

# Algorithms for Optimizing Acyclic Queries

Zheng Luo  



University of California, Los Angeles, CA, USA

Wim Van den Broeck  

University of Bergen, Norway

Guy Van den Broeck  

University of California, Los Angeles, CA, USA

Yisu Remy Wang  

University of California, Los Angeles, CA, USA

---

## 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  $\alpha$ -acyclic query *by edits* in linear time with amortized constant delay, which forms the basis of a cost-based optimizer for acyclic joins. Second, we show the Maximum Cardinality Search algorithm by Tarjan and Yannakakis constructs the unique *shallowest* join tree for any Berge-acyclic query, thus enabling parallel execution of large join queries. Finally, we prove that a simple algorithm by Hu et al. converts any connected left-deep linear plan of a  $\gamma$ -acyclic query into a join tree, allowing reuse of optimizers developed for binary joins.

**2012 ACM Subject Classification** Information systems  $\rightarrow$  Query optimization; Theory of computation  $\rightarrow$  Database theory; Mathematics of computing  $\rightarrow$  Graph theory

**Keywords and phrases** Query Optimization, Join Trees, Enumeration

**Digital Object Identifier** 10.4230/LIPIcs.ICDT.2026.17

**Related Version** *Extended report*: <https://arxiv.org/abs/2509.14144> [27]

**Funding** *Wim Van den Broeck*: Wim Van den Broeck acknowledges support from the L. Meltzers Høyskolefond and the Research Council of Norway, grant number 326537.

## 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 performance. 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 [42] for acyclic queries, which runs in linear time in the input and output size,  $\mathcal{O}(|IN| + |OUT|)$ . 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.



© Zheng Luo, Wim Van den Broeck, Guy Van den Broeck, and Yisu Remy Wang; licensed under Creative Commons License CC-BY 4.0

29th International Conference on Database Theory (ICDT 2026).

Editors: Balder ten Cate and Maurice Funk; Article No. 17; pp. 17:1–17:18

Leibniz International Proceedings in Informatics



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

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:

1. Given the line graph  $L$  (Definition 6) of an  $\alpha$ -acyclic query  $H$ , Algorithm 2 enumerates all join trees *by edits*<sup>1</sup> with amortized constant delay, i.e., in  $\mathcal{O}(|L| + k)$  where  $k$  is the number of trees generated (Theorem 27). If the query is  $\gamma$ -acyclic, Algorithm 4 further reduces the total time complexity to  $\mathcal{O}(|H| + k)$  (Theorem 28).
2. Given a Berge-acyclic query, we prove that the classic Maximum Cardinality Search algorithm by Tarjan and Yannakakis [38] yields a unique *shallowest* join tree (Theorem 35), enabling parallel execution of very large queries.
3. Given a connected left-deep linear join plan for any  $\gamma$ -acyclic query, we prove that a simple algorithm by Hu et al. [20] always converts the plan into a valid join tree (Corollary 39), 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 algorithms; Section 5 introduces the unique shallowest join tree, namely the canonical join tree and its construction; Section 6 discusses the conversion from binary plans to join trees; Section 7 concludes the paper by pointing to avenues for future work. For easy reference, the key notations are summarized in Table 1. Additional technical details and proofs are provided in the extended report [27].

## 2 Related Work

Join order optimization is well studied, with algorithms based on dynamic programming (DP) from the bottom up [29, 33, 31], cost-based rewriting from the top down [11, 15], greedy heuristics [5, 14, 37], and randomized search [35]. Since the plan space is exponential, most methods prune it: some restrict to left-deep plans [21, 23], while others avoid Cartesian products [29, 30]. 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,<sup>2</sup> 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 [29, 30]. This is desirable because subplans are grouped into equivalence classes, and the optimal plan can be constructed in a bottom-up manner. 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, 40, 45]. For example, Zhao et al. [45] 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 [38]. Shallow trees are desirable for parallel execution, where the depth of the tree determines the number of sequential steps. Other practical implementations of

<sup>1</sup> To avoid redundant work, enumeration *by edits* [22] outputs the difference between consecutive elements.

<sup>2</sup> The 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 6.

■ **Table 1** Key notations introduced in Section 3 and Section 4.

Variable	Definition
$H, X=X(H), R=R(H)$	Hypergraph, its vertex set and hyperedge set
$\chi : R \rightarrow 2^X$	Incidence function mapping each hyperedge to its vertices
$ H $	Size of hypergraph: $ H  = \sum_{r \in R}  r $
$H _x$	Neighborhood of $x$ in $H$ : set of hyperedges containing $x$
$H^*, X^*, \chi^*$	Equivalent hypergraph, its vertices, and incidence function
$G, R=R(G), E=E(G)$	(Multi-)graph, its nodes (corresponding to hyperedges) and edges
$\rho : E \rightarrow 2^X, \omega : E \rightarrow \mathbb{N}$	Incidence function mapping each edge to its endpoints, weight function
$G^{\equiv}, \rho^{\equiv}$	Equivalent graph (Def. 15) and its incident function
$G \leq_T G'$	$G$ can be slid into $G'$ (Def. 16)
$L=L(H), \chi(e), \omega(e)$	Line graph of $H$ , edge label $\chi(\{r_1, r_2\}) = r_1 \cap r_2$ , weight $\omega(e) =  \chi(e) $
$ L $	Size of line graph: $ L  = \sum_{e \in E(L)} \omega(e)$
$G _x$	Subgraph of $G \subseteq L(H)$ induced by $H _x$
$T_r, \hat{T}_r$	A join tree (JT) and a monotonic weight JT (Def. 20) rooted at $r$
$\mathcal{T}(L(H))=\mathcal{T}(H)$	Set of (unrooted) join trees of $L(H)$ and also of $H$
$d(T_r, r_i)$	Depth of node $r_i$ in the rooted tree $r$
$p(\cdot), c(\cdot), s(\cdot)$	Parent, children and siblings of a node or an edge in a rooted tree
$\text{LCA}(r_i, r_j), \text{LA}(r_i, d)$	Lowest common ancestor of $r_i, r_j$ and level ancestor of $r_i$ at depth $d$
$\lambda(e)$	LCA edges of a non-tree edge $e = (r_i, r_j)$

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  $\gamma$ -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 [36, 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 [36, 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 each vertex represents a variable and each hyperedge represents a relation. Table 1 summarizes key notations introduced throughout the paper.

► **Definition 1** (Hypergraph). A hypergraph  $H = (X, R, \chi)$  consists of a set of vertices  $X$ , a set of hyperedges  $R$  and an incidence function  $\chi : R \rightarrow \mathcal{P}(X)$ .

## 17:4 Algorithms for Optimizing Acyclic Queries

We only consider hypergraphs without isolated vertices, empty hyperedges or duplicated hyperedges over the same set of vertices. We assume each hyperedge  $r$  contains a bounded number ( $\mathcal{O}(1)$ ) of vertices exclusive to  $r$ . When there is no ambiguity, we will use  $r$  interchangeably with  $\chi(r)$ ,  $x \in r$  with  $x \in \chi(r)$ , and  $H(X, R)$  with  $H(X, R, \chi)$ . We will also apply common set operations directly to hyperedges, e.g.,  $r_1 \cap r_2$  for  $\chi(r_1) \cap \chi(r_2)$ .

