Succinct Structure Representations for Efficient Query Optimization

Zhekai Jiang*
EPFL
Lausanne, Switzerland
zhekai.jiang@epfl.ch

Qichen Wang*
EPFL
Lausanne, Switzerland
qichen.wang@epfl.ch

Christoph Koch
EPFL
Lausanne, Switzerland
christoph.koch@epfl.ch

Abstract

Structural decomposition methods for conjunctive queries offer powerful theoretical guarantees for efficient join evaluation, yet they are rarely used in real-world query optimizers. A major reason is the disconnection between cost-based plan search and structure-based evaluation. In this work, we bridge this gap by integrating structural properties of queries into cost-based optimization. To achieve this, we introduce meta-decompositions for acyclic queries, as the first universal representation that compactly represents and allows for efficient enumeration of all possible join trees, but can be constructed in polynomial time and has size linear in the query size. Building on this, we design a polynomial-time cost-based optimizer, based directly on the meta-decomposition, without having to explicitly enumerate all possible join trees. We characterize plans that can be found by this approach by a novel notion of width, which effectively connects the join tree structure, and thus the theoretical guarantees of running time, to any query plan. Experimental results demonstrate that the plans in our class are consistently comparable to the optimal plans found by the state-of-the-art dynamic programming approach, while our planning process runs orders of magnitude faster, especially on large, complex queries.

Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/epfldata/metaDecomp.

1 Introduction

Joins are the cornerstone of relational database queries. However, query optimizers face extreme difficulty when planning large multi-way join queries. 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 [11, 32]. For this reason, beyond a handful of joins (e.g., more than 12), current query optimizers must abandon exhaustive dynamic programming and resort to heuristic or greedy algorithms. As a result, real-world optimizers often miss the best plans and produce suboptimal ones that can be orders of magnitude slower.

Nevertheless, the database theory community has long been providing tools that exploit query structures to tackle this complexity. The *Yannakakis algorithm* [41] is one classic example that utilizes join trees of acyclic queries to obtain an instance-optimal guarantee on the asymptotic complexity for evaluating these queries. Hypertree decompositions [17] and related width measures [3, 4] generalize this approach to cyclic queries, providing strong worst-case optimal guarantees of evaluation time. Over the years, these structure-based methods have been further generalized to evaluate joins with projections [5, 13, 22], aggregations [2, 26], differences [23, 44], top-k [38], comparisons [39], etc., all with strong theoretical performance guarantees. These results suggest that leveraging a query's structural properties could be of great help for the search of query plans and could dramatically improve evaluation performance with robust worst-case bounds. Yet, in practice, despite limited attempts by, e.g., Scarcello et al. [31] and RelationalAI [3], query optimizers rarely utilize structural decompositions and, therefore, continue to struggle with large, complex queries.

Why are structural methods hardly adopted in mainstream systems? A core obstacle is the dichotomy between cost-based plan search and structure-based evaluation. Traditional cost-based optimizers usually find query plans with the optimal or near-optimal costs under some cost model using (1) **exact algorithms**, mostly based on dynamic programming, such as [28, 33–36], which have to run in exponential time, thus not scalable to very large numbers of relations; (2) **greedy strategies** based on heuristics such as greedy operator ordering (GOO) [9, 14, 29], which have guarantees of neither optimality nor a reasonable approximation factor unless under some assumption, such as independence of predicate selectivities, of the cost function; and/or (3) **transformation-based techniques**, such as the Volcano/Cascades framework [19, 20] and genetic algorithms [7]. Although certain approaches leverage the structures of, e.g., *query graphs* [9, 28, 29], they do not necessarily correspond to the requirements of structural decompositions. Therefore, join orders generated by these methods may not necessarily correspond to any optimal structural decomposition. This makes it impossible to directly apply structure-based methods on them.

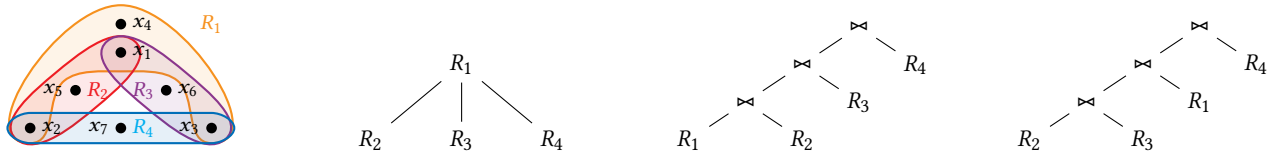
Example 1.1. Consider the query

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

This is a Boolean acyclic query. Its hypergraph is shown in Fig. 1a and a join tree in Fig. 1b. Fig. 1c is a query plan that follows the structure of the join tree. Fig. 1d does not, but it is still within the search space of traditional algorithms.

On the other hand, theoretical research usually focuses on data complexity and assumes the optimal tree decomposition is given in advance. In practice, there can be exponentially many possible optimal tree decompositions for a given query, and choosing the right one is difficult without a cost-based exploration of alternatives. Furthermore, despite the appealing asymptotic running-time guarantee, Yannakakis-style algorithms have few adoptions in practice due to their considerable hidden overhead. Even though recent advances [6, 8, 37, 40] improve the performance and achieve orders of magnitude speedups on certain queries, they still do not integrate the cost-based decision process of real optimizers and thus cannot

*Both authors contributed equally to this research.



(a) Hypergraph of the query **(b) A valid join tree** **(c) A width-1 query plan** **(d) A width-2 query plan**

Figure 1: The hypergraph, a join tree, a width-1 query plan, and a width-2 query plan of the query in Example 1.1

guarantee truly optimal plans for all queries. In summary, existing optimizers lack a way to incorporate structural decompositions into plan search, and scanning through all possible decompositions explicitly would be computationally prohibitive.

1.1 Our Contributions

In this work, we take a major step towards bridging structural decomposition techniques with cost-based query optimization. We **introduce 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, which gives guidance for query planning even in the absence of any cost information. In particular, we provide an example of evaluating acyclic Boolean conjunctive queries in *instance-optimal time*, using width-1 query 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 query plans with the minimal width, we remove the friction caused by higher-width query plans and limit the plan space to only those that follow the query's optimal tree decompositions. To make this practical, we present a notion of *meta-decompositions* for acyclic conjunctive queries that **compactly represents the set of all possible join trees of the query in a single structure**. Despite the possibly exponential number of join trees of a given query, this representation *(1) has size linear in the query, (2) can be constructed in polynomial time, and (3) supports efficient enumeration of all join trees without repetition*, providing a practical way to navigate the exponentially large search space of join trees.

Armed with these tools, we then develop a ***structure-guided cost-based optimization* framework, to find the optimal width-1 query plan among all candidate join trees**. We prove that if the query's meta-decomposition has bounded fan-out, then *the global optimal width-1 query plan can be found in linear time* $O(|Q|)$. For general queries, we give efficient heuristics that leverage the meta-decomposition to guide the optimizer towards good query plans without exhaustive search.

We empirically demonstrate that our approach is potentially game-changing for query optimization for large joins. We **implement a new optimization framework, metaDecomp**, and evaluate it on join queries from the join order benchmark (JOB) [27] and a new extension JOBLarge, with even larger, more complex queries. We compare the results against the native optimizer of DuckDB and a state-of-the-art dynamic programming algorithm, DPconv [34]. The results show that width-1 plans consistently match or outperform the optimal plans found by these baselines. Moreover, by exploiting query structures, our optimization process is orders of magnitude faster for large queries. In summary, integrating structural decompositions into cost-based optimization not only yields better plans but also reduces the optimization overhead, leading to robust and scalable query optimization by truly understanding both the cost information and the query structure at the same time.

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, and the way to enumerate all join trees based on them. In Section 5, we show how to perform cost-based optimization based on meta-decompositions. In Section 6, we present experimental results to show the efficiency of query plans generated by this approach. We discuss the possible future works in Section 7.

Due to the space constraints, all missing proofs in this paper can be found in the appendix of our full technical report [25].

2 Preliminaries

We write 2^S for the set of all subsets of a set S. If $f : A \to B$ and $S \subseteq A$, we let $f(S) = \{f(a) \mid a \in S\}$. 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 [15–18, 37]. 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

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

where $\boldsymbol{\sigma_i}$ 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 $\bar{y} = \cup_{i \in [n]} \bar{x}_i$, the query is 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 $H(Q) = (V(H), E(H))$, where the vertices set $V(H)$ consists of all attributes appearing in the query, i.e., $V(H) = \cup_{i \in [n]} \bar{x}_i$, 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 $e_i = \bar{x}_i$.

The *primal graph* of a hypergraph H is the *graph* on $V(H)$ which is obtained from H by turning each hyperedge of H into a clique.

A hypergraph is connected if its primal graph is connected. *All hypergraphs in this paper are assumed to be finite, connected, and to have a non-empty set of hyperedges.*

2.2 Acyclicity and Join Trees

A conjunctive query is acyclic if its associated hypergraph is acyclic, and we define the notion of acyclicity of a hypergraph in the standard way in database theory, e.g., as in [1, Section 6.4]. A hypergraph H is acyclic if and only if the output of the GYO algorithm [21, 42] on H is empty [1, Theorem 6.4.5]. Additionally, a conjunctive query Q is acyclic if and only if there exists a valid width-1 hypertree decomposition, e.g., a valid *join tree* for the hypergraph $H(Q)$ [1, Theorem 6.4.5].

Different from the usual definition of join trees [1], where each vertex is simply a relation, in this paper, we define join trees using notations for hypertree decompositions [17], to make it easier to generalize to the meta-decompositions that we will define later.

Let Q be an acyclic conjunctive query with associated hypergraph H. A *join tree* for Q is a triple (T, χ, λ) where T is a tree, $\chi : V(T) \to 2^{V(H)}$ is a function s.t. $\chi(V(T)) = \bigcup_{p \in V(T)} \chi(p) = V(H)$, and $\lambda : V(T) \to E(H)$, and it satisfies the following conditions:

- (H_1) **Coverage condition.** For each $e \in E(H)$, there exists exactly one $p \in V(T)$ with $\lambda(p) = \{e\}$.
- (H_2) **Connectedness condition.** For each $v \in V(H)$, the set of vertices $\{p \in V(T) \mid v \in \chi(p)\}$ 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 $\chi(p) \cap \chi(q) \subseteq \chi(s)$.
- (H_3) **Width-1.** For all $p \in V(T)$, $|\lambda(p)| = 1$ and $\chi(p) = V(\lambda(p))$.

For simplicity, we assume that all trees are rooted. For two trees T_1, T_2 of $H(Q)$, rooted at r_1 and r_2 respectively, with $E(T_1) = E(T_2)$, we still consider them to be different trees if $r_1 \neq r_2$. When writing "$(p, q) \in E(T)$" in a tree T, we refer to the edge $\{p, q\} \in E(T)$ with p being closer to the root than q. Moreover, we write T_p as the subtree rooted at p, i.e., T_p contains all vertices q such that p is on the unique path from the root of T to q. For every neighbor q of vertex p in T, we write $T_{p \to q}$ for the subtree with vertices $V(T_{p \to q}) = \{s \in V(T) \mid \text{the path from } p \text{ to } s \neq p \text{ in } T \text{ contains } q\}$.

We also define the *fan-out* of any $p \in V(T)$ as

$$f(p) = \begin{cases} \deg(p) - 1, & \text{if } p \text{ is not the root} \\ \deg(p), & \text{if } p \text{ is the root} \end{cases}$$

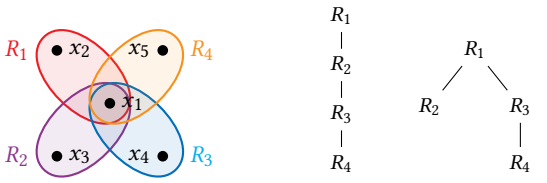
where $\deg(p)$ is the degree (number of neighbours) of node p, and $f(p)$ corresponds to the number of children of p on T. The *fan-out* of the tree T is

$$f(T) = \max_{p \in V(T)} f(p).$$

Given a join tree T, we recursively define the *induced query* for every $p \in V(T)$ as

$$Q(p) = \begin{cases} p & \text{if } p \text{ is a leaf node} \\ p \bowtie \left(\bowtie_{c \text{ is a child node of } p} Q(c) \right) & \text{Otherwise.} \end{cases}$$

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



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

Figure 2: Hypergraph and two valid join trees of the star query Q in Example 2.2

Theorem 2.1. *Consider* star queries[1] *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 2.2. Consider the query $Q = R_1[x_1, x_2] \bowtie R_2[x_1, x_3] \bowtie R_3[x_1, x_4] \bowtie R_4[x_1, x_5]$. Q is a star query of the form we consider. Its associated hypergraph is as shown in Fig. 2a, and Fig. 2b shows two of its valid join trees.

In fact, the possible join trees of such queries in Theorem 2.1 and Example 2.2 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 (H_2). There are a total of n^{n-2} distinct non-rooted trees, and each tree has n possible root nodes. Each such tree can be encoded by a *Prüfer sequence* [30] of length $n - 2$ (where each element in the sequence can take values between 1 and n for a tree with n vertices), plus one extra indicator for the root node.

2.3 Query Plans and Join Ordering

A *query plan* $\mathcal{P} = (V(\mathcal{P}), E(\mathcal{P}))$ for a query Q is a rooted tree, where each non-leaf node is a relational operator (e.g, join, projection, selection), and each leaf node corresponds to a base relation. A plan specifies an order of operations to evaluate the query, where the outputs of child nodes serves as inputs to the parent. For the purpose of join ordering, *we assume selections and projections are inherently integrated in the join operation at the lowest possible level, and therefore we use only join operators in internal nodes.*

Given a query plan $\mathcal{P}$, we recursively define the *induced query* for every $p \in V(\mathcal{P})$ as

$$Q(p) = \begin{cases} p & \text{if } p \text{ is a leaf node,} \\ Q(c_1) \bowtie Q(c_2) & \text{if } p \text{ is join operation.} \end{cases}$$

In addition, we exclude Cartesian products. For every join node with two child nodes c_1 and c_2, inducing subqueries $Q(c_1)$ and $Q(c_2)$, with output attributes $\bar{y}_1$ and $\bar{y}_2$, respectively, we require $\bar{y}_1 \cap \bar{y}_2 \neq \emptyset$.

