Output-Optimal Algorithms for Join-Aggregate Queries

XIAO HU, University of Waterloo, Canada

One of the most celebrated results of computing join-aggregate queries defined over commutative semi-rings is the classic Yannakakis algorithm proposed in 1981. It is known that the runtime of the Yannakakis algorithm is $O(N + \text{OUT})$ for any free-connex query, where N is the input size of the database and OUT is the output size of the query result. This is already output-optimal. However, only an upper bound $O(N \cdot \text{OUT})$ on the runtime is known for the large remaining class of acyclic but non-free-connex queries. Alternatively, one can convert a non-free-connex query into a free-connex one using tree decomposition techniques and then run the Yannakakis algorithm. This approach takes $O\left(N^{\#\text{fn-subw}} + \text{OUT}\right)$ time, where #fn-subw is the *free-connex sub-modular width* of the query. But, none of these results is known to be output-optimal.

In this paper, we show a matching lower and upper bound $\Theta\left(N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}}} + \text{OUT}\right)$ for computing general acyclic join-aggregate queries by *semiring algorithms*, where fn-fhtw is the *free-connex fractional hypertree width* of the query. For example, fn-fhtw $= 1$ for free-connex queries, fn-fhtw $= 2$ for line queries (a.k.a. chain matrix multiplication), and fn-fhtw $= k$ for star queries (a.k.a. star matrix multiplication) with k relations. Although free-connex fractional hypertree width is a natural and well-established measure of how far a join-aggregate query is from being free-connex, we demonstrate that it precisely captures the output-optimal complexity of these queries. To our knowledge, this has been the first polynomial improvement over the Yannakakis algorithm in the last 40 years and completely resolves the open question of computing acyclic join-aggregate queries in an output-optimal way. As a by-product, our output-optimal algorithm for acyclic queries also yields new output-sensitive algorithms for cyclic queries via tree decomposition techniques.

CCS Concepts: • **Theory of computation → Database query processing and optimization (theory)**.

Additional Key Words and Phrases: join-aggregate query, output-optimality, semi-ring algorithm, free-connex fractional hypertree width, Yannakakis algorithm

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

1 INTRODUCTION

Join-aggregate queries defined over commutative semi-rings have wide applications in data analytical tasks. For example, join-aggregate queries over Boolean semiring can capture the CNF satisfiability problem, the k-colorability problem on graphs, the Boolean conjunctive query [2], the constraint satisfaction problem, and the list recovery problem in coding theory [25]. As another example, join-aggregate queries over sum-product semiring have been widely used in complex network analysis (such as clustering coefficients and transitivity ratio), discrete Fourier transforms, graph analysis (such as homomorphism and Holant problem [15]), counting quantified conjunctive query, and permanent computation of matrices. Finally, join-aggregate queries over max-product

Author's address: Xiao Hu, xiaohu@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, Ontario, Canada, N2L 3G1.

ACM 2836-6573/2025/5-ART104
https://doi.org/10.1145/3725241

semiring can capture the maximum a posteriori problem in probabilistic graph models and maximum likelihood decoding in linear codes [5]. We refer interested readers to [3, 5, 20, 38] for many more applications.

Finding efficient algorithms for computing *join-aggregate queries* has been a holy grail in database theory since 1981. Previous results have achieved two flavors of runtimes: *worst-case optimal* [8, 36, 43, 53, 60] and *output-sensitive* [36, 60]. Worst-case optimal bounds are tight only on pathological instances with huge outputs, which are rare in practice. In contrast, output-sensitive bounds express the runtime as a function of the input size N and output size OUT, which are more practically meaningful, especially for queries where the aggregation may significantly reduce the output size. In addition, output-sensitive algorithms can imply worst-case optimal ones: the classical Yannakakis algorithm [60] is the best-known example, which achieves an output-optimal bound of $O(N + \text{OUT})$ for *free-connex* queries (including *acyclic joins* as special cases). Although output-optimal algorithms are practically desirable, they are much more difficult to design. Figure 1 illustrates the relationships between different classes of acyclic join-aggregate queries. For the largest class of *acyclic but non-free-connex* queries (in shadow area in Figure 1), prior works have yet to discover an output-optimal algorithm. Currently, there are only two approaches for computing these queries: (1) run the Yannakakis algorithm; (2) convert the query into a *free-connex one* using the *tree decomposition* technique and the *worst-case optimal join* algorithm [42, 44], and then run the Yannakakis algorithm on the tree decomposition. For (1), Yannakakis only gave an upper bound $O(N \cdot \text{OUT})$ on its runtime. Later, this bound has been tightened to $O\left(N \cdot \text{OUT}^{1-\frac{1}{k}}\right)$ for *star queries* with k relations (the *matrix multiplication* query is the special case with two relations), which is already output-optimal [48]. For (2), Khamis et al. showed an upper bound $O\left(N^{\#\text{fn-subw}} + \text{OUT}\right)$ on its runtime, where #fn-subw is the *#free-connex submodular width* of the query [36]. Both algorithms are worst-case optimal (see full version [27]), but neither is *output-optimal*.

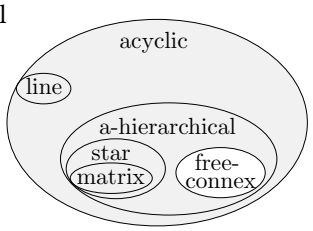

Fig. 1. Classification of acyclic join-aggregate queries.

In this work, we identify the free-connex fractional hypertree width (fn-fhtw) for join-aggregate queries to characterize the output-optimal complexity. We develop a matching lower and upper bound $\Theta\left(N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}}} + \text{OUT}\right)$ for computing general acyclic join-aggregate queries. Note that fn-fhtw $= 1$ for the free-connex queries and fn-fhtw $= k$ for star queries with k relations, thereby generalizing the previous results on these two special cases. Furthermore, since this bound is output-optimal, it unifies and improves previously mentioned incomparable approaches for acyclic queries. As a by-product, our output-optimal algorithm for acyclic queries also yields new output-sensitive algorithms for cyclic queries, although their optimality remains unclear. In addition to our algorithmic contribution, we prove that several important notions of width identified in the literature, such as #free-connex submodular width [37] and free-connex submodular width [37], *collapse* to free-connex fractional hypertree width fn-fhtw on acyclic queries. This surprising finding further verifies our intuition that this is the right notion to capture the output-optimal complexity of acyclic join-aggregate queries.

1.1 Problem Definition

Join Queries. A (natural) *join* is defined as a hypergraph $q = (\mathcal{V}, \mathcal{E})$, where the set of vertices $\mathcal{V} = \{x_1, \ldots, x_\ell\}$ model the *attributes* and the set of hyperedges $\mathcal{E} = \{e_1, \ldots, e_k\} \subseteq 2^{\mathcal{V}}$ model the *relations*. Let $\text{dom}(x)$ be the *domain* of attribute $x \in \mathcal{V}$. Let $\text{dom}(X) = \prod_{x \in X} \text{dom}(x)$ be the *domain* of a subset $X \subseteq \mathcal{V}$ of attributes. An *instance* of q is associated a set of relations $\mathcal{R} = \{R_e : e \in \mathcal{E}\}$. Each relation R_e consists of a set of *tuples*, where each tuple $t \in R_e$ is an

assignment that assigns a value from $\mathrm{dom}(x)$ to x for every attribute $x \in e$. The hypergraph q is called *self-join-free* if every relation R_e is distinct. In this work, we focus on self-join-free queries. More specifically, our lower bounds assume self-join-free queries, but our algorithm can be applied to the case when self-join exists. The *full join result* of q on $\mathcal{R}$, denoted as $q(\mathcal{R})$, is defined as $q(\mathcal{R}) = \{t \in \mathrm{dom}(\mathcal{V}) : \forall e \in \mathcal{E}, \pi_e t \in R_e\}$, i.e., all combinations of tuples, one from each relation, such that they share the same values on their common attributes.

Given a join query q and an instance $\mathcal{R}$, the *effective domain* of a subset of attributes $X \subseteq \mathcal{V}$ is defined as the collection of tuples in $\mathrm{dom}(X)$ that appears in at least one full join result of $q(\mathcal{R})$, i.e., the projection of $q(\mathcal{R})$ onto X.

Join-Aggregate Queries. A *join-aggregate query* is defined as a triple $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$, where $q = (\mathcal{V}, \mathcal{E})$ is a (natural) join, and $\mathbf{y} \subseteq \mathcal{V}$ is the set of *output attributes*. Let $(\mathbf{D}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ be a commutative semi-ring. We consider an instance $\mathcal{R}$ for Q with *annotated relations* [24, 35]. Every tuple t is associated with an *annotation* $w(t) \in \mathbf{D}$. The annotation of a full join result $t \in q(\mathcal{R})$ is $w(t) := \bigotimes_{e \in \mathcal{E}} w(\pi_e t)$. The *query result* of Q on $\mathcal{R}$ is defined as

$$Q(\mathcal{R}) = \bigoplus_{\mathcal{V}-\mathbf{y}} q(\mathcal{R}) = \left\{ (t_\mathbf{y}, w(t_\mathbf{y})) : t_\mathbf{y} \in \pi_\mathbf{y} q(\mathcal{R}), w(t_\mathbf{y}) = \bigoplus_{t \in q(\mathcal{R}) : \pi_\mathbf{y} t = t_\mathbf{y}} w(t) \right\}.$$

In plain language, a join-aggregate query (semantically) first computes the full join result $q(\mathcal{R})$ and the annotation of each result, which is the $\otimes$-aggregate of the tuples comprising the join result. Then it partitions $q(\mathcal{R})$ into groups by the attributes in $\mathbf{y}$. Finally, for each group, it computes the $\oplus$-aggregate of the annotations of the join result in that group. As mentioned, join-aggregate queries include many common database queries as special cases. For example, if we ignore the annotations, it becomes a join-project query $\pi_\mathbf{y} q(\mathcal{R})$, also known as a *conjunctive query*. If we take $\mathbf{D}$ be the domain of integers and set $w(t) = 1$ for every tuple t, it becomes the `COUNT(*) GROUP BY` $\mathbf{y}$ query; in particular, if $\mathbf{y} = \emptyset$, the query computes the full join size $|q(\mathcal{R})|$. If we take $\mathcal{V} = \{A, B, C\}$ with $\mathbf{y} = \{A, C\}$, and $\mathcal{E} = \{\{A, B\}, \{B, C\}\}$, it becomes the *matrix multiplication query*. If we take $\mathcal{V} = \{A_1, A_2, \cdots, A_k, B\}$ with $\mathbf{y} = \{A_1, A_2, \cdots, A_k\}$, and $\mathcal{E} = \{\{A_1, B\}, \cdots \{A_k, B\}\}$, it becomes a *star query*. If we take $\mathcal{V} = \{A_1, A_2, \cdots, A_{k+1}\}$ for $k \geqslant 3$ with $\mathbf{y} = \{A_1, A_{k+1}\}$, and $\mathcal{E} = \{\{A_1, A_2\}, \{A_2, A_3\}, \cdots, \{A_k, A_{k+1}\}\}$, it becomes a *line query*.

