

Algorithms for Optimizing Acyclic Queries

Zheng Luo

UCLA

luo@cs.ucla.edu

Wim Van den Broeck

University of Bergen

wim.broeck@uib.no

Guy Van den Broeck

UCLA

guyvdb@cs.ucla.edu

Yisu Remy Wang

UCLA

remywang@cs.ucla.edu

Abstract

Most research on query optimization has centered on binary join algorithms like hash join and sort-merge join. However, recent years have seen growing interest in theoretically optimal algorithms, notably Yannakakis' algorithm. These algorithms rely on *join trees*, which differ from the operator trees for binary joins and require new optimization techniques. We propose three approaches to constructing join trees for acyclic queries. First, we give an algorithm to enumerate all join trees of an α -acyclic query *by edits* with amortized constant delay, which forms the basis of a cost-based optimizer for acyclic joins. Second, we show that the Maximum Cardinality Search algorithm by Tarjan and Yannakakis constructs a unique *shallowest* join tree, rooted at any relation, for a Berge-acyclic query; this tree enables parallel execution of large join queries. Finally, we prove that any connected left-deep linear plan for a γ -acyclic query can be converted into a join tree by a simple algorithm, allowing reuse of optimization infrastructure developed for binary joins.

1 Introduction

The query optimizer sits at the heart of a database system. It takes a query as input and generates a plan for efficient execution, allowing users to program declaratively without worrying about evaluation. Among the many relational algebra operators, join has received significant attention in optimization research. Its *compositional* nature allows for combining information from multiple relations, constructing complex queries from simple ones, and producing an output asymptotically larger than the inputs. The primary challenge is the *join ordering problem* to find the best arrangement of many join operations. Most existing research has focused on binary join algorithms such as hash join and sort-merge join, but these can produce unnecessarily large intermediates. Recent work has revived interest in optimal join algorithms, notably Yannakakis' instance-optimal algorithm [44] for acyclic queries, which runs in linear time in the input and output size, $\mathcal{O}(|\text{IN}| + |\text{OUT}|)$. Its execution is guided by *join trees* whose nodes are relations, different from traditional binary join plans with relations at the leaves and join operators at the internal nodes. Although the algorithm is optimal regardless of the join tree, the choice of plan can affect practical performance. In this paper, we study the optimization problem in the context of Yannakakis-style algorithms.

A query optimizer typically has two parts: a *plan generator* and a *cost model* to assess each plan. This paper focuses on plan generation and presents three approaches:

- An algorithm that enumerates all join trees of an acyclic query *by edits*, with amortized constant time delay.
- A proof that Maximum Cardinality Search [40] yields a unique *shallowest* join tree for any Berge-acyclic query, enabling parallel execution for very large queries.
- A characterization of when the algorithm by Hu et al. [20] can convert a binary join plan into a valid join tree, allowing reuse of existing optimizers.

The rest of the paper is organized as follows: Section 2 discusses related work; Section 3 introduces relevant concepts and notations; Section 4 presents the join tree enumeration algorithm; Section 5 introduces canonical join trees and their construction; Section 6 discusses the conversion from binary plans to join trees; finally, Section 7 points to avenues for future work and concludes the paper. All missing proofs can be found in the appendix, which also contains additional figures, tables and algorithms.

2 Related Work

Join order optimization is well studied, with algorithms based on dynamic programming (DP) from the bottom up [28, 34, 30], cost-based pruning from the top down [11, 15], greedy heuristics [5, 14, 38], and randomized search [36]. Since the plan space is exponential, most methods prune it: some restrict to left-deep plans [21, 23], while others avoid Cartesian products [28, 29]. Our algorithms restrict the query plans to those running in linear time for acyclic queries. In particular, while avoiding Cartesian products requires each subplan to form a *spanning tree* of the corresponding subquery’s join graph,¹ our algorithms find *maximum spanning trees* of the *weighted* join graph. Several algorithms for ordering binary joins are based on dynamic programming and tabulate shared structures among different plans [28, 29]. This is desirable because subplans are grouped into equivalence classes, and the optimal plan can be constructed in a bottom-up manner. Our enumeration algorithm avoids redundant work by enumerating join trees *by edits*, i.e., it outputs the difference between consecutive join trees instead of the trees themselves. Future work may explore constructing compact representations of join trees, suitable for dynamic programming.

Several recent papers have proposed practical implementations of Yannakakis’ algorithm for acyclic queries [2, 20, 42, 47]. For example, Zhao et al. [47] find that different query plans perform similarly, thanks to the optimality of Yannakakis’ algorithm. They adopt a simple heuristic to construct the join tree by picking the largest input relation as the root, and then greedily attaching the remaining relations into the tree. Inspired by this algorithm, we prove in Section 5 that for Berge-acyclic queries there is a unique *shallowest* join tree for any given root where the depth of each node is minimized. Furthermore, this tree can be constructed in linear time by Tarjan and Yannakakis’ Maximum Cardinality Search algorithm [40]. Shallow trees are desirable for parallel execution, where the depth of the tree determines the number of sequential steps. Other practical implementations of Yannakakis’ algorithm leverage existing optimizers for binary joins and convert a binary plan into a join tree [2, 20]. In particular, Hu et al. [20] find that every left-deep linear plan encountered in practice can be converted into a join tree by a simple algorithm. This is not surprising, as we will prove in Section 6 that every connected left-deep linear plan of a γ -acyclic query must traverse some join tree from root to leaves.

On the theoretical side, attention has been focused on finding *(hyper-)tree decompositions* to improve the asymptotic complexity of query processing [37, 19, 16]. The general goal is to find a decomposition with small *width* which can be used to guide the execution of join algorithms. Most algorithms find a single decomposition with minimum width to achieve the optimal asymptotic complexity [37, 19, 16]. Nevertheless, different decompositions with the same width may still lead to different performance in practice, and cost-based optimization remains crucial. For this, Carmeli et al. [6] propose an algorithm to enumerate tree decompositions with polynomial delay. In this paper, we focus on acyclic queries and their join trees, which are precisely the decompositions with width 1. Our enumeration algorithm can generate all join trees by edits with amortized constant delay.

3 Preliminaries

We focus on full conjunctive queries [1] in this paper and identify each query with its hypergraph, where there is a vertex for each variable and a hyperedge for each relation.

Definition 1. A hypergraph is a tuple (X, R, χ) where X is a set of vertices, R a set of hyperedges, and $\chi : R \rightarrow 2^X$ is a function that assigns to each hyperedge a set of vertices.

Without loss of generality, we limit the scope of our discussion to queries which admit connected hypergraphs. For queries with disconnected hypergraphs, we treat each component separately. We do not consider isolated vertices, empty hyperedges or duplicated hyperedges containing the same set of vertices. Therefore, each hyperedge can be identified with its vertex set. We will use $x \in r$ interchangeably with $x \in \chi(r)$ and apply the common set operations to hyperedges. The size of a hypergraph is the sum of all hyperedges $|H| = \sum_{r \in R} |r|$. We let $H|_x$ denote the *neighborhood* of x in H , consisting of all hyperedges containing vertex x .

¹The join graph of a query has a vertex for each relation and an edge for each pair of relations that join with each other. We later define this as the line graph of the query hypergraph in Definition 2.

Algorithm 1: MCS from $r \in R(H)$

Input: Hypergraph H , root r
Output: Join tree T_r rooted at r

```

1  $p(r) \leftarrow \text{null}$ 
2  $i \leftarrow 0$ 
3 while  $R \neq \emptyset$  do
4    $i \leftarrow i + 1$ 
5    $r_i \leftarrow r$ 
6   for  $x \in r_i \cap X$  do
7     for  $r' \in R : x \in r'$  do
8        $p(r') \leftarrow r_i$ 
9    $X \leftarrow X \setminus r_i$ 
10   $R \leftarrow R \setminus \{r_i\}$ 
11   $r \leftarrow \arg \max_{r \in R} |r \setminus X|$ 

```

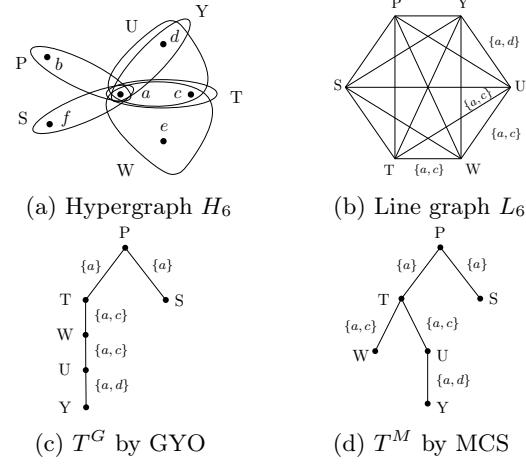


Figure 1: A hypergraph H_6 , its line graph L_6 , and two join trees T^G and T^M .

Definition 2. The line graph $L(H)$ of a hypergraph $H = (X, R, \chi)$ is an undirected simple graph (R, E) whose vertices R are the hyperedges of H , and there is an edge $(r_1, r_2) \in E$ whenever $r_1 \cap r_2 \neq \emptyset$. We extend χ such that $\chi(e) = \chi(r_1, r_2) := r_1 \cap r_2$ for each edge $e = (r_1, r_2) \in E$, and define the edge-weight function $\omega(r_1, r_2) := |r_1 \cap r_2|$.

When there is no ambiguity from the context, we simply write $L(H)$ as L . The size of a line graph is the sum of all edge weights $|L| = \sum_{e \in E(L)} \omega(e)$. It reflects the number of join clauses in a SQL query with one clause per variable shared between two relations. We let $L|_x$ denote the subgraph of $L(H)$ induced by $H|_x$.

Example 3. Figure 1a shows an example hypergraph of size 14. The line graph $L_6 = L(H_6)$ is shown in Figure 1b. For instance, hyperedges Y and U share two vertices a and d , so they are connected by an edge in the line graph. The edge weight is given by $\omega(Y, U) = |\{a, d\}| = 2$. Each unannotated edge in L_6 connects two hyperedges sharing only the vertex a . Otherwise, the common vertices shared by a pair of hyperedges are annotated next to the corresponding edge. The size of the line graph is $|L_6| = 19$.

Definition 4. A join tree T of hypergraph H is a spanning tree of $L(H)$ such that $T|_x$ is a connected subtree for each $x \in X(H)$. If a certain vertex is specified as the root, T becomes a rooted join tree where the edges are oriented away from the root.

A join tree $T(L)$ is a subgraph of L , so we use $R(T)$ to denote the set of tree nodes and $E(T)$ to denote the set of tree edges. Because T is spanning and acyclic, we always have $R(T) \equiv R(L)$ and $E(T) \subseteq E(L)$. The requirement that $T|_{x \in X(H)}$ is connected is also known as the *running intersection property* [13]. We write $\mathcal{T}(L(H))$ or $\mathcal{T}(H)$ to denote the set of unrooted join trees. The *union join graph* $\mathcal{U}(L(H))$ or $\mathcal{U}(H)$ is a spanning subgraph of $L(H)$ where the edge set is the union of all join trees $E(\mathcal{U}(H)) = \bigcup_{T \in \mathcal{T}(H)} E(T)$. We use T_r to denote a join tree rooted at $r \in R(T)$. When there is no ambiguity, we simply write T . The depth of a node r_i in the rooted tree T_r , denoted $d(T_r, r_i)$, is defined as its distance from the root. Join trees can be constructed by a procedure called *GYO reduction*.

Definition 5. A GYO reduction order is a sequence of hyperedges r_1, r_2, \dots, r_k such that for each $r_{i < k}$, there is some $r_{p > i}$, called the parent of r_i , such that $\forall r_{j > i} : r_i \cap r_j \subseteq r_p$.

The GYO reduction algorithm finds such an order iteratively, and attaches each hyperedge to its parent to form a join tree. It generates the join tree T^G as shown in Figure 1c. Readers may refer to the work of Yu and Ozsoyoglu [45] for details.