In this work, we focus on query optimization of binary joins, where, for every join operation on the query plan, there are exactly two child nodes. Given a conjunctive query with more than 2 relations, the query plan also specifies the sequence and structure in which joins are to be evaluated. A query plan is called *left-deep* (or *right-deep*) if all join operations have their right (left) child node cor-

[1]We note that, in some literature, e.g., [9], such queries are called "*clique*" queries, because their *query graphs* are cliques. Regardless, it has been shown that optimizing such queries exactly is NP-hard and takes exponential time [9, 32] (unless P = NP).

responding to a base relation; otherwise, the query plan is referred to as *bushy*.

We apply a cost function C to select the best query plan, where C can be represented as a recursive function along the query plan $\mathcal{P}$ and root node r, as follows:

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

where $c(r)$ is the cardinality of the output of r, and $\mathcal{P}_1$ and $\mathcal{P}_2$ (if applicable) are the left and right subtrees of r, respectively. In this work, we adopt the cost function C_{out} to minimize the intermediate result size by substituting the $\oplus$ operator simply with arithmetic +, as is standard in, e.g., [34].

The standard approach to find an optimal plan is to recursively find the optimal split of a set of relations S into two disjoint subsets S_1 and S_2 with $S = S_1 \cup S_2$ [28, 33–36]. With a given cost function C, the optimization problem can be solved by dynamic programming (DP) with a worst-case time complexity $O(3^n)$ given by the algorithm DPccp [28]. The recently proposed algorithm DPconv [34] improves this computation by utilizing fast subset convolution, reducing the complexity for this 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 & Width-1 Plans

We start by introducing the width notion for a given query plan P, which connects the plan's structure to the potential size of its intermediate results. This measure helps quantify the complexity of a query plan before looking at the specific database instance. Given a query plan $\mathcal{P}(V(\mathcal{P}), E(\mathcal{P}))$, for any node $p \in V(\mathcal{P})$, we consider the subquery evaluated at that node. Let $H_p = (V(H_p), E(H_p))$ be the hypergraph induced by the leaf nodes in the subtree rooted at p with 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 corresponding to the rest of the query, where $E(\overline{H_p}) = E(H) \setminus E(H_p)$, and $V(\overline{H_p}) = \bigcup_{e \in E(\overline{H_p})} e$.

The set of attributes that separates H_p and $\overline{H_p}$ is the interface $I(p) = V(H_p) \cap V(\overline{H_p})$. We then define

$$Q'(p) \leftarrow \pi_{I(p)} Q(p),$$

which captures the minimal set of attributes that need to be kept in the intermediate result at node p, because this intermediate result must then be joined with the results from the rest of the query. The set of join attributes is precisely the interface $I(p)$. The width of node p is the size of the smallest subset of its relations needed to cover this interface, i.e.,

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

and the width of P is the maximum width over all nodes, i.e.,

$$w(P) = \max_{p \in V(\mathcal{P})} w(p).$$

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

Theorem 3.1. *Given a Boolean conjunctive query Q and a query plan $\mathcal{P}$ for the query,*

$$\max_{p \in V(H)} |Q'(p)|$$

is upper bounded by $O(N^{w(\mathcal{P})})$ in the worst case, where N is the maximum cardinality of any base relation.

Such a width notation can draw a direct connection with the join tree, as follows.

Definition 3.2. We say a query plan $\mathcal{P} = (V(\mathcal{P}), E(\mathcal{P}))$ is *induced by* a join tree $T = (V(T), E(T))$ if

(1) for each node $p \in V(T)$, there exists $q \in V(\mathcal{P})$ with $p = Q(q)$, or $Q(p) = Q(q)$, i.e., the induced query of node p in the join tree is exactly the induced query of node q in the query plan, unless $Q(q)$ involves only a single relation; 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 $E(T)$.

Theorem 3.3. *Given a query plan $\mathcal{P}(V(\mathcal{P}), E(\mathcal{P}))$ for query Q, the width $w(\mathcal{P}) = 1$ if and only if there exists a join tree $T = (V(T), E(T))$ for Q that induces the query plan $\mathcal{P}$.*

Theorem 3.3 directly lead to the relationship between width-1 query plans and acyclic queries:

Theorem 3.4. *A conjunctive query Q has width-1 query plans if and only if Q is acyclic.*

Example 3.5. Consider the query

$$Q = R_1[x_1, x_2, x_3] \bowtie R_2[x_1, x_4, x_5] \bowtie R_3[x_5, x_6] \bowtie R_4[x_3, x_7].$$

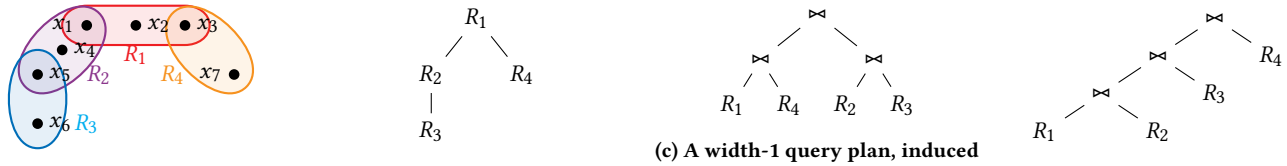
Its hypergraph is shown in Fig. 3a and a join tree in Fig. 3b. Given this join tree, Fig. 3c is an example of a valid width-1 query plan, whereas Fig. 3d shows a query plan that is width-2: taking p as the node on the join tree with R_2, there is no q on the query plan such that $Q(p) = Q(q) = R_2 \bowtie R_3$.

The width-1 query plan plays an important role in tree-based query evaluation methods. Tree-based query evaluation methods follow a common pattern: given a join tree, they apply a series of reductions starting from the leaves and working upwards until only a single node remains. The selected join tree heavily influences the efficiency of the execution plan produced by these methods. In essence, width-1 query plans exploit the join tree structure that evaluates the query in a bottom-up fashion, where 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.

Early elimination of irrelevant attributes. By the connectedness condition of join trees, attributes that are not part of the output but only participate in the subtree can be projected out in an early stage, as long as the parent no longer contains the attribute.

Example 3.6. With the query in Example 3.5, assuming that x_3 is not an output attribute, 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 at once, we can locally optimize the join order at each node of the 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 $Q(c_1), \ldots, Q(c_{f(p)})$.



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

Figure 3: The hypergraph, a join tree, a width-1 query plan, and a width-2 query plan of the query in Example 3.5

The order of doing these joins can be optimized independently for 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 practical queries. Our experimental results show that the maximum fan-out $f(T)$ tends to be very low. In the queries we experimented with in the join ordering benchmark (JOB), it is at most 5 and on average 3.

Example 3.7. Consider again the join tree in Fig. 3b for the query in Example 3.5. Given this join tree, a width-1 query plan requires that a subtree with $R_2 \bowtie R_3$ exist in the query plan. 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. Suppose that 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.

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

Proof sketch. We observe, based on Theorem 3.3, that, given a width-1 query plan $\mathcal{P}$ induced by a join tree T, for each $t \in V(T)$, with children $c_1, \ldots, c_{f(t)}$ and the corresponding $p \in V(\mathcal{P})$, such that $Q(t) = Q(p)$, if we consider the sub-query-plans $\mathcal{P}_{c_1}, \ldots, \mathcal{P}_{c_{f(t)}}$, induced by the sub-join-trees $T_{c_1}, \ldots, T_{c_{f(t)}}$ rooted at the children of t, each as a single relation, then the sub-query-plan $\mathcal{P}_p$ rooted at p is a *left-deep plan* consisting of $t, \mathcal{P}_{c_1}, \ldots, \mathcal{P}_{c_{f(t)}}$, and t is an operand of the deepest join operation. Therefore, we use dynamic programming with a table DP that stores the optimal plan to join each subset of children plus the node p. There are $2^{f(p)}$ states (subsets of $f(p)$ children), and, for each state, we try at most $f(p)$ ways to add the next child, which gives $O(2^{f(p)}f(p))$ for each node p, and $O\left(f(T)2^{f(T)}|Q|\right)$ in total. □

Admittedly, this approach comes at the cost of excluding some plan alternatives with higher widths. However, those excluded plans are not compatible for structure-guided algorithms and will not preserve the theoretical guarantees provided by the query structure.

Structure-guided query evaluation. An additional advantage of width-1 query plans is that they naturally accommodate those newly developed structure-guided query evaluation methods. For example, we can simulate the Yannakakis algorithm for evaluating *relation-dominated* conjunctive queries [37] 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 $\bar{y} = \emptyset$.

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 in a bottom-up fashion. 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 a linear running time.

Example 3.10. Consider again the query in Example 1.1, with the hypergraph, a join tree, and two query plans shown in Fig. 1. Assume that it is evaluated over a database instance with $R_1 = \{(i,i,i,i) \mid 1 \le i \le n\}$, $R_2 = R_3 = R_4 = \{(i,j,j) \mid 1 \le i,j \le n\}$ for some $n > 1$. The query plan shown in Fig. 1c has width 1, as the interface of the node p that induces the query $Q(p) = R_1 \bowtie R_2$ is $I(p) = \{x_1, x_2, x_3\}$, which can be covered by only one hyperedge, R_1. The case is similar for the subquery $(R_1 \bowtie R_2) \bowtie R_3$, whose interface $\{x_2, x_3\}$ can be covered by R_1. In this case, *any width-1 query plan can give a linear runtime*, as we can start from R_1, apply a series of semi-joins, and the intermediate sizes will always be at most the size of $|R_1|$. On the contrary, the query plan in Fig. 1d has width 2 because the interface of q that induces the query $Q(q) = R_2 \bowtie R_3$ is $I(q) = \{x_1, x_2, x_3\}$, which cannot be covered by any single hyperedge $e \in \{R_2, R_3\}$. *This would cause a quadratic blowup of the intermediate result size*, as each tuple $(i,j,j) \in R_2$ can be joined with n tuples (i, j', j') with $1 \le j' \le n$ from R_3.

4 Meta-Decompositions

Although we have shown a connection between join trees and width-1 query plans in Theorem 3.1 and Theorem 3.3, it is impractical to find the optimal width-1 query plan by enumerating every possible join tree—As noted earlier, a query can have an exponential number of possible join trees. Therefore, in this section, we introduce and explore a generalized representation that allows us to efficiently represent all join trees of an acyclic query. This enables us to perform query optimization 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 2.1 and illustrated by Example 2.2. In such queries, any pair of relations in Q_{star} has valid join predicates, meaning there are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairwise predicates. Any specific join tree selects only $n-1$ of

those predicates and infers the remaining predicates through the tree structure, where $\frac{(n-1)(n-2)}{2}$ predicates are not selected for that particular tree. On the contrary, in the case of a query where, for any attribute x, there are at most two relations $R_i(\bar{x}_i)$ and $R_j(\bar{x}_j)$ such that $x \in \bar{x}_i$ and $x \in \bar{x}_j$, as in, e.g., Example 3.5, the query demonstrates a more "tree-like" structure compared to Q_{star}. 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, it can capture all possible join trees but requires only linear space.

Definition 4.1. Given a conjunctive query Q and the associated hyergraph $H(Q)$, its *meta-decomposition* $M = (V(M), r(M), E(M), \lambda, \chi, \kappa)$ is a labeled tree, where

- $(V(M), r(M), E(M))$ is a tree with vertices $V(M)$, a unique root $r(M) \in V(M)$, and edges $E(M)$,
- $\lambda : V(M) \to 2^{E(H)}$ maps each tree vertex to either a set of hyperedges in H or to $\emptyset$, and
- $\chi : V(M) \to 2^{V(H)}$ and $\kappa : V(M) \to 2^{V(H)}$ each maps each tree vertex in M to a set of vertices in H,

such that it satisfies (H_1), (H_2), and

(H_3') For all $p \in V(M)$, $\chi(p) \subseteq V(\lambda(p))$ if $\lambda(p) \neq \emptyset$.

(H_4) **Interface condition.** For every non-root node $p \in V(M)$, let q be p's parent node, $V(M)$ be the set of vertices in M, $\kappa(p)$ 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 root node $p = r(M)$, we set $\kappa(p) = \emptyset$.

(H_5) **Uniqueness condition.** For every pair of nodes $p, q \in V(M)$ such that $\kappa(p) = \kappa(q)$, there exists one unique node $o \in V(M)$ with $\lambda(o) = \emptyset$ and $\chi(o) = \kappa(p) = \kappa(q)$.

For acyclic queries, the meta-decompositions additionally satisfy

(H_3'') For all $p \in V(M)$ such that $\lambda(p) \neq \emptyset$, we have $|\lambda(p)| = 1$ and $\chi(p) = V(\lambda(p))$.

The meta-decomposition differs from the standard join tree in the following ways. First, we introduce a κ-label using the interface condition (H_4): Given a meta-decomposition M, for any node $p \in V(M)$, $\kappa(p)$ captures the *interface* between the induced hypergraph $H(V(T_p))$ and the complementary hypergraph $H \setminus H(V(T_p))$. Removing hyperedge $\lambda(p)$ from the query hypergraph H partitions the hypergraph into multiple subhypergraphs, each intersecting with $\lambda(p)$ precisely at these interfaces, where for p, the interfaces include $\kappa(p)$ and $\kappa(c)$ for every child node c of p in M. Condition (H_4)(b) limits the shape of the meta-decomposition M: there can be multiple nodes that can satisfies $\kappa(p) \subseteq \chi(q)$, while Condition ($H_4$)(b) limits q to be the highest node with $\kappa(p) \subseteq \chi(q)$.

Furthermore, we introduce nodes with $\lambda = \emptyset$. We call such nodes *minor nodes*, and other nodes *physical nodes*, in the meta-decomposition. Minor nodes serve exactly to represent scenarios where multiple relations share a common interface, as captured by condition (H_5). When several relations have a common interface k, as for the example of a star query, let p be a relation that contains the interface with $k \subseteq \chi(p)$, we can either place them in the same subtree of p on the join tree or separate them into different subtrees. The use of minor nodes facilitates this flexibility without the need to introduce an exponential number of edges in the representation.