Below, if not specified, a query always refers to a join-aggregate query. We use $N = \sum_{e \in \mathcal{E}} |R_e|$ to denote the *input size* of $\mathcal{R}$ and $\mathrm{OUT} = |Q(\mathcal{R})|$ to denote the *output size* of Q over $\mathcal{R}$. We study the data complexity of this problem by assuming the query size (i.e., $|\mathcal{V}|$ and $|\mathcal{E}|$) as constants.

Model of Computation. We use the standard RAM model with uniform cost measures. A tuple or a semiring element is stored in a word. Copying one semiring element or combining two semiring elements via a semiring operation ($\oplus$ and $\otimes$) can be done in $O(1)$ time. Inheriting from [47], we confine ourselves to *semiring algorithms* that work with semiring elements as an abstract type and can only copy them from existing semiring elements or combine them using $\oplus$ or $\otimes$. No other operations on semi-ring elements are allowed, such as division, subtraction, or equality check.

Output-Optimality. To establish *output-optimality* for algorithm design, we consider a unified output-sensitive upper and lower bound in Definition 1.1.

Definition 1.1 (Output-sensitive Bound). For a self-join-free query Q, let $f(Q)$ be the smallest exponent such that for any parameters $1 \leqslant N$ and $\mathrm{OUT} \leqslant \max_{\mathcal{R}' \in \mathfrak{R}(N)} |Q(\mathcal{R}')|$, a semi-ring algorithm exists that can compute $Q(\mathcal{R})$ for any instance $\mathcal{R}$ of input size N and output size OUT within $O\left(N \cdot \mathrm{OUT}^{1 - \frac{1}{f(Q)}} + \mathrm{OUT}\right)$ time, where $\mathfrak{R}(N)$ is the set of all instances over Q of input size N.

Join-Aggregate Query	Yannakakis [36, 60]	Deep et al. [22]	Our Algorithm
a-Hierarchical	$N \cdot \text{OUT}^{1-\frac{1}{\#\text{fn-subw}}}$		
Line	$\min\{N \cdot \text{OUT}, N^2\}$	$N \cdot \text{OUT}^{1-\frac{1}{k}}$	$N \cdot \sqrt{\text{OUT}}$
Acyclic	$\min \begin{cases} N \cdot \text{OUT} \\ N^{\#\text{fn-subw}} + \text{OUT} \end{cases}$	$N \cdot \text{OUT}^{1-\frac{1}{\text{projw}}} + \text{OUT}$	$N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}}} + \text{OUT}$

Fig. 2. Comparison between previous and our new upper bounds. All results are in $\Theta(\cdot)$. N is the input size, and OUT is the output size. k is the number of relations. fn-fhtw is the free-connex fractional hypertree width (Definition 3.1). projw is the project-width (Definition 3.6). #fn-subw is the #free-connex submodular width. As shown in Lemma 3.7, $\#\text{fn-subw}(Q) = \text{fn-fhtw}(Q)$ for any acyclic query Q.

This bound is a monotonic function of $f(Q)$. A smaller $f(Q)$ implies a smaller runtime, i.e., a better upper bound. For example, $f(Q) \leqslant 1$ implies $O(N + \text{OUT})$; and $f(Q) \leqslant +\infty$ implies $O(N \cdot \text{OUT})$. This definition can also express lower bounds. For example, $f(Q) \geqslant 1$ implies $\Omega(N + \text{OUT})$. Below, we characterize $f(\cdot)$ for acyclic queries with both lower and upper bounds.

1.2 Our New Lower Bound for Acyclic Queries

Prior work has provided lower bounds of $f(Q)$. First, $f(Q) \geqslant 1$ for all queries since any algorithm must read the input data and output all query results. Pagh et al. showed $f(Q) \geqslant k$ for star queries with k relations [47]. Hu identified the *free-width* for an acyclic query Q, denoted as $\text{freew}(Q)$, and showed that $f(Q) \geqslant \text{freew}(Q)$ [28]. In this paper, we prove:

Theorem 1.2. *For any acyclic query Q, $f(Q) \geqslant \text{fn-fhtw}(Q)$.*

We next give some simple observations to understand our significant improvement over [28] and defer the detailed comparison between $\text{fn-fhtw}(Q)$ and $\text{freew}(Q)$ to Section 3. First, $\text{fn-fhtw}(Q) \geqslant \text{freew}(Q) \geqslant 1$ for all queries. In some cases, $\text{fn-fhtw}(Q) = \text{freew}(Q)$, such as free-connex queries, line queries, and star queries. But, for many other cases, $\text{fn-fhtw}(Q) > \text{freew}(Q)$.

This fn-fhtw-dependent lower bound can be broken beyond semiring algorithms. For example, some works use fast matrix multiplication techniques to speed up conjunctive queries (as a special case of join-aggregate queries defined over Boolean semiring) processing [1, 7, 21, 28] or graph pattern search (as a special case of self-joins) [14, 18, 33]. However, we cannot apply these techniques to arbitrary join-aggregate queries, since a general semiring does not necessarily have an additive inverse (such as the tropical semi-ring), so we won't pursue this dimension further in this paper.

1.3 Our New Upper Bound for Acyclic Queries

In this paper, we propose a new algorithm by exploring a hybrid version of the Yannakakis algorithm for computing acyclic queries, and therefore we can prove:

Theorem 1.3. *For any acyclic query Q, $f(Q) \leqslant \text{fn-fhtw}(Q)$.*

Combine Theorem 1.2 and Theorem 1.3, we obtain a full understanding of $f(\cdot)$ for acyclic queries:

Corollary 1.4. *For any acyclic query Q, $f(Q) = \text{fn-fhtw}(Q)$.*

Comparison with [60]. One question remains for the Yannakakis algorithm: *Is the unsatisfactory upper bound $O(N \cdot \text{OUT})$ due to a fundamental limitation of the algorithm itself or just because we do*

not have a tight analysis of its runtime? The runtime bound of the Yannakakis algorithm has been tightened on free-connex queries and star queries [48]. Recently, Hu showed that the Yannakakis algorithm indeed requires $\Theta(N \cdot \text{OUT})$ time for *line queries* [28], which has first demonstrated the limitation of the Yannakakis algorithm. We provide a tight analysis for all acyclic queries. More specifically, the runtime bound can be tightened to $O\left(N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}(Q)}} + \text{OUT}\right)$ for *a-hierarchical queries*. In contrast, $O(N \cdot \text{OUT})$ is tight for all non-a-hierarchical queries. As shown in Figure 1, a-hierarchical queries include free-connex and star queries, but not line queries. Hence, our new algorithm strictly outperforms the Yannakakis algorithm on all non-a-hierarchical queries.

Comparison with [36]. Another approach (even applying for cyclic queries) that converts a query into a free-connex one and then runs the Yannakakis algorithm takes $O\left(N^{\#\text{fn-subw}(Q)} + \text{OUT}\right)$ time, where $\#\text{fn-subw}(Q)$ is the #free-connex sub-modular width of query Q [36]. As shown in Lemma 3.10, both notions of width collapse on all acyclic queries, i.e., $\#\text{fn-subw}(Q) = \text{fn-fhtw}(Q)$ for any acyclic query Q. It is not hard to see that this result is always worse (or at least not better) than our new result due to $N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}(Q)}} + \text{OUT} = O\left(N^{\text{fn-fhtw}(Q)} + \text{OUT}\right)$. Moreover, this result is worse than our new result by a factor of $\left(\frac{N^{\text{fn-fhtw}(Q)}}{\text{OUT}}\right)^{1-\frac{1}{\text{fn-fhtw}(Q)}}$ when $\text{OUT} < N^{\text{fn-fhtw}(Q)}$.

Comparison with [22]. In an independent work from ours, Deep et al. identified the *project-width* for a query Q, denoted as $\text{projw}(Q)$, and showed that $f(Q) \leqslant \text{projw}(Q)$ [22]. Again, we give some observations to understand our advantages over this upper bound and defer a detailed comparison to Section 3. First, $\text{fn-fhtw}(Q) \leqslant \text{projw}(Q)$ for all queries. For a-hierarchical queries, $\text{fn-fhtw}(Q) = \text{projw}(Q)$. For line queries with k relations, $\text{fn-fhtw}(Q) = 2 < \text{projw}(Q) = k$. Also, for many other cases, $\text{fn-fhtw}(Q) < \text{projw}(Q)$.

We summarize the runtime of our new algorithm and comparable algorithms in Figure 2.

1.4 Implications to Cyclic Queries

The common approach that converts a query into an acyclic one using tree decomposition techniques and then runs the Yannakakis algorithm takes $O\left(N^{\#\text{subw}(Q)} \cdot \text{OUT}\right)$ time, where $\#\text{subw}(Q)$ is the #sub-modular width of the query Q [40]. If restricting tree decompositions to be free-connex, this takes $O\left(N^{\#\text{fn-subw}(Q)} + \text{OUT}\right)$ time. Note that $\#\text{subw}(Q) \leqslant \#\text{fn-subw}(Q)$ for all queries. These two results are incomparable unless the value of OUT is known. By replacing the Yannakakis algorithm with our new output-optimal algorithm, we can get new output-sensitive algorithms for cyclic queries. However, their optimality is unclear, which we leave as an open question.

1.5 Organization of This Paper

Our paper is organized as follows. In Section 2, we introduce the preliminaries. In Section 3, we define the free-connex fractional hypertree width and investigate its properties for acyclic queries. In Section 4, we review the Yannakakis algorithm and introduce our algorithm for line queries as an introductory example. We present our algorithm for general acyclic queries in Section 5. In Section 6, we show output-sensitive algorithms for cyclic queries. Finally, we review other related works in Section 8 and conclude in Section 9.