Four common notions of hypergraph acyclicity are defined in decreasing order of strictness, following the seminal work by Fagin [13], namely *Berge-acyclic* \Rightarrow γ -acyclic \Rightarrow β -acyclic \Rightarrow α -acyclic. The detailed definitions are given by Definition 41 in Section A.

Another algorithm for checking acyclicity of hypergraphs and constructing join trees is the Maximum Cardinality Search (MCS) algorithm by Tarjan and Yannakakis [40]. Algorithm 1 shows a simplified pseudocode of MCS. We discuss the properties of MCS and how it constructs the join trees T^M in Figure 1d with Example 42 in Section A.

Computation model. Throughout the paper we assume the Random Access Machine model of computation, where one can allocate an array of size n in $\mathcal{O}(n)$ time. Constant-time operations include accessing and updating an array element, adding or deleting an element in a linked list, and the common arithmetic operations on integers.

4 Enumerating Join Trees

Our strategy for enumerating join trees is based on the following classic result relating join trees of a hypergraph to maximum spanning trees (MSTs) of its line graph:

Proposition 6 (Maier [27]). *A subgraph of $L(H)$ is a join tree of H if and only if it is a maximum spanning tree of $L(H)$ according to the weight function ω .*

A naïve approach is to apply an off-the-shelf algorithm for enumerating the maximum spanning trees of $L(H)$. The best known algorithm for this purpose is due to Eppstein [12], and it works by deriving from the input graph G a so-called *equivalent graph* G^\equiv . Every spanning tree of G^\equiv corresponds to an MST of G and vice versa. Eppstein proves a lower bound of $\Omega(m + n \log n)$ on constructing the equivalent graph from an arbitrary weighted graph with m edges and n vertices. Then to enumerate all k MSTs of G , Eppstein applies existing algorithms to enumerate the spanning trees of G^\equiv . Since there are optimal spanning tree enumeration algorithms that run in $\mathcal{O}(m + n + k)$ [22, 35], the overall time complexity to enumerate MSTs is $\mathcal{O}(m + n \log n + k)$.

The main result of this section is an algorithm for enumerating join trees leveraging the structure of acyclic hypergraphs and their line graphs. In particular:

- for any α -acyclic hypergraph H , we can construct an equivalent graph in $\mathcal{O}(|L|) = \mathcal{O}(m + n)$, where m and n are the numbers of edges and vertices in $L(H)$, thus enumerating the join trees in $\mathcal{O}(|L| + k)$.
- for any γ -acyclic hypergraph H , we can construct an equivalent graph from the union join graph $\mathcal{U}(H)$ which can be built in $\mathcal{O}(|H| + |\mathcal{U}(H)|)$, lowering the overall time complexity of enumeration to $\mathcal{O}(|H| + k)$.

In the rest of this section, we formally introduce the notion of the equivalent graph G^\equiv . Thereafter, we outline the main algorithm for enumerating join trees, prove its correctness and analyze its complexity. Finally, we consider the case of γ -acyclic queries.

4.1 Equivalent Graph

The construction of the equivalent graph G^\equiv for a given edge-weighted graph G , is based on the following local graph modification, *sliding transformation*.

Definition 7 (Sliding Transformation [12]). *Let G be an edge-weighted graph with weight function ω . Given edges $e^* = (u, v)$ and $e = (v, w)$ where $\omega(e) < \omega(e^*)$, sliding e along e^* moves e 's end node v to u such that $e = (u, w)$.*

We can construct an equivalent graph by repeatedly performing sliding transformations. Sliding may create parallel edges, cycles, and self-loops, so the result is a multigraph.

Definition 8 (Equivalent Graph [12]). *Given weighted graph G with MST T rooted, an equivalent graph G^\equiv is obtained from G by exhaustively sliding every edge $e = (v, w) \in E(G)$ along another $e^* = (u, v) \in T$ as long as $d(T, u) < d(T, v)$.*

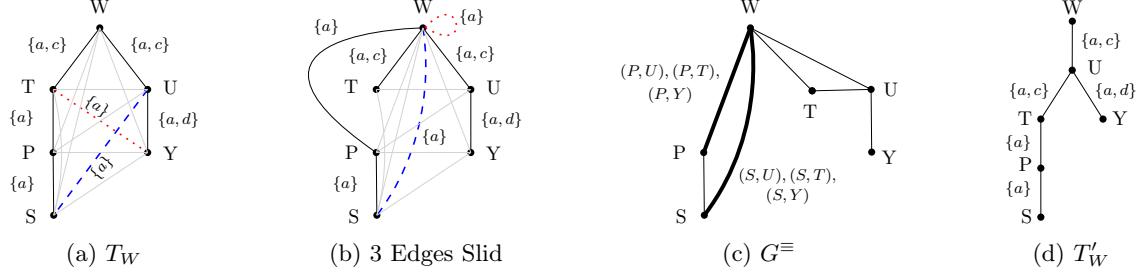


Figure 2: T_W is an MST of the line graph in Figure 1b where the black solid lines stand for the tree edges, the dashed line for an MST edge (S, U) , the dotted line for a non-MST edge (T, Y) and gray solid lines for the remaining non-tree edges. $G^=$ is the resulting equivalent graph where the thick lines highlight the parallel edges.

Sliding transformation preserves the identity of each edge $e \in E(G)$ as explained in Example 10. Every spanning tree of $G^=$ is an MST of G and vice versa.

Theorem 9 (by Eppstein [12]). *The maximum spanning trees of a weighted graph G are in one-to-one correspondence with the spanning trees of its equivalent graph $G^=$.*²

In the following, we use $p(r)$ to denote the parent of r in a given rooted tree, and $c(r)$ to denote its children. We extend the notation of parent to edges as follows: for any edge $e \in E(T)$, its parent $p(e) \in E(T)$ is another tree edge incident to e and closer to the root r . The edges incident to the root do not have a parent and are all siblings. Otherwise, the siblings of e are incident to the same parent, denoted as $s(e) = \{e' \in E(T) \setminus \{e\} \mid p(e') = p(e)\}$.

Example 10. Given the 6-clique line graph L_6 in Figure 1b, we find a rooted MST T_W as shown in Figure 2a. Among all the tree edges shown as black solid lines, we can only apply the sliding transformation to the edge (T, P) whose weight $\omega(T, P) = |\{a\}| = 1$ is lighter than its parent tree edge (W, T) with $\omega(W, T) = |\{a, c\}| = 2$. We slide along the tree edge (W, T) to the root so that the edge (T, P) becomes (W, P) as shown by the solid curve in Figure 2b. Non-tree edges can slide similarly. We consider two examples, (S, U) illustrated with a dashed line and (Y, T) with a dotted line in Figure 2a. All other non-tree edges are shown in light gray. We can slide along tree edge (U, W) to the root so that the edge (S, U) becomes (S, W) as shown by the dashed curve in Figure 2b. Both ends of (Y, T) can slide along the tree edges to the root so that the edge becomes a self-loop as shown by the dotted loop in Figure 2b. This edge will not appear in any spanning tree of $G^=$, and therefore not a part of any MST of L_6 . We refer to such an edge as a non-MST edge, as opposed to an MST edge. By applying sliding transformations to a fixpoint, we obtain an equivalent graph $G^=$, where there are two sets of parallel edges highlighted by thick lines in Figure 2c. For example, the tree edge (T, P) in T_W and non-tree edges $(P, U), (P, T), (P, Y)$ are parallel in $G^=$ between P and W . We can easily verify that each spanning tree of $G^=$ corresponds to an MST of L_6 , such as T'_W in Figure 2d.

The choice of the initial spanning tree can affect the structure of the equivalent graph, but the order of sliding transformations performed has no impact [12].

4.2 Enumerating Join Trees of α -Acyclic Hypergraphs

The bottleneck of Eppstein’s algorithm for constructing equivalent graphs of general graphs comes from a subroutine that identifies where each edge will eventually slide to. Because each edge can only slide along a heavier edge, it will eventually be “blocked” by a lighter or equally weighted edge along its path to the root. The subroutine essentially performs binary search to find the blocking edge, leading to the $\log n$ factor in the overall complexity. The key to our improvement is to show that for every acyclic hypergraph H , we can construct an “equivalent hypergraph” H^* that shares the same structure with H and has the same set of join trees, but H^* admits a special join tree with *monotonically increasing weight* from root to leaf, which enables constant-time identification of the blocking edge.

²Minimum spanning trees are considered in the original literature [12].

Definition 11. A monotonic weight join tree $\hat{T} \in \mathcal{T}(H)$ is a rooted join tree of H such that for any $e \in \hat{T}$ that has a parent $p(e)$, $\omega(e) > \omega(p(e))$.

Example 12. H_6^* in Figure 3a is a hypergraph with similar structure to H_6 in Figure 1a. H_6^* differs from H_6 by one vertex d' and admits the same set of join trees. Therefore, finding an equivalent graph $G^{\equiv}(H_6^*)$ is sufficient for enumerating the join trees of H_6 . H_6^* also admits a monotonic weight join tree \hat{T}_P in Figure 3b. This means each tree edge in \hat{T}_P is already in place, because it cannot slide along the lighter parent. For each non-tree edge, we can identify its destination in constant time using algorithms for finding lowest common ancestors [3] and level ancestors [4], as detailed in Algorithm 3.

The example shows we can enumerate join trees of H by constructing H^* and its equivalent graph $G^{\equiv}(H^*)$, then enumerate spanning trees of $G^{\equiv}(H^*)$. However, constructing H^* has a cost that may exceed our target time complexity. We show that, perhaps surprisingly, running Algorithm 3 directly on H produces the same equivalent graph $G^{\equiv}(H^*)$, so we can skip the costly construction of H^* .

Before presenting the algorithm, we first formalize the relationship between H and H^* using the concept of *hypergraph homomorphisms*. Note the definition of homomorphisms is not standardized in the literature, and we choose ours carefully to simplify the proofs. In particular, our definition is equivalent to the homomorphism of incidence graphs [32].

Definition 13 (Hypergraph Homomorphism). For two hypergraphs H_1, H_2 , a homomorphism f is a pair of functions $(f_X : X(H_1) \rightarrow X(H_2), f_R : R(H_1) \rightarrow R(H_2))$ such that:

$$\forall r_1 \in R(H_1), x_1 \in r_1 : f_X(x_1) \in f_R(r_1)$$

We write $H_1 \rightarrow H_2$ to denote that there is a homomorphism from H_1 to H_2 .

When there is no ambiguity from the context, we simply write f for f_X or f_R . Intuitively, homomorphisms preserve relations. A *strong* homomorphism also reflects relations:

Definition 14 (Strong Homomorphism). A homomorphism $f : H_1 \rightarrow H_2$ is called strong if it also satisfies:

$$\forall r_1 \in R(H_1), x'_1 \notin r_1 : f_X(x'_1) \notin f_R(r_1)$$

We write $H_1 \twoheadrightarrow H_2$ to denote that there is a strong homomorphism from H_1 to H_2 .

In this paper we only consider hypergraphs over the same set of edges,³ and from now on we assume f_R is the identity function.

We now show that if there is a strong homomorphism $f : H' \twoheadrightarrow H$, then every join tree of H is also a join tree of H' .

Lemma 15. If $H' \twoheadrightarrow H$ then $\mathcal{T}(H) \subseteq \mathcal{T}(H')$.

We construct H^* such that $H^* \twoheadrightarrow H$ and H^* admits a monotonic weight join tree \hat{T} , by first using the MCS algorithm to construct a join tree T of H called the *MCS tree*. Then, we *perturb* the weights of T so that they increase monotonically from the root to leaves. Finally, we construct H^* such that the perturbed tree is a join tree of H^* .