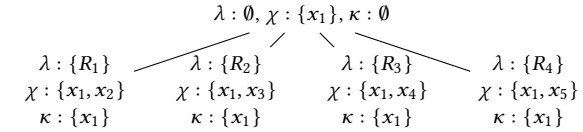


Figure 4: Meta-decomposition of the query in Example 2.2

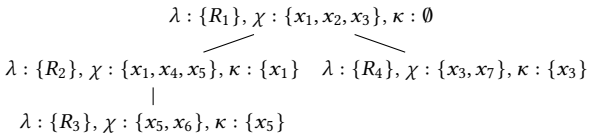


Figure 5: Meta-decomposition of the query in Example 3.5

Example 4.2. The meta-decomposition of the query in Example 2.2 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 are exactly $\{x_1\}$.

Example 4.3. The meta-decomposition of the query in Example 3.5 is shown in Fig. 5. It has the exact same structure as the join tree shown in Fig. 3b and has no minor node.

Example 4.4. Consider the query

$$Q = R_1[x_1, x_2, x_6] \bowtie R_2[x_1, x_2, x_3, x_7] \bowtie R_3[x_1, x_3, x_4, x_8] \bowtie R_4[x_1, x_4, x_9] \bowtie R_5[x_1, x_5]).$$

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 which should be intuitive based on the meta-decomposition. We also highlight the fact that R_5 could in fact be the child of any other node on 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.

4.1 Constructing Meta-Decompositions

A meta-decomposition can be constructed by Algorithm 1. In the algorithm, $o(e, H) = e \cap \left(\cup_{e' \in E(H) \setminus \{e\}} e'\right)$ denotes the interface, as a set of vertices, of hyperedge e with the rest of the hyperedges $E(H) \setminus \{e\}$ in the hypergraph H. A hyperedge e is *reducible* if there exists another hyperedge e', such that $o(e, H) \subseteq e'$. We call such hyperedges *ears*. Similar to the GYO algorithm, the algorithm is also based on reduction. The GYO algorithm removes one ear for each round, thereby establishing a total order for all hyperedges in order to identify a specific join tree. In contrast, Algorithm 1 *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.

For every round, the algorithm first handles the case that some edges e_1, e_2 are *mutually reducible*, i.e., $o(e_1, H) = o(e_2, H)$ (Lines 4-12). The algorithm will create a minor node p with $\chi(p) = o(e_1, H)$ if such a node does not exist, and then remove e_1 and e_2 from the

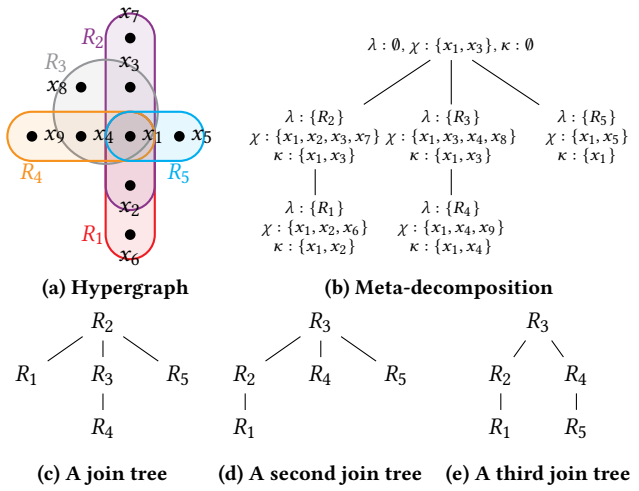


Figure 6: Hypergraph, meta-decomposition, and three join trees of the query in Example 4.4

hypergraph. We then add a hyperedge $e_m = o(e_1, H)$, which conceptually corresponds to the minor node, back to the hypergraph, so that the acyclicity of H is not broken. The rationale for adding e_v can be illustrated by the following example:

Example 4.5. Consider the query

$$Q = \pi_\emptyset(R_1[x_1, x_2, x_3, x_4] \bowtie R_2[x_1, x_2, x_5] \bowtie R_3[x_1, x_3, x_6] \bowtie R_4[x_2, x_3, x_6] \bowtie R_5[x_1, x_2, x_3, x_7]),$$

which is similar to Example 3.10 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_v, 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:

Lemma 4.6. *Given an acyclic hypergraph H, if there exist two distinct ears $e_1, e_2 \in E(H)$ such that $o(e_1, H) = o(e_2, H)$, then, the hypergraph H', with $E(H') = E(H) \setminus \{e_1, e_2\} \cup \{o(e_1, H)\}$ and $V(H') = \cup E(H')$, is acyclic.*

Proof sketch. In the GYO algorithm for H, since $o(e_1, H) = o(e_2, H) \subseteq e_2$, e_1 is an ear and can already be removed, with e_2 being its witness. So, there exists a join tree T of H where there is a leaf node p_1 with $\lambda(p_1) = \{e_1\}$, and its parent p_2 has $\lambda(p_2) = \{e_2\}$. If we merge p_1 and p_2 into a single node p_m with $\lambda(p_m) = \{o(e_1, H)\}$ to obtain a new tree T', T' is a join tree of H'. □

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 physical node p and set $\kappa(p) = o(e, H')$. During the procedure, the algorithm creates minor nodes if necessary (Lines 9, 14, 17–18, and 20).

Theorem 4.7. *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 $E(H')$. □

Theorem 4.8. *Given an acyclic query with an acyclic hypergraph H, Algorithm 1 returns a valid meta-decomposition of H.*

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 hypergraph with $V(H') = V(H)$ and $E(H') = E(H)$
2 $V(M) \leftarrow \emptyset$
3 **while** $|E(H')| > 1$ **do**
4 **foreach** *maximal* $S \subseteq E(H')$ *such that* $|S| > 1$, *and, for any* $e_1, e_2 \in S$, $o(e_1, H') = o(e_2, H')$ **do**
5 $o \leftarrow$ the value of $o(e_i, H')$ that all $e_i \in S$ share
6 **if** *there exists* $v \in V(M)$ *such that* $\lambda(v) = \emptyset$ *and* $\chi(v) = o$ **then** // There is already a minor node. We remove it and create a new one later
7 Remove v from $V(M)$
8 **foreach** $e \in S$ **do**
9 Add a node p to $V(M)$ with $\lambda(p) = \{e\}$ if $e \in E(H)$, or $\lambda(p) = \emptyset$ otherwise, $\chi(p) = e$, and $\kappa(p) = o(e, H')$
10 Remove e from $E(H')$
11 Add a special hyperedge $e_m = o$ to $E(H')$
12 $\mathcal{E} \leftarrow$ the set of all ears $e \in E(H')$, i.e., for every $e \in \mathcal{E}$, there exists a distinct $e' \in E(H')$ with $o(e, H') \subseteq e'$
13 **foreach** $e \in \mathcal{E}$ **do**
14 Add a new node p to $V(M)$ with $\lambda(p) = \{e\}$ if $e \in E(H)$, or $\lambda(p) = \emptyset$ otherwise, $\chi(p) = e$, and $\kappa(p) = o(e, H')$
15 Remove e from $E(H')$
16 $S \leftarrow \{p' \in V(M) \cup \mathcal{E} \mid \kappa(p') = \kappa(p)\}$
17 **if** $|S| > 1$ *and, for all* $p' \in S$, $\lambda(p') \neq \emptyset$ **then**
18 Add a new node v to $V(M)$ with $\lambda(v) = \emptyset$, $\chi(v) = \kappa(p)$, and $\kappa(v) = \kappa(p)$
19 $e \leftarrow$ the remaining edge in $E(H')$
20 Add a new node r to $V(M)$ with with $\lambda(r) = \{e\}$ if $e \in E(H)$, or $\lambda(r) = \emptyset$ otherwise, $\chi(r) = e$, and $\kappa(r) = \emptyset$
21 Remove e from $E(H')$
22 $E(M) \leftarrow \emptyset$
23 **foreach** $p \in V(M)$ *such that* $\kappa(p) \neq \emptyset$ **do**
24 **if** *there exists* $q \in V(M)$ *such that* $\lambda(q) = \emptyset$ *and* $\chi(q) = \kappa(p)$ **then**
25 Add an edge (q, p) in $E(M)$
26 **else**
27 Find $q \in V(M)$ such that $\kappa(p) \subseteq \chi(q)$ and $\kappa(p) \not\subseteq \kappa(q)$
28 Add an edge (q, p) in $E(M)$
29 **return** $M = (V(M), r, E(M), \lambda, \chi, \kappa)$

Theorem 4.9. *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 $V(M)$ in M are either physical nodes or minor nodes. We create one physical node per hyperedge in $E(H)$. And for minor nodes, we use the fact that each minor node m has fan-out $f(m) \geq 2$. □

4.2 Join Tree Enumeration

With the meta-decomposition, it is possible to enumerate all possible join trees of a hypergraph. The distinct join trees of the same hypergraph can be different in the following ways:

(1) **"Re-branching".** If a vertex p in a join tree has children c_1 and c_2 such that $\kappa(c_1) = \chi(c_1) \cap \chi(p) \subseteq \chi(c_2)$, then c_1 could have been a child of c_2 without violating any condition. In the meta-decomposition, condition (H_4) ensures that c_1 is positioned at the highest possible node. Therefore, when enumerating a join tree, we can make c_1 a child of c_2 if $\kappa(c_1) \subseteq \chi(c_2)$. The join tree Fig. 6e based on meta-decomposition Fig. 6b from Example 4.4 demonstrates such a case.
(2) **"Rerooting".** A rerooting of a join tree is still a valid join tree.
(3) **Minor nodes.** If multiple relations share the same set of attributes pairwise, they can be joined in arbitrary order. This situation is represented by the minor nodes in the meta-decomposition. An example is the root node with χ-label $\{x_1\}$ in Fig. 4 for Example 2.2, based upon which we should be able to enumerate, among others, the join trees in Fig. 2b.

These alternatives should be considered in order to enumerate all join trees, and the meta-decomposition keeps all the information to allow for that. 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, and then enumerates all possible ways to combine these join trees into one. For minor nodes r, we find its neighbours that share the common set of attributes given by $\chi(r)$ and use the Prüfer sequence to generate all possible distinct structures of join trees without repetition, where each vertex will be replaced by the join tree for a subtree given by a neighbour, and at this point we consider all rerootings of some 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 the space constraints, the algorithms for generating all Prüfer sequences and rerootings are given in the technical report.

Example 4.10. Take the query in Example 4.4 as an example. 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 $\lambda(v_i) = \{R_i\}$. r is a minor node, with a $\chi(r) = \{x_1, x_3\}$. 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. We then enumerate all possible trees with vertices v_2 and v_3, having $\kappa(v_2) = \kappa(v_3) = \chi(r)$, using Prüfer sequences. There are only two such trees, each of

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

input : A meta-decomposition $M = (V(M), r, E(M), \lambda, \chi, \kappa)$
output : All possible non-homomorphic join trees up to rerooting

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 **if** $\lambda(v) = \emptyset$ **then** // minor node
4 $S \leftarrow \{c \in C \mid \kappa(c) = \chi(v)\}$ // they can be found at the head of the sorted C
5 **foreach** $c \in S$ **do**
6 $\mathcal{T}_c \leftarrow$ enumerate(c) // All non-homomorphic join trees for subtree M_c rooted at c, up to rerooting
7 **if** $\kappa(v) = \chi(v)$ **then**
8 $\mathcal{T}_P \leftarrow$ Prüfer($S \cup \{v\}$), all rerooted to v // Enumerate all possible trees with vertices $S \cup \{v\}$ // The minor node v will eventually be replaced on Line 33
9 **else** // $\kappa(v) \neq \chi(v)$, the parent should not participate
10 $\mathcal{T}_P \leftarrow$ Prüfer(S)
11 **foreach** $T_P \in \mathcal{T}_P$ **do** // A skeleton of a way to connect subtrees, given by a Prüfer sequence
12 **foreach** $c \in V(T_P)$ **do**
13 **if** *c is a leaf node* **then**
14 $\mathcal{T}'_c \leftarrow \mathcal{T}_c$
15 **else**
16 $\mathcal{T}'_c \leftarrow \emptyset$
17 **foreach** *non-leaf $c \in V(T_P)$, in bottom-up order, $T_c \in \mathcal{T}_c$, child d of c on T_P, $T_d \in \bigcup_{T_d \in \mathcal{T}_d}$ rerootings$(T_d, \emptyset, \kappa(d))$, $u \in V(T_c)$ such that $\kappa(d) \subseteq \chi(u)$* **do**
18 $T'_c \leftarrow T_c$, with T_c added as a child of u
19 Add T'_c to $\mathcal{T}'_c$
20 $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'_{\text{root}(T_P)}$
21 $X \leftarrow C \setminus S$, preserving the order
22 **else**
23 $\mathcal{T} \leftarrow$ a set of one tree with one vertex v only
24 $X \leftarrow C$, preserving the order
25 **foreach** *child $c \in X$ in order* **do**
26 $\mathcal{T}_c \leftarrow \bigcup_{T \in \text{enumerate}(c)}$ rerootings$(T, \emptyset, \kappa(c))$
27 $\mathcal{T}' \leftarrow \emptyset$
28 **foreach** $T_c \in \mathcal{T}_c, T \in \mathcal{T}$ **do** // Enumerate all possible v in some T which T_c can be attached to
29 **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.
30 **foreach** *child d of the root of T_c* **do**
31 $\mathcal{T}'' \leftarrow \emptyset$
32 **foreach** $T' \in \mathcal{T}', u \in T'$ *such that* $\kappa(d) \subseteq \chi(u)$ **do**
33 $T'' \leftarrow T'$ with T_d attached as a child of u
34 Add T'' to $\mathcal{T}''$
35 $\mathcal{T}' \leftarrow \mathcal{T}''$
36 **else**
37 **foreach** $u \in T$ *such that* $\kappa(c) \subseteq \chi(u)$ **do**
38 $T' \leftarrow T$ with T_c added as a child of u
39 Add T' to $\mathcal{T}'$
40 $\mathcal{T} \leftarrow \mathcal{T}'$
41 **return** $\mathcal{T}$
42 **return** $\bigcup_{T \in \text{enumRec}(r)}$ rerootings$(T, \emptyset, \emptyset)$

which contains one edge connecting v_2 and v_3, and they are re-rootings 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 $\kappa(v_5) \neq \chi(r)$. It can be attached as 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 spanning tree generated by the Prüfer sequence is rooted at R_2, and R_5 is attached as a child of R_2. Fig. 6d is a similar example where the spanning tree is rooted at R_3, and R_5 is attached as a child of R_3. And Fig. 6e is based on the same spanning tree as for Fig. 6d, but R_5 is attached as a child of R_4 instead of R_3.