2 PRELIMINARIES

2.1 Fractional Edge Covering and AGM Bound

For a join query $q = (\mathcal{V}, \mathcal{E})$, we use $\mathcal{E}_x = \{e \in \mathcal{E} : x \in e\}$ to denote the set of relations containing attribute x. An attribute $x \in \mathcal{V}$ is *unique* if $|\mathcal{E}_x| = 1$, and *joint* otherwise. For a subset of attributes $S \subseteq \mathcal{V}$, we use $q[S] = (S, \mathcal{E}[S])$ to denote the sub-query induced by S, where $\mathcal{E}[S] = \{e \cap S : e \in \mathcal{E}\}$.

A *fractional edge covering* is a function $\rho : \mathcal{E} \to [0, 1]$ such that $\sum_{e:A\in e} \rho(e) \geqslant 1$ for each attribute $A \in \mathcal{V}$. The *fractional edge covering number* of q, denoted as $\rho^*(q)$, is defined as the minimum sum of weight over all possible fractional edge coverings ρ for q, i.e., $\rho^*(q) = \min_\rho \sum_{e\in\mathcal{E}} \rho(e)$. For a join query $q = (\mathcal{V}, \mathcal{E})$ and any parameter $N \in \mathbb{Z}^+$, the AGM bound [8] states that the maximum number of join results produced by any instance of input size N is $\Theta(N^{\rho^*(q)})$. For a join-aggregate query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$ with $q = (\mathcal{V}, \mathcal{E})$ and any parameter $N \in \mathbb{Z}^+$, the maximum number of join results produced by any instance of input size N is $\Theta(N^{\rho^*(q[\mathbf{y}])})$.

2.2 Tree Decompositions

A *tree decomposition* (TD) of $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$ is a pair $(\mathcal{T}, \chi)$, where $\mathcal{T}$ is a tree and $\chi : \text{nodes}(\mathcal{T}) \to 2^{\mathcal{V}}$ is a mapping from the nodes of $\mathcal{T}$ to subsets of $\mathcal{V}$, that satisfies the following properties:

- For each relation $e \in \mathcal{E}$, there is a node $u \in \text{nodes}(\mathcal{T})$ such that $e \subseteq \chi(u)$.
- For each attribute $A \in \mathcal{V}$, the set $\{u \in \text{nodes}(\mathcal{T}) : A \in \chi(u)\}$ forms a connected sub-tree of $\mathcal{T}$.

Each set $\chi(u)$ is called a *bag* of the TD. Wlog, we assume $\chi(u) \neq \chi(u')$ for any pair of nodes $e, e' \in \text{nodes}(\mathcal{T})$. The *width* of $(\mathcal{T}, \chi)$ noted as $\text{width}(\mathcal{T}, \chi)$ is defined as

$$\text{width}(\mathcal{T}, \chi) \quad = \quad \max_{u\in\text{nodes}(\mathcal{T})} \quad \rho^*\left(q[\chi(u)]\right)$$

i.e., the maximum fractional edge covering number of the subqueries induced by all bags in $\mathcal{T}$. A TD is *non-redundant* if no bag is a subset of another. Below, we can only consider non-redundant TDs. A TD $(\mathcal{T}, \chi)$ is *free-connex* if there is a connected subtree S of $\mathcal{T}$ such that $\bigcup_{u\in\text{nodes}(S)} \chi(u) = \mathbf{y}$, i.e., the union of attributes appearing in S is exactly the output attributes. S is called a *connex* of $\mathcal{T}$.

2.3 Classification of Queries

Acyclic [11, 32]. A query is acyclic if and only if it has a width-1 TD.

Free-connex [9]. A query is free-connex if and only if it has a width-1 free-connex TD.

Hierarchical [51]. A query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$ is hierarchical if for any pair of attributes $A, B \in \mathcal{V}$, either $\mathcal{E}_A \subseteq \mathcal{E}_B$, or $\mathcal{E}_B \subseteq \mathcal{E}_A$, or $\mathcal{E}_A \cap \mathcal{E}_B = \emptyset$.

∃-Connected. The *existential connectivity* of a query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$ is defined on a graph $G^{\exists}_Q$, where each relation $e \in \mathcal{E}$ is a vertex, and an edge exists between $e, e' \in \mathcal{E}$ if $e \cap e' - \mathbf{y} \neq \emptyset$. Q is *∃-connected* if $G^{\exists}_Q$ is connected, and *∃-disconnected* otherwise. If Q is ∃-disconnected, we can *decompose* it as follows. We find all connected components of $G^{\exists}_Q$, in which the set of relations corresponding to the set of vertices in one connected component of $G^{\exists}_Q$ form a connected subquery of Q. There are four ∃-connected subqueries of Q: 1 is a complicated acyclic query, 2 is a line query, 3 is a star query, and 4 is a single relation.

Cleansed. A query Q is *cleansed* if every unique attribute is an output attribute, and there exist no relations whose attributes are fully contained by another one. If Q is not cleansed, we *cleanse* it by iteratively removing a unique non-output attribute or a relation whose attributes are fully contained by another. The resulting query is the *cleansed* version of Q.

Separated. A query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$ is *separated* if every output attribute is unique, every unique attribute is an output attribute, and for each $e \in \mathcal{E}$ with $e \cap \mathbf{y} \neq \emptyset$, there exists some $e' \in \mathcal{E} - \{e\}$ with $e - \mathbf{y} \subseteq e'$. Among these 4 queries, all of 2, 3, 4 are separated, but 1 is not.

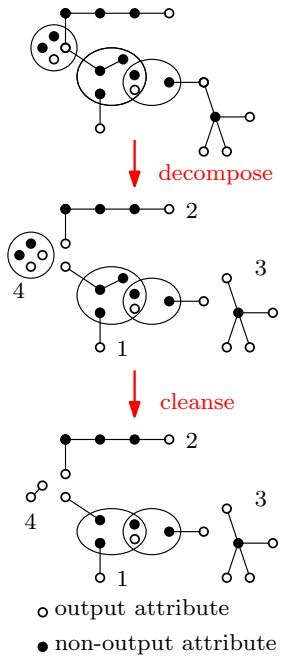

Fig. 3. An illustration of the decompose and cleanse process.

Agg-hierarchical. A query Q is agg-hierarchical if every $\exists$-connected subquery in $G^{\exists}_{Q'}$ is hierarchical, where Q' is the cleansed version of Q.

3 FREE-CONNEX FRACTIONAL HYPERTREE WIDTH

In this section, we give a structural definition of free-connex fractional hypertree width for general queries. For acyclic queries, we characterize an equivalent procedural definition, which is fundamental in facilitating comparisons between free-connex fractional hypertree width with other notions of width in Section 3.2, and inspiring our output-optimal algorithm in Section 5.

3.1 Definition

The structural definition of free-connex fractional hypertree width is defined on free-connex tree decompositions. Let $\mathsf{FTD}(Q)$ denote the set of all free-connex TDs for a query Q.

*Definition 3.1 (Free-connex Fractional Hypertree Width (*fn-fhtw*)).* For a query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$, its free-connex fractional hypertree width $\mathsf{fn\text{-}fhtw}(Q)$ is defined as:

$$\mathsf{fn\text{-}fhtw}(Q) \quad = \quad \min_{(\mathcal{T},\chi)\in\mathsf{FTD}(Q)} \quad \mathsf{width}(\mathcal{T},\chi) \tag{1}$$

i.e., the minimum width of all possible free-connex TDs.

It is easy to see $\mathsf{fn\text{-}fhtw}(Q) \geqslant 1$ for all queries and $\mathsf{fn\text{-}fhtw}(Q) = 1$ for free-connex queries. Furthermore, fn-fhtw is preserved in the cleanse process (see Lemma 3.2). When restricting our scope to acyclic queries, we prove two important properties of fn-fhtw in Lemma 3.3 and Lemma 3.4, which serves as a procedural definition of free-connex fractional hypertree width. Their proofs are rather technical and deferred to the full version [27].

Lemma 3.2. *For any query Q, $\mathsf{fn\text{-}fhtw}(Q) = \mathsf{fn\text{-}fhtw}(Q')$, where Q' is cleansed version of Q.*

Lemma 3.3. *For any acyclic query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$, if Q is $\exists$-disconnected with $\exists$-connected subqueries $Q_1, Q_2, \cdots, Q_h$, then $\mathsf{fn\text{-}fhtw}(Q) = \max_{i\in[h]} \mathsf{fn\text{-}fhtw}(Q_i)$.*

Lemma 3.4. *For any acyclic query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$, if Q is $\exists$-connected, then $\mathsf{fn\text{-}fhtw}(Q) = \rho^*(q[\mathbf{y}])$ for $q = (\mathcal{V}, \mathcal{E})$, i.e., the fractional edge covering number of the sub-query induced by output attributes.*

Corollary 3.5. *For any acyclic query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$, its free-connex fractional hypertree width $\mathsf{fn\text{-}fhtw}(Q)$ is recursively defined as:*

- *If Q is $\exists$-disconnected with $\exists$-connected subqueries $Q_1, Q_2, \cdots, Q_h$, $\mathsf{fn\text{-}fhtw}(Q) = \max_{i\in[h]} \mathsf{fn\text{-}fhtw}(Q_i)$.*
- *If Q is $\exists$-connected, $\mathsf{fn\text{-}fhtw}(Q) = \rho^*(q[\mathbf{y}])$ for $q = (\mathcal{V}, \mathcal{E})$, i.e., the fractional edge covering number of the sub-query induced by output attributes.*

3.2 Comparison with Other Notions of Width

Free-width and Project-width. We first review the definitions for *free-width* and *project-width*:

Definition 3.6 (Free-width [28] and Project-width [22]). For any acyclic query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$, its free-width $\mathsf{freew}(Q)$ and project-width $\mathsf{projw}(Q)$ are defined as follows:

- If Q is $\exists$-disconnected with $\exists$-connected subqueries $Q_1, Q_2, \cdots, Q_h$, $\mathsf{freew}(Q) = \max_{i\in[h]} \mathsf{freew}(Q_i)$, and $\mathsf{projw}(Q) = \max_{i\in[h]} \mathsf{projw}(Q_i)$.
- If Q is $\exists$-connected but not cleansed, $\mathsf{freew}(Q) = \mathsf{freew}(Q')$ and $\mathsf{projw}(Q) = \mathsf{projw}(Q')$, where Q' is the cleansed version of Q.
- If Q is $\exists$-connected and cleansed, $\mathsf{freew}(Q) = |\{e \in \mathcal{E} : e \cap \mathcal{V}_\bullet \neq \emptyset\}|$, where $\mathcal{V}_\bullet$ is the set of unique (output) attributes in Q; and $\mathsf{projw}(Q) = |\mathcal{E}|$.