The next result shows every MCS tree is already “somewhat monotonic”, in that every edge must contain some variable not in its parent. With abuse of notation we write $e_1 \subseteq e_2$ for $\chi(e_1) \subseteq \chi(e_2)$ and use the common set operations directly on edges to reduce clutter.

Lemma 16. Let T be an MCS tree. For any edge $e \in T$ that has a parent, then $e \not\subseteq p(e)$.

Furthermore, if two edges share any variable not in their parents, they must be siblings.

Lemma 17. Let T be an MCS tree. For two edges $e, e' \in T$ that have parents,

$$(e \setminus p(e)) \cap (e' \setminus p(e')) \neq \emptyset \implies p(e) = p(e').$$

³The hypergraphs share the same hyperedge set R but differ in the hyperedge-to-vertices functions χ , i.e., an edge may have different sets of vertices in different hypergraphs.

We now show how to derive from an α -acyclic hypergraph H an equivalent hypergraph H' that preserves $\mathcal{T}(H)$ and admits a monotonic weight join tree \hat{T} .

Definition 18 (Vertex Duplication). *Given a hypergraph H and a vertex $x \in X(H)$, duplicating x produces a new hypergraph H' that differs from H by only one vertex $\{x'\} = X(H') \setminus X(H)$ which is added to each $r \in H|_x$.*

We will refer to such a hypergraph H' as a *vertex duplication* of H or simply a *duplicated* hypergraph of H . We generalize duplication to a set of vertices in the natural way. Importantly, vertex duplication preserves the set of join trees of the original hypergraph.

Lemma 19. *Let H' be a duplicated hypergraph of H . Then,*

1. $H' \Rightarrow H$ and $H' \Leftarrow H$
2. $\mathcal{T}(H') = \mathcal{T}(H)$
3. $L(H') = L(H)$

We are now ready to construct H^* by vertex duplication.

Algorithm 2: Construction of H^*

```

Input: MCS tree  $T_r(H)$ 
1 for  $r' \in c(r)$  do
2    $e \leftarrow (r, r');$   $q.\text{enqueue}(e)$ 
3 while  $q \neq \emptyset$  do
4    $e \leftarrow q.\text{dequeue}()$ 
5    $\Delta \leftarrow \omega(p(e)) - \omega(e) + 1$ 
6   if  $\Delta > 0$  then
7      $x \leftarrow$  a vertex in  $\chi(e) \setminus \chi(p(e))$ 
8     make  $\Delta$  duplicates of  $x$ 
9   for  $e' \in c(e)$  do
10     $q.\text{enqueue}(e')$ 

```

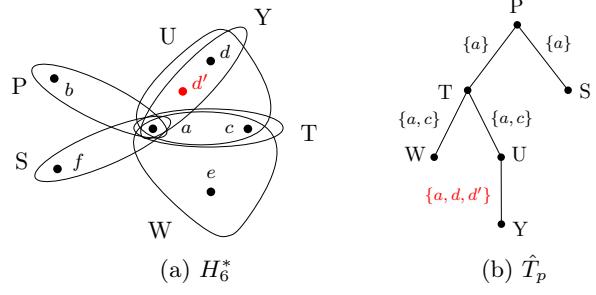


Figure 3: An equivalent hypergraph H_6^* and its monotonic weight join tree \hat{T}_P

Theorem 20. *Given an MCS tree T of an α -acyclic hypergraph H , there exists a hypergraph H^* that*

- *admits a monotonic weight join tree $\hat{T} \in \mathcal{T}(H^*)$ whose edges are in a one-to-one correspondence to those of T*
- *satisfies $\mathcal{T}(H^*) = \mathcal{T}(H)$.*

Proof. To prove the existence, we construct such a hypergraph H^* by vertex duplication. Lemma 19 guarantees that $\mathcal{T}(H^*) = \mathcal{T}(H)$. We only need to show the existence of $\hat{T} \in \mathcal{T}(H^*)$. Algorithm 2 constructs H^* by vertex duplication in a breadth-first manner from the root of T to make it a monotonic weight join tree in H^* .

Algorithm 2 starts with enqueueing each edge incident to the root. Once a tree edge $e \in E(T_r)$ is dequeued, we check for the number of duplications needed to make it heavier than its parent edge $p(e)$, namely $\Delta = \omega(p(e)) - \omega(e) + 1$. The duplication is performed only if $\Delta > 0$. Lemma 16 guarantees that $\chi(e) \setminus \chi(p(e)) \neq \emptyset$. Therefore, we can always duplicate a vertex $x \in \chi(e) \setminus \chi(p(e))$ for Δ times such that $\omega(e) > \omega(p(e))$.

Let $e = (r_u, r_d)$ where $d(T, r_u) < d(T, r_d)$, x only occurs in the subtree rooted at r_u . Otherwise, it violates the running intersection property. Therefore, the duplication of x does not affect the weight of any edge closer to the root than e in T . By Lemma 17, duplication in a sibling edge $e' \in s(e)$ would not affect the weight of e either, and $\omega(e) > \omega(p(e))$ will continue to hold. The tree has monotonic weights when Algorithm 2 terminates. \square

Given a rooted join tree T of a hypergraph H , a non-tree edge is $e = (r_i, r_j) \in E(L(H)) \setminus E(T)$. There is a path in T between r_i and r_j via their lowest common ancestor $\text{LCA}(r_i, r_j)$. We define the *LCA edges* $\lambda(e) = \lambda(r_i, r_j)$ as a set of at most two tree edges on the path and incident to $\text{LCA}(r_i, r_j)$. If r_i, r_j are ancestor and child, then $|\lambda(e)| = 1$, otherwise, $|\lambda(e)| = 2$.

In the process of building $G^{\equiv}(H^*)$ from a monotonic weight join tree \hat{T} of H^* , Definition 7 and Definition 11 guarantee that:

- no tree edge in \hat{T} needs to be slid;
- each non-tree edge e can be removed if it is lighter than all $e' \in \lambda(e)$;
- otherwise, the non-tree edge e can be slid directly to be incident to each $e' \in \lambda(e)$.

Algorithm 3 constructs $G^{\equiv}(H^*)$ from the line graph $L(H)$ weighted by ω and an MCS tree T_r . Full details on the constant-time operations on graphs and data structures to support operations performed in Algorithm 3 can be found in Section B. As a part of pre-processing, we first conduct a breadth-first search on \hat{T}_r to obtain the depth table \mathbf{d} of each tree node in $\mathcal{O}(|R(\hat{T}_r)|)$. We will also use the algorithms proposed by Michael Bender [3, 4] to build the data structures in $\mathcal{O}(|R(\hat{T}_r)|)$ to facilitate the constant-time query of lowest common ancestors (LCAs) and level ancestors (LAs). The constant-time query of depth, LCA and LA allows for finding the $\lambda(e)$ of any non-tree edge e in constant time as shown from Line 4 to Line 6 in Algorithm 3.

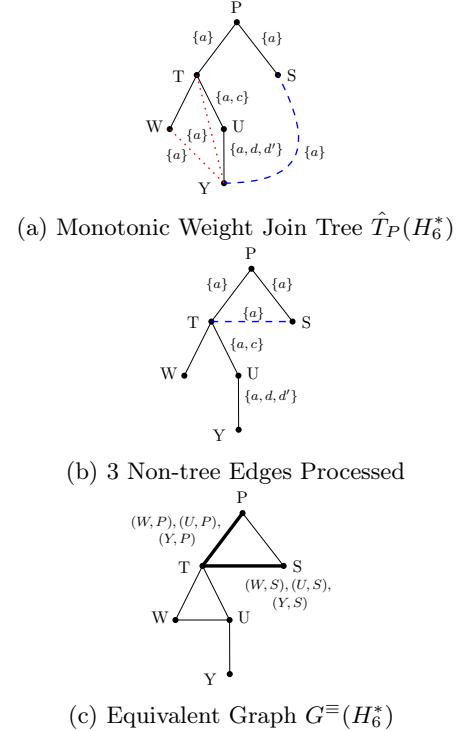
Algorithm 3: buildEG

Input: MCS tree $T_r(H)$ of line graph $L(H)$ with weight function ω

```

1  $G^{\equiv} \leftarrow L(H)$ 
2 for  $e^* = (r_i, r_j) \in R(L(H)) \setminus R(T_r(H))$  do
3   // where  $\mathbf{d}(T_r(H), r_i) \leq \mathbf{d}(T_r(H), r_j)$ 
4    $w^* = \omega(e^*)$ 
5    $l \leftarrow \text{LCA}(r_i, r_j); d \leftarrow \mathbf{d}(l) + 1$ 
6    $e_1 = (l, r_1 \leftarrow \text{LA}(r_i, d)); w_1 = \omega(e_1)$ 
7    $e_2 = (l, r_2 \leftarrow \text{LA}(r_j, d)); w_2 = \omega(e_2)$ 
8   if  $w^* < w_1 \wedge w^* < w_2$  then
9     delete  $e^*$  from  $R(G^{\equiv})$ 
10    if  $e_1 = e_2 \wedge w^* = w_1$  then
11      slide  $r_j$  to  $r_1$ 
12    else if  $e_1 \neq e_2 \wedge w^* = w_1 = w_2$  then
13      slide  $r_i$  to  $r_1$  and  $r_j$  to  $r_2$ 
14    else if  $e_1 \neq e_2 \wedge w^* = w_1 < w_2$  then
15      slide  $r_i$  to  $l$  and  $r_j$  to  $r_1$ 
16    else if  $e_1 \neq e_2 \wedge w^* = w_2 < w_1$  then
17      slide  $r_i$  to  $l$  and  $r_j$  to  $r_2$ 
17 return  $G^{\equiv}$ 

```



Example 21. We consider \hat{T}_P in Figure 3b of an equivalent hypergraph H_6^* in Figure 3a. Three of \hat{T}_P 's non-tree edges are shown in Figure 4a where the non-MST edges $(Y, W), (Y, T)$ are dotted and the MST edge (Y, S) is dashed. We can identify $\lambda(Y, W) = \{(T, W), (T, U)\}$ in constant time. Because $\omega(Y, W) = |\{a\}| = 1$ is lighter than $\omega(T, W) = \omega(T, U) = |\{a, c\}| = 2$, it is removed. (Y, T) is also removed, for it is lighter than its LCA edge (T, U) . We compare (Y, S) with $\lambda(Y, S) = \{(P, T), (P, S)\}$. Because they have the same weight with $\omega(Y, S) = \omega(P, T) = \omega(P, S) = |\{a\}| = 1$, we slide (Y, S) directly to (T, S) . The intermediate graph

with those three non-tree edges processed is shown in Figure 4b. The resulting equivalent graph $G^{\equiv}(H_6^*)$ is shown in Figure 4c.

The next result follows immediately from the construction of the equivalent hypergraph.

Lemma 22. *Let $H^* = (X^*, R, \chi^*)$ be an equivalent hypergraph derived from $H = (X, R, \chi)$ by vertex duplication. For any non-tree edge e of an MCS tree T of H ,*

- *If $\chi(e) = \chi(e')$ for some $e' \in \lambda(e)$, then $\chi^*(e) = \chi^*(e')$*
- *If $\chi(e) \subset \chi(e')$ for some $e' \in \lambda(e)$, then $\chi^*(e) \subset \chi^*(e')$*

Proof. The duplication of a vertex $x \in \chi(e)$ adds x' to both $\chi(e)$ and $\chi(e')$. If $\chi(e) = \chi(e')$, $\chi^*(e) = \chi(e) \cup \{x'\} = \chi(e') \cup \{x'\} = \chi^*(e')$. A similar argument applies if $\chi(e) \subset \chi(e')$. \square

Lemma 22 allows us to produce $G^{\equiv}(H^*)$ by keeping all tree edges of the MCS tree $T(H)$ in place and sliding the non-tree edges directly to their LCA edges. For example, running Algorithm 1 on the MCS tree $T(H_6)$ in Figure 1d produces an identical $G^{\equiv}(H_6^*)$ in Figure 4c.