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

Theorem 4.11 (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 4.12 (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 4.12, for each join tree T, we can pinpoint exactly one way in which T can be enumerated by the algorithm based on the meta-decomposition M. Noting also that the number of children of each node on the meta-decomposition M is bounded by the fan-out $f(M)$, we get the following running time of Algorithm 2, following the control flow.

Theorem 4.13 (Running time). *Given a meta-decomposition M of an acyclic hypergraph H, Algorithm 2 enumerates all possible join trees of H in $O(f(M) \cdot |V(H)| \cdot |\mathcal{T}|)$ time, where $|\mathcal{T}|$ is the number of all possible join trees.*

In fact, this algorithm can be written in a way that enumerates join trees in polynomial delay, taking polynomial space. The details are given in the appendix in our technical report [25].

5 Cost-Based Optimization from Meta-Decompositions

Although 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 2.1. 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 re-branching possibilities, and (4) determining the local join order for each node and its neighbours. In this section, we detail the cost-based optimization procedure, which systematically addresses all four factors.

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 queries 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 will show in the appendix of the technical report, accommodating a possible re-branching adds only a constant factor to the overall complexity.

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

input : A meta-decomposition $M = (V, r, E)$
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
  $\mathsf{plan}(T_{p\to q}) \leftarrow \mathsf{optimizeLocal}(q, \{\mathsf{plan}(T_{q\to c}) \mid c \in C(q)\})$ // $C(q)$ is the set of all children of node q
$\mathsf{plan}(M) \leftarrow \mathsf{optimizeLocal}(r, \{\mathsf{plan}(T_{r\to c}) \mid c \in C(r)\})$
// Top-down traversal
foreach $c \in C(r)$ **do**
  $\mathsf{plan}(T_{c\to r}) \leftarrow \mathsf{optimizeLocal}(r, \{\mathsf{plan}(T_{r\to c'}) \mid c' \in C(r) \setminus \{c\}\})$
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**
    $\mathsf{plan}(T_{c\to q}) \leftarrow \mathsf{optimizeLocal}(q, \{\mathsf{plan}(T_{q\to p})\} \cup \{\mathsf{plan}(T_{q\to c'}) \mid c' \in C(r) \setminus \{c\}\})$
return *the plan* $\mathsf{plan}(T_{p\to q}) \bowtie \mathsf{plan}(T_{q\to p})$ *with the minimum cost among all* $(p, q) \in E(M)$

5.2 The Overall Algorithm

Given a meta-decomposition, our algorithm to produce a width-1 query plan is shown in Algorithm 3. This algorithm makes two traversals of the meta-decomposition M, one bottom-up and one top-down. In the bottom-up traversal, for each node q on the meta-decomposition, 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 induced query on subtree of M rooted at q. Then, 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 neighbours (parent and children) of q except c, which gives the optimal plan to join all relations in $T_{c\to q}$. 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 $(p, q) \in E(M)$. Alternatively, we can think of this algorithm as *finding the optimal rerooting* of M, so that the query is evaluated by (1) replacing each node with a join with its children and (2) removing its children, in a bottom-up order.

The function optimizeLocal called in the algorithm optimizes the local join for each node and its neighbours. We will elaborate on this in Section 5.3.

5.3 Optimizing the Local Join with Neighbours

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 neighbours (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 $i = 1, \ldots, |C_q|$, 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 a Cartesian product $S_1 \times S_2 \times \cdots \times S_n$—which is already shown to be NP-hard [32]—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 [28, 34], the problem can be solved in $\widetilde{O}\left(2^{|C_q|}\right)^2$, where, in our context, $|C_q|$ equals the fan-out of q on the meta-decomposition. However, as demonstrated 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) [14], which iteratively joins the neighbor that results in the smallest intermediate result and results in a left-deep plan. This method is outlined in Algorithm 8 in the appendix of our full technical report [25].

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

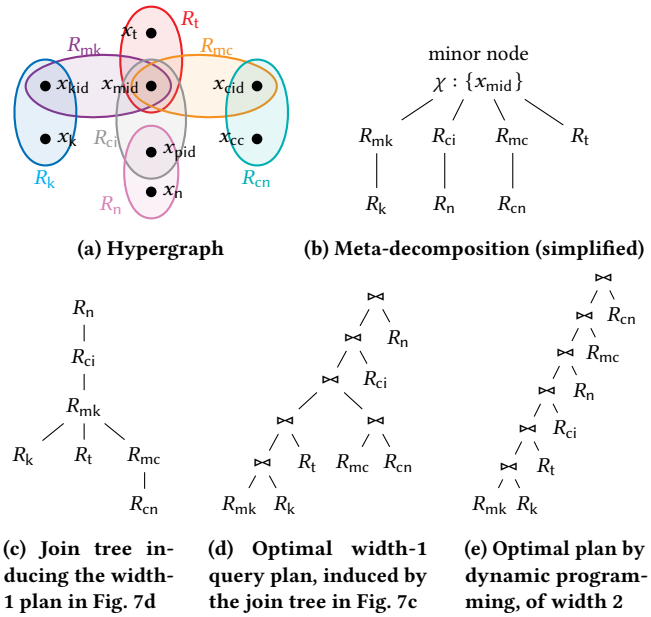
Our framework naturally incorporates projection pushdown for early cardinality reduction. When building a plan for a subquery, such as $\text{plan}(T_{p \to q})$ in Algorithm 3, we can immediately project the result to only the attributes that are necessary for subsequent operations. Specifically, after computing the joins within the subtree T_q, we only need to retain (1) attributes required for the final output $(O \cup \chi(T_{p \to q})) \cup (\chi(p) \cap \chi(q))$, 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 there exists one relation that 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 often struggle to identify these opportunities for projection pushdown, as they lack the comprehensive structural view provided by the meta-decompositions and the width-1 query plans.

Example 5.1. 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

$$\begin{aligned} Q \leftarrow \pi_{x_n}(R_{\text{ci}}[x_{\text{mid}}, x_{\text{pid}}] \bowtie R_{\text{cn}}[x_{\text{cid}}, x_{\text{cc}}] \bowtie \sigma_{x_k=\ldots}(R_{\text{k}}[x_{\text{kid}}, x_{\text{k}}]) \\ \bowtie R_{\text{mc}}[x_{\text{mid}}, x_{\text{cid}}] \bowtie R_{\text{mk}}[x_{\text{mid}}, x_{\text{kid}}] \\ \bowtie \sigma_{x_n=\ldots}(R_{\text{n}}[x_{\text{pid}}, x_{\text{n}}]) \bowtie R_{\text{t}}[x_{\text{mid}}, x_{\text{t}}]). \end{aligned}$$

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* $((R_{\text{mk}} \bowtie R_{\text{k}}) \bowtie R_{\text{t}}) \bowtie R_{\text{ci}}$ *has to be projected to three attributes,* x_{mid}, x_{pid}, *and* x_{n}—the former two are join keys, and x_{n} is the output attribute. Executing these plans, we indeed observe that *width-1 plan has a* 1.72x *speedup over the width-2 plan.*



(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**

Figure 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 5.1

Like evaluating join-projection queries, one can modify the cost model and utilize a join-tree-based query evaluation algorithm that incorporates cost-based optimization through meta-decomposition. We hope this work inspires further research in related areas and encourages the integration of these theoretically desirable query evaluation techniques in practical systems.

6 Experimental Evaluation & Analysis

In this section, we present experimental results to provide clear and insightful answers to the following research questions:

(RQ1) How effective are width-1 query plans, compared to the optimal query plan?

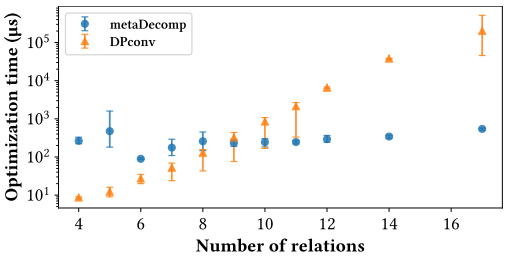
(RQ2) How effective is query optimization using meta-decompositions?

(RQ3) How effectively can the proposed techniques enhance structural-based evaluation methods?

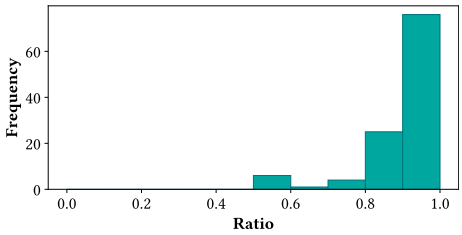
6.1 Experimental Setup

6.1.1 Experimental Environment. Experiments are conducted on an Apple M4 Pro machine with 14 cores and 48 GB RAM, running

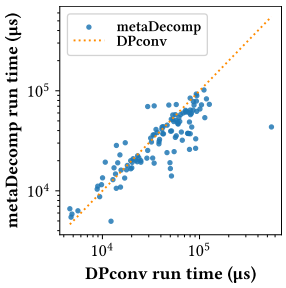
[2]Suppressing poly-logarithmic factors, and assuming the largest join cardinality is polynomial to the query size



(a) Optimization time. For each number of relations, the point shows the mean, and the error bar shows the range of values observed.

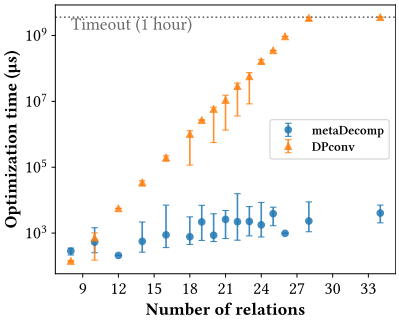


(b) Histogram of the ratio between the costs of the globally optimal plan and the optimal width-1 plan. A ratio of 1 indicates they have the same cost, and is included in the bar between 0.9 and 1.0.

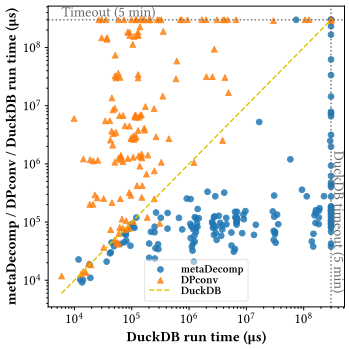


(c) Total run time (optimization plus execution)

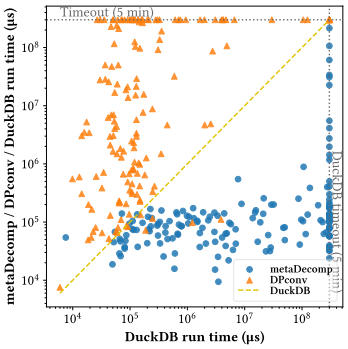
Figure 8: Comparison of metaDecomp and DPconv in the original Join Order Benchmark (JOB), given exact cardinalities



(a) Optimization time. For each number of relations, the point shows the mean, and the error bar shows the range of values observed.



(b) Total run time, given estimated cardinalities by DuckDB's `EXPLAIN`



(c) Total run time, given all-0 cardinality estimation

Figure 9: Comparison of metaDecomp, DPconv on queries in JOBLarge, with DuckDB as the baseline for the total run time

macOS Sequoia 15.5. We use DuckDB version 1.2.2, Scala 3.6.3, GraalVM 17.0.12, and LLVM 20.1.3 in the experiments.

We implement metaDecomp in Scala. It first computes the metadecomposition for a given query (Section 4) and then applies our cost-based optimization algorithm (Section 5) to select the best width-1 query plan. During the experiments, we compare the runtime of the query plans selected by (1) metaDecomp, (2) the state-of-the-art dynamic programming optimizer DPconv [34], and (3) DuckDB's native optimizer, which uses DPccp [28] for queries with up to 12 relations and Greedy Operator Ordering (GOO) [14] otherwise. Query plans by metaDecomp and DPconv are enforced by rewriting queries into a sequence of subqueries, creating temporary views, and are executed within DuckDB.

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 runtime measurements. The full dataset is preloaded into the DuckDB database, and the estimated cardinalities are preloaded into memory, with all loading time excluded from optimization timing.

6.1.2 Queries, Benchmarks, and Cardinalities. We evaluate our approach on the standard Join Order Benchmark (JOB) [27]. To test performances on larger and more complex workloads, we introduce a new benchmark called *JOBLarge*, which extends JOB to simulate practical but larger analytical scenarios, where analysts or automated systems often combine independent analytical queries into one larger query to gain deeper insights. Each query in JOBLarge is generated by selecting two queries from the JOB that share at least one common output attribute and then merging these two queries into a single query, with a join on the shared output attribute. JOBLarge significantly increases query complexity, with queries containing up to 34 relations (average 18 relations), compared to the standard JOB benchmark (maximum 17, average 8). All queries used are available in our repository [24] as SQL files.

For the JOB benchmark, we provide true cardinalities of possible intermediate join results to both metaDecomp and DPconv (precomputed by running appropriate `COUNT(*)` queries). This gives DPconv perfect information to identify the optimal plan, and allows metaDecomp to determine the optimal width-1 plan. For the JOBLarge benchmark, computing exact cardinalities for all possible intermediate joins is infeasible due to the much larger query sizes. So we instead use estimated cardinalities via DuckDB's `EXPLAIN` command. Note, however, that these estimates can be highly inaccurate and may not reflect the true relative sizes of subqueries.

6.2 Results

6.2.1 (RQ1) & (RQ2) Efficiency of width-1 query plans & optimization using metaDecomp.