Lemma 3.7. *For any acyclic query* Q, $\mathsf{freew}(Q) \leqslant \mathsf{fn\text{-}fhtw}(Q) \leqslant \mathsf{projw}(Q)$.

These three notions of width share the exact definition if Q is $\exists$-disconnected or not cleansed. The only difference comes when Q is $\exists$-connected and cleansed. In this case, it is not hard to see that $\mathsf{freew}(Q) \leqslant \mathsf{fn\text{-}fhtw}(Q) \leqslant \mathsf{projw}(Q)$. First, every relation containing a unique attribute (which must be an output attribute because Q is cleansed) should be assigned a weight of 1 in any fractional edge covering of $q[\mathbf{y}]$. This is why $\mathsf{freew}(Q) \leqslant \mathsf{fn\text{-}fhtw}(Q)$. Moreover, assigning all relations with weight 1 forms a trivial fractional edge covering for $q[\mathbf{y}]$, hence $\mathsf{fn\text{-}fhtw}(Q) \leqslant |\mathcal{E}| = \mathsf{projw}(Q)$. In Example 3.8, we show a query Q with $\mathsf{freew}(Q) < \mathsf{fn\text{-}fhtw}(Q) < \mathsf{projw}(Q)$.

Example 3.8. Consider an $\exists$-connected and cleansed query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$ in Figure 4, where $\mathcal{V} = \{A_1, A_2, A_3, B_1, B_2, B_3, C_1, C_2\}, \mathcal{E} = \{e_1 = \{A_1, B_1\}, e_2 = \{A_2, B_2\}, e_3 = \{A_3, B_3\}, e_5 = \{B_1, B_2, C_1, C_2\}, e_6 = \{B_3, C_1, C_2\}\}$, and $\mathbf{y} = \{A_1, A_2, A_3, C_2\}$. Note that A_1, A_2, A_3 are unique output attributes. All of e_1, e_2, e_3 contain some unique output attribute(s), so $\mathsf{freew}(Q) = 3$. It has the fractional edge covering number as 4, so $\mathsf{fn\text{-}fhtw}(Q) = 4$. It has 5 relations, so $\mathsf{projw}(Q) = 5$.

#Free-connex submodular width. The #free-connex submodular width (#fn-subw) is also defined based on tree decompositions but in a rather complicated formula. First, we can show $\#\mathsf{fn\text{-}subw}(Q) \leqslant \mathsf{fn\text{-}fhtw}(Q)$ for all queries. From [36], we can further draw a clear ordering by additionally involving submodular width (subw), free-connex submodular width (fn-subw), and #submodular width (#subw) as follows:

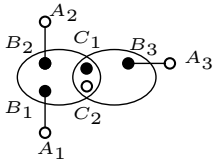

Fig. 4. An example query Q with $\mathsf{freew}(Q) < \mathsf{fn\text{-}fhtw}(Q) < \mathsf{projw}(Q)$.

Lemma 3.9. *For any query* Q, $\mathsf{subw}(Q) \leqslant \{\#\mathsf{subw}(Q), \mathsf{fn\text{-}subw}(Q)\} \leqslant \#\mathsf{fn\text{-}subw}(Q) \leqslant \mathsf{fn\text{-}fhtw}(Q)$.

Surprisingly, some widths collapse when we restrict our scope to acyclic queries!

Lemma 3.10. *For any acyclic query* Q, $\mathsf{fn\text{-}subw}(Q) = \#\mathsf{fn\text{-}subw}(Q) = \mathsf{fn\text{-}fhtw}(Q)$.

All missing proofs are deferred to the full version [27]. This result also implies the fundamental difference between existing algorithms and our new algorithm. On acyclic queries, the algorithm in [36] always picks one free-connex TD and performs computation according to it; in contrast, our algorithm partitions the input instance into multiple sub-instances, picks a set of free-connex TDs, and applies them to different sub-instances. The power of this *hybrid strategy* will become much clearer in our next section.

4 WARM UP: YANNAKAKIS REVISITED AND LINE QUERY

In this section, we first review the Yannakakis algorithm [35, 60] over acyclic join-aggregate queries. To illustrate some of the high-level ideas behind our general algorithm, we examine line queries (a.k.a. chain matrix multiplication), which represent the simplest scenario where the Yannakakis algorithm is not optimal:

$$Q_{\mathsf{line}} = \bigoplus_{A_2, A_3, \cdots, A_k} R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie \cdots \bowtie R_k(A_k, A_{k+1})$$

Notably, chain matrix multiplication has independently garnered significant attention in previous research [10, 23, 39, 45].

4.1 Yannakakis Algorithm Revisited

Suppose we are given an acyclic query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$ with $q = (\mathcal{V}, \mathcal{E})$ and an instance $\mathcal{R}$ for Q. For simplicity, we assume that there exists no pair of relations $e, e' \in \mathcal{E}$ such that $e \subseteq e'$; otherwise, we

Algorithm 1: YANNAKAKIS$(Q = (\mathcal{V}, \mathcal{E}, \mathbf{y}), \mathcal{R}, \mathcal{T})$ [35, 60]

foreach *node* e *of* $\mathcal{T}$ **do** $p_e \leftarrow$ the parent node of e in $\mathcal{T}$;
while *visit nodes of* $\mathcal{T}$ *in a bottom-up way (excluding the root* r*)* **do**
  foreach *node* e *visited* **do** $R_{p_e} \leftarrow R_{p_e} \ltimes R_e$;
while *visit nodes of* $\mathcal{T}$ *in a top-down way (excluding the root* r*)* **do**
  foreach *node* e *visited* **do** $R_e \leftarrow R_e \ltimes R_{p_e}$;
while *visit nodes of* $\mathcal{T}$ *in a bottom-up way (excluding the root* r*)* **do**
  foreach *node* e *visited* **do**
    $R_e \leftarrow \oplus_{e-p_e-\mathbf{y}} R_e$;
    $R_{p_e} \leftarrow R_e \bowtie R_{p_e}$;
return $\oplus_{r-\mathbf{y}} R_r$ for the root r;

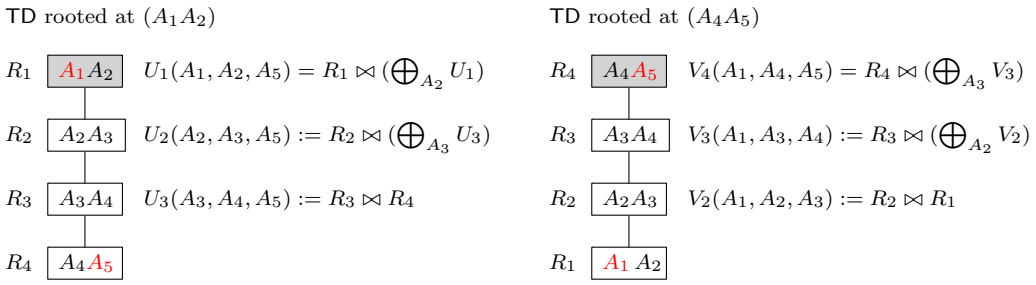

Fig. 5. An illustration of two width-1 TDs for a line query with $k = 4$. For the TD rooted at (A_1A_2), U_3, U_2, U_1 are the intermediate results materialized by the Yannakakis algorithm, implying a free-connex TD (left) in Figure 6. For the TD rooted at (A_4A_5), V_2, V_3, V_4 are the intermediate results materialized by the Yannakakis algorithm, implying a free-connex TD (right) in Figure 6.

can preprocess[1] these relations within $O(N)$ time. Let $(\mathcal{T}, \mathcal{X})$ be a width-1 (non-redundant) TD for Q. There is a one-to-one correspondence between relations in $\mathcal{E}$ and nodes in $\mathcal{T}$. For simplicity, we also use e to denote the node u in $\mathcal{T}$ that corresponds to relation e, i.e., $\chi(u) = e$. The algorithm consists of two phases:

Semi-Joins. A tuple is *dangling* if it does not participate in any full join result. The first phase removes all *dangling tuples* via a bottom-up and top-down pass of semi-joins along $\mathcal{T}$.

Pairwise Join-Aggregation. The second phase performs joins and aggregations in a bottom-up way along $\mathcal{T}$. Specifically, it takes two nodes R_e and R_{p_u} such that u is a leaf and p_u is the parent of u, aggregate over non-output attributes that do not appear in $\chi(p_u)$ by replacing R_e with $\oplus_{\chi(u)-\chi(p_u)-\mathbf{y}} R_e$, and replaces R_{p_u} with $R_e \bowtie R_{p_u}$. Then R_e is removed, and the step repeats until only one node remains, i.e., the root node r. Hence, it aggregates over all remaining non-output attributes in r, and outputs $\oplus_{r-\mathbf{y}} R_r$ as the final result. See an example in Figure 5.

The runtime is proportional to the largest number of intermediate results materialized (after dangling tuples are removed). But, this number can vary significantly depending on the sepecific *query plans* chosen. Each *query plan* corresponds to one width-1 TD together with a particular sequence of pairwise joins and aggregations. So, the runtime of the Yannakakis algorithm refers to the runtime of the *fastest* available query plan.

[1]For any pair of $e, e' \in \mathcal{E}$ with $e \subseteq e'$, we simply replace $R_{e'}$ by $R_{e'} \bowtie R_e$ and remove R_e. Note that the annotations of tuples in R_e are "passed" to tuples in $R_{e'}$ via the join. As $e \subseteq e'$, the join can be done within $O(N)$ time.