Theorem 23. *Let H be an α -acyclic hypergraph with weight function ω , line graph L and MCS tree T . Let H^* be the equivalent hypergraph of H with weight function ω^* . Applying Algorithm 3 to inputs L , T and ω produces the same equivalent graph as applying Algorithm 3 to inputs L , T and ω^* .*

Proof. One can readily check that for any non-tree edge e of T , the conditions of each if statement in Algorithm 3 hold under ω if and only if they hold under ω^* by Lemma 22. \square

Theorem 24. *Given line graph L of α -acyclic hypergraph H with weight function ω and an MCS tree T of H , Algorithm 3 returns equivalent graph G^{\equiv} of equivalent hypergraph H^* in $\mathcal{O}(m+n)$ where n is the number of vertices and m is the number of edges in L .*

Proof. The construction of each data structure $d(\cdot)$, $LA(\cdot)$ and $LCA(\cdot)$ can be done in $\mathcal{O}(m+n)$ [3, 4]. Execution of the rest of the algorithm also takes time $\mathcal{O}(m+n)$, as it requires examining each non-tree edge to either delete it or slide it directly to its LCA edges in the MCS tree. For each non-tree edge, the lookup of at most two LCA edges takes time $\mathcal{O}(1)$, as does the comparison of weights and the direct sliding transformation. \square

Once we have the equivalent graph G^{\equiv} , the existing algorithm allows for enumerating all its spanning trees with an amortized constant delay. The total time complexity of enumerating $|\mathcal{T}(H)|$ given the line graph $L(H)$ as input is $\mathcal{O}(|L(H)| + |\mathcal{T}(H)|)$.

Starting from an α -acyclic query \mathcal{Q} , Section C.1 describes how to convert it into a hypergraph H . Section C.2 describes how to convert it into a weighted line graph $L(H)$. Both conversions take linear time $\mathcal{O}(|\mathcal{Q}|)$ with respect to the query size as defined in Section C. Algorithm 1 constructs an MCS tree T in $\mathcal{O}(|H|) \leq \mathcal{O}(|\mathcal{Q}|)$. With the inputs $L(H)$ and T , we can run Algorithm 3 to construct $G^{\equiv}(H^*)$ in $\mathcal{O}(m+n) \leq \mathcal{O}(|\mathcal{Q}|)$ (where n is the number of vertices and m is the number of edges in $L(H)$). Finally, we apply the spanning tree enumeration algorithm by Kapoor and Ramesh [22] on $G^{\equiv}(H^*)$ to enumerate all join trees $|\mathcal{T}(H)|$ in $\mathcal{O}(m+n+|\mathcal{T}(H)|) \leq \mathcal{O}(|\mathcal{Q}|+|\mathcal{T}(H)|) = \mathcal{O}(|L(H)|+|\mathcal{T}(H)|)$.

4.3 γ -Acyclic Queries

The run time of Algorithm 3 depends on the size of the line graph which can be quadratically larger than the input hypergraph. But if H is γ -acyclic, we can bring the total time complexity of enumeration down to $\mathcal{O}(|H|+|\mathcal{T}(H)|)$. The line graph of a γ -acyclic hypergraph H is its union join graph [25] $L(H) = \mathcal{U}(H)$, i.e., every edge $e \in E(L(H))$ in the line graph is an MST edge. By modifying the MCS algorithm to Algorithm 7 in Section F, we can construct a weighted MCS tree T and an unweighted line graph $L(H)$, namely the union join graph, in $\mathcal{O}(|H|+|E(\mathcal{U}(H))|)$. We simplify Algorithm 3 to Algorithm 8 in Section F, which compares the LCA edges $\lambda(e)$ of each non-tree edge e and slides it to construct the equivalent graph $G^{\equiv}(H)$ in $\mathcal{O}(|E(\mathcal{U}(H))|)$. As γ -acyclicity guarantees that every non-tree edge e is an MST edge, we can use the weights of the LCA edges to determine how to slide e . Let $\lambda(e) = \{e_l, e_r\}$. If $e_l = e_r$, we slide e to be parallel

to e_l . Otherwise, if e_l and e_r have different weights, we slide e to be parallel to the heavier one. If they have the same weight, we slide e to form a triangle with them.

The number of join trees is lower bounded by the number of non-tree edges in its line graph, namely $|\mathcal{T}(H)| = \Omega(|E(\mathcal{U}(H))| - |E(T)|)$. The total time complexity of enumerating all join trees is $\mathcal{O}(|H| + |E(\mathcal{U}(H))| + |\mathcal{T}(H)|) = \mathcal{O}(|H| + |\mathcal{T}(H)|)$.

5 The Canonical Join Tree of a Berge-Acyclic Query

An acyclic query can have exponentially many join trees with respect to its size. For example, the line graph of a clique query with n relations is an n -clique K_n with n^{n-2} join trees by Cayley's formula [7]. Enumerating all join trees can be prohibitive for large queries. On the other hand, the query optimizer does not need to consider all possible join trees to achieve good performance. For example, the implementation of Yannakakis' algorithm by Zhao et al. [47] has similar performance on any join tree rooted at the largest relation. An alternative to enumeration is therefore to simply construct an arbitrary join tree for a given root. This can be done in linear time by the Maximum Cardinality Search (MCS) algorithm from a chosen relation as shown in Algorithm 1. In this section, we show that the MCS algorithm produces a *shallowest tree* for a Berge-acyclic query, in the sense that the depth of every node in the tree is minimized. We prove that this shallowest tree is unique, and therefore call it the *canonical join tree*:

Definition 25. A join tree T_r rooted at r is canonical if $d(T_r, r_i) \leq d(T'_r, r_i)$ for any other join tree T'_r rooted at r and any $r_i \in R(T_r) = R(T'_r)$.

A shallow join tree has practical benefits. For example, the depth of the join tree determines the number of sequential steps required in a parallel join algorithm. A shallow join tree tends to be wide and have more leaves, allowing better utilization of indices.

Although Berge-acyclicity was thought to be too restrictive when it was first introduced to database theory [13], we found it to be general enough to cover almost all acyclic queries encountered in the wild. As shown in Table 1 of Section F, among 10 454 queries from five popular benchmarks, 9285 are α -acyclic, and only 8 of these are not Berge-acyclic. In retrospect, this should not be surprising, as most joins in relational databases are over primary/foreign keys. Emerging workloads in graph databases usually involve simple graphs and seldom require composite key joins. A query without composite key joins admits a *linear* hypergraph, where each pair of hyperedges shares at most one vertex. The following result establishes an equivalence between α -acyclicity with linearity and Berge-acyclicity.

Proposition 26. An α -acyclic hypergraph is Berge-acyclic if and only if it is linear.

By Proposition 26, every edge in the line graph of a Berge-acyclic hypergraph is labeled with a single variable, thus having a weight of 1. Every spanning tree is a maximum spanning tree, therefore a join tree.

Corollary 27. For a Berge-acyclic hypergraph H , any spanning tree of $L(H)$ is a join tree.

In the rest of this section, we prove the existence and uniqueness of the canonical join tree rooted at any relation of a Berge-acyclic hypergraph, and show that it can be constructed by MCS as in Algorithm 1.

5.1 Existence & Uniqueness of the Canonical Join Tree

The key insight leading to the existence and uniqueness of the canonical join tree is that the line graph of a Berge-acyclic hypergraph is *geodetic* [31], meaning that there is a unique shortest path between any pair of vertices. To prove this, we first review some relevant concepts and properties of graphs. All graphs below are simple and undirected.

Definition 28. A cycle in a graph is a sequence of distinct vertices $v_0, \dots, v_{k-1} \geq 2$ such that there is an edge between v_i and $v_{(i+1) \bmod k}$ for all $1 \leq i \leq k$.

A special class of graphs called *chordal graphs* are intimately related to acyclic hypergraphs:

Definition 29. A graph is chordal if every cycle of length at least 4 has a chord, i.e., an edge that is not part of the cycle but connects two vertices of the cycle.

The key step in this section is to show the line graph of a Berge-acyclic hypergraph is a *block graph*, which is a special class of chordal graphs. There are many different characterizations of block graphs [18], and we present a recent one:

Definition 30 (Block Graph [10]). *A graph $G = (R, E)$ with the vertex set R and the edge set E is a block graph if it is chordal and diamond-free.*

Removing an edge from a 4-clique K_4 results in a *diamond*. Diamond-free requires that no induced subgraph $G|_{R'}$ is a diamond, where $R' \subseteq R$ is any subset of vertices. We now present the main result relating block graphs to Berge-acyclic hypergraphs:

Lemma 31. *The line graph L of a Berge-acyclic hypergraph H is a block graph.*

Together with the fact that every block graph is *geodetic* (it has a unique shortest path between any two vertices) [31], Lemma 31 implies the following corollary.

Corollary 32. *Let L be the line graph of a Berge-acyclic hypergraph H . There is a unique shortest path between any two vertices in L .*

We are now ready to prove the existence and uniqueness of the canonical join tree.

Theorem 33. *Let L be the line graph of a Berge-acyclic hypergraph H , and $P(r, r')$ be the shortest path in L between $r, r' \in R(L)$, $T_r = \bigcup_{r' \in R(L)} P(r, r')$ is the unique canonical join tree for H rooted at r .*

Proof. A spanning tree of L is its join tree by Corollary 27. To show T_r is a join tree, it is sufficient to show T_r is spanning, connected and acyclic.

T_r is spanning and connected because it contains a path from r to each $r' \in R(L)$.

We prove T_r is acyclic by induction on the distance $\text{dist}(r, r')$ between r and r' . Let $R_d = \{r' \in R(L) \mid \text{dist}(r, r') \leq d\}$. $R_0 = \{r\}$ contains only the root. The subgraph $T_r|_{R_0}$ is trivially acyclic. Assuming that $T_r|_{R_{d-1}}$ is acyclic, we consider a vertex $r' \in R_{d+1} \setminus R_d$. Corollary 32 guarantees a unique shortest path between each pair of vertices $r, r' \in R(L)$. Each r' is connected to a unique neighbor $r'' \in R_d \setminus R_{d-1}$ that is at distance d from r . Otherwise, there are at least two distinct shortest paths from r to r' . Therefore $T_r|_{R_{d+1}}$ is acyclic, and T_r is a join tree.

The join tree T_r is canonical, because the path from r to each $r' \in R(L)$ is the shortest and therefore minimizing the depth $d(T_r, r')$. The canonical tree is unique by Corollary 32. \square

5.2 Construction of the Canonical Join Tree

Given a Berge-acyclic hypergraph H and $r \in R(H)$ as the root, we can construct the canonical tree $T_r(H)$ by running Algorithm 1. We first present a useful lemma on the structure of the canonical join tree:

Lemma 34. *Let T_r be the canonical join tree rooted at r of a Berge-acyclic hypergraph. Along its root-to-leaf path r, r_1, \dots, r_k , each pair of adjacent vertices shares a distinct variable.*

We now show that the MCS algorithm constructs the canonical join tree.

Theorem 35. *Given a Berge-acyclic hypergraph H , Algorithm 1 from $r \in R(H)$ constructs the canonical tree $T_r(H)$.*

Proof. Algorithm 1 labels each hyperedge in $R(H)$ in ascending order from the root $r_1 = r$ to $r_{n=|R(H)|}$. When Algorithm 1 has labeled $i \in [n]$ hyperedges, we consider a graph G_i whose vertices are the labeled hyperedges $\{r_1, \dots, r_i\}$ and edges are the parent-child relationships $\{(r_2, p(r_2)), \dots, (r_i, p(r_i))\}$.