On the original Join Order Benchmark. The results are shown in Fig. 8. We first compare the optimization time of metaDecomp and DPconv in Fig. 8a, where DPconv will find the optimal query plan

under the cost model. The optimization time of metaDecomp remains almost constant as the number of relations increases, whereas DPconv shows an exponential trend of growth, and its optimization time starts to overtake that of metaDecomp from approximately 9 relations. Nevertheless, even though metaDecomp restricts its search to width-1 plans, as shown in Fig. 8b, it *consistently finds query plans with costs comparable to the optimal ones found by* DPconv. Taking into account both optimization and execution time, we can observe, in Fig. 8c, that metaDecomp shows an advantage overall, especially for queries that are more complex, for which DPconv takes a long time. Overall, compared to DPconv, metaDecomp *delivers a speedup of* 1.21x *on average (geometric mean), and* 12.66x *maximum*. This demonstrates that finding and executing width-1 plans are indeed highly efficient in practice.

On JOBLarge. With larger, more complex queries in JOBLarge, the results are shown in Fig. 9. We increase the timeout to 1 hour to examine the trends of optimization time, as shown in Fig. 9a. The results remain consistent with the observation with original JOB queries, where the optimization time of DPconv grows exponentially, but that of metaDecomp remains almost constant. Even with 34 relations, for which DPconv is not able to terminate within 1 hour, metaDecomp finds a plan in merely 2 ms.

We continue to show the total run time that includes both optimization and execution. On these larger queries, metaDecomp shows even clearer speedups, as, for many of the JOBLarge queries, DPconv fails to find any valid plan within the time limit of 5 minutes, again highlighting that traditional exponential-time dynamic-programming-based optimizers do not scale to such large queries. For these queries, we also include the time of executing the queries directly in DuckDB, which includes DuckDB's internal optimization time based on similar cardinality estimations. And metaDecomp turns out to have a 1.34x average speedup over DuckDB's optimizer.

Nevertheless, we also observe a few JOBLarge queries (045, 073, 126, and 153) for which the plans by metaDecomp could not complete within the time limit either. We discover that these queries had extremely inaccurate, even highly misleading, cardinality estimates. For example, some intermediate results had actual cardinalities over 10^9, yet the estimate by DuckDB's `EXPLAIN` command was 0. In this case, the cost information actually had a negative effect. Therefore, we ran an additional experiment *with all cardinality estimates set to* 0. In this scenario, metaDecomp has no reliable cost information and thus simply picks a random width-1 plan with minimal join-tree height. The results are shown in Fig. 9c. Under this condition, metaDecomp achieves an even higher average speedup of 1.75x over DuckDB's optimizer, and, importantly, all queries finish within the time limit. This result indicates that *even without accurate cardinality information, choosing a width-1 plan based solely on structural information can still yield a reasonably efficient execution plan.*

6.2.2 (RQ3) Effectiveness for structure-based optimization techniques. Finally, we demonstrate the effectiveness of metaDecomp for recently developed structure-based optimization techniques. The standard way to further optimize those techniques relies on selecting a join tree. Thus, we first evaluate metaDecomp's efficiency in *enumerating join trees*, compared to the standard GYO reduction[3].

[3]Using the implementation in https://github.com/hkustDB/SparkSQLPlus [12]

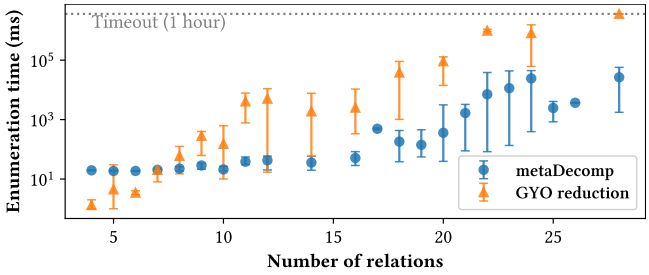


Figure 10: Time taken by metaDecomp and GYO reduction to enumerate join trees. For GYO reduction, queries that are not supported by the system are excluded.

As shown in Fig. 10, metaDecomp can exhibit up to *two orders-of-magnitude of speedup* over the traditional approach. In particular, for queries involving 28 relations, the traditional approach already takes over 1 hour, whereas metaDecomp can enumerate the entire join order search space in between 2 seconds and 1 minute.

A more effective approach to optimizing structure-based techniques is to apply cost-based optimization, possibly with a tailored cost model, directly on the meta-decomposition. This eliminates the need for join tree enumeration. In our second experiment, we replace the join tree in Yannakakis$^+$ [37] with the join tree selected by metaDecomp that induces the optimal width-1 query plan. *This modification achieves an average speedup of* 1.11x *on the tested JOB queries.* These results demonstrate that structure-based evaluation algorithms, such as Yannakakis$^+$, can benefit from our cost-based optimization framework, where using metaDecomp to guide join ordering leads to improved query execution efficiency.

7 Conclusions and Future Work

In this work, we introduce novel techniques for query optimization by leveraging structural properties via meta-decompositions. They offer promising opportunities to integrate theoretically-desirable structure-based optimization strategies into practical database systems, to efficiently evaluate large, complex queries.

A natural next step is to extend our width notion and meta-decomposition-based framework to accommodate cyclic queries. While width-1 query plans are infeasible for cyclic queries based on Theorem 3.4, we hope to generalize our framework to identify minimal-width query plans and represent all minimal-width hypertree decompositions. In addition, improving cardinality estimation methods is crucial, as we have shown experimentally that misleading estimation can significantly impact the performance. Recent research [10, 43] shows promising results for upper-bound estimators to prevent severe underestimations, which, as we encounter in our experiments, can dramatically mislead query optimization. With the help of these advances, we aim to develop efficient, reliable cardinality estimators suitable within our proposed framework.

References

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

[2] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. Faq: Questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, page 13–28, New York, NY, USA, 2016. Association for Computing Machinery.

[3] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '17, page 429–444, New York, NY, USA, 2017. Association for Computing Machinery.

[4] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

[5] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic*, pages 208–222, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[6] Liese Bekkers, Frank Neven, Stijn Vansummeren, and Yisu Remy Wang. Instance-optimal acyclic join processing without regret: Engineering the yannakakis algorithm in column stores. *arXiv preprint arXiv:2411.04042*, 2024.

[7] Kristin Bennett, Michael C. Ferris, and Yannis E. Ioannidis. A genetic algorithm for database query optimization. Technical Report #1004, Center for Parallel Optimization, 1991.

[8] Altan Birler, Alfons Kemper, and Thomas Neumann. Robust join processing with diamond hardened joins. *Proc. VLDB Endow.*, 17(11):3215–3228, July 2024.

[9] Altan Birler, Mihail Stoian, and Thomas Neumann. Optimizing linearized join enumeration by adapting to the query structure. *Datenbanksysteme für Business, Technologie und Web (BTW 2025)*, pages 171–194, 2025.

[10] Walter Cai, Magdalena Balazinska, and Dan Suciu. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 18–35, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Sophie Cluet and Guido Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In Georg Gottlob and Moshe Y. Vardi, editors, *Database Theory — ICDT '95*, pages 54–67, Berlin, Heidelberg, 1995. Springer.

[12] Binyang Dai, Qichen Wang, and Ke Yi. SparkSQL+: Next-generation Query Planning over Spark. In *Companion of the 2023 International Conference on Management of Data*, pages 115–118, Seattle WA USA, June 2023. ACM.

[13] Shaleen Deep, Hangdong Zhao, Austen Z. Fan, and Paraschos Koutris. Output-sensitive conjunctive query evaluation. *Proc. ACM Manag. Data*, 2(5), November 2024.

[14] Leonidas Fegaras. A new heuristic for optimizing large queries. In Gerald Quirchmayr, Erich Schweighofer, and Trevor J.M. Bench-Capon, editors, *Database and Expert Systems Applications*, pages 726–735, Berlin, Heidelberg, 1998. Springer.

[15] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. Hypertree decompositions: Questions and answers. In Tova Milo and Wang-Chiew Tan, editors, *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*, pages 57–74. ACM, 2016.

[16] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In Dieter Kratsch, editor, *Graph-Theoretic Concepts in Computer Science, 31st International Workshop, WG 2005, Metz, France, June 23-25, 2005, Revised Selected Papers*, volume 3787 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2005.

[17] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.

[18] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. *J. Comput. Syst. Sci.*, 66(4):775–808, 2003.

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

[20] Goetz Graefe and William J. McKenna. The Volcano optimizer generator: extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 209–218, Vienna, Austria, 1993. IEEE Comput. Soc. Press.

[21] Marc H. Graham. On the universal relation. Technical report, University of Toronto, 1979.

[22] Xiao Hu. Output-optimal algorithms for join-aggregate queries. *Proc. ACM Manag. Data*, 3(2), June 2025.

[23] Xiao Hu and Qichen Wang. Computing the difference of conjunctive queries efficiently. *Proc. ACM Manag. Data*, 1(2), June 2023.

[24] Zhekai Jiang, Qichen Wang, and Christoph Koch. metaDecomp: Succinct structure representations for efficient query optimization. GitHub repository, https://github.com/epfldata/metaDecomp.

[25] Zhekai Jiang, Qichen Wang, and Christoph Koch. Succinct structure representations for efficient query optimization. Technical report, https://github.com/epfldata/metaDecomp/blob/main/technical-report.pdf, 2025.

[26] Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. Ajar: Aggregations and joins over annotated relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, page 91–106, New York, NY, USA, 2016. Association for Computing Machinery.

[27] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *The VLDB Journal*, 27(5):643–668, October 2018.

[28] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 539–552, New York, NY, USA, June 2008. Association for Computing Machinery.

[29] Thomas Neumann and Bernhard Radke. Adaptive Optimization of Very Large Join Queries. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 677–692, New York, NY, USA, May 2018. Association for Computing Machinery.

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

[31] Francesco Scarcello, Gianluigi Greco, and Nicola Leone. Weighted hypertree decompositions and optimal query plans. *Journal of Computer and System Sciences*, 73(3):475–506, May 2007.

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

[33] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD '79, pages 23–34, New York, NY, USA, May 1979. Association for Computing Machinery.

[34] Mihail Stoian and Andreas Kipf. Dpconv: Super-polynomially faster join ordering. *Proc. ACM Manag. Data*, 2(6), December 2024.

[35] Bennet Vance. *Join-order optimization with Cartesian products*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1998.

[36] Bennet Vance and David Maier. Rapid bushy join-order optimization with Cartesian products. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, pages 35–46, New York, NY, USA, June 1996. Association for Computing Machinery.

[37] Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. Yannakakis+: Practical acyclic query evaluation with theoretical guarantees. *Proc. ACM Manag. Data*, 3(3), June 2025.

[38] Qichen Wang, Qiyao Luo, and Yilei Wang. Relational algorithms for top-k query evaluation. *Proc. ACM Manag. Data*, 2(3), May 2024.

[39] Qichen Wang and Ke Yi. Conjunctive queries with comparisons. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 108–121, New York, NY, USA, 2022. Association for Computing Machinery.

[40] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. 2024.

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

[42] C. T. Yu and M. Z. Özsoyoğlu. 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*, pages 306–312. IEEE, 1979.

[43] Haozhe Zhang, Christoph Mayer, Mahmoud Abo Khamis, Dan Olteanu, and Dan Suciu. Lpbound: Pessimistic cardinality estimation using lp-norms of degree sequences. *Proc. ACM Manag. Data*, 3(3), June 2025.

[44] Hangdong Zhao, Austen Z. Fan, Xiating Ouyang, and Paraschos Koutris. Conjunctive queries with negation and aggregation: A linear time characterization. *Proc. ACM Manag. Data*, 2(2), May 2024.

A Missing Proofs in Section 3

A.1 Proof of Theorem 3.1

Let $p^* \in V(\mathcal{P})$ be a node in the plan where the maximum width is achieved, i.e., $w(p^*) = w(\mathcal{P})$. By the definition of $w(p^*)$, there must exist a set of relations $S \subseteq E(H_{p^*})$ such that $|S| = w(p^*)$ and $I(p^*) \subseteq \bigcup_{e \in S} e$. Therefore, the result of $Q(p^*)$ must be a subset of the join of all relations in S. In the worst-case scenario, where all relations in S share no common attributes and their join is a Cartesian product, the size of this join is $O(N^{|S|})$, which gives an upper bound of $O(N^{w(\mathcal{P})})$.

A.2 Proof of Theorem 3.3

When $Q(p)$ only involves one relation, $Q(p) = q$ holds for all join trees because of the coverage condition of a join tree. Let T be the join tree satisfies the condition, for any $q \in V(T)$, the interface I_q between the induced hypergraph H_q for subtree T_q rooted by q and the residual hypergraph $\overline{H_q}$ satisfies $I_q \subseteq q$, because of the connectedness condition of the join tree. In addition, $Q(p) = Q(q)$ implies the induced hypergraph $H_p = H_q$, therefore, for every $p \in \mathcal{P}$, we can find a set $S = \{q\}$ that covers I_p, and $w(p) = 1$ holds for p, which completes the proof of if-direction.

For the only-if direction, given a width-1 query plan P, we can construct such a join tree T with induction. For any join operation $p \in V(\mathcal{P})$. Assuming there exists a valid join tree for both of the two child nodes c_1, c_2, denoted T_{c_1}, T_{c_2}. Let $q_1 \in V(T_{c_1})$ be the relation that covers the interface I_{c_1} and $q_2 \in V(T_{c_2})$ covers I_{c_2}, we make q_1 be the root of T_{c_1} and q_2 be the root of T_{c_2}, then $I_{c_1} \cap I_{c_2} = q_1 \cap q_2$, and we can merge T_1 and T_2 to obtain T' by connecting q_1 and q_2. In addition, there exists $q \in V(T_{c_1}) \cup V(T_{c_2})$ since $w(p) = 1$, such tree T' is a valid join tree for node p. Combining with the base case where T contains only one node, we then complete the proof.

