WG 2005, Metz, France, June 23-25, 2005, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3787)*, Dieter Kratsch (Ed.). Springer, 1–15. https://doi.org/10.1007/11604686_1

[20] Georg Gottlob, Matthias Paul Lanzinger, Davide Mario Longo, Cem Okulmus, Reinhard Pichler, and Alexander Selzer. 2023. Reaching Back to Move Forward: Using Old Ideas to Achieve a New Level of Query Optimization. In *Proceedings*

of the 15th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW 2023). CEUR-WS.org. https://doi.org/10.34726/5396

[21] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2002. Hypertree Decompositions and Tractable Queries. *J. Comput. Syst. Sci.* 64, 3 (2002), 579–627. https://doi.org/10.1006/JCSS.2001.1809

[22] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2003. Robbers, marshals, and guards: Game theoretic and logical characterizations of hypertree width. *J. Comput. Syst. Sci.* 66, 4 (2003), 775–808. https://doi.org/10.1016/S0022-0000(03)00030-8

[23] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data(base) Engineering Bulletin* 18 (1995), 19–29.

[24] Goetz Graefe and William J. McKenna. 1993. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*. IEEE Comput. Soc. Press, Vienna, Austria, 209–218. https://doi.org/10.1109/ICDE.1993.344061

[25] Marc H. Graham. 1979. *On the universal relation.* Technical Report. University of Toronto.

[26] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality estimation in DBMS: a comprehensive benchmark evaluation. *Proc. VLDB Endow.* 15, 4 (Dec. 2021), 752–765. https://doi.org/10.14778/3503585.3503586

[27] Xiao Hu. 2025. Output-Optimal Algorithms for Join-Aggregate Queries. *Proc. ACM Manag. Data* 3, 2, Article 104 (June 2025), 27 pages. https://doi.org/10.1145/3725241

[28] Xiao Hu and Qichen Wang. 2023. Computing the Difference of Conjunctive Queries Efficiently. *Proc. ACM Manag. Data* 1, 2, Article 153 (June 2023), 26 pages. https://doi.org/10.1145/3589298

[29] Zhekai Jiang, Qichen Wang, and Christoph Koch. 2026. metaDecomp. GitHub repository. https://github.com/epfldata/metaDecomp

[30] Zhekai Jiang, Qichen Wang, and Christoph Koch. 2026. Succinct Structure Representations for Efficient Query Optimization. arXiv:2603.15465 [cs.DB] https://arxiv.org/abs/2603.15465

[31] Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. 2016. AJAR: Aggregations and Joins over Annotated Relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (San Francisco, California, USA) *(PODS '16)*. ACM, New York, NY, USA, 91–106. https://doi.org/10.1145/2902251.2902293

[32] Donald Kossmann and Konrad Stocker. 2000. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Transactions on Database Systems* 25, 1 (March 2000), 43–82. https://doi.org/10.1145/352958.352982

[33] Paraschos Koutris, Stijn Vansummeren, Qichen Wang, Yisu Remy Wang, and Xiangyao Yu. 2026. Database Theory in Action: Yannakakis' Algorithm. In *29th International Conference on Database Theory (ICDT 2026)*. 21:1–21:6. https://doi.org/10.4230/LIPIcs.ICDT.2026.21

[34] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *The VLDB Journal* 27, 5 (Oct. 2018), 643–668. https://doi.org/10.1007/s00778-017-0480-7

[35] Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. 2024. LLM-R^2 : A Large Language Model Enhanced Rule-Based Rewrite System for Boosting Query Efficiency. *Proc. VLDB Endow.* 18, 1 (Sept. 2024), 53–65. https://doi.org/10.14778/3696435.3696440

[36] Zheng Luo, Wim Van den Broeck, Guy Van den Broeck, and Yisu Remy Wang. 2026. Algorithms for Optimizing Acyclic Queries. In *29th International Conference on Database theory (ICDT 2026)*.

[37] Riccardo Mancini, Srinivas Karthik, Bikash Chandra, Vasilis Mageirakos, and Anastasia Ailamaki. 2022. Efficient Massively Parallel Join Optimization for Large Queries. In *Proceedings of the 2022 International Conference on Management of Data*. ACM, Philadelphia PA USA, 122–135. https://doi.org/10.1145/3514221.3517871

[38] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. ACM, New York, NY, USA, 1275–1288. https://doi.org/10.1145/3448016.3452838

[39] Guido Moerkotte and Thomas Neumann. 2008. Dynamic programming strikes back. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 539–552. https://doi.org/10.1145/1376616.1376672

[40] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 677–692. https://doi.org/10.1145/3183713.3183733

[41] Heinz Prüfer. 1918. Neuer Beweis eines Satzes über Permutationen. *Archiv der Mathematischen Physik* 27 (1918), 742–744.