In fact, the reader can often ignore  $\chi$  and identify a hyperedge with its set of vertices; the purpose of  $\chi$  is to relate hyperedges over different vertex sets across hypergraphs in the correctness proofs of our algorithms.

We write  $X(H)$  and  $R(H)$  to refer to the vertices and edges of  $H$  respectively. The size of a hypergraph is the total size of all hyperedges  $|H| = \sum_{r \in R} |r|$ . We let  $H|_x$  denote the *neighborhood* of  $x \in X$ , consisting of all hyperedges containing  $x$ .

► **Definition 2 (Multigraph).** A multigraph  $G = (R, E, \rho)$  consists of a set of vertices  $R$ , a set of edges  $E$ , and the incidence function  $\rho : E \rightarrow \mathcal{P}(R)$  such that  $1 \leq |\rho(e)| \leq 2$  for any  $e \in E$ . An edge  $e$  is a self-loop if  $|\rho(e)| = 1$ . Edges  $e_1, e_2 \in E$  are parallel if  $\rho(e_1) = \rho(e_2)$ .

We denote the set of vertices in  $G$  with  $R$ , because we will soon define the *line graph*  $L$  of a hypergraph where each vertex in  $L$  represents a hyperedge. We write  $R(G)$  and  $E(G)$  to refer to the vertices and edges of  $G$  respectively.

► **Definition 3 (Simple Graph).** A simple graph is a multigraph with no parallel edges or self-loops, i.e., the incidence function is injective and always returns two distinct vertices.

Because each edge  $e$  in a simple graph can be identified with its two endpoints  $\rho(e) = \{r_1, r_2\}$ , we will omit  $\rho$  and write  $\{r_1, r_2\}$  for  $e$  when there is no ambiguity.

► **Definition 4 (Cycle, Clique and Diamond).** In a simple graph  $G = (R, E)$ , a sequence of  $n \geq 3$  distinct vertices  $(v_0, \dots, v_{n-1})$  where  $v_i \in R$  is

- a cycle if there is an edge between  $v_i$  and  $v_{(i+1) \bmod n}$  for all  $0 \leq i < n - 1$ ;
- an  $n$ -clique  $K_n$  if the induced graph  $G|_{\{v_0, \dots, v_n\}}$  has an edge between each pair of vertices;
- a diamond if  $n = 4$  and  $G|_{\{v_0, \dots, v_3\}}$  has one fewer edge than a  $K_4$ .

A *weighted graph* is a (multi-)graph where each edge is assigned a weight:

► **Definition 5 (Weighted Graph).** A weighted graph  $(G, \omega)$  consists of a multigraph  $G = (R, E, \rho)$  and weight function  $\omega : E \rightarrow \mathbb{N}$  assigning a natural number to each edge in  $G$ .

The *line graph* of a hypergraph is the “intersection graph” of its hyperedges:

► **Definition 6 (Line Graph).** For a hypergraph  $H$ , the line graph  $L(H) = (G, \chi)$  consists of a simple graph  $G = (R, E)$  and an edge labeling function  $\chi : E \rightarrow \mathcal{P}(X)$ . The vertices of  $G$  are the hyperedges  $R$  of  $H$ , and there is an edge  $e = \{r_1, r_2\} \in E$  when  $r_1 \cap r_2 \neq \emptyset$ . The edge labeling function maps each edge  $e$  to the intersection of the hyperedges represented by the endpoints of  $e$ :  $\chi(\{r_1, r_2\}) = r_1 \cap r_2$ .

In addition, we define a weight function mapping each edge to the size of its label:

► **Definition 7 (Line Graph Edge Weight).** For each edge  $e \in E(L(H))$ , the weight function  $\omega : E(L(H)) \rightarrow \mathbb{N}$  is defined as  $\omega(e) = |\chi(e)|$ .

When there is no ambiguity from the context, we simply write  $L(H)$  as  $L$ , and we may treat the line graph  $(G, \chi)$  as a weighted graph  $(G, \omega)$ . The size of a line graph is the sum of all edge weights  $|L| = \sum_{e \in E(L)} \omega(e)$ . For any subgraph  $G$  of  $L(H)$ , we let  $G|_x$  denote the subgraph of  $G$  induced by  $H|_x$ .

■ **Algorithm 1** MCS from  $r \in R(H)$ .

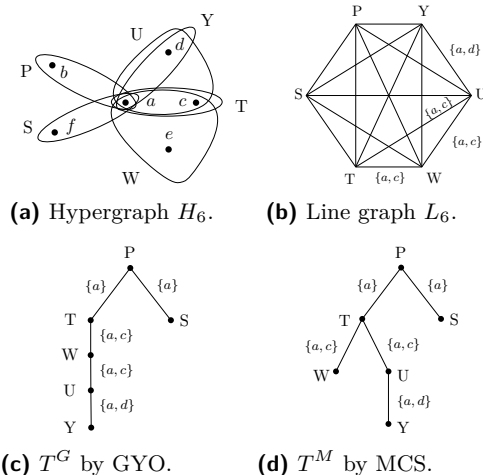
**Input:**  $\alpha$ -acyclic hypergraph  $H$   
and root  $r$

**Output:** Join tree  $T_r$  rooted at  $r$

```

1  $p(r) \leftarrow \text{null}; E(T_r) \leftarrow \emptyset$ 
2 while  $R \neq \emptyset$  do
3   for  $x \in r \cap X$  do
4     for  $r' \in R : x \in r'$  do
5        $p(r') \leftarrow r$ 
6    $X \leftarrow X \setminus r$ 
7    $R \leftarrow R \setminus \{r\}$ 
8    $E(T_r) \leftarrow E(T_r) \cup \{\{r, p(r)\}\}$ 
9    $r \leftarrow \arg \max_{r \in R} |r \setminus X|$ 
10 return  $T_r$ 

```



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

► **Example 8.** 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 9 (Join Tree).** 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.

We use  $R(T)$  to denote the set of nodes and  $E(T)$  to denote the set of edges of tree  $T$ . The requirement that  $T|_{x \in X(H)}$  be a connected subtree 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 of  $H$ . 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.

Four common notions of hypergraph acyclicity are defined in increasing order of strictness [13], namely  $\alpha$ -acyclic  $\supset \beta$ -acyclic  $\supset \gamma$ -acyclic  $\supset$  Berge-acyclic.