Algorithm 2: $\textsc{Line}(\mathcal{Q}_{\text{line}}, \mathcal{R})$

foreach $i \in [k-1]$ **do**
  if $i = 1$ **then** $T_1(A_1, A_2) \leftarrow R_1(A_1, A_2)$;
  else $T_i(A_1, A_{i+1}) \leftarrow \bigoplus_{A_i} S_{i-1}(A_1, A_i) \bowtie R_i(A_i, A_{i+1})$;
  $A_{i+1}^{\text{heavy}} \leftarrow \left\{a \in \text{dom}(A_{i+1}) : \left|\sigma_{A_{i+1}=a} T_i\right| > \sqrt{\tilde{\text{OUT}}}\right\}$;
  $A_{i+1}^{\text{light}} \leftarrow \left\{a \in \text{dom}(A_{i+1}) : 1 \leqslant \left|\sigma_{A_{i+1}=a} T_i\right| \leqslant \sqrt{\tilde{\text{OUT}}}\right\}$;
  $R_i^{\text{heavy}}, R_i^{\text{light}} \leftarrow R_i \ltimes A_{i+1}^{\text{heavy}},\ R_i \ltimes A_{i+1}^{\text{light}}$;
  $S_i(A_1, A_{i+1}) \leftarrow T_i(A_1, A_{i+1}) \ltimes A_{i+1}^{\text{light}}$;
foreach $i \in [k-1]$ **do**
  $Q_i \leftarrow \bigoplus_{A_2, A_3, \cdots, A_k} \left(\bowtie_{j \in [i-1]} R_j^{\text{light}}\right) \bowtie R_i^{\text{heavy}} \bowtie \left(\bowtie_{j=i+1}^{k} R_j\right)$;
$Q_* \leftarrow \bigoplus_{A_2, A_3, \cdots, A_k} \left(\bowtie_{j \in [k-1]} R_j^{\text{light}}\right) \bowtie R_k$;
return $Q_1 \oplus Q_2 \oplus \cdots \oplus Q_{k-1} \oplus Q_*$;

4.2 Line Queries

As pointed out by [28], there exists some instance for line queries such that every query plan of the Yannakakis algorithm must materialize $\Omega(N \cdot \text{OUT})$ intermediate results. From their hard instance, we are inspired to leverage *the power of multiple query plans* to overcome the fundamental limitation of the Yannakakis algorithm. Given the wide variety of TDsfor a line query, one natural question arises: which query plans should we select? For line queries, we only consider two query plans that correspond to two width-1 TDs: (a) one is rooted at (A_1A_2); (b) one is rooted at (A_kA_{k+1}). See Figure 5 and 6. Behind our hybrid strategy, the idea is to partition the input instance into a set of sub-instances and then choose one of two plans for each sub-instance. To determine which plan to use, we need a deeper analysis of data statistics. For example, if the effective domain of A_{k+1} is small, we choose the one rooted at (A_1A_2); and if that of A_1 is small, we choose the one rooted at (A_kA_{k+1}).

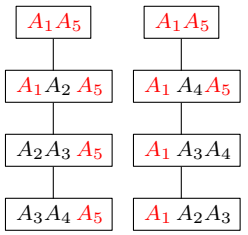

Fig. 6. An illustration of two free-connex TDs for a line query with $k = 4$.

Algorithm. As described in Algorithm 2, our new approach consists of two stages. In **Stage I**, we partition the input instance. In **Stage II**, we choose different query plans for each sub-instance, apply the Yannakakis algorithm, and aggregate all subqueries. See an example in Figure 7. We now assume a parameter $\tilde{\text{OUT}}$ is known such that $\tilde{\text{OUT}} \leqslant \text{OUT} \leqslant c \cdot \tilde{\text{OUT}}$ for some constant c. This assumption can be removed without increasing the complexity asymptotically [27].

Stage I: Partition. For each value $a \in \text{dom}(A_2)$, we define its *degree* as the number of tuples from R_1 displaying a in A_2, i.e., $\triangle(a) = |\pi_{A_1} \sigma_{A_2=a} R_1|$. A value $a \in \text{dom}(A_2)$ is *heavy* if $\triangle(a) > \sqrt{\tilde{\text{OUT}}}$, and *light* otherwise. Let $A_2^{\text{heavy}}, A_2^{\text{light}}$ be the set of heavy and light values in A_2. Let $R_1^{\text{heavy}} = R_1 \ltimes A_2^{\text{heavy}}$ and $R_1^{\text{light}} = R_1 \ltimes A_2^{\text{light}}$ be the set of heavy and light tuples in R_1 respectively. Below, we partition relations by ordering $R_2, R_3, \cdots, R_k$. Suppose we are done with $R_1, R_2, \cdots, R_{i-1}$. We next partition values in $\text{dom}(A_{i+1})$ that can be joined with any value in $\text{dom}(A_1)$ via $R_1^{\text{light}}, R_2^{\text{light}}, \cdots, R_{i-1}^{\text{light}}$. Let $\triangle(a) = \left|\pi_{A_1}\left\{\left(\bowtie_{j \in [i-1]} R_j^{\text{light}}\right) \bowtie (\sigma_{A_{i+1}=a} R_i)\right\}\right|$ be the *degree* of each value $a \in \text{dom}(A_{i+1})$. A

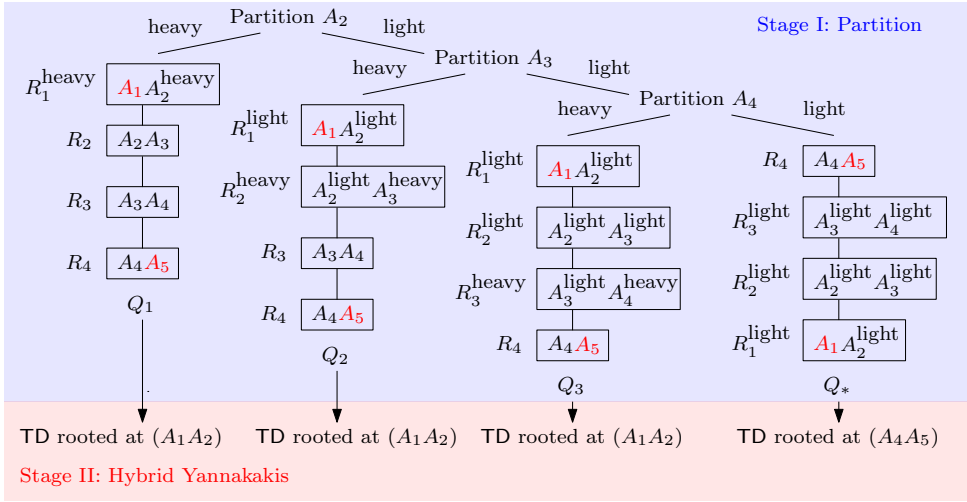

Fig. 7. An illustration of Algorithm 2 for line query with $k = 4$.

value $a \in \text{dom}(A_{i+1})$ is *heavy* if $\triangle(a) > \sqrt{\widetilde{\text{OUT}}}$, and *light* otherwise. Let $A_{i+1}^{\text{heavy}}, A_{i+1}^{\text{light}}$ be the set of heavy, light values in A_i. Then, $R_i^{\text{heavy}} = R_i \ltimes A_{i+1}^{\text{heavy}}$ and $R_i^{\text{light}} = R_i \ltimes A_{i+1}^{\text{light}}$. Note that some values in A_{i+1} may be undefined, as well as some tuples in R_i.

Instead of computing $\triangle(\cdot)$ directly, we introduce the following intermediate relations:

$$T_i(A_1, A_{i+1}) = \bigoplus_{A_2, A_3, \cdots, A_i} \left(\bowtie_{j \in [i-1]} R_j^{\text{light}}\right) \bowtie R_i, \text{ and } S_i(A_1, A_{i+1}) = \bigoplus_{A_2, A_3, \cdots, A_i} \left(\bowtie_{j \in [i]} R_j^{\text{light}}\right)$$

and recursively compute them as follows (with $T_1 = R_1$ in line 2):

$$T_i(A_1, A_{i+1}) = \bigoplus_{A_i} S_{i-1}(A_1, A_i) \bowtie R_i(A_i, A_{i+1}); \quad \text{(line 3)}$$

$$S_i(A_1, A_{i+1}) = T_i(A_1, A_{i+1}) \ltimes A_{i+1}^{\text{light}}; \quad \text{(line 7)}$$

Once we have computed T_i, we can identify the heavy and light values in A_{i+1}, i.e., A_{i+1}^{heavy} and A_{i+1}^{light} (lines 4-5). We can use $A_{i+1}^{\text{heavy}}, A_{i+1}^{\text{light}}$ to partition R_i into $R_i^{\text{heavy}}, R_i^{\text{light}}$ (line 6). Then, S_i can be computed based on T_i and A_{i+1}^{light}. Furthermore, T_{i+1} can be computed based on S_i and R_{i+1}.

We partition Q_{line} into k sub-instances: $Q_i = \bigoplus_{A_2, A_3, \cdots, A_k} \left(\bowtie_{j \in [i-1]} R_j^{\text{light}}\right) \bowtie R_i^{\text{heavy}} \bowtie \left(\bowtie_{j=i+1}^{k} R_j\right)$ for each $i \in [k-1]$, and $Q_* = \bigoplus_{A_2, A_3, \cdots, A_k} \left(\bowtie_{j \in [k-1]} R_j^{\text{light}}\right) \bowtie R_k$.

Stage II: Hybrid Yannakakis. We invoke the Yannakakis algorithm to compute Q_i for each $i \in [k-1]$ using the TD (a), and Q_* using the TD (b). In the latter case, it is equivalent to computing $\bigoplus_{A_k} S_{k-1}(A_1, A_k) \bowtie R_k(A_k, A_{k+1})$. Finally, we aggregate the results of all subqueries.

Analysis. In **Stage I**, consider any $i \in [k] - \{1\}$. As there are N tuples in R_i, and each of them can be joined with at most $\sqrt{\widetilde{\text{OUT}}}$ tuples in S_{i-1}, T_i can be computed in $O\left(N \cdot \sqrt{\text{OUT}}\right)$ time. Also, $|T_i| = O\left(N \cdot \sqrt{\text{OUT}}\right)$. The cost of computing S_i is $O(|T_i|) = O\left(N \cdot \sqrt{\text{OUT}}\right)$. In **Stage II**, for Q_i, each value in the effective domain of A_{k+1} can be joined with at least $\sqrt{\widetilde{\text{OUT}}}$ values in A_1, implied by A_{i+1}^{heavy}. As there are OUT results in total, the effective domain size of A_{k+1} is $O\left(\sqrt{\text{OUT}}\right)$.