A.3 Proof of Theorem 3.8

Following Theorem 3.3, we have

Lemma A.1. *For all width-1 query plans $\mathcal{P}$ induced by a join tree T, for all non-leaf nodes $t \in V(T)$ with distinct children c_1 and c_2, if $p_1, p_2 \in V(\mathcal{P})$ are the nodes on $\mathcal{P}$ such that $Q(p_1) = Q(c_1)$ and $Q(p_2) = Q(c_2)$, then there exists no $q \in \mathcal{P}$ such that $Q(q) = Q(p_1) \bowtie Q(p_2) = Q(c_1) \bowtie Q(c_2)$.*

Proof. Suppose there is such a q, then by Definition 3.2, there would have been an edge between c_1 and c_2 on T, contradicting with the assumption that c_1 and c_2 are distinct children of some node t. □

This means that, given a width-1 query plan $\mathcal{P}$ induced by a join tree T, for each $t \in V(T)$, with children $c_1, \ldots, c_{f(t)}$ and the corresponding $p \in V(\mathcal{P})$ such that $Q(t) = Q(p)$, if we consider the sub-query-plans $\mathcal{P}_{c_1}, \ldots, \mathcal{P}_{c_{f(t)}}$, induced by the sub-join-trees $T_{c_1}, \ldots, T_{c_{f(t)}}$ rooted at the children of t, each as a single relation, then the sub-query-plan $\mathcal{P}_p$ rooted at p is a *left-deep plan* consisting of $t, \mathcal{P}_{c_1}, \ldots, \mathcal{P}_{c_{f(t)}}$, and t is an operand of the deepest join operation.

This can be done by a dynamic programming algorithm, as shown in Algorithm 4. In the algorithm, $\mathsf{children}(v)$ is the set of all children of v.

Algorithm 4: Finding the optimal width-1 query plan given a join tree of an acyclic query

input : A join tree T
output : The optimal width-1 query plan induced by T

foreach $v \in V(T)$, *in bottom-up order* **do**
  foreach *child* c *of* v **do**
    if $\lambda(c) \neq \emptyset$ **then**
      $P(v, \{c\}) \leftarrow P(c, \mathsf{children}(c) \cup \{c\})$
    else
      $P(v, \{c\}) \leftarrow P(c, \mathsf{children}(c))$
  if $\lambda(v) \neq \emptyset$ **then**
    $P(v, \{v\}) \leftarrow$ query plan with one node v
  if $\lambda(v) = \emptyset$ **then**
    $S \leftarrow 2^{\mathsf{children}(v)} \setminus \emptyset$
  else
    $S \leftarrow \{T \cup \{v\} \mid T \in 2^{\mathsf{children}(v)} \setminus \emptyset\}$
  foreach $T \in S$, *ordered by increasing size* **do**
    $P(v, T) \leftarrow \operatorname{argmin}_{p \in \{P(v, T \setminus \{u\}) \bowtie P(v, \{u\}) \mid u \in T\}} \mathsf{cost}(p)$
return $P(\mathsf{root}(T), S(\mathsf{root}(T)))$

In the algorithm, the outer loop (Line 1) is executed $|V(T)| = |Q|$ times. The complexity of the loop body is dominated by enumerating all possible T (Line 13) and $u \in T$ (Line 14). Noting that the number of children of v is bounded by the fan-out $f(p)$, the complexity of this part is given by

$$\begin{aligned}
\sum_{i=1}^{f(v)} \binom{f(v)}{i} \times i &= \sum_{i=1}^{f(v)} \frac{f(v)!}{i!(f(v)-i)!} \times i \\
&= \sum_{i=1}^{f(v)} \frac{f(v)!}{(i-1)!(f(v)-i)!} \\
&= \sum_{i=1}^{f(v)} f(v) \times \frac{(f(v)-1)!}{(i-1)!((f(v)-1)-(i-1))!} \\
&= \sum_{i=1}^{f(v)} f(v) \times \binom{f(v)-1}{i-1} \\
&= f(v) \sum_{i=0}^{f(v)-1} \binom{f(v)-1}{i} \\
&= f(v) 2^{f(v)-1} \\
&\le \max_{v \in V(T)} f(v) 2^{f(v)-1} = f(T) 2^{f(T)-1}.
\end{aligned}$$

So, overall, the complexity is $O(f(T)2^{f(T)-1}|Q|)$.

B Missing Materials in Section 4

B.1 Missing Algorithms that Algorithm 2 is Based on

These algorithms are shown in Algorithm 5 and Algorithm 6.

Algorithm 5: Prüfer(V): Enumerate all non-homomorphic trees, up to rerooting, with vertices V, using Prüfer sequences

input : Set of vertices V
output : A set $\mathcal{T}$ of all non-isomorphic trees with vertices V

1 Arbitrarily order the vertices in V and let v_i be the i-th vertex in V under this ordering
2 $\mathcal{T} \leftarrow \emptyset$
3 **foreach** *sequence* $P = p_1, \ldots, p_{|V|-2}$ *of* $|V|-2$ *integers, each with value in range* $[1, |V|]$ **do**
4 **foreach** $i = 1, \ldots, |V|$ **do**
5 $d(i) \leftarrow 1$
6 $S \leftarrow [1, |V|]$ // maintains the set of indices i with $d(v_i) = 1$, in increasing order
7 **foreach** $i = 1, \ldots, |V|-2$ **do**
8 $d(s_i) \leftarrow d(p_i) + 1$
9 $S \leftarrow S \setminus \{i\}$
10 Initialize a tree T with set of vertices V and no edge
11 **foreach** $i = 1, \ldots, |V|-2$ **do**
12 $j \leftarrow$ remove the least element in S
13 $d(i) \leftarrow d(i) - 1$
14 $d(j) \leftarrow d(j) - 1$
15 **if** $d(i) = 1$ **then**
16 Add i into S
17 In T, add v_j as a child of v_i
18 $i, j \leftarrow$ the remaining two elements in S
19 In T, add v_j as a child of v_i // the direction does not matter
20 $\mathcal{T} \leftarrow \mathcal{T} \cup \{T\}$
21 **return** $\mathcal{T}$

Algorithm 6: rerootings(T, p, κ): Enumerate all valid rerootings of a tree T such that the new root r' has $\kappa \subseteq \chi(r')$

input : A tree T, optionally the previous parent p of the root r of T before the previous rotation, the expected key κ
output : A set $\mathcal{T}$ of all valid rerootings of T such that the new root r' has $\kappa \subseteq \chi(r')$

```
1 T ← ∅
2 foreach child c of the root r of T do
3 |   if κ ⊆ χ(c) and c ≠ p then
4 |   |   T' ← T rerooted to c
5 |   |   T ← T ∪ {T'} ∪ rerootings(T', r, κ)
6 return T
```

B.2 Proof of Lemma 4.6

We start with the following lemma:

Lemma B.1. *Given an acyclic hypergraph H, if there exist two ears $e_1, e_2 \in E(H)$ such that $o(e_1, H) = o(e_2, H)$, then $o(e_1, H) = o(e_2, H) = e_1 \cap e_2$.*

Proof. ($\subseteq$): $o(e_1, H) = e_1 \cap (\cup(E(H) \setminus \{e_1\})) \subseteq e_1$, and, similarly, $o(e_2, H) \subseteq e_2$. So it follows that $o(e_1, H) = o(e_2, H) \subseteq e_1 \cap e_2$.
($\supseteq$): Since $e_1 \in E(H) \setminus \{e_2\}$, we have $o(e_1, H) = o(e_2, H) = e_2 \cap (\cup(E(H) \setminus \{e_2\})) \supseteq e_1 \cap e_2$. □

A hypergraph is acyclic if and only if it has a join tree, satisfying conditions (H_1)–(H_3). Since e_1 and e_2 are ears in H such that $o(e_1, H) = o(e_2, H) \subseteq e_2$, in the GYO algorithm for H, e_1 can already be removed, with e_2 being its witness. Therefore, there exists a join tree T of H where there is a leaf node p_1 with $\lambda(p_1) = \{e_1\}$, and its parent p_2 has $\lambda(p_2) = \{e_2\}$. We construct a tree T' that is exactly the same as T except that p_1 and p_2 are merged into a single node m with $\lambda(m) = \{o(e_1, H)\}$ and $\chi(m) = o(e_1, H)$, and we show that T' is a valid join tree of H'. (H_1) and (H_3) are satisfied by construction. It remains to show that T' satisfies (H_2). Suppose by way of contradiction that this is not the case. By (H_2) on the join tree T, this means, on T', either (1) there exist $s, t \in V(T')$ such that m is on the path between s and t, and $\chi(s) \cap \chi(t) \not\subseteq \chi(m)$, or (2) there exists $s \in V(T')$ and t on the path between s and m, such that $\chi(s) \cap \chi(m) \not\subseteq \chi(t)$.

For case (1), since, by Lemma B.1, $\chi(m) = o(e_1, H) = e_1 \cap e_2 = \chi(p_1) \cap \chi(p_2)$, if $\chi(s) \cap \chi(t) \not\subseteq \chi(m)$, this means either $\chi(s) \cap \chi(t) \not\subseteq \chi(p_1)$ or $\chi(p) \cap \chi(q) \not\subseteq \chi(p_2)$. On T, since p_1 is a leaf node, p_2 should be on the path between s and t. By (H_2) for T, we have $\chi(s) \cap \chi(t) \subseteq \chi(p_2)$. So it can only be that $\chi(s) \cap \chi(t) \not\subseteq \chi(p_1)$, i.e., there exists some $v \in \chi(s) \cap \chi(t) \subseteq \chi(p_2)$ such that $v \notin \chi(p_1)$. But then, $v \in o(e_2, H)$ but $v \notin o(e_1, H)$, contradicting $o(e_1, H) = o(e_2, H)$.

For case (2), we first note that $\chi(s) \cap o(e_1, H) = \chi(s) \cap e_1$. This is because ($\subseteq$) $o(e_1, H) = e_1 \cap (\cup(E(H) \setminus \{e_1\})) \subseteq e_1$, and ($\supseteq$) $\chi(s) \cap e_1 \subseteq (\cup(E(H) \setminus \{e_1\})) \cap e_1 = o(e_1, H)$. Then, $\chi(s) \cap \chi(m) = \chi(s) \cap o(e_1, H) = \chi(s) \cap e_1 = \chi(s) \cap \chi(p_1)$. So, $\chi(s) \cap \chi(p_1) = \chi(s) \cap \chi(m) \not\subseteq \chi(p_2)$. Since p_1 is a leaf and p_2 is the parent of p_1 on T, p_2 must be on the path between s and p_1, so this contradicts (H_2) for T.

B.3 Proof of Theorem 4.7

We start by considering the while-loop (Line 3). At the start of each iteration, an ear must exist because the hypergraph H' must be acyclic, as guaranteed by Lemma 4.6. Furthermore, $E(H')$ always decreases in size by at least 1 in each iteration, because we only remove ears from $E(H')$, except when we have $e_1, e_2 \in S$ such that $o(e_1, H') = o(e_2, H')$, in which case we remove all (at least 2) those ears sharing the same intersection but add only one special hyperedge e_m to $E(H')$. Therefore, the body of the while loop is executed at most $|E(H)|$ times.

For each iteration of the while loop, all sets S (Line 4) can be enumerated in $O(|E(H')|)$, if we maintain two hash maps: (1) T_1, which maps each $o \subseteq V(H')$ to all hyperedges $e \in E(H')$ such that $o(e, H') = o$, and (2) T_2, which maps each $v \in V(H')$ to the number of hyperedges $e \in E(H')$ such that $v \in e$. Each time a hyperedge e is added or removed from $E(H')$, we update T_2 for each $v \in e$, and then update T_1 if some $v \in e$ is/was covered by exactly one hyperedge. These updates can be done in $O(|E(H')|)$, which is $O(|E(H)|)$.

With these observations, the runtime $O(|E(H)|^3)$ can be obtained by following the control flow.

B.4 Proof of Theorem 4.8

We show that M output by the algorithm satisfies (H_1)–(H_5).

(H_1). For each $e \in E(H)$, the algorithm has to remove e from $E(H')$ on Line 10, Line 15, or Line 21 before it terminates. In any of these cases, the algorithm will create a node p in M with $\lambda(p) = \{e\}$, on Line 9, Line 14, or Line 20.

Before proceeding with (H_2), we have the following lemmas:

Lemma B.2. *For each edge $(q, p) \in E(M)$, where $\chi(p) = \{e_p\}$ and $\chi(q) = \{e_q\}$, we have either*

(1) e_p is removed as ear by the algorithm before e_q is, or
(2) $\lambda(q) = \emptyset$ and $\kappa(q) = \chi(q)$

Additionally, $\kappa(p) = \chi(p) \cap \chi(q)$.

Proof. We first have $\kappa(p) \subseteq \chi(p)$ by definition on Lines 9, 14, 18, and 20. Since the algorithm adds the edge (q, p), by Lines 24 and 27, we have $\kappa(p) \subseteq \chi(q)$. Therefore, $\kappa(p) \subseteq \chi(p) \cap \chi(q)$.

We then consider the relative order in which e_p and e_q are removed from $E(H')$ in the algorithm.

(1) If e_p is removed before e_q is, then at the time e_p is removed, $e_q \in E(H')$, and therefore $\chi(p) \cap \chi(q) = e_p \cap e_q \subseteq e_p \cap (\cup(E(H') \setminus \{e_p\})) = \kappa(p)$. Since also $\kappa(p) \subseteq \chi(p) \cap \chi(q)$, we have $\kappa(p) = \chi(p) \cap \chi(q)$.

(2) If e_p is removed after e_q is, then, symmetric to case (1), we have $\chi(p) \cap \chi(q) \subseteq \kappa(q)$. Then, since we have shown that $\kappa(p) \subseteq \chi(p) \cap \chi(q)$, we have $\kappa(p) \subseteq \chi(p) \cap \chi(q) \subseteq \kappa(q)$. By Line 24 and Line 27, this is only possible if $\lambda(q) = \emptyset$ and $\kappa(p) = \chi(q)$. Since $\kappa(p) \subseteq \chi(p)$ and $\kappa(p) = \chi(q)$, it is true that $\kappa(p) = \chi(p) \cap \chi(q)$. □

It follows that

Corollary B.3. *p is a descendant of q on M if and only if e_p is removed as ear by the algorithm before e_q is.*

(H_2). The algorithm iteratively adds edges (q, p) to attach a connected subtree M_p rooted at p to a connected subtree M^q that contains q. Suppose by way of contradiction that (H_2) is violated at the end, and suppose it is when processing $p \in V(M)$, that the addition of edge (q, p) violates this condition for the first time. Namely, M_p and M^q each satisfies (H_2), but the resulting M violates (H_2), i.e., for some $s, t \in V(M)$, there exists u on the path betwween s and t such that $\chi(s) \cap \chi(t) \not\subseteq \chi(u)$. Without loss of generality, we take the shortest possible path from s to t where the condition is violated, where u is the node immediately before t on the path. Then, $\kappa(t) = \chi(t) \cap \chi(u)$.