► **Definition 10 (Hypergraph Acyclicity).** A hypergraph  $H$  is:

- $\alpha$ -acyclic if it admits a join tree as in Definition 9;
- $\beta$ -acyclic if every subgraph of  $H$  is  $\alpha$ -acyclic;
- $\gamma$ -acyclic if it does not contain any  $\gamma$  cycle. A  $\gamma$  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 following classic result relates join trees of  $H$  to maximum spanning trees of  $L(H)$ :

► **Theorem 11 (Maier [28]).** Given an  $\alpha$ -acyclic hypergraph  $H = (X, R)$ , a tree with nodes in  $R$  is a join tree of  $H$  if and only if it is a maximum spanning tree (MST) of  $L(H)$ .

Join trees for acyclic hypergraphs can be constructed by *GYO reduction*:

► **Definition 12.** 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 [43] 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, where, for example, the node  $T$  is the parent of  $W$ , denoted as  $p(W) = T$  and the root  $P$  has no parent.

Another way to construct join trees is via the Maximum Cardinality Search (MCS) algorithm [38], and we present a simplified version in Algorithm 1. The join tree produced by the algorithm is called an MCS tree. Given an input hypergraph  $H$  and a hyperedge  $r \in R(H)$  chosen as the root, Algorithm 1 constructs a rooted MCS tree by iteratively expanding toward hyperedges connected with those already processed. The algorithm maintains a set  $R$  of unprocessed hyperedges and a set  $X$  of unvisited vertices. Initially, the MCS tree contains no edges, and the first hyperedge to be processed is the root  $r$ . In each iteration, the algorithm processes the current hyperedge  $r$  by examining all of its unvisited vertices in  $r \cap X$ . Whenever such a vertex  $x$  is encountered, every remaining hyperedge  $r' \in R$  that contains  $x$  records  $r$  as its tentative parent  $p(r') \leftarrow r$ . Once all vertices of  $r$  are marked as visited, they are removed from  $X$ . The hyperedge  $r$  is marked as processed and removed from  $R$ . After processing each  $r$ , the edge between  $r$  and its recorded parent  $p(r)$  is added to the MCS tree as  $\{r, p(r)\}$ . The algorithm selects as the next hyperedge an element of  $R$  that maximizes the number of already visited vertices, namely a hyperedge  $r$  that maximizes  $|r \setminus X|$ , breaking ties arbitrarily. This process continues until all hyperedges have been processed, when the recorded edges  $\{r, p(r)\}$  form an MCS tree rooted at  $r$ .

► **Example 13.** On  $H_6$  in Figure 1a with  $r = P$ , Algorithm 1 examines all vertices of  $P$ .  $P$  also becomes the tentative parent of five hyperedges,  $p(S) = p(T) = p(U) = p(W) = p(Y) \leftarrow P$ . It proceeds to check the next hyperedge with the most visited vertices as in Line 9. At this point, each of the five has one visited vertex, Algorithm 1 breaks the tie arbitrarily, say proceeding with  $S$ , and examining all its vertices. Its parent is finalized as  $p(S) = P$ . Each of the remaining four again has one visited vertex, so Algorithm 1 proceeds to check  $T$ . Its parent is finalized as  $p(T) = P$ . While checking  $T$ , Algorithm 1 examines the vertex  $c$  that is also in  $U, W$ , their parents are updated to  $p(U) = p(W) \leftarrow T$ . It continues until all hyperedges are processed. By connecting each hyperedge with its parent, Algorithm 1 constructs the MCS tree  $T^M$  in Figure 1d.

Algorithm 1 can generate different join trees by varying the choice of root and tie-breaking strategy. However, it does not guarantee to generate all possible trees. For example, it never generates the join tree  $T^G$  shown in Figure 1c.

We extend the notions of parent, children and siblings to edges of a rooted tree. Let  $e \in E(T)$  be an edge of tree  $T$  rooted at  $r$  (and  $e$  not incident to  $r$ ). Its parent  $p(e) \in E(T)$  is the unique tree edge incident to  $e$  and closer to the root  $r$ . We let  $c(e)$  denote the set of children of  $e$  in  $T$ . The siblings of  $e$  are the set of edges incident to the parent of  $e$ , namely  $s(e) = \{e' \in E(T) \setminus \{e\} \mid p(e') = p(e)\}$ . Edges incident to the root do not have a parent and are all siblings.

Lemma 14 shows that an MCS tree is “somewhat monotonic” in that every edge must contain some variable not in its parent. If two edges share any variable not in their parents, they must be siblings.

► **Lemma 14.** Let  $T$  be an MCS tree. For an edge  $e \in T$  that has a parent, then

1.  $e \not\subseteq p(e)$
2. For another edge  $e' \in T : (e \setminus p(e)) \cap (e' \setminus p(e')) \neq \emptyset \implies p(e) = p(e')$ .

**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 the join trees of a hypergraph  $H$  starts from Theorem 11 which allows us to reduce the problem to the enumeration of maximum spanning trees (MSTs) of the line graph  $L(H)$ . The best known algorithm for MST enumeration<sup>3</sup> is due to Eppstein [12] 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 gives an algorithm (and proves a matching lower bound) in time  $\mathcal{O}(m + n \log n)$  to construct the equivalent graph from an arbitrary weighted graph with  $m$  edges and  $n$  vertices. Then to enumerate all  $k$  MSTs of  $G$ , he applies existing algorithms to enumerate the spanning trees of  $G^\equiv$ . Since there are optimal spanning tree enumeration algorithms that run in  $\mathcal{O}(m + n + k)$  time [22, 34], 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:

- Given the line graph  $L$  of an  $\alpha$ -acyclic hypergraph, we can construct an equivalent graph of  $L$  in  $\mathcal{O}(|L|)$ , thus enumerating the join trees in  $\mathcal{O}(|L| + k)$  time (Theorem 27).
- Given any  $\gamma$ -acyclic hypergraph  $H$ , we can construct an equivalent graph of  $L(H)$  in  $\mathcal{O}(|H|)$ , lowering the overall time complexity of enumeration to  $\mathcal{O}(|H| + k)$  (Theorem 28). Note that  $|L(H)|$  can be quadratic in  $|H|$  while  $|H| \in \mathcal{O}(|L(H)|)$ , so the second item above yields a better bound.

In the rest of this section, we first define the equivalent graph. Then, we present the algorithm for enumerating join trees of  $\alpha$ -acyclic hypergraphs. Finally, we adapt the algorithm to enumerate join trees of  $\gamma$ -acyclic hypergraphs more efficiently.

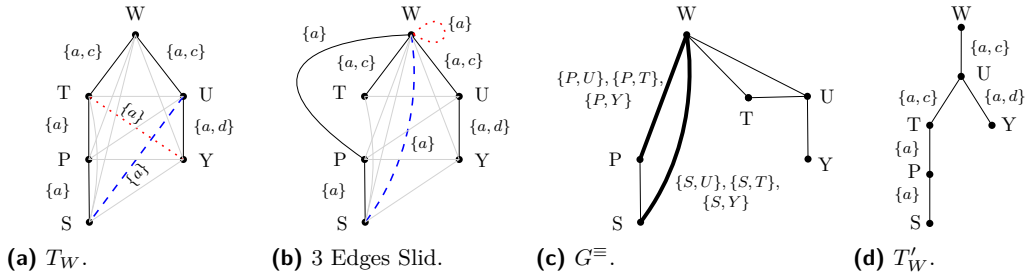
### 4.1 Equivalent Graph

The key idea of Eppstein’s algorithm [12] is to construct an *equivalent graph*  $G^\equiv$  whose spanning trees one-to-one correspond to the MSTs of the input graph  $G$ , thereby reducing MST enumeration to spanning tree enumeration.