Algorithm 3: AcyclicJoinAggregate$(Q, \mathcal{R})$

1 Remove dangling tuples in $\mathcal{R}$;
2 $Q_1, Q_2, \cdots, Q_h \leftarrow \exists$-connected subqueries of Q;
3 **foreach** $i \in [h]$ **do** $\mathcal{R}_i \leftarrow$ sub-instance of $\mathcal{R}$ for subquery Q_i;
4 **foreach** $i \in [h]$ **do**
5 $\quad (Q_i', \mathcal{R}_i') \leftarrow$ Cleanse$(Q_i, \mathcal{R}_i)$; ▸ Algorithm 4;
6 $\quad (Q_i'', \mathcal{R}_i'') \leftarrow$ Separate$(Q_i', \mathcal{R}_i')$; ▸ Algorithm 5;
7 $\quad \mathcal{S}_i \leftarrow$ HybridYannakakis$(Q_i'', \mathcal{R}_i'')$ ▸ Algorithm 6;
8 **return** $\bowtie_{i\in[h]} \mathcal{S}_i$ by Yannakakis algorithm [60]; ▸ Algorithm 1;

Algorithm 4: Cleanse$(Q = (\mathcal{V}, \mathcal{E}, \mathbf{y}), \mathcal{R})$

1 **while** $(\mathcal{V}, \mathcal{E}, \mathbf{y})$ *is not cleansed* **do**
2 $\quad$ **if** $\exists B \in \mathcal{V} - \mathbf{y}$ *s.t.* $|\mathcal{E}_B| = 1$, *say* $\mathcal{E}_B = \{e\}$ **then**
3 $\quad\quad R_e \leftarrow \oplus_B R_e$, $e \leftarrow e - \{B\}$, $\mathcal{V} \leftarrow \mathcal{V} - \{B\}$;
4 $\quad$ **if** $\exists e, e' \in \mathcal{E}$ *s.t.* $e \subseteq e'$ **then**
5 $\quad\quad R_{e'} \leftarrow R_{e'} \bowtie R_e$, $\mathcal{E} \leftarrow \mathcal{E} - \{e\}$;
6 **return** Updated $(\mathcal{V}, \mathcal{E}, \mathbf{y})$ and $\mathcal{R}$;

Hence, the number of intermediate join results materialized for each node is at most $O\left(N \cdot \sqrt{\text{OUT}}\right)$. For Q_*, the total number of intermediate join results is $O\left(N \cdot \sqrt{\text{OUT}}\right)$, as there are N tuples in R_k and each of them can be joined with at most $\sqrt{\tilde{\text{OUT}}}$ tuples in S_{k-1}. Hence, this step takes $O\left(N \cdot \sqrt{\text{OUT}}\right)$ time. Finally, as each sub-query produces at most OUT results, and there are $O(1)$ subqueries, the aggregation step takes $O(\text{OUT})$ time. Putting everything together, we obtain:

Theorem 4.1. *For Q_{line} and an arbitrary instance $\mathcal{R}$ of input size N and output size* OUT*, the query result $Q(\mathcal{R})$ can be computed in $O\left(N \cdot \sqrt{\text{OUT}}\right)$ time.*

5 ACYCLIC QUERIES

After obtaining some high-level ideas behind our hybrid strategy, we are now ready to move to general acyclic queries. But, there are several challenging questions in front of us:

- *How to relate free-connex fractional hypertree width to the structure of an acyclic query?*
- *What optimal condition is required to run the Yannakakis algorithm within the targeted time?*
- *How to partition the input instance into a set of sub-instances satisfying the optimal condition?*

We will answer these questions step by step.

5.1 Outline of Our Algorithm

The procedural definition of free-connex fractional hypertree width in Corollary 3.5 essentially outlines our algorithm. As described in Algorithm 3, given an acyclic query Q and an instance $\mathcal{R}$ of input size N and output size OUT, we first remove dangling tuples (line 1), decompose Q into a set of connected subqueries (line 2-3), compute the results of each $\exists$-connected subquery separately (line 4-6) and combine their results via join (line 7).

From now on, we focus exclusively on $\exists$-connected queries. Given an arbitrary subquery Q and an instance $\mathcal{R}$ of input size N, if Q is not cleansed, we apply Algorithm 4 to obtain a cleanse version

Algorithm 5: SEPARATE$(Q = (\mathcal{V}, \mathcal{E}, \mathbf{y}), \mathcal{R})$

1 $\mathcal{E}_\circ \leftarrow$ a set of fn-fhtw relations with $\mathbf{y} \subseteq \bigcup_{e \in \mathcal{E}'} e$;
2 $\kappa \leftarrow$ assign each output attribute $A \in \mathbf{y}$ to an arbitrary relation $e \in \mathcal{E}'$ with $A \in e$;
3 **foreach** $A \in \mathbf{y}$ *with* $|\mathcal{E}_A| > 1$ **do**
4 $\quad x_A \leftarrow$ an attribute not appearing $\mathcal{V}$;
5 $\quad \mathcal{V} \leftarrow \mathcal{V} \cup \{x_A\}$, $\kappa(A) \leftarrow \kappa(A) \cup \{x_A\}$, $\mathbf{y} \leftarrow \mathbf{y} \cup \{x_A\} - \{A\}$;
6 $\quad$ **foreach** *tuple* $t \in R_{\kappa(A)}$ **do** extend t with value $\pi_A t$ in attribute x_A;
7 **foreach** $e \in \mathcal{E}_\circ$ *s.t.* $\nexists e' \in \mathcal{E}$ *with* $e - \mathbf{y} \subseteq e'$ **do**
8 $\quad x_e \leftarrow$ an attribute not appearing in $\mathcal{V}$;
9 $\quad e'' \leftarrow \{x_e\} \cup (e \cap \mathbf{y})$;
10 $\quad \mathcal{V} \leftarrow \mathcal{V} \cup \{x_e\}$, $\mathbf{y} \leftarrow \mathbf{y} \cup \{x_e\} - (e \cap \mathbf{y})$, $\mathcal{E} \leftarrow \mathcal{E} \cup \{e''\}$;
11 $\quad R_{e''} \leftarrow \emptyset$;
12 $\quad$ **foreach** $t \in \pi_{e \cap e''} R_e$ **do**
13 $\quad\quad t' \leftarrow$ a tuple over attributes e'' with $\pi_{e \cap e''} t' = \pi_{x_e} t' = t$ and $w(t') = \mathbf{1}$;
14 $\quad\quad R_{e''} \leftarrow R_{e''} \cup \{t'\}$;
15 $\quad \mathcal{R} \leftarrow \mathcal{R} \cup \{R_{e''}\}$;
16 **return** Updated Q and $\mathcal{R}$;

Q' and an updated instance $\mathcal{R}'$ of input size $O(N)$ such that $Q(\mathcal{R}) = Q'(\mathcal{R}')$. Algorithm 4 iteratively applies the following two steps: (lines 2-3) removes a unique non-output attribute $B \in \mathcal{V} - \mathbf{y}$ (suppose $\mathcal{E}_B = \{e\}$) and aggregate R_e over B; or (lines 4-5) removes relation $e \in \mathcal{E}$ if there exists another relation $e' \in \mathcal{E}$ with $e \subseteq e'$ and update $R_{e'}$ by $R_{e'} \bowtie R_e$, i.e., update the annotation of each tuple $t \in R_{e'}$ by $w(t) \otimes w(\pi_e t)$. This step only takes $O(N)$ time.

THEOREM 5.1. *Any $\exists$-connected query Q and an instance $\mathcal{R}$ of input size N, can be transformed into a cleansed query Q' and an instance $\mathcal{R}'$ of input size $O(N)$ within $O(N)$ time, such that $Q(\mathcal{R}) = Q'(\mathcal{R}')$.*

Next, we narrow our scope to $\exists$-connected and cleansed queries. Given an arbitrary query Q and an instance $\mathcal{R}$, if Q is not separated, we apply the Algorithm 5 to obtain a separated version Q' and an updated instance $\mathcal{R}'$ of input size $O(N)$ such that $Q(\mathcal{R}) = Q'(\mathcal{R}')$ and $\mathsf{fn\text{-}fhtw}(Q) = \mathsf{fn\text{-}fhtw}(Q')$. Algorithm 5 begins by handling joint output attributes (lines 1-6). As shown in Lemma 5.2, it is always feasible to find a subset $\mathcal{E}_\circ \subseteq \mathcal{E}$ of fn-fhtw relations that contain all output attributes (line 1), and to assign each output attribute $A \in \mathbf{y}$ to an arbitrary relation $e \in \mathcal{E}_\circ$ with $A \in e$. Let κ be such an *assignment* (line 2). For each joint output attribute $A \in \mathbf{y}$, we introduce a unique output attribute x_A to relation $\kappa(A)$ and convert A into a non-output attribute (line 5). To preserve the equivalence of query results, we force a one-to-one mapping between $\mathrm{dom}(A)$ and $\mathrm{dom}(x_A)$ (line 6). The annotations of all tuples remain unchanged. By applying this procedure to every joint output attribute, we ultimately obtain a query in which the set of unique attributes is exactly the set of output attributes. Algorithm 5 then examines every relation $e \in \mathcal{E}_\circ$ with $e \cap \mathbf{y} \neq \emptyset$, for which there is no other relation $e' \in \mathcal{E}$ such that $e - \mathbf{y} \subseteq e'$ (lines 7-15). For each such relation, we add another relation e'' that includes a unique output attribute x_e along with all output attributes present in e, and then convert all output attributes in e into non-output attributes (line 10). To preserve the equivalence of the query results, we force a one-to-one mapping between $\mathrm{dom}(e \cap e'')$ and $\mathrm{dom}(x_e)$, and set the annotation of each tuple in $R_{e''}$ as $\mathbf{1}$ (lines 12-14).