To prove the claim by induction, we show a loop invariant that G_i is a subtree of T_r for all $i \in [n]$. An empty graph is trivially a subtree of any graph. When the root r is labeled as r_1 , this graph of a single vertex is also trivially a subtree of T_r . Assuming G_k is a subtree of T_r , Algorithm 1 proceeds to label r_{k+1} and assigns its parent $p(r_{k+1}) = r_{p \leq k}$. Suppose for the sake of contradiction that r_{k+1} has a different parent r_q in T_r . Let us consider the possible relationships between r_p and r_q . First, r_q cannot be a descendant of r_p , otherwise there would be a shorter path from r_{k+1} to the root r going through r_p instead of r_q , violating Theorem 33. Suppose r_q is an ancestor of r_p . By the running intersection property, $r_q \cap r_{k+1}$ must

contain $r_p \cap r_{k+1}$, but that would imply every vertex in $r_p \cap r_{k+1}$ is already marked by r_q and the algorithm would not have assigned r_p as the parent of r_{k+1} . Therefore, r_p and r_q are not descendants of each other. Denote their lowest common ancestor in T_r as $r_a = \text{LCA}(r_p, r_q)$. Let $\lambda(r_p, r_q) = \{e_l, e_r\}$ as shown by the solid lines in Figure 5 of Section F, P_p be the path from r_a to r_p excluding e_l and P_q be the path from r_a to r_q excluding e_r as shown by the dashed lines in Figure 5. By Lemma 34, the variables on P_p and P_q are all distinct. Consider the following possible cases:

- Case 1:** If $r_{k+1} \cap r_p = r_{k+1} \cap r_q$ and $\chi(e_l) = \chi(e_r)$, then P_p, P_q form a Berge-cycle.
- Case 2:** If $r_{k+1} \cap r_p = r_{k+1} \cap r_q$ and $\chi(e_l) \neq \chi(e_r)$, then P_p, r_a, P_q form a Berge-cycle.
- Case 3:** If $r_{k+1} \cap r_p \neq r_{k+1} \cap r_q$ and $\chi(e_l) = \chi(e_r)$, then P_p, P_q, r_{k+1} form a Berge-cycle.
- Case 4:** If $r_{k+1} \cap r_p \neq r_{k+1} \cap r_q$ and $\chi(e_l) \neq \chi(e_r)$, then P_p, r_a, P_q, r_{k+1} form a Berge-cycle.

All possible cases above contradict Berge-acyclicity, therefore, Algorithm 1 must correctly assign r_p as the parent of r_{k+1} . \square

6 Converting a Binary Join Plan to a Join Tree

A recent approach [20, 2] that converts a binary join plan into a join tree has gained popularity as it allows system builders to leverage existing query optimizers designed for binary join plans. In this section, we focus on an algorithm by Hu et al. [20] that converts left-deep linear join plans into join trees. The algorithm can convert all acyclic queries in standard benchmarks, including those in Table 1 of Section F. We prove that the algorithm converts any connected left-deep linear join plan into a join tree if and only if the query is γ -acyclic. Our result yields a new characterization of γ -acyclic queries. We formally define binary join plans and describe the algorithm by Hu et al. [20] with Algorithm 6 in Section F.

Definition 36. A left-deep linear plan is a sequence of relations r_1, r_2, \dots, r_n . It is connected if for each $r_{i \geq 2}$, $\exists r_{j < i} : r_i \cap r_j \neq \emptyset$.

Query optimizers strive to produce connected plans, to avoid expensive Cartesian products. Many optimizers produce exclusively left-deep linear plans. Plans that are not left-deep are called *bushy*, and such plans may still be decomposed into left-deep fragments [43].

Given a left-deep linear plan, Hu et al. [20] generate a join tree with Algorithm 6 in Section F. The algorithm chooses the first relation r_1 as the root and iterates through the rest of the plan. For each relation $r_{i \in [2, n]}$, it finds the first relation r_j that contains all attributes shared by r_i with all previous relations $r_i \cap \bigcup r_{k < i}$, and assigns r_j as the parent of r_i . The algorithm constructs a join tree if it finds a parent for each r_i .

Hu et al. [20] proved that the algorithm succeeds whenever the input plan is the reverse of a GYO-reduction order. They also observed that every left-deep linear plan produced for queries in standard benchmarks is indeed the reverse of a GYO-reduction order. This is not a coincidence, as we show that every connected left-deep linear join plan must be the reverse of a GYO-reduction order if and only if the query is γ -acyclic.

Theorem 37. A query is γ -acyclic if and only if every connected left-deep linear join plan for the query is the reverse of a GYO-reduction order.

The “only if” direction is more difficult. We prove its contrapositive by constructing a γ -cycle from a query plan that is not the reverse of a GYO-reduction order. We start with a simple observation that directly follows from the definition of GYO-reduction order.

Proposition 38. If a left-deep linear plan is not the reverse of a GYO-reduction order, then there exists a relation r_i that has no parent among $\{r_{j < i}\}$, formally $\neg \exists r_{j < i} : (r_i \cap \bigcup r_{k < i}) \subseteq r_j$. We call r_i an orphan, and denote it as \hat{r}_i .

We introduce an ordering on relations to compare them in a plan.

Definition 39. Given a relation r , we write $r_1 \geq_r r_2$ if $r_1 \cap r \supseteq r_2 \cap r$.

We first observe that \geq_r is a preorder (reflexive and transitive), but not a partial order (antisymmetry may fail). Moreover, if there is a greatest element r_p with respect to \geq_r , i.e., $\forall r' : r_p \geq_r r'$, then r_p is a parent of r . Those observations lead to the following.

Lemma 40. Let \hat{r} be a maximal relation with respect to $\geq_{\hat{r}}$ for an orphan \hat{r} , meaning $\neg \exists r : r >_{\hat{r}} \hat{r}$, then there is a relation \bar{r} incomparable with \hat{r} , i.e., $\hat{r} \not\geq_{\hat{r}} \bar{r}$ and $\bar{r} \not\geq_{\hat{r}} \hat{r}$.

Proof. If every relation is comparable with \hat{r} , the maximal relation \hat{r} is the greatest element with respect to $\geq_{\hat{r}}$, thus a parent of \hat{r} . It contradicts the fact that \hat{r} is an orphan. \square

Given a pair of incomparable relations \hat{r} and \bar{r} with respect to \hat{r} , we can find at least one variable in each relation that does not appear in the other relation but appears in \hat{r} , namely $\exists \bar{x} \in \bar{r} \cap \hat{r} : \bar{x} \notin \hat{r}$ and $\exists \hat{x} \in \hat{r} \cap \bar{r} : \hat{x} \notin \bar{r}$. We say that \hat{x} is a *dangling variable* of \hat{r} and \bar{x} is a *dangling variable* of \bar{r} . We are now ready to prove Theorem 37. Our strategy is to derive a γ -cycle from any orphan relation, leading to a contradiction.

Proof of Theorem 37. To prove the “only if” direction by contradiction, we assume that a connected left-deep linear plan for a γ -acyclic query is not the reverse of a GYO-reduction order. By Proposition 38, there is an orphan \hat{r} in the plan. Assuming that \hat{r} is the t -th relation in the plan, we now consider the prefix of the plan up to \hat{r} , namely $\text{pre}(t) = r_1, \dots, r_{t-1}, \hat{r}$. Let $\hat{r} \in \text{pre}(t-1)$ be a maximal relation with respect to $\geq_{\hat{r}}$, Lemma 40 guarantees an $\bar{r} \in \text{pre}(t-1)$ incomparable with \hat{r} . There are at least two dangling variables $\bar{x} \in \bar{r} \cap \hat{r} \setminus \hat{r}$ and $\hat{x} \in \hat{r} \cap \bar{r} \setminus \bar{r}$.

Because the shorter prefix $\text{pre}(t-1) = r_1, \dots, r_{t-1}$ is connected, there is a path in $\text{pre}(t-1)$ between \hat{r} and \bar{r} . The incomparability guarantees that $\bar{r} \cap \hat{r} \not\subseteq \hat{r} \cap \bar{r}$. We can choose a shortest path in $\text{pre}(t-1)$ that connects a variable in $\hat{r} \cap \bar{r}$ to a variable in $\bar{r} \setminus \hat{r}$. We now consider two exhaustive cases where each yields a γ -cycle:

Case 1: If the shortest path consists of a single relation \tilde{r} , $\tilde{r} \not\geq_{\hat{r}} \hat{r}$ (because \hat{r} is maximal) and $\tilde{r} \not\leq_{\hat{r}} \hat{r}$ (because \tilde{r} contains some variable in $\hat{r} \setminus \hat{r}$). Therefore, \tilde{r} is incomparable with \hat{r} . Since $\tilde{r} \cap \hat{r} \neq \emptyset$, $\hat{r}, \hat{x}, \tilde{r}, \tilde{x}, \bar{r}$ form a γ -cycle where $\hat{x} \in \hat{r} \cap \tilde{r}$, $\tilde{x} \in \tilde{r} \cap \bar{r}$, and $\bar{x} \in \bar{r} \cap \hat{r}$.

Case 2: If the shortest path consists of at least two relations, let $r_0 = \hat{r}$, and the path be r_1, \dots, r_{k-1} .

We now show that r_0, r_1, \dots, r_{k-1} form a *pure cycle* where each r_i is only adjacent to $r_{(i-1) \bmod k}$ and $r_{(i+1) \bmod k}$. First, if a $r_{i \in [2, k-2]}$ on the path is adjacent to a $r_{j \in [1, k-1]}$ other than its neighbors r_{i-1} or r_{i+1} , the path can be shortened, contradicting the shortest path assumption. Second, if the relation $r_{i \in [2, k-2]}$ is adjacent to r_0 , then r_i must contain variables in either $r_0 \cap \hat{r}$ or $r_0 \setminus \hat{r}$. We can shrink the path to r_1, \dots, r_i or r_i, r_{k-1} , which also contradicts the shortest path assumption. Therefore, each $r_{i \in [0, k-1]}$ on the cycle is only adjacent to $r_{(i-1) \bmod k}$ and $r_{(i+1) \bmod k}$. The cycle r_0, r_1, \dots, r_{k-1} is a pure cycle of length $k \geq 3$. A pure cycle is always a γ -cycle.

This concludes the proof for the “only if” direction.

Now we prove the “if” direction by contrapositive, namely that if a query contains a γ -cycle, then there exists a connected left-deep linear plan that is not the reverse of a GYO-reduction order. Let the γ -cycle of length $k \geq 3$ be $r_0, \dots, r_i, \dots, r_{k-1}$ where $r_{i \in [1, k-2]}$. By definition, there is an x_{i-1} appearing exclusively in r_{i-1} and r_i and an x_i appearing exclusively in r_i and r_{i+1} . We modify the original plan by moving r_i after r_{k-1} such that $r_0, \dots, r_{i-1}, r_{i+1}, \dots, r_{k-1}, r_i$ becomes part of the new query plan.

The prefix $\text{pre}(k-1) = r_0, \dots, r_{i-1}, r_{i+1}, \dots, r_{k-1}$ is connected. Therefore, the new query plan is connected. However, r_i has no parent, because no relation in $\text{pre}(k-1)$ contains both x_{i-1} and x_i . This implies the plan is not the reverse of a GYO-reduction order. \square

7 Conclusion and Future Work

We proposed three approaches for constructing join trees. Our enumeration algorithm in Section 4 generates join trees by edits with amortized constant delay, forming a basis for cost-based optimization of acyclic joins.

In Section 5, we showed that the Maximum Cardinality Search algorithm constructs the unique shallowest join tree for any Berge-acyclic query, supporting very large queries and enabling parallel execution. Finally, in Section 6, we gave theoretical justification for converting binary join plans to join trees, allowing reuse of existing optimization infrastructure.

Future work includes compact representations of join trees for dynamic programming, as in binary plan optimizers, and the challenging cost estimation for Yannakakis-style algorithms: the random-walk approach [26] models joint probabilities for binary joins, whereas an efficient and accurate solution for semijoins remains to be found.