► **Definition 15** (Equivalent Graph [12]). *Given a weighted graph  $G = ((R, E, \rho), \omega)$ , a multigraph  $G^\equiv = ((R, E, \rho^\equiv), \omega)$  is an equivalent graph of  $G$  if the spanning trees of  $G^\equiv$  one-to-one correspond to the maximum spanning trees of  $G$ : a set of edges  $E^T \subseteq E$  induces a spanning tree of  $G^\equiv$  if and only if  $E^T$  induces an MST of  $G$ .*

In what follows, we may use the notation  $G^\equiv(L(H))$  or  $G^\equiv(H)$  to refer to an equivalent graph of the line graph of hypergraph  $H$ . Note that  $G^\equiv$  shares the same vertices and edges of  $G$ , and they only differ in the incidence function mapping each edge to its endpoints. Such an equivalent graph can be constructed from  $G$  by applying a series of *sliding transformations*: for two incident edges  $e, e'$  with  $\rho(e) = \{r_1, r_2\}$  and  $\rho(e') = \{r_2, r_3\}$ , we can *slide*  $e'$  along  $e$  by updating  $\rho(e') = \{r_1, r_3\}$ , if  $\omega(e') < \omega(e)$ . Note that the edge  $e'$  retains its identity after sliding, which explains the need for the incidence function  $\rho$ . Furthermore, we will use a given rooted MST  $T$  as a guide, and only slide edges towards the root. More formally:

<sup>3</sup> Eppstein focuses on minimum spanning trees, but the same algorithm applies to maximum spanning trees with flipped comparisons.



■ **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.

► **Definition 16** (Sliding Transformation [12]). *Let  $T$  be a MST of  $G$  rooted at  $r$ . Let  $e$  be an edge such that  $\rho(e) = \{r_1, r_2\}$  with  $r_1$  closer to the root than  $r_2$  in  $T$ . If another edge  $e'$  shares  $r_2$  with  $e$ , i.e.,  $\rho(e') = \{r_2, r_3\}$ , and  $\omega(e') < \omega(e)$ , then sliding  $e'$  along  $e$  results in a graph  $G'$  that is identical to  $G$ , except that the incidence function maps  $e'$  to  $\{r_1, r_3\}$ .*

The key result by Eppstein shows that applying sliding transformations on  $G$  along a rooted MST  $T$  to a fixpoint results in an equivalent graph of  $G$ . To state this formally, we first define an ordering on graphs based on sliding transformations. It is easy to verify the following is a partial order:

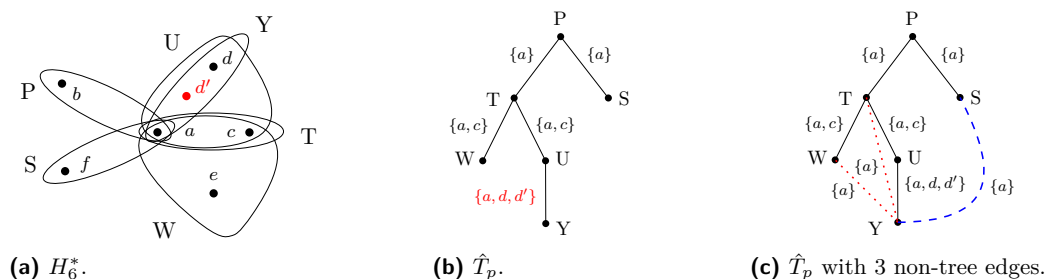
► **Definition 17** (Sliding Partial Order). *We write  $G \leq_T G'$  if  $G'$  can be obtained from  $G$  by applying a sequence of sliding transformations along a rooted MST  $T$  of  $G$ .*

Applying sliding transformations to a fixpoint therefore yields a maximal element under the sliding partial order.

► **Theorem 18** (Sliding produces  $G^=$  at fixpoint [12]). *Given a weighted graph  $G$  and a rooted MST  $T$ , and let  $G^=$  be a maximal element under  $\leq_T$ . Then  $G^=$  is an equivalent graph of  $G$ .*

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

► **Example 19.** 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  $\{S, U\}$  along tree edge  $\{U, W\}$  to become  $\{S, W\}$  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  $L_6^=$ , 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*. Denoting the graph after sliding as  $L'_6$ , ordering  $L_6 \leq_{T_W} L'_6$  holds. By applying sliding transformations to a fixpoint, we obtain an equivalent graph  $L_6^=$ , where there are two sets of parallel edges highlighted by thick lines in Figure 2c (self-loops are omitted). For example, the tree edge  $\{T, P\}$  in  $T_W$  and non-tree edges  $\{P, U\}$ ,  $\{P, T\}$ ,  $\{P, Y\}$  become parallel in  $L_6^=$  between  $P$  and  $W$ . Therefore  $L_6 \leq_{T_W} L'_6 \leq_{T_W} L_6^=$  holds under the sliding partial order. We can easily verify that each spanning tree of  $L_6^=$  corresponds to an MST of  $L_6$ , such as  $T'_W$  in Figure 2d.



■ **Figure 3** An equivalent hypergraph  $H_6^*$  and its monotonic weight join tree  $\hat{T}_P$  rooted at  $P$ .

## 4.2 Enumerating Join Trees of $\alpha$ -Acyclic Hypergraphs

The bottleneck of Eppstein’s algorithm for constructing equivalent graphs of arbitrary weighted graphs lies in 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*  $H^*$  whose join trees one-to-one correspond to those of  $H$ , but one special join tree of  $H^*$  has *monotonically increasing weight* from root to leaf, which enables constant-time identification of the blocking edge.

► **Definition 20.** A monotonic weight join tree  $\hat{T} \in \mathcal{T}(H)$  is a rooted join tree of  $H$  such that for any  $e \in E(\hat{T})$  that has a parent edge  $p(e)$ ,  $\omega(e) > \omega(p(e))$ .

We can always construct such a  $H^*$  given any  $\alpha$ -acyclic hypergraph  $H$ :

► **Theorem 21.** Given an  $\alpha$ -acyclic hypergraph  $H = (X, R, \chi)$ , there exists an equivalent hypergraph  $H^* = (X^*, R, \chi^*)$  over the same hyperedge set  $R$  that

- admits a monotonic weight join tree  $\hat{T} \in \mathcal{T}(H^*)$ , and
- $\mathcal{T}(H^*) = \mathcal{T}(H)$ .

► **Example 22.**  $H_6^*$  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  $G \equiv (L(H_6^*))$  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.