Since M_p and M^q each satisfies (H_2), the only way it can be violated is if $s \in V(M_p)$, i.e., s is a descendent of p, $t \in M^q$, and thus p and q are on the path between s and t. For the purpose of the proof, we think of the subtree M_p as being added to the subtree M^q one node at a time, in a top-down order, starting from the root p. We then show that each time a node is added, the connectedness condition is preserved. Let $\lambda(p) = \{e_p\}$ and $\lambda(t) = \{e_t\}$.

As the base case, when p is added as a child of q, we consider two cases on the relative order in which e_p and e_t are removed from $E(H')$ by the algorithm.

(1) If e_p is removed as the ear by the algorithm before e_t is, then $\chi(p) \cap \chi(t) = e_p \cap e_t \subseteq \kappa(p) \subseteq \chi(q)$. And, for all u between q and t on the path between p and t, we have $\chi(p) \cap \chi(t) = \chi(p) \cap \chi(t) \cap \chi(t) \subseteq \chi(q) \cap \chi(t) \subseteq \chi(u)$, by the connectedness condition on M^q.

(2) If e_t is removed as the ear by the algorithm before e_p is, then $\chi(p) \cap \chi(t) = e_p \cap e_t \subseteq \kappa(t)$. By Corollary B.3, t is a descendant of q, and so the parent u of t is on the path between p and t. Then, $\chi(p) \cap \chi(t) \subseteq \kappa(t) = \chi(t) \cap \chi(u) \subseteq \chi(u)$

As the induction step, assuming that the condition holds for the connected subtree with nodes in M^q and all p' that are added before s. Following a similar argument as for the base case, substituting q with the parent of s, we can also derive that $\chi(s) \cap \chi(t) \subseteq \chi(u)$ for all u.

(H_3'). This is guaranteed by construction on Lines 9, 14, 18, and 20.

(H_4)(a). By Lemma B.2, for each $p \in V(M)$ and its parent q, we have $\kappa(p) = \chi(p) \cap \chi(q)$. ($\subseteq$): $\kappa(p) = \chi(p) \cap \chi(q) \subseteq \chi(q)$. ($\supseteq$): For all $s \in V(M) \setminus V(M_p)$, since q is the parent of p, q has to be on the path between p and s. Then, by the connectedness condition (H_2) on M, we have $\chi(p) \cap \chi(s) \subseteq \chi(q)$. Therefore, $\kappa(p) = \chi(p) \cap \chi(q) \supseteq \chi(p) \cap \left(\bigcup_{s \in V(M) \setminus V(M_p)} \chi(s)\right) = \chi(p) \cap \chi(V(M) \setminus V(M_p))$.

(H_4)(b). This is guaranteed by Lines 24 and 27.

(H_5). Lines 4–11 and Lines 17–18 create such minor nodes when necessary. Lines 6–7 and the condition on Line 17 before creating a minor node on Line 18 ensures there are no duplicate minor nodes.

B.5 Proof of Theorem 4.9

Vertices $V(M)$ in M are either physical nodes or minor nodes. Let $V^P(M)$ be the set of physical nodes and $V^M(M)$ be the set of minor nodes, such that $V(M) = V^P(M) \cup V^M(M)$. Each time we create a physical node p with $\lambda(p) = \{e\}$ (Lines 9, 14, and 20), we remove e from $E(H')$ (Lines 10, 15, and 21). So there are at most $|E(H)|$ physical nodes, $|V^P(M)| \leq |E(H)|$. Now let us consider the number of minor nodes, $|V^M(M)|$. Note that each minor node $m \in V^M(M)$ must have at least two children (see Line 4 and Line 17), i.e., the fanout $f(m) \geq 2$. Then, the total number of vertices

$$\begin{aligned} |V(M)| &= |V^P(M)| + |V^M(M)| \\ &= |E(M)| + 1 = 1 + \sum_{p \in V(M)} f(p) \\ &> 1 + \sum_{m \in V^M(M)} f(m) \geq 1 + \sum_{m \in V^M(M)} 2 = 2|V^M(M)| + 1. \end{aligned}$$

Then, $|V^P(M)| + |V^M(M)| > 2|V^M(M)| + 1$, so $|V^M(M)| < |V^P(M)| - 1 = |E(H)| - 1$. So, $|V(M)| = |V^P(M)| + |V^M(M)| < |E(H)| + |E(H)| - 1 = 2|E(H)| - 1$. And $|E(M)| = |V(M)| - 1 \leq 2|E(H)| - 2$.

B.6 Proof of Theorem 4.11

We prove by showing that all trees T output by Algorithm 2 satisfy (H_1), (H_2), and (H_3).

For (H_1), note that this condition must already hold for the meta-decomposition M. In Algorithm 2, all physical nodes, for which $\lambda(r) \neq \emptyset$, will be kept in $\mathcal{T}$, on Line 23. Minor nodes, for which $\lambda(r) = \emptyset$, will no longer exist in the output. If $\kappa(r) \neq \chi(r)$, trees in $\mathcal{T}_p$, and thus $\mathcal{T}$, will not contain minor nodes (Line 10) but only nodes in the subtrees rooted at the children of r (Lines 11–20). If $\kappa(r) = \chi(r)$, the minor node r on Line 8 will eventually be removed and replaced by v on Line 33. Therefore, the trees output by Algorithm 2 contain only physical nodes. (H_3) then follows, by (H_3') for M.

It now remains to show (H_2), the connectedness condition. We prove by structural induction on vertices $v \in V(M)$, i.e., for each $v \in V(M)$, all trees T returned by the function enumRec satisfy (H_2). In the base case, for all leaf nodes v, v has to be physical, and the function simply returns a tree with one single node, so these conditions hold. For non-leaf nodes v, we assume as induction hypothesis that for all children c of v, all trees returned by the function call enumRec(c) satisfy (H_2). And we now show that it remains satisfied throughout the operations in enumRec(v).

We start with the trees in $\mathcal{T}$ constructed on Lines 3–23. If $\lambda(v) \neq \emptyset$, this vacuously holds as there is only one tree with one vertex r in $\mathcal{T}$ (Line 23). If $\lambda(v) = \emptyset$, we will show that it still holds for all T_c' (constructed on Lines 11–20), assuming that it holds for $T_c \in \mathcal{T}_c$ from the previous iteration. It suffices to show that, for each (T_p, c, T_c, d, T_d, u) processed in the loop, for all $p \in T_c$ and $q \in T_d$, all s on the path between p and q has $\chi(p) \cap \chi(q) \subseteq \chi(s)$. Let the root of T_c be r_c and the root of T_d be r_d. The path from p to q needs to go through u then r_d. For s on the path between p and u, we have $\chi(p) \cap \chi(q) \subseteq \chi(v) = \kappa(d) \subseteq \chi(u)$. Since T_c satisfies (H_2), we have $\chi(p) \cap \chi(q) \subseteq (\chi(p) \cap \chi(q)) \cap \chi(q) \subseteq \chi(u) \cap \chi(q) \subseteq \chi(s)$. For s on the path between r_d and q, we have $\chi(p) \cap \chi(q) \subseteq \chi(v) = \kappa(d) \subseteq \chi(r_d)$. Since T_d satisfies (H_2), we have $\chi(p) \cap \chi(q) \subseteq (\chi(p) \cap \chi(q)) \cap \chi(q) \subseteq \chi(r_d) \cap \chi(q) \subseteq \chi(s)$.

We now continue to show that lines Line 25–Line 40 preserve the condition (H_2) for all $T \in \mathcal{T}$, by showing that the condition is preserved for all $T' \in \mathcal{T}'$ after each child $c \in X$ is processed, assuming that all the previous $T \in \mathcal{T}$ satisfy (H_2).

Let the root of T_c be r_c. By the connectedness condition on M, we have $\chi(p) \cap \chi(q) \subseteq \chi(v)$, and $\chi(p) \cap \chi(q) \subseteq \chi(c)$. Since c is a child of v, $\chi(p) \cap \chi(q) \subseteq \chi(v) \cap \chi(c) = \kappa(c)$.

(1) If $\lambda(c) \neq \emptyset$ or $\kappa(c) \neq \chi(c)$, T_c is added directly as a child of u on T' (Line 38). For all $p \in V(T')$ and $q \in V(T_c)$, the path from p to q needs to go through u then r_c. For s on the path between p and u, we have $\chi(p) \cap \chi(q) \subseteq \kappa(c) \subseteq \chi(u)$. Since T' satisfies (H_2), we have $\chi(p) \cap \chi(q) \subseteq (\chi(p) \cap \chi(q)) \cap \chi(q) \subseteq \chi(u) \cap \chi(q) \subseteq \chi(s)$. For s on the path between r_c and q, we have $\chi(p) \cap \chi(q) \subseteq \kappa(c) \subseteq \chi(r_c)$. Since T_c satisfies (H_2), we have $\chi(p) \cap \chi(q) \subseteq (\chi(p) \cap \chi(q)) \cap \chi(q) \subseteq \chi(r_c) \cap \chi(q) \subseteq \chi(s)$.

(2) If $\lambda(c) = \emptyset$ and $\kappa(c) \neq \chi(c)$, for each child d of the root of T_c, we attach T_d as a child of u. Let the root of T_d be r_d. Then, $\kappa(d) \subseteq \chi(r_d)$ and $\kappa(d) \subseteq \chi(u)$. For all $p \in V(T')$ and $q \in V(T_d)$, the path from p to q needs to go through u then r_d. For s on the path between p and u, we have $\chi(p) \cap \chi(q) \subseteq \kappa(c) = \kappa(d) \subseteq \chi(u)$. Since T' satisfies (H_2), we have $\chi(p) \cap \chi(q) \subseteq (\chi(p) \cap \chi(q)) \cap \chi(q) \subseteq \chi(u) \cap \chi(q) \subseteq \chi(s)$. For s on the path between r_c and q, we have $\chi(p) \cap \chi(q) \subseteq \kappa(c) = \kappa(d) \subseteq \chi(r_d)$. Since T_c satisfies (H_2), we have $\chi(p) \cap \chi(q) \subseteq (\chi(p) \cap \chi(q)) \cap \chi(q) \subseteq \chi(r_d) \cap \chi(q) \subseteq \chi(s)$.

B.7 Proof of Theorem 4.12

We start from a few lemmas:

Lemma B.4. *For any non-root node p on a meta-decomposition M with parent q, $\kappa(p) = \chi(p) \cap \chi(q)$.*

Proof. ($\supseteq$): Since $q \in V(M) \setminus V(M_p)$, $\chi(p) \cap \chi(q) \subseteq \chi(p) \cap \chi(V(M) \setminus V(M_p)) = \kappa(p)$. ($\subseteq$): For any $v \in \kappa(p) = \chi(p) \cap \chi(V(M) \setminus V(M_p))$, i.e., any $v \in \chi(p) \cap \chi(s)$ for some $s \in V(M) \setminus V(M_p)$, the parent q of p lies on the path on M between p and s. Then, by the connectedness condition (H_2) for M, $v \in \chi(q)$. □

Lemma B.5. *For all $p \in V(M)$ with $\lambda(p) = \emptyset$ and $\kappa(p) = \chi(p)$, all children c of p satisfy $\kappa(c) = \chi(p)$.*

Proof. Suppose by way of contradiction that there exists such a $p \in V(T)$ with child c such that $\kappa(c) \neq \chi(p)$. By Lemma B.4, we have $\kappa(c) = \chi(c) \cap \chi(p) \subseteq \chi(p)$. Since $\kappa(p) = \chi(p) \neq \emptyset$, p is not the root of M and thus has a parent q. Again by Lemma B.4, we have $\kappa(p) = \chi(p) \cap \chi(q) \subseteq \chi(q)$. But then, $\kappa(c) \subseteq \chi(p) = \kappa(p) \subseteq \chi(q)$, contradicting ($H_4$)(b) since $\kappa(c) \neq \chi(p)$ but $q \in V(T) \setminus V(T_p)$. □

Lemma B.6. *Given a meta-decomposition M, if some non-root node $p \in V(M)$ has $\lambda(p) = \emptyset$ and $\kappa(p) = \chi(p)$, then its parent q has either $\lambda(q) \neq \emptyset$ or $\kappa(q) \neq \chi(q)$.*

Proof. Suppose by way of contradiction that, on a meta-decomposition M, there is $p \in V(M)$ such that $\lambda(p) = \emptyset$ and $\kappa(p) = \chi(p)$, and its parent q has $\lambda(q) = \emptyset$ and $\kappa(q) = \chi(q)$. Then, $\kappa(p) \subseteq \chi(q) = \kappa(q)$. Since $\kappa(q) = \chi(q) \neq \emptyset$, q has a parent as well. Let the parent of q be s. Then, $\kappa(q) = \chi(q) \cap \chi(s) \subseteq \chi(s)$. So, $\kappa(p) \subseteq \kappa(q) \subseteq \chi(s)$. By ($H_5$), $\kappa(p) \neq \kappa(q) = \chi(q)$, contradicting ($H_4$)(b). □

Lemma B.7. *Given a meta-decomposition M of an acyclic hypergraph H, for all $v \in V(M)$,*

(1) if $\lambda(v) = \emptyset$ and $\kappa(v) = \chi(v)$, then, for each join tree T of H, there exists some $S = \{p \in V(M) \mid \lambda(p) \neq \emptyset, \chi(v) \subseteq \chi(p)\}$ such that the set $V_v^P = \{p \in V(M_v) \mid \lambda(p) \neq \emptyset\} \cup S$ induces a connected subtree T_v of T;

(2) if $\lambda(v) \neq \emptyset$ or $\kappa(v) \neq \chi(v)$, then, for each join tree T of H, the set $V_v^P = \{p \in V(M_v) \mid \lambda(p) \neq \emptyset\}$, i.e., physical nodes on the subtree M_v rooted at v, induces a connected subtree T_v of T.

Proof. We prove by structural induction on M. In the base case, for leaf nodes $v \in V(M)$, there is only one node on the subtree M_v, so the statement is vacuously true. For non-leaf nodes $v \in V(M)$, we assume as induction hypothesis that the statement is true for all children c of v. Let c and d be the children of v such that $p \in V(M_c)$ and $q \in V(M_d)$. We now consider two cases of v.