Our results also raise theoretical questions. Can join tree enumeration achieve worst-case constant delay? We proved *Berge-acyclicity* sufficient for the existence and uniqueness of the canonical join tree, but it is not necessary, and γ -*acyclicity* is insufficient as shown in Figure 6 of Section F. What is the precise characterization of hypergraphs that admit a unique canonical join tree for any root, or for *some* root?

References

- [1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: <http://webdam.inria.fr/Alice/>.
- [2] Liese Bekkers, Frank Neven, Stijn Vansumeren, and Yisu Remy Wang. Instance-optimal acyclic join processing without regret: Engineering the yannakakis algorithm in column stores. *Proc. VLDB Endow.*, 18(8):2413–2426, 2025. URL: <https://www.vldb.org/pvldb/vol18/p2413-vansumeren.pdf>.
- [3] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.
- [4] Michael A Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.
- [5] Nicolas Bruno, César A. Galindo-Legaria, and Milind Joshi. Polynomial heuristics for query optimization. In *Proceedings of the 26th International Conference on Data Engineering (ICDE)*, pages 589–600. IEEE, 2010.
- [6] Nofar Carmeli, Batya Kenig, and Benny Kimelfeld. Efficiently enumerating minimal triangulations. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 273–287. ACM, 2017. doi:10.1145/3034786.3056109.
- [7] Arthur Cayley. A theorem on trees. *Quarterly Journal of Pure and Applied Mathematics*, 23:376–378, 1889.
- [8] Jeremy Chen, Yuqing Huang, Mushi Wang, Semih Salihoglu, and Ken Salem. Accurate summary-based cardinality estimation through the lens of cardinality estimation graphs. *Proceedings of the VLDB Endowment*, 15(8):1533–1545, 2022.
- [9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 4th edition, 2022. Sections 22.2–22.3.
- [10] Pablo De Caria and Marisa Gutierrez. On basic chordal graphs and some of its subclasses. *Discrete Applied Mathematics*, 210:261–276, 2016. LAGOS’13: Seventh Latin-American Algorithms, Graphs, and Optimization Symposium, Playa del Carmen, México — 2013. URL: <https://www.sciencedirect.com/science/article/pii/S0166218X15002243>, doi:10.1016/j.dam.2015.05.002.
- [11] David DeHaan and Frank Wm. Tompa. Optimal top-down join enumeration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 785–796, 2007.

[12] David Eppstein. Representing all minimum spanning trees with applications to counting and generation. UC Irvine, 1995.

[13] Ronald Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM (JACM)*, 30(3):514–550, 1983.

[14] Leonidas Fegaras. A new heuristic for optimizing large queries. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications (DEXA)*, volume 1460 of *Lecture Notes in Computer Science*, pages 726–735. Springer, 1998.

[15] Pit Fender and Guido Moerkotte. Counter strike: Generic top-down join enumeration for hypergraphs. *Proceedings of the VLDB Endowment (VLDB)*, 6(14):1822–1833, 2013.

[16] Georg Gottlob, Matthias Lanzinger, Cem Okulmus, and Reinhard Pichler. Fast parallel hypertree decompositions in logarithmic recursion depth. *ACM Trans. Database Syst.*, 49(1):1:1–1:43, 2024. doi: 10.1145/3638758.

[17] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. Cardinality estimation in DBMS: A comprehensive benchmark evaluation. *Proc. VLDB Endow.*, 15(4):752–765, 2021. URL: <https://www.vldb.org/pvldb/vol15/p752-zhu.pdf>, doi:10.14778/3503585.3503586.

[18] Frank Harary. A characterization of block-graphs. *Canadian Mathematical Bulletin*, 6(1):1–6, 1963.

[19] Zongyan He and Jeffrey Xu Yu. A branch-\u0026-bound algorithm for fractional hypertree decomposition. *Proc. VLDB Endow.*, 17(13):4655–4667, 2024. URL: <https://www.vldb.org/pvldb/vol17/p4655-he.pdf>.

[20] Zeyuan Hu, Yisu Remy Wang, and Daniel P Miranker. Treetracker join: Simple, optimal, fast. *arXiv preprint arXiv:2403.01631*, 2024.

[21] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984. doi:10.1145/1270.1498.

[22] Sanjiv Kapoor and H. Ramesh. Algorithms for enumerating all spanning trees of undirected and weighted graphs. *SIAM Journal on Computing*, 24(2):247–265, 1995. arXiv:<https://doi.org/10.1137/S009753979225030X>, doi:10.1137/S009753979225030X.

[23] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 128–137. Morgan Kaufmann, 1986. URL: <http://www.vldb.org/conf/1986/P128.PDF>.

[24] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[25] Arne Leitert. Computing the union join and subset graph of acyclic hypergraphs in subquadratic time. In Anna Lubiw and Mohammad R. Salavatipour, editors, *Algorithms and Data Structures - 17th International Symposium, WADS 2021, Virtual Event, August 9-11, 2021, Proceedings*, volume 12808 of *Lecture Notes in Computer Science*, pages 571–584. Springer, 2021. doi:10.1007/978-3-030-83508-8_41.

[26] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join: Online aggregation via random walks. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 615–629. ACM, 2016. doi:10.1145/2882903.2915235.

[27] David Maier. *The theory of relational databases*, volume 11. Computer science press Rockville, 1983.

[28] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 930–941. VLDB Endowment, 2006.

[29] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 539–552. ACM, 2008. doi:10.1145/1376616.1376672.

[30] Thomas Neumann. Query simplification: Graceful degradation for join-order optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 403–414, 2009.

[31] Øystein Ore. *Theory of Graphs*, volume 38 of *American Mathematical Society Colloquium Publications*. American Mathematical Society, Providence, RI, 1962. doi:10.1090/coll/038.

[32] Benjamin Scheidt and Nicole Schweikardt. Counting homomorphisms from hypergraphs of bounded generalised hypertree width: A logical characterisation. In Jérôme Leroux, Sylvain Lombardy, and David Peleg, editors, *48th International Symposium on Mathematical Foundations of Computer Science, MFCS 2023, August 28 to September 1, 2023, Bordeaux, France*, volume 272 of *LIPICS*, pages 79:1–79:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: <https://doi.org/10.4230/LIPIcs.MFCS.2023.79>. doi:10.4230/LIPIcs.MFCS.2023.79.

[33] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, Boston, MA, 4th edition, 2011. Chapter 4: BreadthFirstPaths.

[34] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[35] Akiyoshi Shioura, Akihisa Tamura, and Takeaki Uno. An optimal algorithm for scanning all spanning trees of undirected graphs. *SIAM J. Comput.*, 26(3):678–692, 1997. doi:10.1137/S0097539794270881.

[36] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, 1997.

[37] Vaishali Surianarayanan, Anikait Mundhra, Ajaykrishnan E. S, and Daniel Lokshtanov. Fast hypertree decompositions via linear programming: Fractional and generalized. *Proc. ACM Manag. Data*, 3(3):159:1–159:27, 2025. doi:10.1145/3725296.

[38] Arun N. Swami. Optimization of large join queries: Combining heuristic and combinatorial techniques. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 367–376, 1989.

[39] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. arXiv:<https://doi.org/10.1137/0201010>, doi:10.1137/0201010.

[40] Robert E Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on computing*, 13(3):566–579, 1984.

[41] Transaction Processing Performance Council. TPC Benchmark H (Decision Support). https://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf, 2023. Standard Specification, Revision 3.0.1.

[42] 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):235:1–235:28, 2025. doi:10.1145/3725423.

[43] Yisu Remy Wang, Max Willsey, and Dan Suciu. Free join: Unifying worst-case optimal and traditional joins. *Proceedings of the ACM on Management of Data*, 1(2):1–23, 2023.

[44] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, page 82–94. VLDB Endowment, 1981.

[45] C. T. Yu and M. Z. Ozsoyoglu. 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. doi: 10.1109/CMPSC.1979.762509.

[46] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun’ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 3911–3921. Association for Computational Linguistics, 2018. URL: <https://doi.org/10.18653/v1/d18-1425>, doi:10.18653/V1/D18-1425.

[47] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huachen Zhang. Debunking the myth of join ordering: Toward robust sql analytics. *Proceedings of the ACM on Management of Data*, 3(3):1–28, 2025.

[48] Yun-zhou Zhu. Line graph of gamma-acyclic database schemes and its recognition algorithm. In *VLDB*, pages 218–221, 1984.

A Extended Preliminaries

Definition 41 (Hypergraph Acyclicity). *A hypergraph H is:*

- α -acyclic if it admits a join tree;
- β -acyclic if every subgraph of H is α -acyclic;
- γ -acyclic if it does not contain any γ cycle. A γ cycle is a sequence of length $k \geq 3$ of distinct hyperedges and distinct vertices $(r_0, x_0, \dots, r_{k-1}, x_{k-1})$ such that every $x_{i \in [0, k-2]}$ belongs to $r_i \cap r_{i+1}$ and no other r_j while x_{k-1} belongs to $r_{k-1} \cap r_0$ and possibly other hyperedges;
- Berge-acyclic if it does not contain any Berge cycle. A Berge cycle is a sequence of length $k \geq 2$ of distinct vertices and distinct hyperedges $(r_0, x_0, \dots, r_{k-1}, x_{k-1})$ such that $\forall i \in [k] : x_i \in r_i \cap r_{(i+1) \bmod k}$.

The four notions of acyclicity form a strict hierarchy that $\text{Berge-acyclic} \Rightarrow \gamma\text{-acyclic} \Rightarrow \beta\text{-acyclic} \Rightarrow \alpha\text{-acyclic}$ [13].

Given a hypergraph H with n hyperedges, Algorithm 1 labels each hyperedge with an integer $i \in [1, n]$ in ascending order and assigns a parent $p(r_i)$ to each hyperedge r_i . Upon labeling an edge the algorithm marks all previously unmarked vertices contained in that edge. Initially, all hyperedges are unlabeled and all vertices are unmarked. The algorithm begins by assigning *null* as the parent of the root r and initializing label i to 0. In each following while loop iteration, the algorithm increments i by one and labels the current hyperedge as r_i . For each unmarked vertex x in r_i (a vertex is considered marked when it is removed from X), it assigns r_i tentatively as the parent of each unlabeled hyperedge r' containing x . The algorithm marks all unmarked vertices in r_i and removes r_i from future consideration. Finally, it selects a hyperedge with the most marked vertices to label next, breaking ties arbitrarily. The algorithm terminates when all hyperedges are labeled. Tarjan and Yannakakis [40] describes an efficient implementation of Algorithm 1 running in $\mathcal{O}(|H|)$.

Example 42. Considering the hypergraph H_6 shown in Figure 1a, we run Algorithm 1 from the hyperedge P . It is labeled as r_1 and all its vertices are marked. P is also tentatively assigned as the parent of five hyperedges such that $p(S) = p(T) = p(U) = p(W) = p(Y) = r_1$. We proceed to label the next hyperedge with the most marked vertices. At this point, each unlabeled hyperedge has one marked vertex, so we can choose any of them, say S , and label it as r_2 .

Because the labeling of S marks no additional vertices in T, U, W, Y , their parents remain unchanged. These remaining hyperedges are again tied with the same number of marked vertices, so we can choose any of them, say T , and label it as r_3 . Since the labeling of T marks a vertex c in U, W , their parents are updated to $p(U) = p(W) = r_3$.

We continue the process until all hyperedges are labeled and all vertices are marked. By connecting each labeled hyperedge with its parent, we construct the join tree T^M shown in Figure 1d.

Notice that Algorithm 1 can generate different join trees for the same hypergraph by breaking ties differently. However, it does not guarantee to generate all possible trees. For example, it never generates the Join Tree T^G shown in Figure 1c.