A monotonic weight join tree  $\hat{T}$  can accelerate the construction of the equivalent graph in several ways. First, all tree edges are already “in place”, because they cannot slide along their lighter parent edges. We therefore need only consider non-tree edges. For each non-tree edge  $e = \{r_i, r_j\}$ , there are two possible cases. First, if one endpoint, say  $r_i$ , is an ancestor of the other, say  $r_j$ , then  $e$  can only slide to become parallel to the edge right below  $r_i$  on the path connecting  $r_i$  and  $r_j$  in  $\hat{T}$ . This is because all edges on that path must be no lighter than  $e$  due to the running intersection property of join trees, but since  $\hat{T}$  has monotonically increasing weight, at most one edge (the one right below  $r_i$ ) can have equal weight to  $e$ . If this edge has weight equal to  $e$ , it becomes the blocking edge, and  $e$  slides to become parallel to it; otherwise,  $e$  slides to a self-loop at  $r_i$ . In the second case, if neither endpoint is an ancestor of the other, there are at most two blocking edges, namely those below the lowest common ancestor of  $r_i$  and  $r_j$  on the path between them. We therefore only need to compare the weight of  $e$  with these two potentially blocking edges to determine the destination of  $e$ .

■ Algorithm 2 *buildEG*.

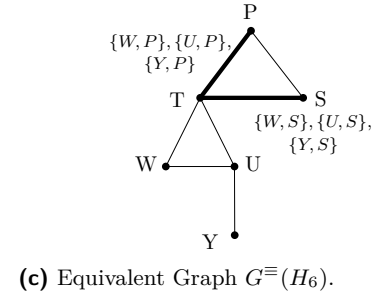
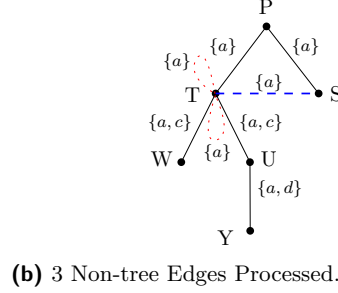
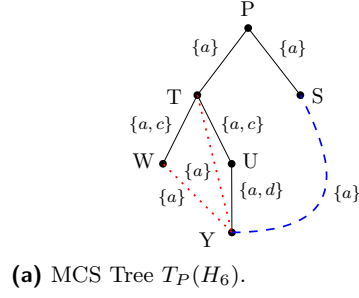
---

**Input:** Line graph  $L = ((R, E, \rho), \omega)$  of  $\alpha$ -acyclic  $H$  and an MCS tree  $T_r$

**Output:** Equivalent Graph  $G^\equiv$

- 1  $G^\equiv(R, E, \rho^\equiv) \leftarrow (R, E, \rho)$
- 2 **for**  $e \in E(L) \setminus E(T_r)$  **do**
- 3      $\omega^* \leftarrow \omega(e)$
- 4      $r_i, r_j \leftarrow \rho(e)$
- 5     **if**  $|\lambda(e)| = 1$  **then**
- 6          $\{\{l, l'\}\} \leftarrow \lambda(e)$
- 7         **if**  $\omega^* < \omega(l, l')$  **then**
- 8             set  $\rho^\equiv(e)$  to  $\{l\}$
- 9         **else**
- 10             set  $\rho^\equiv(e)$  to  $\{l, l'\}$
- 11     **else**
- 12          $\{\{l, l_1\}, \{l, l_2\}\} \leftarrow \lambda(e)$
- 13         **if**  $\omega^* < \omega(l, l_1) \wedge \omega^* < \omega(l, l_2)$  **then**
- 14             set  $\rho^\equiv(e)$  to  $\{l\}$
- 15         **else if**  $\omega^* = \omega(l, l_1)$  **then**
- 16             set  $\rho^\equiv(e)$  to  $\{l, l_1\}$
- 17         **else if**  $\omega^* = \omega(l, l_2)$  **then**
- 18             set  $\rho^\equiv(e)$  to  $\{l, l_2\}$
- 19         **else**
- 20             set  $\rho^\equiv(e)$  to  $\{l_1, l_2\}$
- 21 **return**  $G^\equiv$

---


 ■ Figure 4 Algorithm 2 on  $H_6$  of Figure 1a

► **Example 23.** Consider again  $\hat{T}_p$  in Figure 3c. First note that every tree edge is heavier than its parent edge, and each non-tree edge is no heavier than any edge on the path connecting its endpoints in the tree. There is one edge,  $\{T, Y\}$ , where one endpoint ( $T$ ) is an ancestor of the other ( $Y$ ). The only potentially blocking edge is  $\{T, U\}$  right below  $T$ . But because  $\{T, U\}$  is heavier,  $\{T, Y\}$  slides to become a self-loop at  $T$ . For the edge  $\{W, Y\}$ , all edges on the tree path connecting the endpoints are heavier, so  $\{W, Y\}$  also slides to a self-loop at their LCA  $T$ . Finally,  $\{Y, S\}$  is blocked by the two edges  $\{P, T\}$  and  $\{P, S\}$  below their LCA  $P$ , as all three edges have a weight of 1. Therefore  $\{Y, S\}$  slides to  $\{T, S\}$ .

Our algorithm for constructing the equivalent graph leverages the insight above to slide each edge in constant time. In the following, we first define a few helpful notations and describe a preprocessing step to build helper data structures, before presenting the algorithm.

Given a rooted MCS tree  $T_r$  of a line graph  $L$ , a non-tree edge is  $e = \{r_i, r_j\} \in E(L) \setminus E(T_r)$ . There is a path in  $T_r$  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,  $|\lambda(e)| = 2$ .

During pre-processing, we first conduct a breadth-first search on  $T_r$  to obtain the depth table  $\mathbf{d}$  of each tree node in  $\mathcal{O}(|T_r|)$ . Then we build two data structures in  $\mathcal{O}(|T_r|)$  to facilitate the constant-time query of the lowest common ancestor (LCA) [3] given two nodes, and the

level ancestor (LA) [4] of a node  $r$ , which is the ancestor of  $r$  at a given depth. Algorithm 2 assumes these data structures are prebuilt and available, which allows finding  $\lambda(e)$  of any non-tree edge  $e$  in constant time as follows:

$$\lambda(\{r_i, r_j\}) = \begin{cases} \{\{r_i, \text{LA}(r_j, d(r_i) + 1)\}\} & \text{if } r_i = \text{LCA}(r_i, r_j) \\ \{\{r_j, \text{LA}(r_i, d(r_j) + 1)\}\} & \text{if } r_j = \text{LCA}(r_i, r_j) \\ \{\{l, \text{LA}(r_i, d)\}, \{l, \text{LA}(r_j, d)\}\} & \text{otherwise, } l := \text{LCA}(r_i, r_j), d := d(l) + 1 \end{cases}$$

We are now ready to present Algorithm 2 which constructs  $G^\equiv$  by sliding each non-tree edge  $e = \{r_i, r_j\}$  with weight  $\omega^*$ . There are two cases. First, if  $\lambda(e)$  returns a single edge, then one endpoint is an ancestor of the other. By the definition above  $\lambda(e)$  always returns the ancestor node  $l$  first, while the other node is denoted as  $l'$ . The weight of  $\{l, l'\}$  is then compared with  $\omega^*$ : if  $\omega^*$  is lighter,  $e$  slides to a self-loop at  $l$ ; otherwise the weights must be equal, and  $e$  slides to be parallel to  $\{l, l'\}$ . Second, if  $\lambda(e)$  returns two edges, then neither endpoint is an ancestor of the other. In this case we compare  $\omega^*$  with the two edges in  $\lambda(e)$ . If  $\omega^*$  is lighter than both,  $e$  slides to a self-loop at  $l$ ; if  $\omega^*$  equals one of them,  $e$  slides to be parallel to that edge; otherwise,  $\omega^*$  equals both, and  $e$  slides to be incident to each of the highest depth endpoints of both edges. The four cases are exhaustive, as the non-tree edge never outweighs any of its LCA edges by the running intersection property.