LEMMA 5.2. *There exists a subset $\mathcal{E}_\circ \subseteq \mathcal{E}$ of* fn-fhtw *relations such that $\mathbf{y} \subseteq \bigcup_{e \in \mathcal{E}_\circ} e$.*

Proof of Lemma 5.2. As Q is acyclic, $q[\mathbf{y}]$ is also acyclic. Every acyclic query has an optimal fractional edge covering ρ^* that is also integral [26], i.e., $\rho^*(e) = 1$ or $\rho^*(e) = 0$ for any $e \in \mathcal{E}$. Let $\mathcal{E}_\circ \subseteq \mathcal{E}$ be the set of relations for which $\rho^*(e) = 1$ holds for every $e \in \mathcal{E}_\circ$. For every attribute $A \in \mathbf{y}$, there must exist a relation $e \in \mathcal{E}_\circ$ such that $A \in e$. Implied by Corollary 3.5, $|\mathcal{E}_\circ| = \text{fn-fhtw}$. □

Example 5.3. We continue with Example 3.8 in which Q is not separated. Let $\mathcal{E}_\circ = \{e_1, e_2, e_3, e_6\}$ be the chosen subset of relations in line 1. Recall that $\mathbf{y} = \{A_1, A_2, A_3, C_2\}$. There is only one way to assign: A_1 to e_1, A_2 to e_2, A_3 to e_3, and C_2 to e_6. As C_2 is not unique, we introduce a unique output attribute B_4 to e_6 and convert C_2 into a non-output attribute. As no relation contains all non-output attributes of e_6, we add $e_4 = \{A_4, B_4\}$ with a unique output attribute A_4 to $\mathcal{E}$ and convert B_4 into a non-output attribute. The resulting query $(\mathcal{V}', \mathcal{E}', \mathbf{y}')$ is separated, where $\mathcal{V}' = \{A_1, A_2, A_3, A_4, B_1, B_2, B_3, B_4, C_1, C_2\}$, $\mathcal{E}' = \{e_1 = \{A_1, B_1\}, e_2 = \{A_2, B_2\}, e_3 = \{A_3, B_3\}, e_4 = \{A_4, B_4\}, e_5 = \{B_1, B_2, C_1, C_2\}, e_6 = \{B_3, B_4, C_1, C_2\}\}$ and $\mathbf{y}' = \{A_1, A_2, A_3, A_4\}$.

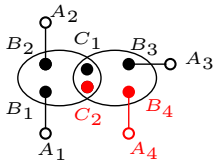

Fig. 8. An illustration of the Separate procedure.

Theorem 5.4. *Any ∃-connected and cleansed acyclic query Q and an instance $\mathcal{R}$ of input size N, can be transformed into a separated acyclic query Q' and an instance $\mathcal{R}'$ of input size $O(N)$ within $O(N)$ time, such that* fn-fhtw(Q) = fn-fhtw(Q') *and* $Q(\mathcal{R}) = Q'(\mathcal{R}')$.

Thanks to these helper procedures, we can ultimately focus on separated acyclic queries at last, which constitute the most technical part of this section. We delve into the structural properties of separated acyclic queries in Section 5.2 and present our output-optimal algorithm in Section 5.3. The main result achieved is summarised in Theorem 5.4.

Theorem 5.5. *For any separated acyclic query Q, and an instance $\mathcal{R}$ of input size N and output size* OUT, *the query result $Q(\mathcal{R})$ can be computed in $O\left(N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}(Q)}}\right)$ time.*

We now briefly analyze the complexity of Algorithm 3. Both Cleanse and Separate procedures take $O(N)$ time. From Theorem 5.5, each invocation of the HybridYannakakis procedure takes $O\left(N \cdot \left|Q_i'\left(\mathcal{R}_i'\right)\right|^{1-\frac{1}{\text{fn-fhtw}(Q_i')}}\right)$ time. Implied by Lemma 3.2 and Theorem 5.4, fn-fhtw $\left(Q_i'\right)$ = fn-fhtw$(Q_i) \leqslant$ fn-fhtw(Q). Moreover, when there are no dangling tuples, $\left|Q_i'\left(\mathcal{R}_i'\right)\right| \leqslant |Q(\mathcal{R})|$. Putting everything together, we obtain:

Theorem 5.6. *For any acyclic query Q, and an instance $\mathcal{R}$ of input size N and output size* OUT, *the query result $Q(\mathcal{R})$ can be computed in $O\left(N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}(Q)}} + \text{OUT}\right)$ time, where* fn-fhtw(Q) *is the free-connex fractional hypertree width of Q.*

5.2 Structural Properties of Separated Acyclic Queries

In the remainder of this section, we focus on separated acyclic queries. We discover a nice structural property of separated acyclic queries regarding free-connex fractional hypertree width. Recall that in a separated query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$, every output attribute is unique, every unique attribute is an output attribute, and for each relation $e \in \mathcal{E}$ with $e \cap \mathbf{y} \neq \emptyset$, there exists some other relation $e' \in \mathcal{E} - \{e\}$ with $e - \mathbf{y} \subseteq e'$. In an arbitrary TD $(\mathcal{T}, \chi)$, a node is said to be incident to another node if an edge exists between them. For a node e, let $\mathcal{N}_e$ denote the set of nodes incident to it. A node e is a *leaf* if it is only incident to another node, i.e., $|\mathcal{N}_e| = 1$.

Lemma 5.7. *Any separated acyclic query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$ has a width-1* TD $(\mathcal{T}, \chi)$ *such that there is a one-to-one correspondence between the set of nodes in $\mathcal{T}$ and the set of relations containing output attribute(s) in $\mathcal{T}$. $(\mathcal{T}, \chi)$ is called a* separated TD *for Q.*

Proof of Lemma 5.7. Let $\mathcal{E}_\bullet = \{e \in \mathcal{E} : e \cap \mathbf{y} = \emptyset\}$ be the set of relations without non-output attributes. Implied by the GYO reduction [2], the join query $(\mathcal{V} - \mathbf{y}, \mathcal{E}_\bullet)$ derived by relations in $\mathcal{E}_\bullet$ is also acyclic since all (unique) output attributes can be removed first and then all relations containing output attributes. Let $(\mathcal{T}_\bullet, \chi_\bullet)$ a width-1 TD for $(\mathcal{V} - \mathbf{y}, \mathcal{E}_\bullet)$. Note that there is a one-to-one correspondence between nodes in $\mathcal{T}_\bullet$ and relations in $\mathcal{E}_\bullet$. For each $e \in \mathcal{E} - \mathcal{E}_\bullet$, we identify an arbitrary $e' \in \mathcal{E} - \{e\}$ such that $e - \mathbf{y} \subseteq e'$, and add e as a child node of e'. It can be checked that the resulting TD $(\mathcal{T}, \chi)$ is a valid width-1 TD for $\mathcal{Q}$, and each relation containing an output attribute is a leaf node. It remains to show that each leaf node of $\mathcal{T}$ also contains (unique) output attributes. By contradiction, we assume that there exists some leaf node e of $\mathcal{T}$ such that $e \in \mathcal{E}_\bullet$. Let e' be the unique node incident to e. As $\mathcal{Q}$ is cleansed, $e - e' \neq \emptyset$. Together with $e \cap \mathbf{y} = \emptyset$, e must contain some unique non-output attribute, leading to a contradiction. □

See an example in Figure 9. From Lemma 5.7, the number of leaf nodes in $\mathcal{T}$ is exactly fn-fhtw($\mathcal{Q}$). For a pair of incident nodes e_1, e_2, we use $\{e_1, e_2\}$ to denote the undirected edge between them, use (e_1, e_2) (resp. (e_2, e_1)) to denote the directed edge from e_1 to e_2 (resp. from e_2 to e_1). Removing edge $\{e_1, e_2\}$ separates $\mathcal{T}$ into two connected subtrees $\mathcal{T}_{e_1,e_2}$ and $\mathcal{T}_{e_2,e_1}$, which contains e_1 and e_2 separately. Let $\mathcal{L}_{e_1,e_2}$ be the set of leaf nodes in $\mathcal{T}_{e_1,e_2}$. For simplicity, we use $\phi_{e_1,e_2} = \frac{|\mathcal{L}_{e_1,e_2}|}{\text{fn-fhtw}(\mathcal{Q})}$ to measure the *fraction* of the number of nodes containing output attributes, or equivalently the number of leaf nodes in $\mathcal{T}_{e_1,e_2}$. These parameters will be frequently used for partitioning the input instance in the next subsection. For each edge (e_1, e_2), the subtree $\mathcal{T}_{e_1,e_2}$ derives a sub-query $\mathcal{Q}_{e_1,e_2} := \bigoplus_{\mathcal{V}-\mathbf{y}-(e_1\cap e_2)} \bowtie_{u\in\text{nodes}(\mathcal{T}_{e_1,e_2})} R_e$, i.e., aggregates over all non-output attributes except the join attributes between e_1 and e_2. Note that these join attributes will appear in some nodes of $\mathcal{T} - \mathcal{T}_{e_1,e_2}$, and should be kept in the subsequent computation.

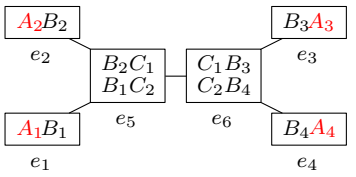

Fig. 9. A separated width-1 TD for query in Figure 8. e_1, e_2, e_3 and e_4 are leaf nodes. $\mathcal{E}_\bullet = \{e_5, e_6\}$. All (unique) output attributes A_1, A_2, A_3, A_4 are highlighted in red. Leaf nodes e_1, e_2, e_3, e_4 correspond to relations e_1, e_2, e_3, e_4 . Removing edge $\{e_5, e_6\}$ leads to two sub-trees, where $\mathcal{T}_{e_5,e_6}$ contains nodes e_1, e_2, e_5 and $\mathcal{T}_{e_6,e_5}$ contains nodes e_3, e_4, e_6. $\mathcal{Q}_{e_5,e_6} = \bigoplus_{B_1,B_2} R_1(A_1, B_1) \bowtie R_2(A_2, B_2) \bowtie R_5(B_1, B_2, C_1, C_2)$. And, $\phi(e_5, e_6) = \phi(e_6, e_5) = \frac{1}{2}$.

5.3 Our Algorithm for Separated Acyclic Queries

Given an arbitrary separated acyclic query, we next characterize an *optimal condition* on the input instances for which the runtime of the Yannakakis algorithm can be bounded as stated in Theorem 5.5. In other words, the Yannakakis algorithm is already output-optimal in these cases. We then explore how to partition an arbitrary input instance into a collection of sub-instances, each of which satisfies this optimal condition individually. Likewise, we assume that a parameter $\tilde{\text{OUT}}$ is known such that $\tilde{\text{OUT}} \leqslant \text{OUT} \leqslant c \cdot \tilde{\text{OUT}}$ for some constant c. This assumption can be removed without increasing the complexity asymptotically [27]. Finally, we introduce the notion of *edge label*, which encapsulates both data statistics and query structure.