B Data Structures for Constant Time Operations on Graphs

Given line graph L and MCS tree T , Algorithm 3 iterates over non-tree edges of L . Moreover, the algorithm uses data structures $d(\cdot)$, $LA(\cdot)$ and $LCA(\cdot)$, in determining the endpoints of a non-tree edge's LCA edges. In this section we describe briefly how iterating over non-tree edges can be achieved in time linear in the number of vertices and edges of L and how determining the endpoints of a non-tree edge's LCA edges can be achieved in constant time.

Given a graph $G = (V, E)$, we assume the vertices are represented by integers from 1 to $|V|$. Given a tree T of G rooted at node r , recall that data structure $d_T(\cdot)$, stores for each node r_i the length of the shortest path between r and r_i in T . Data structure $LCA_T(\cdot)$, stores for each pair of nodes r_i, r_j the lowest common ancestor of r_i, r_j in T . Finally, data structure $LA_T(\cdot)$ stores for each node-integer pair r_i, j , the ancestor of node r_i at depth j . Each of these data structures can be established from a tree T in time linear in the size of the tree and support constant time look-ups.

Observation 43. Given rooted tree T , data structures $d(\cdot)$, $LCA(\cdot)$ and $LA(\cdot)$ can be established in time $\mathcal{O}(|T|)$ and support $\mathcal{O}(1)$ time look-ups [9, 33, 3, 4].

Given the line graph L of an α -acyclic hypergraph H and an MCS tree T of H , recall that the LCA edges $\lambda(e)$ of a non-tree edge $e = (r_i, r_j)$ of T is defined as the set of edges incident to $LCA(r_i, r_j)$ that lie on the unique path between r_i and r_j in T . The following observation shows that an edges LCA edges can be found in constant time given data structures $d(\cdot)$, $LCA(\cdot)$ and $LA(\cdot)$.

Observation 44. Given the line graph L of an α -acyclic hypergraph H , an MCS tree T and data structures $d(\cdot)$, $LCA(\cdot)$ and $LA(\cdot)$, the LCA edges of any non-tree edge can be found in constant time.

Proof. Given a non-tree edge $e = (r_i, r_j)$ we find $LCA(r_i, r_j) = l$ in constant time by Observation 43. The depth $d = d(l) + 1$ is likewise found in constant time. The edge $e_1 = (l, r_1)$, where node $r_1 = LA(r_i, d)$ is found by a constant time look-up, is adjacent to l and lies on the unique path between r_i and r_j . The same holds for edge $e_2 = (l, r_2)$ where node $r_1 = LA(r_j, d)$. \square

In what remains of this section we describe how Algorithm 3 iterates over and performs sliding transformations on non-tree edges.

We use an array L to represent the adjacency list of a line graph where entry $L[i]$ consists of a linked list of tuples (j, ω) where j is a neighbor of i and ω is the weight of the edge (i, j) . A rooted tree T_r is likewise implemented as an array, where the index i represents the vertex i , and the i -th entry $T_r[i]$ stores (p, ω) where p is the parent of i and ω is the weight of the edge (i, p) . The depth table of the tree is also implemented as an array to enable constant time lookup for the depth of a vertex $d(T_r, i)$.

Section C shows how to construct the weighted line graph and the hypergraph given the original query. Then, using the hypergraph we can construct T_r and d during the MCS algorithm in linear time [40].

As we iterate over each edge (j, ω) of $L[i]$, we check if the current edge (i, j) is a tree edge by performing constant-time query of $T_r[i]$ and $T_r[j]$ to see if one of them is the parent of the other. From Observation 44 we know that we can find the LCA edges $\lambda(i, j) = \{e_1, e_2\}$ in constant time. There's no general solution to lookup an entry in a linked list in constant time. Instead of iterating over the linked-list representation of L to find the edge weights of e_1 and e_2 , we can lookup their weights from the array representation of T_r in constant time. Let $e_1 = (r_1, p(r_1))$ and $e_2 = (r_2, p(r_2))$ with weights ω_1 and ω_2 . Because both e_1 and e_2 are tree edges, we can find their weights by querying $T_r[r_1] = (p(r_1), \omega_1)$ and $T_r[r_2] = (p(r_2), \omega_2)$. Each of these operations takes constant time. When we need to delete an edge $e^* = (i, j)$ from L , we first check if e^* is a tree edge by checking $T_r[i]$ and $T_r[j]$. Deleting e^* from the linked list takes constant time. Sliding an edge involves updating the endpoints of the edge which takes constant time.

C From Queries to (Hyper-)Graphs

Algorithm 3 assumes the weighted line graph and an initial MCS tree are given as input. In practice, we need to construct these graphs from the original SQL query.

We model a query involving m relations as a set of join predicates each in the form of $R_a.x_i = R_b.x_j$ such that $a \neq b \wedge a, b \in [m]$, namely no self-joins or duplicates, and assume each relation is uniquely identified by an integer in $[m]$. For each relation R_a with arity k_a , each attribute $x_{i \in [k_a]}$ is uniquely identified by an integer in $[k_a]$. Therefore each predicate is uniquely identified by a tuple (a, i, b, j) . Each query Q is represented by a set of such tuples. We will assume that query Q is transitively closed over join predicates, i.e. if $R_a.x_i = R_b.x_j$ and $R_b.x_j = R_c.x_k$ are in Q , then $R_a.x_i = R_c.x_k$ is also in Q . The tuple (a, i, b, j) is present in Q if and only if $R_a.x_i = R_b.x_j$ is a join predicate over relations R_a, R_b . We define the size of a query $|Q|$ as the number of join predicates in Q , equivalent to the number of (a, i, b, j) tuples. Our goal is to construct the query hypergraph and the weighted line graph from this set of tuples.

C.1 Construction of Hypergraph

We construct the hypergraph H by finding connected components of the *predicate graph*, which is a graph where there is a vertex for each attribute $R_a.x_i$, and there is an edge between $R_a.x_i$ and $R_b.x_j$ for every predicate $R_a.x_i = R_b.x_j$. Each connected component of the predicate graph corresponds to an “equivalence class” of attributes. We then assign a unique variable to each connected component, and map each attribute to the variable of its component. This can be done in linear time using standard algorithms for computing connected components [9, 39].

Algorithm 4 generates hypergraph H from query Q . H is represented as an array of size m where each entry is a list of integers. The algorithm starts by initializing H to an array of size m and P to an array of size p , where p is the number of distinct relation-attribute pairs $R_a.x_i$ in Q as shown by Lines 1-4. If numbers m and p are not known a priori they can be found in a constant number of passes over elements in Q . Array H will serve as a hyperedge list of the query hypergraph and array P will serve as an adjacency list representation of the query predicate graph. The subsequent for-loop fills out adjacency list P at Lines 5-7. We assume here that each (a, i) pair is mapped to a unique integer in $[p]$ by a perfect hash function f at Line 6, otherwise such a mapping can be established in a constant number of passes over Q . Next, a standard algorithm [9, 39] for the enumeration of connected components of a graph is called in Line 8. This produces array \mathcal{C} where each entry is a list of vertices that forms a connected component in P . The subsequent nested for-loop iterates over each relation in each component to fill out hyperedge list H at Lines 9-11.

Observation 45. *Algorithm 4 generates the hypergraph of Q in time $\mathcal{O}(|Q|)$.*

Proof. Arrays H and P can be initialized in time $\mathcal{O}(|Q|)$ since $m \leq |Q|$ and $p \leq |Q|$. The subsequent for-loop iterates over all elements in $|Q|$ making a single insertion to P thus performing at most $\mathcal{O}(|Q|)$ steps. Predicate graph P therefore has size at most $\mathcal{O}(|Q|)$. Generating a list of connected components of predicate graph P is done in time linear in $|P| = \mathcal{O}(|Q|)$ by standard algorithms [9, 39]. The subsequent nested for-loop iterates over each relation in each component performing a single insertion on each iteration. The sum of the sizes of each component is exactly number of vertices of P and therefore $\mathcal{O}(|Q|)$ elementary operations are performed. \square

Algorithm 4: Generating query hypergraph H from query Q .

```

Input: Query  $Q$ 
Output: Query hypergraph  $H$ 
1 for  $i \in [1 \dots m]$  do
2    $H[i] \leftarrow \text{linkedList}()$ 
3 for  $i \in [1 \dots p]$  do
4    $P[i] \leftarrow \text{linkedList}()$ 
5 for  $(a, i, b, j) \in Q$  do
6    $p = f(a, i)$ 
7    $P[p].append(b, j)$ 
8  $\mathcal{C} \leftarrow \text{connectedComponent}(P)$ 
9 for  $i \in [|\mathcal{C}|]$  do
10  for  $R_a \in \mathcal{C}[i]$  do
11     $H[a].append(i)$ 
12 return  $H$ 

```

C.2 Construction of Weighted Line Graph

To construct the weighted line graph, we will ignore the attribute indices (i, j) and focus on the relation indices (a, b) for each join predicate in Q . The set of predicates can then be seen as representing the edge list of a multigraph. Algorithm 5 generates the weighted line graph of Q by converting this multigraph into a weighted graph L where the weight of each edge (u, v) is the number of parallel edges between u and v in the multigraph.

The algorithm first initializes arrays L and M both of size m , where M will be an adjacency list representation of a multigraph and L will be adjacency list representation of the final weighted line graph in Line 2-3. Auxiliary array **weight** is initialized in Line 4. Entries of **weight** are initialized to be an empty linked list. In the subsequent for-loop, "edges" in Q are inserted into multigraph M in Line 6. Array **weight**[j] stores the weight of the edge (i, j) for the current i . Entries of **weight**[j] are initialized to 0. The subsequent for-loop iterates over adjacency list M in Line 7. The first nested for-loop at Line 8 iterates over each neighbor j of i stored in $M[i]$. For each neighbor j , array **weight**[j] is incremented by 1. Array **weight**[j] now contains the weights of all edges adjacent to i at its non-zero entries. The second nested for-loop from Line 10 to Line 13 iterates over each neighbor j one more time and inserts neighbor and weight pairs into array L if **weight**[j] is not 0, otherwise skipping to the next neighbor. After an insertion is made, **weight**[j] is set to 0. A weight of 0 at **weight**[j] indicates the edge at (i, j) and its weight has already been outputted. This prevents the insertion of duplicate edges and resets array **weight**[j] to be reused for the next relation in M . Since each array operation takes constant time and we visit each edge a constant number of times, the overall complexity of this procedure is linear in the size of the input.

Observation 46. *Algorithm 5 generates the weighted line graph of Q in time $\mathcal{O}(|Q|)$.*

Proof. Array initializations for L and M take $\mathcal{O}(m)$ time where $m \leq |Q|$. The first for-loop scans all predicates in Q and inserts the corresponding edges in the multigraph M , performing $\mathcal{O}(|Q|)$ insertions, so $|M| = \mathcal{O}(|Q|)$. The auxiliary array **weight** has size m and is initialized in $\mathcal{O}(m)$ time.

In the subsequent traversal of M , each adjacency list is visited once. For every neighbor j of a vertex i , the first inner loop performs a constant-time update of **weight**[j]; the second inner loop performs at most one constant-time insertion of $(j, \text{weight}[j])$ into L and resets **weight**[j]. Thus each stored neighbor (i.e., each edge occurrence in M) incurs only $\mathcal{O}(1)$ work overall. Since the total number of stored neighbors is $\Theta(|M|) = \mathcal{O}(|Q|)$, the time for these loops is $\mathcal{O}(|Q|)$.

Summing up, all steps are linear in $|Q|$, hence Algorithm 5 runs in time $\mathcal{O}(|Q|)$. \square

Algorithm 5: Generating weighted line graph L from query Q .