► **Theorem 24.** *Given a hypergraph  $H^* = (X^*, R, \chi^*)$  with a monotonic weight join tree  $\hat{T}$ , Algorithm 2 produces an Equivalent Graph  $G^\equiv(L(H^*))$ .*

The notation  $H^*$  highlights later applications to equivalent hypergraphs, although Theorem 24 applies to any hypergraph admitting a monotonic weight join tree.

Although Algorithm 2 constructs an equivalent graph of  $L(H^*)$  in constant time per edge, such an  $H^*$  may be asymptotically larger than  $H$ . Perhaps surprisingly, running Algorithm 2 *directly on  $H$*  produces the same equivalent graph! This is because  $H^*$  is constructed in a way that preserves the structure of  $H$ , keeping the relative weights of an edge and its LCA edges invariant. Intuitively, we construct  $H^*$  by making “local copies” of the vertices in  $H$ : this perturbs the weights to be monotonic along  $\hat{T}$ , while ensuring the weights are adjusted consistently. In particular, when making a new copy  $x'$  of a vertex  $x$ , we add  $x'$  to *all hyperedges* that contain  $x$ , namely  $H|_x$ . This way,  $\chi(e) \subset \chi(e')$  in  $L(H)$  if and only if  $\chi(e) \subset \chi(e')$  in  $L(H^*)$ . As a result, all comparisons in Algorithm 2 return the same result when running on  $H$  or  $H^*$ . We illustrate this with the following example.

► **Example 25.** In Figure 3a, we made a copy  $d'$  of  $d$  and added it to all hyperedges containing  $d$  in  $H_6$ . This maintains the relative weights between the tree edge  $\{U, Y\}$  and the non-tree edges highlighted in Figure 3c: all three non-tree edges remain lighter than  $\{U, Y\}$ . Were we to make a copy of  $a$  to add to  $H_6|_a$  (which contains every hyperedge in  $H_6$ ), the weights of the non-tree edges relative to their respective LCA edges would also remain unchanged.

Finally, because  $H$  and  $H^*$  share the same join trees, there must be  $G^\equiv(L(H))$  that is the same as  $G^\equiv(L(H^*))$ . Together with Theorem 24, this means running Algorithm 2 directly on  $L(H)$  produces an equivalent graph  $G^\equiv(L(H))$  of  $L(H)$ :

► **Theorem 26.** *Given the line graph  $L$  and an MCS tree  $T_r$  of an  $\alpha$ -acyclic hypergraph  $H$ , Algorithm 2 returns an equivalent graph  $G^\equiv(L)$  in time  $\mathcal{O}(|L|)$ .*

Figure 4 shows an example of applying Algorithm 2 to  $H_6$  in Figure 1a. Once we compute the equivalent graph  $G^\equiv(L)$ , we can apply existing algorithms [22, 34] to enumerate all its spanning trees by edits with amortized constant delay. And because the MCS tree can be

---

**Algorithm 3** MCS ( $\gamma$ -acyclic).

---

**Input:**  $\gamma$ -acyclic Hypergraph  $H$   
and root  $r$

**Output:**  $T_r$  and  $\omega$

```

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

```

---



---

**Algorithm 4** *buildEG* ( $\gamma$ -acyclic).

---

**Input:** Weighted  $T_r$ , unweighted  $L(H)$

**Output:** Equivalent Graph  $G^\equiv$

```

2 for  $e \in E(L) \setminus E(T_r)$  do
3   ...
5   if  $|\lambda(e)| = 1$  then
6     ... set  $\rho^\equiv(e)$  to  $\{l, l'\}$ 
7   else
8      $\{\{l, l_1\}, \{l, l_2\}\} \leftarrow \lambda(e)$ 
9     else if  $\omega(l, l_1) < \omega(l, l_2)$  then
10       $\rho^\equiv(e)$  to  $\{l, l_1\}$ 
11     else if  $\omega(l, l_1) > \omega(l, l_2)$  then
12       $\rho^\equiv(e)$  to  $\{l, l_2\}$ 
13     else  $\rho^\equiv(e)$  to  $\{l_1, l_2\}$ 
14 return  $G^\equiv$ 

```

---

constructed in linear time from  $H$  which in turn can be recovered from  $L(H)$  in linear time, the overall time complexity of enumerating all join trees of  $H$  is linear in the size of  $L$  plus the number of join trees:

► **Theorem 27.** *Given the line graph  $L$  of an  $\alpha$ -acyclic hypergraph  $H$ , the join trees of  $H$  can be enumerated in time  $\mathcal{O}(|L| + |\mathcal{T}(H)|)$ .*

### 4.3 $\gamma$ -Acyclic Queries

The run time of Algorithm 2 depends on the size of the line graph which can be quadratically larger than the input hypergraph. If  $H$  is  $\gamma$ -acyclic, we can bring the total time complexity of enumeration down to  $\mathcal{O}(|H| + |\mathcal{T}(H)|)$ . Using an algorithm by Leitert [25], we can construct  $L(H)$  from  $H$  in time  $\mathcal{O}(|H| + |L(H)|)$  when  $H$  is  $\gamma$ -acyclic; furthermore, because every edge in  $L(H)$  of a  $\gamma$ -acyclic  $H$  is an MST edge [25], we have  $|H| + |\mathcal{T}(H)| \in \Omega(|L(H)|)$ , and so  $\mathcal{O}(|H| + |L(H)| + |\mathcal{T}(H)|) = \mathcal{O}(|H| + |\mathcal{T}(H)|)$ , hiding the  $|L(H)|$  term in the overall complexity. However, the  $L(H)$  constructed by Leitert is *unweighted*, so we need to modify Algorithm 2 slightly to slide each non-tree edge based on the weights of its LCA edges alone. We first modify MCS (Algorithm 1) to track the weight of each tree edge as in Algorithm 3. Then, we update the cases of Algorithm 2 to those in Algorithm 4. First, if  $\lambda(e)$  returns a single edge, we directly slide  $e$  to be parallel to it, because we know  $e$  is an MST edge and therefore will not become a self-loop. Otherwise, if  $\lambda(e)$  returns two edges, we *compare the weights of those two edges*, instead of comparing them with  $\omega(e)$ . This is again because we know  $e$  is an MST edge, so it must have weight equal to at least one of its LCA edges. If one of the LCA edges is lighter,  $e$  must have equal weight to that edge, and we slide  $e$  to be parallel to it. Otherwise, if both LCA edges have the same weight,  $e$  must also have the same weight, and we slide  $e$  to be under both edges to form a triangle. In all cases we only use the weights of the tree edges and do not need to know  $\omega(e)$ .