[42] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, New York, NY, USA, 1981–1984. https://doi.org/10.1145/

3299869.3320212

[43] Francesco Scarcello, Gianluigi Greco, and Nicola Leone. 2007. Weighted hypertree decompositions and optimal query plans. *J. Comput. System Sci.* 73, 3 (May 2007), 475–506. https://doi.org/10.1016/j.jcss.2006.10.010

[44] Wolfgang Scheufele and Guido Moerkotte. 1996. *Constructing Optimal Bushy Processing Trees for Join Queries is NP-hard.* Technical Report. Universität Mannheim.

[45] Patricia Griffiths Selinger, Morton Michael Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas Gordon Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data (SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/582095.582099

[46] Mihail Stoian and Andreas Kipf. 2024. DPconv: Super-Polynomially Faster Join Ordering. *Proc. ACM Manag. Data* 2, 6, Article 234 (Dec. 2024), 26 pages. https://doi.org/10.1145/3698809

[47] Transaction Processing Council (TPC). 2022. TPC Benchmark H. Available: https://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf.

[48] Bennet Vance. 1998. *Join-order optimization with Cartesian products.* Ph. D. Dissertation. Oregon Graduate Institute of Science and Technology. https://digitalcollections.ohsu.edu/record/177

[49] Bennet Vance and David Maier. 1996. Rapid bushy join-order optimization with Cartesian products. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data (SIGMOD '96)*. Association for Computing Machinery, New York, NY, USA, 35–46. https://doi.org/10.1145/233269.233317

[50] Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. 2025. Yannakakis$^+$: Practical Acyclic Query Evaluation with Theoretical Guarantees. *Proc. ACM Manag. Data* 3, 3, Article 235 (June 2025), 28 pages. https://doi.org/10.1145/3725423

[51] Qichen Wang, Qiyao Luo, and Yilei Wang. 2024. Relational Algorithms for Top-k Query Evaluation. *Proc. ACM Manag. Data* 2, 3, Article 168 (May 2024), 27 pages. https://doi.org/10.1145/3654971

[52] Qichen Wang and Ke Yi. 2022. Conjunctive Queries with Comparisons. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 108–121. https://doi.org/10.1145/3514221.3517830

[53] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. 2024. Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. In *14th Annual Conference on Innovative Data Systems Research (CIDR '24)*.

[54] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the 2022 International Conference on Management of Data*. ACM, Philadelphia PA USA, 931–944. https://doi.org/10.1145/3514221.3517885

[55] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *VLDB '81: Proceedings of the seventh international conference on Very Large Data Bases*, Vol. 7. 82–94.

[56] Clement T. Yu and Meral Z. Özsoyoğlu. 1979. An algorithm for tree-query membership of a distributed query. In *The IEEE Computer Society's Third International Computer Software and Applications Conference, COMPSAC 1979, 6-8 November, 1979, Chicago, Illinois, USA*. IEEE, 306–312. https://doi.org/10.1109/CMPSAC.1979.762509

[57] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, Brussels, Belgium, 3911–3921. https://doi.org/10.18653/v1/D18-1425

[58] Haozhe Zhang, Christoph Mayer, Mahmoud Abo Khamis, Dan Olteanu, and Dan Suciu. 2025. LpBound: Pessimistic Cardinality Estimation Using ℓ_p-Norms of Degree Sequences. *Proc. ACM Manag. Data* 3, 3, Article 184 (June 2025), 27 pages. https://doi.org/10.1145/3725321

[59] Hangdong Zhao, Austen Z. Fan, Xiating Ouyang, and Paraschos Koutris. 2024. Conjunctive Queries with Negation and Aggregation: A Linear Time Characterization. *Proc. ACM Manag. Data* 2, 2, Article 75 (May 2024), 19 pages. https://doi.org/10.1145/3651138

[60] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. 2025. Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. *Proc. ACM Manag. Data* 3, 3, Article 146 (June 2025), 28 pages. https://doi.org/10.1145/3725283

[61] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A Learned Query Rewrite System Using Monte Carlo Tree Search. *Proc. VLDB Endow.* 15, 1 (Sept. 2021), 46–58. https://doi.org/10.14778/3485450.3485456

[62] Xuanhe Zhou, Guoliang Li, Jianming Wu, Jiesi Liu, Zhaoyan Sun, and Xinning Zhang. 2023. A Learned Query Rewrite System. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 4110–4113. https://doi.org/10.14778/3611540.3611633

Received October 2025; revised January 2026; accepted February 2026