Input: Query Q
Output: Weighted line graph L

```

1 for  $i \in [1 \dots m]$  do
2    $L[i] \leftarrow \text{linkedList}()$ 
3    $M[i] \leftarrow \text{linkedList}()$ 
4    $\text{weight}[i] \leftarrow 0$ 
5 for  $(a, i, b, j) \in Q$  do
6    $M[a].append(b)$ 
7 for  $R_i \in M$  do
8   for  $R_j \in M[i]$  do
9      $\text{weight}[j] \leftarrow \text{weight}[j] + 1$ 
10  for  $R_j \in M[i]$  do
11    if  $\text{weight}[j] \neq 0$  then
12       $L[i].append(R_j, \text{weight}[j])$ 
13       $\text{weight}[j] \leftarrow 0$ 
14 return  $L$ 

```

D Missing Proofs in Section 4

Lemma 15 *If $H' \twoheadrightarrow H$ then $\mathcal{T}(H) \subseteq \mathcal{T}(H')$.*

Proof. Let T be a join tree of H , and x' be any vertex in H' . Consider the neighborhood $H'|_{x'}$ of x' in H' comprising the set of hyperedges $\{r' \in H' \mid x' \in r'\}$. Since $H' \twoheadrightarrow H$, we have that $x' \in r'$ if and only if $f(x') \in f(r') = r'$. The neighborhood $H|_{f(x')}$ of $f(x')$ in H comprises of the set of hyperedges $\{r \in H \mid f(x') \in r\}$. Therefore, $r' \in H'|_{x'} \iff f(r') = r' \in H|_{f(x')}$. Because T is a join tree of H , $H|_{f(x')}$ is connected in T , therefore $H'|_{x'}$ is also connected in T and T is also a join tree of H' . \square

Lemma 16 *Let T be an MCS tree. For any edge $e \in T$ that has a parent, then $e \not\subseteq p(e)$.*

Proof. In Algorithm 1, a child r_c is only connected to a parent r if they share some previously unmarked vertex x . Because every vertex is marked only once, and the parent r_p of r was labeled before r , x cannot be marked by r 's parent r_p . Let the e be edge between r and r_c , then the edge between r and r_p is $p(e)$, and $x \in e$ but $x \notin p(e)$, therefore $e \not\subseteq p(e)$. \square

Lemma 17 *Let T be an MCS tree. For two edges $e, e' \in T$ that have parents,*

$$(e \setminus p(e)) \cap (e' \setminus p(e')) \neq \emptyset \implies p(e) = p(e').$$

Proof. We prove by contrapositive, assuming $p(e) \neq p(e')$.

Case 1: $p(e') = e$, then $(e \setminus p(e)) \cap (e' \setminus e) \subseteq (e \cap e') \setminus e = \emptyset$.

Case 2: $p(e') \neq e$. Without loss of generality, we suppose e is no further from the root than e' . Let $p(e') = e''$, then e'' is on the path between e and e' in T . If $(e \setminus p(e)) \cap (e' \setminus e'') \neq \emptyset$, then $\exists x \in (e \cap e') \setminus e''$, which violates the running intersection property of join trees.

\square

Lemma 19 *Let H' be a duplicated hypergraph of H . Then,*

1. $H' \twoheadrightarrow H$ and $H' \leftarrow H$

2. $\mathcal{T}(H') = \mathcal{T}(H)$

3. $L(H') = L(H)$

Proof. 1. We only need to show that $H' \rightarrow H$ ($H' \leftarrow H$) holds for H' derived from H by a single vertex duplication. $H' \rightarrow H$ ($H' \leftarrow H$) will then hold for any duplicate H' derived by a series of vertex duplications since strong homomorphisms are closed under composition.

Suppose H' is derived from H by duplicating $x \in X(H)$ to x' .

Consider a pair of functions $(f_X : X(H') \mapsto X(H), R(H)_{\text{id}})$ where f_X maps x' to x . Otherwise, it is an identity map $X(H)_{\text{id}}$. This function pair constitutes a strong hypergraph homomorphism $H' \rightarrow H$. Likewise, the pair of functions $(X(H)_{\text{id}}, R(H)_{\text{id}})$ constitutes a homomorphism $H \rightarrow H'$.

2. It follows from Lemma 15 and the facts above.

3. By the strong homomorphisms, the hyperedge sets have the same cardinality $|R(H)| = |R(H')|$. So do the vertex sets of the line graphs, $R(L(H)) = R(H)$ and $R(L(H')) = R(H')$. By construction, the vertex duplication adds x' to each hyperedge containing x , therefore does not create any additional edges in the line graph, $E(L(H')) = E(L(H))$. \square

E Missing Proofs in Section 5

Proposition 26 *An α -acyclic hypergraph is Berge-acyclic if and only if it is linear.*

Proof. Fagin [13] showed that every Berge-acyclic hypergraph is linear. We prove the other direction by contradiction, by assuming that H is α -acyclic and linear. We also assume that there is a Berge cycle $(r_0, x_0, \dots, r_{k-1}, x_{k-1})$ of length $k \geq 2$. Without loss of generality, we suppose r_1 is the first hyperedge on the cycle to be removed by GYO reduction. Then r_1 's parent r_p must satisfy $r_p \cap r_1 \supseteq (r_1 \cap \bigcup_{i \in [k] \setminus \{1\}} r_i) = \{x_0, x_1\}$, contradicting H 's linearity. \square

Lemma 31 *The line graph L of a Berge-acyclic hypergraph H is a block graph.*

Proof. A Berge-acyclic hypergraph is γ -acyclic [13]. Zhu proves that the line graph of a γ -acyclic hypergraph is chordal [48]. We now prove that the line graph is diamond-free by showing that every diamond must be part of a K_4 . Assuming a diamond (r_1, r_2, r_3, r_4) in L without the edge (r_1, r_3) , it can be treated as two triangles (r_1, r_2, r_4) and (r_2, r_4, r_3) sharing an edge (r_2, r_4) . We now consider the possible edge labelings in each triangle:

Case 1: If all edges are labeled with distinct variables, each triangle forms a Berge-cycle, contradicting the Berge-acyclicity.

Case 2: If two edges are labeled with the same variable, all three vertices share that variable. The third edge must be labeled with the same variable.

Since the two triangles share the edge (r_2, r_4) , all their edges are labeled with the same variable x , namely $x \in r_1 \cap r_2 \cap r_3 \cap r_4$. There has to be an edge $(r_1, r_3) \in L$, contradicting the assumption that (r_1, r_2, r_3, r_4) is a diamond. \square

Lemma 34 *Let T_r be the canonical join tree rooted at r of a Berge-acyclic hypergraph. Along its root-to-leaf path r, r_1, \dots, r_k , each pair of adjacent vertices shares a distinct variable.*

Proof. Let $r_0 = r$, Proposition 26 requires that $|r_{i-1} \cap r_i| = 1$ for all $i \in [k]$. By Theorem 33, the path r_0, \dots, r_k is the shortest path from r_0 to r_k . The variable shared by two adjacent vertices r_{i-1}, r_i is equivalent to the edge label $\chi(r_{i-1}, r_i)$. The edge labels along the path are all distinct. Otherwise, two consecutive edges with the same label allows for shortening the path by removing either, contradicting the shortest path property. Two non-consecutive edges with the same label violate the running intersection property. \square

F Additional Figures, Tables and Algorithms

Name	# Queries	# α -Acyclic	# Composite-Key Joins	# Berge-Acyclic
TPC-H[41]	22	21	2	19
JOB[24]	113	113	0	113
STATS-CEB[17]	2603	2603	0	2603
CE[8]	3004	1839	0	1839
Spider-NLP[46]	4712	4709	6	4703

Table 1: Acyclic queries in the benchmarks (all α -acyclic queries are also γ -acyclic.)

Algorithm 6: Converting a binary plan to a join tree

Input: Left-deep plan r_1, r_2, \dots, r_n
Output: Tree rooted at r_1

```

1  $p(r_1) \leftarrow \text{null}$ 
2 for  $i \leftarrow 2$  to  $n$  do
3    $\text{key} \leftarrow r_i \cap \bigcup_{k < i} r_k$ 
4   for  $j \leftarrow 1$  to  $i - 1$  do
5     if  $\text{key} \subseteq r_j$  then
6        $p(r_i) \leftarrow r_j$ 
7       break
8     end
9   end
10 end

```

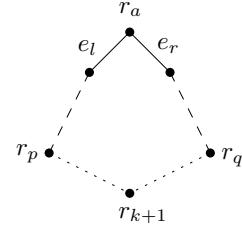


Figure 5: Possible cases of how r_{k+1} is connected to G_k

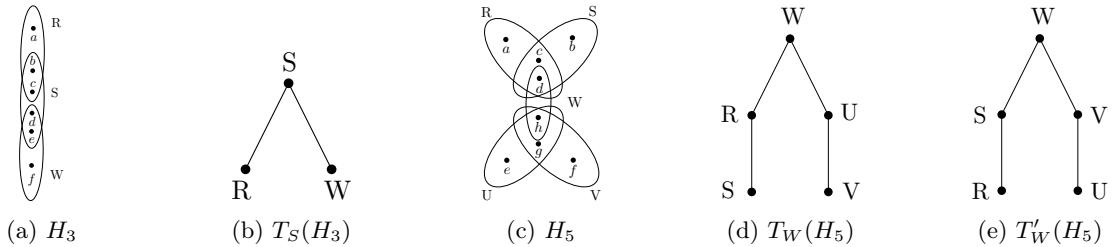


Figure 6: H_3 is a non-Berge hypergraph that admits a unique canonical join tree $T_S(H_3)$ with any relation chosen as root, such as $T_S(H_3)$. H_5 is a γ hypergraph that does not admit unique canonical join trees at any relation chosen as root. For example, $T_W(H_5)$ and $T'_W(H_5)$ are MCS trees generated by Algorithm 1. Neither of them is a canonical join tree rooted at W .

Algorithm 7: MCS for the γ -acyclic

Input: Hypergraph H , root r
Output: weighted join tree T_r and
 unweighted line graph L

```

1  $R(L) \leftarrow R(H); E(L) \leftarrow \emptyset$ 
2  $p(r) \leftarrow \text{null}; i \leftarrow 0$ 
3 while  $R \neq \emptyset$  do
4    $i \leftarrow i + 1; r_i \leftarrow r$ 
5   for  $x \in r_i \cap X$  do
6     for distinct  $r', r'' \in R : x \in r' \cap r''$ 
7       do
8          $E(L) \leftarrow E(L) \cup \{(r', r'')\}$ 
9          $\omega(r', r_i) \leftarrow |r' \cap r_i|$ 
10         $p(r') \leftarrow r_i$ 
11       $X \leftarrow X \setminus r_i; R \leftarrow R \setminus \{r_i\}$ 
12       $r \leftarrow \arg \max_{r \in R} |r \setminus X|$ 
13 return  $L, p(\cdot), \omega(\cdot)$ 

```

Algorithm 8: $buildEG$ for the γ -acyclic

Input: weighted MCS tree $T_r(H)$ weight
 function ω and unweighted union join
 graph $L(H)$

```

1  $G^{\equiv} \leftarrow L(H)$ 
2 for  $e^* = (r_i, r_j) \in R(L(H)) \setminus R(T_r(H))$  do
3   // where  $d(T_r(H), r_i) \leq d(T_r(H), r_j)$ 
4    $l \leftarrow \text{LCA}(r_i, r_j); d \leftarrow d(l) + 1$ 
5    $e_1 = (l, r_1 \leftarrow \text{LA}(r_i, d)); w_1 = \omega(e_1)$ 
6    $e_2 = (l, r_2 \leftarrow \text{LA}(r_j, d)); w_2 = \omega(e_2)$ 
7   if  $e_1 = e_2$  then
8     | slide  $r_j$  to  $r_1$ 
9   else if  $w_1 = w_2$  then
10    | slide  $r_i$  to  $r_1$  and  $r_j$  to  $r_2$ 
11   else if  $w_1 < w_2$  then
12    | slide  $r_i$  to  $l$  and  $r_j$  to  $r_1$ 
13   else if  $w_2 < w_1$  then
14    | slide  $r_i$  to  $l$  and  $r_j$  to  $r_2$ 
15 return  $G^{\equiv}$ 

```