► **Theorem 28.** *The join trees of a  $\gamma$ -acyclic  $H$  can be enumerated in time  $\mathcal{O}(|H| + |\mathcal{T}(H)|)$ .*

■ **Table 2** Acyclic queries in the benchmarks (all  $\alpha$ -acyclic queries are also  $\gamma$ -acyclic.).

Name	# Queries	# $\alpha$ -Acyclic	# Composite-Key Joins	# Berge-Acyclic
TPC-H[39]	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[44]	4712	4709	6	4703

## 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. 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. [45] achieves similar performance on any join tree rooted at the largest relation. An alternative to enumeration is therefore to simply construct one 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 prove that for a Berge-acyclic query the MCS algorithm produces a *shallowest tree*, where the depth of each tree node is minimized. We prove that this shallowest tree is unique, and therefore call it the *canonical join tree*.

► **Definition 29.** 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 also 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 2, among 10 454 queries from five popular benchmarks, 9285 are  $\alpha$ -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  $\alpha$ -acyclicity with linearity and Berge-acyclicity.

► **Proposition 30.** An  $\alpha$ -acyclic hypergraph is Berge-acyclic if and only if it is linear.

By Proposition 30, every edge in the line graph of a Berge-acyclic hypergraph has a weight of 1. Every spanning tree is a maximum spanning tree, therefore a join tree.

► **Corollary 31.** 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.

The key insight leading to the existence and uniqueness of the canonical join tree is that the line graph  $L$  of a Berge-acyclic hypergraph is *geodetic* [32], meaning that there is a unique shortest path between any pair of vertices. The canonical join tree can then be constructed

## 17:14 Algorithms for Optimizing Acyclic Queries

by taking the union of all shortest paths from the root to each other vertex, as this will guarantee minimal depth for each vertex. The key step to establish the geodetic property is to show that  $L$  is a special class of *chordal graphs* called *block graphs*. Chordal graphs are intimately related to acyclic hypergraphs [9], and a block graph is a special chordal graph defined as follows:

- **Definition 32** (Chordal and Block). *A simple graph  $G = (R, E)$  is*
  - chordal [10] *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;*
  - a block graph [18] *if it is chordal and diamond-free, i.e., no subgraph induced by any  $R' \subseteq R$  is a diamond (Definition 4).*

- **Lemma 33.** *The line graph  $L$  of a Berge-acyclic hypergraph  $H$  is a block graph.*

Together with the fact that every block graph is *geodetic* (with a unique shortest path between any two vertices) [32], Lemma 33 implies the following corollary.

- **Corollary 34.** *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 35.** *A Berge-acyclic hypergraph  $H$  has a unique canonical tree.*

**Proof.** Let  $L$  be the line graph of  $H$ , and  $P(r, r')$  be the shortest path in  $L$  between  $r, r' \in R(L)$ , we prove that  $T_r = \bigcup_{r' \in R(L)} P(r, r')$  is the unique canonical join tree for  $H$  rooted at  $r$ .

By Corollary 31 any spanning tree of  $L$  is a join tree.  $T_r$  is connected and spans all vertices in  $R(L)$ , because it contains the shortest path from  $r$  to every  $r' \in R(L)$ . It remains to show that  $T_r$  is acyclic, which we prove 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  $T_r|_{R_{d>0}}$  is acyclic, we consider a vertex  $r' \in R_{d+1} \setminus R_d$ . Corollary 34 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 spanning tree thus a join tree of  $H$ .

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 34. ◀

The canonical join tree can be constructed by the MCS algorithm as shown in Algorithm 1:

- **Theorem 36.** *Given a Berge-acyclic hypergraph  $H$ , running Algorithm 1 from  $r \in R(H)$  constructs the canonical tree  $T_r(H)$ .*

## 6 Converting a Binary Join Plan to a Join Tree

Recent approaches [20, 2] that convert a binary join plan into a join tree have gained popularity as they allow 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] to convert left-deep linear join plans into join trees as shown in Figure 5. We prove that the algorithm converts any connected left-deep linear join plan into a join tree if and only if the query is  $\gamma$ -acyclic. This can be seen as a new characterization of  $\gamma$ -acyclic queries. We formally define binary join plans and describe the algorithm by Hu et al. in Algorithm 5.

■ **Algorithm 5** Binary-to-JT Conversion.

---

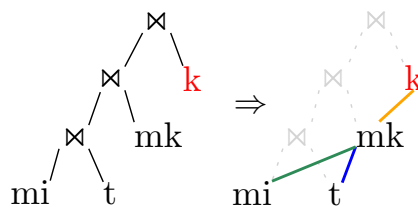
**Input:** Left-deep plan  $(r_1, \dots, r_n)$   
**Output:** Rooted join tree  $T_{r_1}$

```

1  $p(r_1) \leftarrow null; E(T_{r_1}) \leftarrow \emptyset$ 
2 for  $i \leftarrow 2$  to  $n$  do
3    $key \leftarrow r_i \cap \bigcup_{k < i} r_k$ 
4   for  $j \leftarrow 1$  to  $i - 1$  do
5     if  $key \subseteq r_j$  then
6        $E(T_{r_1}) \leftarrow E(T_{r_1}) \cup \{\{r_i, r_j\}\}$ 
7       break
8   return  $T_{r_1}$ 

```

---



■ **Figure 5** Algorithm 5 [20] on JOB-3a [24].

► **Definition 37.** A left-deep linear plan is a sequence of hyperedges  $(r_1, r_2, \dots, r_n)$ . 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 [41].

Given a left-deep linear plan, Hu et al. [20] generate a join tree with Algorithm 5. 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,  $key := r_i \cap \bigcup_{k < i} r_k$ , 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  $\gamma$ -acyclic.

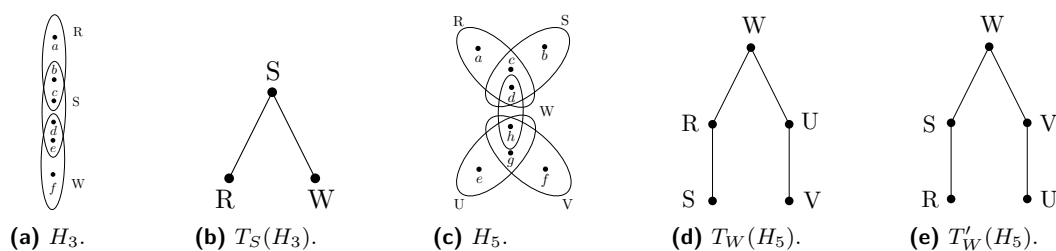
► **Theorem 38.** A query is  $\gamma$ -acyclic if and only if every connected left-deep linear join plan for the query is the reverse of a GYO-reduction order.