(1) If $\lambda(v) = \emptyset$ and $\kappa(v) = \chi(v)$, then, for any s on the path between p and q on T such that $s \notin M_v$, we have $\chi(p) \cap \chi(q) \subseteq \chi(s)$ by the connectedness condition on T. By (the contrapositive of) Lemma B.6, $\lambda(c) \neq \emptyset$ or $\kappa(c) \neq \chi(c)$, so by the inductive hypothesis, V_c^P induces a connected subtree T_c of T. Similarly, V_d^P induces a connected subtree T_d of T. If $c = d$, then p and q are in the same subtree rooted at the same child of v, and the statement would hold. If $c \neq d$, on the path from p to q on T, let the last node on T_c be x and the first node on T_d be y. s is between x and y. Then, the path from c and d has to go through x and y, and hence also s. Therefore, $\chi(c) \cap \chi(d) \subseteq \chi(s)$. By Lemma B.5, in this case, $\kappa(v) = \kappa(c) = \kappa(d) = \chi(c) \cap \chi(d) \subseteq \chi(s)$.

(2) If $\lambda(v) \neq \emptyset$ or $\kappa(v) \neq \chi(v)$, we would like to show that for any s on the path between p and q on T, $s \in V(M_v)$. Suppose by way of contradiction that this is not the case, i.e., there exists some s on the path between p and q on T, but $s \in V(M) \setminus V(M_v)$. By the connectedness condition on T, we have $\chi(p) \cap \chi(q) \subseteq \chi(s)$. Furthermore, on M, $\chi(p) \cap \chi(s) \subseteq \chi(v)$ and $\chi(q) \cap \chi(s) \subseteq \chi(v)$, so $\chi(p) \cap \chi(q) = \chi(p) \cap \chi(q) \cap \chi(s) \subseteq \chi(v)$. Let the parent of p on M be t. Then, $\chi(t) \cap \chi(s) \subseteq \chi(v)$. Then, $\kappa(p) = \kappa(p) \cap \chi(s) = \chi(p) \cap \chi(t) \cap \chi(s) = (\chi(p) \cap \chi(s)) \cap (\chi(t) \cap \chi(s)) \subseteq \chi(v) \cap \chi(v) = \chi(v)$. But, by ($H_4$), this means either $t = v$ or t is a minor node with $\kappa(t) = \chi(t)$. In either case, we have $\chi(t) \cap \chi(q) \subseteq \chi(v)$, because (i) if $t = v$, $\chi(t) \cap \chi(q) = \chi(v) \cap \chi(q) \subseteq \chi(v)$, and (ii) if t is a minor node with $\kappa(t) = \chi(t)$, then $\chi(t) \cap \chi(q) = \kappa(t) \cap \chi(q) = \kappa(p) \cap \chi(q) \subseteq \chi(p) \cap \chi(q) \subseteq \chi(v)$. Also, we have $\kappa(t) = \chi(p) \cap \chi(t) \subseteq \chi(q)$, because either t is on the path between p and q, or p is on the path between t and q. Let the parent of v be u. Then, we have $\kappa(t) = \kappa(t) \cap \chi(q) = \chi(t) \cap \chi(q) = (\chi(t) \cap \chi(q)) \cap (\kappa(p) \cap \chi(q)) \subseteq (\chi(t) \cap \chi(q)) \cap (\chi(p) \cap \chi(q)) \subseteq \chi(v) \cap \chi(s) \subseteq \chi(u)$. But, since it cannot be that $\kappa(v) = \chi(v)$ in this case, this violates (H_4), contradiction. □

Lemma B.8. *Given a meta-decomposition M of hypergraph H, for all $v \in V(M)$, let $V_v^P = \{p \in V(M_v) \mid \lambda(p) \neq \emptyset\}$. Then,*

(1) if $\lambda(v) = \emptyset$ and $\kappa(v) = \chi(v)$, then Algorithm 2 enumerates all join trees of the hypergraph H_v with $E(H_v) = \lambda(V_v^P) \cup \{\chi(v)\}$ and $V(H_v) = \cup E(H_v)$;

(2) if $\lambda(v) \neq \emptyset$ or $\kappa(v) \neq \chi(v)$, then Algorithm 2 enumerates all join trees of the hypergrpah H_v with $E(H_v) = \lambda(V_v^P)$ and $V(H_v) = \cup E(H_v)$.

Proof. We will show that each join tree T of H can be enumerated by the algorithm based on the meta-decomposition M of H. We prove by structural induction on vertices $v \in V(M)$. If v is a leaf node, there is only one node in the subtree M_v, so this statement is vacuously true. If v is a non-leaf node, consider the following cases.

(1) If $\lambda(v) = \emptyset$ and $\kappa(v) = \chi(v)$, by Lemma B.6, all children c of v have either $\lambda(c) \neq \emptyset$ or $\kappa(c) \neq \chi(c)$, so, by the induction hypothesis, Algorithm 2 enumerates all join trees of hypergraph H_c with $E(H_c) = \bigcup_{p \in V_c^P} \lambda(p)$ and $V(H_c) = \bigcup_{e \in E(H_c)} e$. Also, note that it is possible to obtain a meta-decomposition M_v' of the hypergraph H_v from M_v by simply replacing v by a physical node v' with $\lambda(v') = \{\chi(v)\}$, $\chi(v') = \chi(v)$, and $\kappa(v') = \emptyset$. By Lemma B.7, for each child c of v, on any join tree T_c of hypergraph H_c, V_c^P induces a connected subtree T_c of T. Furthermore, T_c is a join tree of the meta-decomposition M_c, and can hence be enumerated on Line 6. So it remains to show that the algorithm enumerates all possible ways to combine these subtrees along with v. For each possible join tree T_v of H_v, without loss of generality, assume that T_v is rooted at v'. T_v can be constructed from the algorithm as follows. We first construct a tree T_P, by a top-down traversal of T_v, where, for each node p, if (i) $\chi(v) \subseteq \chi(p)$, (ii) $p \in M_c$, i.e., in the subtree M_c rooted at some child c of v, and (iii) its parent q is in a different subtree M_d rooted at some child $d \neq c$ of v, then we add an edge from d to c. If $p = v$, and its parent $q \in V(M_d)$ for some child d of v, we add an edge from d to v. If $p \in V(M_c)$ for some child c of v, and its parent $q = v$, we add an edge from v to c. Note that, for all pairs of c and d where we add edges, $\kappa(c) = \kappa(d) = \chi(v)$. Now, this tree T_P is a spanning tree of the clique with vertices $C \cup \{v\}$, where C is the set of all children of v, so it can be enumerated by the Prüfer sequence on Line 8. For each child c of v, by Lemma B.7, V_c^P induces a connected subtree T_c of T_v. We traverse the vertices $c \in V(T_P)$ in bottom-up order. For each c and each child d of c on T_P, let r_d be the root of the connected subtrees T_d of T_v induced by V_d^P, and u be the parent of r_d on T_v. The algorithm will be able to reroot T_d to r_d, find u (since $\kappa(d) = \kappa(v) = \chi(v) = \chi(c) \subseteq \chi(u)$), and attach T_d as a child of u. Finally, by Lemma B.5, in this case, $S = C$ and $X = \emptyset$, so the second part will not be executed at all.

(2) If $\lambda(v) = \emptyset$ and $\kappa(v) \neq \chi(v)$, starting from Lines 11–20, note that it is not possible to have a child $c \in C$ such that $\kappa(c) = \chi(c)$, because if that is the case, $\kappa(c) = \chi(c) = \chi(v)$, violating the uniqueness condition (H_5). Therefore, for all $c \in C$, we have $\lambda(c) \neq \emptyset$ or $\lambda(c) \neq \chi(c)$. Furthermore, we claim that the vertices $V_C^P = \bigcup_{c \in C} V_c^P$ induces a connected subtree T_C on any join tree T_v of M_v. This is because, if this is not the case, i.e., if there is some s on the path between some $p \in V_c^P$ and $q \in V_d^P$, where c and d are children of v with $\kappa(c) = \kappa(d) = \chi(v)$, such that $\kappa(s) \subsetneq \chi(v)$, then $\chi(p) \cap \chi(q) = \chi(v) \not\subseteq \kappa(s)$, violating the connectedness condition (H_2). Therefore, we can follow the same method as in case (1) to construct T_P with vertices C and show that, at the end of this part, it is possible to obtain a partial join tree T with vertices in V_C^P.

Now consider all children $c \in X = \{c \in C \mid \kappa(c) \subsetneq \chi(v)\}$.

(i) If $\lambda(c) = \emptyset$ and $\kappa(c) = \chi(c)$, by Lemma B.6, all children d of c has either $\lambda(d) \neq \emptyset$ or $\kappa(d) \neq \chi(d)$, so, by Lemma B.7, V_d^P induces a connected subtree in any join tree. We build a tree T_c by (a) collecting all maximal connected subtrees of the join tree with nodes $p \in V_c^P$ and (b) adding a common parent c as the root node. This is a valid join tree of the hypergraph H_c with $E(H_c) = \left(\bigcup_{p \in V_c^P} \lambda(p)\right) \cup \{\chi(c)\}$ and

$V(H_c) = \bigcup_{e \in E(H_c)} e$, and, by our inductive hypothesis, can be enumerated. For each child d of c on T_c, let the parent of d on T be u. The algorithm will be able to find such u to attach T_d, because $\kappa(d) = \chi(d) \cap \chi(c) \subseteq \chi(u)$, by the connectedness condition on T.

(ii) If $\lambda(c) \neq \emptyset$ or $\kappa(c) \neq \chi(c)$, similarly to case (i), by Lemma B.7, V_c^P induces a connected subtree on T, which, by our inductive hypothesis, can be enumerated. Then, the algorithm will be able to find such u to attach T_c, because $\kappa(c) = \chi(c) \cap \chi(v) \subseteq \chi(u)$, by the connectedness condition on T.

(3) If $\lambda(v) \neq \emptyset$, we can follow a similar argument for the children $c \in X$ in case (2). □

Finally, based on Lemma B.8, taking $v = r$, the root of M (for which $\kappa(r) = \emptyset \neq \chi(r)$), we finish the proof of completeness for the algorithm.

B.8 Polynomial-Delay Algorithm to Enumerate All Join Trees Based on a Meta-Decomposition

Algorithm 7: Enumerating all join trees based on a meta-decomposition, in polynomial delay

input : A meta-decomposition $M = (V(M), r, E(M), \lambda, \chi, \kappa)$
output : All possible non-homomorphic join trees up to rerooting

Function enumRec(v: *a node on* M, callback(T) *: a callback function once* enumRec *obtains a tree* T):
  $C \leftarrow$ set of children c of v on M, sorted by partial order $\supseteq$ on $\kappa(c)$
  if $\lambda(v) = \emptyset$ **then**
    $S \leftarrow \{c \in C \mid \kappa(c) = \chi(v)\}$ // they can be found at the head of the sorted C
    $X \leftarrow C \setminus S$
    if $\kappa(v) = \chi(v)$ **then**
      $T_v \leftarrow$ tree with one vertex v
      enumSpanningTrees($S \cup \{v\}, X$, callback)
    else
      enumSpanningTrees(S, X, callback)
  else
    $T_v \leftarrow$ tree with one vertex v
    attachChildren(T_v, C, callback)
Function enumSpanningTrees(S: *set of nodes,* C: *set of children,* callback):
  foreach *tree* T_p *with vertices* V *given by a Prüfer sequence* **do**
    enumMinorOriginSubtrees(T_p, S, X, callback)
Function enumMinorOriginSubtrees(T: *partial tree,* S: *remaining origins,* X: *remaining children,* callback):
  if $S = \emptyset$ **then**
    attachChildren(T, X, callback)
  else
    $c \leftarrow$ some node in S
    $S' \leftarrow S \setminus \{c\}$
    enumRec($c, T_c \Rightarrow$ enumOrigins(T with c replaced by T_c, S', X, callback))
Function attachChildren(T: *partial tree,* X: *remaining children,* callback):
  if $X = \emptyset$ **then**
    callback(T) // We finish constructing a valid join tree. Now we call the callback function.
  else
    $c \leftarrow$ some node in S
    $S' \leftarrow S \setminus \{c\}$
    if $\lambda(c) = \emptyset$ *and* $\kappa(c) = \chi(c)$ **then**
      enumRec($c, T_c \Rightarrow$ { **foreach** $u \in V(T)$ such that $\kappa(c) \subseteq \chi(u)$ **do** attachChildren(T with u replaced by T_c, S', callback)}
    else
      foreach $u \in V(T)$ *such that* $\kappa(c) \subseteq \chi(u)$ **do**
        enumRec($c, T_c \Rightarrow$ { **foreach** $T' \in$ rerootings$(T, \kappa(c), \emptyset)$ **do** attachChildren(T with c replaced by T', S', callback}
enumRec($r, T \Rightarrow$ { **foreach** $T' \in$ rerootings$(T, \emptyset, \emptyset)$, **emit** T'}

C Missing Materials for Section 5

C.1 Supporting Re-branching for Algorithm 3

For each node p, if there exists a subtree T_q that can be attached as a child of p, we simply use an additional bit in the DP table (plan) to indicate whether T_q is included. And we call optimizeLocal with or without the relations in T_q accordingly. We note again that this is a very rare possibility in practical queries.

C.2 Greedy Algorithm for Local Join Ordering

The implementation of local join ordering optimization using a greedy algorithm is as shown in Algorithm 8.

Algorithm 8: Greedy implementation of optimizeLocal

input : Node q on meta representation, a set of query plans $\mathcal{P}$
output : A query plan that joins the relation in q with all query plans in $\mathcal{P}$

if *q is a minor node* **then** // $\lambda(q) = \emptyset$, there is no relation in q
  $P \leftarrow$ remove P_1 from $\mathcal{P}$ with the minimum result cardinality
else
  $P \leftarrow$ the relation in $\lambda(q)$
while $\mathcal{P} \neq \emptyset$ **do**
  $P_i \leftarrow$ remove P_i from $\mathcal{P}$ that minimizes the cost to join P with P_i
  $P \leftarrow$ the plan $P_i \bowtie P$
return P