Immediately following Theorem 38, we can conclude the following.

► **Corollary 39.** For any  $\gamma$ -acyclic query, Algorithm 5 converts a given connected left-deep linear join plan to a join tree.

## 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; in Section 5, we showed that the Maximum Cardinality Search algorithm constructs the unique shallowest join tree for any Berge-acyclic query; in Section 6, we characterize the class of binary join plans that can be converted to join trees. Practitioners can choose from the three approaches to integrate instance-optimal algorithms into their system: a cost-based optimizer can use our enumeration algorithm to generate candidate query plans; a system aiming to support very large queries can implement the MCS algorithm to generate shallow join trees to improve parallelism; and a more conservative extension can derive join trees using existing optimization infrastructure, while our final result guarantees the validity of the output.



■ **Figure 6**  $H_3$  is not Berge-acyclic but 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  $\gamma$ -acyclic and 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  $W$ .

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 work also raises further 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  $\gamma$ -*acyclicity* is insufficient as shown in Figure 6. What is the precise characterization of hypergraphs that admit a unique canonical join tree for any root, or for *some* root? How can our algorithms be extended to enumerate tree decompositions à la Carmeli et al. [6]? One possible direction is to further develop connections between MSTs of the line graph and tree decompositions of the hypergraph. For example, we are considering using the determinants of the line graph Laplacian as heuristics to guide the search for good tree decompositions.

## 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 Vansummeren, 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-vansummeren.pdf>.
- 3 Michael A. Bender and Martin Farach-Colton. The lca problem revisited. In *LATIN 2000*, 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. doi:10.1016/J.TCS.2003.05.002.
- 5 Nicolas Bruno, César A. Galindo-Legaria, and Milind Joshi. Polynomial heuristics for query optimization. In *Proceedings of ICDE 2010*, pages 589–600. IEEE, 2010. doi:10.1109/ICDE.2010.5447916.
- 6 Nofar Carmeli, Batya Kenig, and Benny Kimelfeld. Efficiently enumerating minimal triangulations. In *Proceedings of the 36th ACM Symposium on Principles of Database Systems (PODS 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. *Proc. VLDB Endow.*, 15(8):1533–1545, 2022. doi:10.14778/3529337.3529339.

- 9 Alessandro D’atri and Marina Moscarini. On hypergraph acyclicity and graph chordality. *Information Processing Letters*, 29(5):271–274, 1988. doi:10.1016/0020-0190(88)90121-4.
- 10 Pablo De Caria and Marisa Gutierrez. On basic chordal graphs and some of its subclasses. *Discrete Applied Mathematics*, 210:261–276, 2016. doi:10.1016/j.dam.2015.05.002.
- 11 David DeHaan and Frank Wm. Tompa. Optimal top-down join enumeration. In *Proceedings of SIGMOD 2007*, pages 785–796, 2007. doi:10.1145/1247480.1247567.
- 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. *J. ACM*, 30(3):514–550, 1983. doi:10.1145/2402.322390.
- 14 Leonidas Fegaras. A new heuristic for optimizing large queries. In *Proceedings of DEXA 1998*, volume 1460 of *Lecture Notes in Computer Science*, pages 726–735. Springer, 1998. doi:10.1007/BFB0054528.
- 15 Pit Fender and Guido Moerkotte. Counter strike: Generic top-down join enumeration for hypergraphs. *Proc. VLDB Endow.*, 6(14):1822–1833, 2013. doi:10.14778/2556549.2556565.
- 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. 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-&-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. *ACM Trans. Database Syst.*, October 2025. Just Accepted. doi:10.1145/3774325.
- 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. doi:10.1137/S009753979225030X.
- 23 Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB 1986)*, 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? *Proc. VLDB Endow.*, 9(3):204–215, 2015. doi:10.14778/2850583.2850594.
- 25 Arne Leitert. Computing the union join and subset graph of acyclic hypergraphs in subquadratic time. In *WADS 2021*, 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 *Proceedings of the 2016 International Conference on Management of Data (SIGMOD 2016)*, pages 615–629. ACM, 2016. doi:10.1145/2882903.2915235.
- 27 Zheng Luo, Wim Van den Broeck, Guy Van den Broeck, and Yisu Remy Wang. Algorithms for optimizing acyclic queries, 2026. doi:10.48550/arXiv.2509.14144.
- 28 David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.
- 29 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 VLDB 2006*, pages 930–941, 2006. URL: <http://dl.acm.org/citation.cfm?id=1164207>.

- 30 Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2008)*, pages 539–552. ACM, 2008. doi:10.1145/1376616.1376672.
- 31 Thomas Neumann. Query simplification: Graceful degradation for join-order optimization. In *Proceedings of SIGMOD 2009*, pages 403–414, 2009. doi:10.1145/1559845.1559889.
- 32 Øystein Ore. *Theory of Graphs*, volume 38 of *American Mathematical Society Colloquium Publications*. American Mathematical Society, 1962. doi:10.1090/co11/038.
- 33 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 SIGMOD 1979*, pages 23–34, 1979. doi:10.1145/582095.582099.
- 34 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.
- 35 Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB Journal*, 6(3):191–208, 1997. doi:10.1007/S007780050040.
- 36 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.
- 37 Arun N. Swami. Optimization of large join queries: Combining heuristic and combinatorial techniques. In *Proceedings of SIGMOD 1989*, pages 367–376, 1989. doi:10.1145/67544.66961.
- 38 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. doi:10.1137/0213035.
- 39 Transaction Processing Performance Council. TPC Benchmark H (Decision Support). [https://tpc.org/tpc\\_documents\\_current\\_versions/pdf/tpc-h\\_v3.0.1.pdf](https://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf), 2023. Standard Specification, Revision 3.0.1.
- 40 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.
- 41 Yisu Remy Wang, Max Willsey, and Dan Suciu. Free join: Unifying worst-case optimal and traditional joins. *Proc. ACM Manag. Data*, 1(2), 2023. doi:10.1145/3589295.
- 42 Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of VLDB 1981*, pages 82–94, 1981.
- 43 C. T. Yu and M. Z. Ozsoyoglu. An algorithm for tree-query membership of a distributed query. In *COMPSAC 1979*, pages 306–312. IEEE, 1979. doi:10.1109/CMPSAC.1979.762509.
- 44 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 *Proceedings of EMNLP 2018*, pages 3911–3921. Association for Computational Linguistics, 2018. doi:10.18653/v1/d18-1425.
- 45 Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. Debunking the myth of join ordering: Toward robust sql analytics. *Proc. ACM Manag. Data*, 3(3), 2025. doi:10.1145/3725283.