Definition 5.8 (Edge Label). For a separated acyclic query $\mathcal{Q}$ and a separated width-1 TD $(\mathcal{T}, \chi)$, an instance $\mathcal{R}$ and parameter $\tilde{\text{OUT}}$, an edge (e_1, e_2) in $\mathcal{T}$ is

- *large* if $|\mathcal{Q}_{e_1,e_2}(\mathcal{R}) \ltimes t| > \tilde{\text{OUT}}^{\phi_{e_1,e_2}}$ holds for every tuple $t \in \pi_{e_1\cap e_2} R_{e_1}$; and
- *small* if $|\mathcal{Q}_{e_1,e_2}(\mathcal{R}) \ltimes t| \leqslant \tilde{\text{OUT}}^{\phi_{e_1,e_2}}$ holds for every tuple $t \in \pi_{e_1\cap e_2} R_{e_1}$; and
- *unlabeled* otherwise.

Furthermore, a small edge (e_1, e_2) is *limited* if $\left|\bigoplus_{e_1\cap e_2} \mathcal{Q}_{e_1,e_2}(\mathcal{R})\right| \leqslant \tilde{\text{OUT}}^{\phi_{e_1,e_2}}$.

5.3.1 Optimal Condition of Yannakakis Algorithm

Now, we can see a natural connection between the Yannakakis algorithm and edge labels:

Lemma 5.9. *For a separated acyclic query Q with a separated width-1* TD *$(\mathcal{T}, \chi)$, and an instance $\mathcal{R}$ of input size N and output size* OUT*, if edge (e, e') is small, the intermediate join result $R_{e'} \bowtie Q_{e,e'}(\mathcal{R})$ can be computed in $O\left(N \cdot \text{OUT}^{\phi_{e,e'}}\right)$ time.*

Proof of Lemma 5.9. Consider an arbitrary node $e_1 \in \text{nodes}(\mathcal{T}_{e,e'})$. Let p_{e_1} be the parent node of e_1. The intermediate join result materialized at e_1 is $R_{e_1} \bowtie (\bowtie_{e_2} Q_{e_2,e_1})$, where e_2 is over all child nodes of e_1. Without dangling tuples, every tuple $t \in R_{e_1}$ participates in at most $\tilde{\text{OUT}}^{\phi_{e,e'}}$ query result in $Q_{e_1,p_{e_1}}$; otherwise, there must exist some tuple $t \in \text{dom}(e \cap e')$ that can participate in more than $\tilde{\text{OUT}}^{\phi_{e,e'}}$ query result in $Q_{e,e'}$, contradicting the fact that edge (e, e') is small. As there are at most N tuples in R_{e_1}, the number of intermediate join results materialized at e_1 is at most $O\left(N \cdot \text{OUT}^{\phi_{e,e'}}\right)$. When we move to e', the size of $R_{e'} \bowtie Q_{e,e'}(\mathcal{R})$ can be bounded similarly. □

Recall that $\phi_{e,e'} = 1 - \frac{1}{\text{fn-fhtw}}$ if e' is a leaf node. Hence, we come to the optimal condition below:

Corollary 5.10 (Optimal Condition for Yannakakis). *For a separated acyclic query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$ with a separated width-1* TD *$(\mathcal{T}, \chi)$, and an instance $\mathcal{R}$ of input size N and output size* OUT*, if there is a leaf node e' of $\mathcal{T}$ such that edge (e, e') is small for the only node e' incident to e, then $Q(\mathcal{R})$ can be computed in $O\left(N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}(Q)}}\right)$ time.*

5.3.2 Partition Separated Acyclic Queries

The input instance does not necessarily meet the optimal condition above, so we cannot apply the Yannakakis algorithm directly. Even worse, it is unknown how to efficiently *decide* the label of each edge since the definition of edge labeling is based on the sub-query, which may be too expensive to compute. Consequently, the most technically challenging step is to identify an efficient ordering for labeling the edges and to iteratively isolate a sub-instance once it meets the optimal condition specified in Corollary 5.10.

As described in Algorithm 6, let $(\mathcal{T}, \chi)$ be a separated width-1 TD for Q (line 1), with all edges unlabeled initially. Again, there is a one-to-one correspondence between relations in $\mathcal{E}$ and nodes in $\mathcal{T}$. Hence, we also use e to denote the node in $\mathcal{T}$ that corresponds to relation $e \in \mathcal{E}$. We put $(\mathcal{T}, \mathcal{R})$ into a candidate set $\mathcal{P}$ of instances to be partitioned (line 2). In general, consider an arbitrary pair $(\mathcal{T}', \mathcal{R}') \in \mathcal{P}$. From Lemma 5.11, we apply *limited-imply-limited* rule to infer edge labels (lines 5-6). If it meets the optimal condition, we compute the result immediately (lines 7-8) and remove it from $\mathcal{P}$ (line 16). Otherwise, we further partition it (lines 9-15). More specially, we pick a non-labeled edge (e_1, e_2) such that every other incoming edge to e_1 is small, i.e., edge (e_3, t_1) is small for each node $e_3 \in \mathcal{N}_{e_1} - \{e_2\}$ (line 10). We compute $Q_{e_1,e_2}(\mathcal{R}')$ using the Yannakakis algorithm along $\mathcal{T}_{e_1,e_2}$ rooted at e_1 (line 11). A tuple $t \in \text{dom}(e_1 \cap e_2)$ is *heavy* if $|Q_{e_1,e_2}(\mathcal{R}') \ltimes t| > \tilde{\text{OUT}}^{\phi_{e_1,e_2}}$, and *light* otherwise. Now, we construct two sub-instances for $\mathcal{R}'$, which contain heavy and light tuples in R_{e_1} separately (line 15), and two copies of $\mathcal{T}'$ in which edge (e_1, e_2) is further labeled as large and small separately (line 13-14). By Lemma 5.12, we can apply *large-reverse-limited* rule to infer (e_2, e_1) as limited when (e_1, e_2) is labeled as large (line 14). We add these two sub-instances into $\mathcal{P}$ (line 15) and also remove $(\mathcal{T}', \mathcal{R}')$ from $\mathcal{P}$ (line 16). We continue applying this procedure to every remaining instance in $\mathcal{P}$ until $\mathcal{P}$ becomes empty (line 3).

Lemma 5.11 (Limited-Imply-Limited). *For any node e_1, if there exists a node $e_2 \in \mathcal{N}_{e_1}$ such that edge (e_3, e_1) is limited for every node $e_3 \in \mathcal{N}_{e_1} - \{e_2\}$, then edge (e_1, e_2) must be limited.*

Algorithm 6: HybridYannakakis $(Q = (\mathcal{V}, \mathcal{E}, \mathbf{y}), \mathcal{R})$

1 $(\mathcal{T}, \chi) \leftarrow$ a separated width-1 TD of Q with all edges of $\mathcal{T}$ non-labeled;
2 $\mathcal{P} \leftarrow \{(\mathcal{T}, \mathcal{R}\}, \mathcal{S} \leftarrow \mathbf{0}$;
3 **while** $\mathcal{P} \neq \varnothing$ **do**
4 $\quad (\mathcal{T}', \mathcal{R}') \leftarrow$ an arbitrary pair in $\mathcal{P}$;
5 $\quad$ **while** $\exists$ *non-labeled* (e_1, e_2) *in* $\mathcal{T}'$ *such that* (e_3, e_1) *is limited for each* $e_3 \in \mathcal{N}_{e_1} - \{e_2\}$ **do**
6 $\quad\quad$ Label edge (e_1, e_2) as limited;
7 $\quad$ **if** *there is a leaf node* e_1 *with small (or limited) edge* $(*, e_1)$ *in* $\mathcal{T}'$ **then**
8 $\quad\quad \mathcal{S} \leftarrow \mathcal{S} \oplus Q(\mathcal{R}')$ for computing $Q(\mathcal{R}')$ via Yannakakis algorithm on $\mathcal{T}$ rooted at e_1;
9 $\quad$ **else**
10 $\quad\quad (e_1, e_2) \leftarrow$ a non-labeled edge in $\mathcal{T}'$ s.t. (e_3, e_1) is small for each $e_3 \in \mathcal{N}_{e_1} - \{e_2\}$;
11 $\quad\quad$ Compute $Q_{e_1,e_2}(\mathcal{R}')$ by the Yannakakis algorithm on $\mathcal{T}_{e_1,e_2}$ rooted at e_1;
12 $\quad\quad \mathcal{H} \leftarrow \left\{ t \in \mathrm{dom}(e_1 \cap e_2) : |\sigma_{e_1 \cap e_2 = t} Q_{e_1,e_2}(\mathcal{R}')| > \tilde{\mathrm{OUT}}^{\phi_{e_1,e_2}} \right\}$;
13 $\quad\quad \mathcal{T}_1' \leftarrow \mathcal{T}'$ with $(e_1, e_2), (e_2, e_1)$ labeled as large, limited separately;
14 $\quad\quad \mathcal{T}_2' \leftarrow \mathcal{T}'$ with (e_1, e_2) labeled as small;
15 $\quad\quad \mathcal{P} \leftarrow \mathcal{P} \cup \left\{ \left(\mathcal{T}_1', \mathcal{R}' - \{R_{e_1}\} \cup \{R_{e_1} \ltimes \mathcal{H}\}\right), \left(\mathcal{T}_2', \mathcal{R}' - \{R_{e_1}\} \cup \{R_{e_1} \rhd \mathcal{H}\}\right) \right\}$;
16 $\quad \mathcal{P} \leftarrow \mathcal{P} - \{(\mathcal{T}', \mathcal{R}')\}$;
17 **return** $\mathcal{S}$;

Proof of Lemma 5.11. For Q_{e_1,e_2}, we observe the following:

$$\left| \bigoplus_{\mathcal{V}-\mathbf{y}} Q_{e_1,e_2} \right| \leqslant \prod_{e_3 \in \mathcal{N}_{e_1} - \{e_2\}} \left| \bigoplus_{\mathcal{V}-\mathbf{y}} Q_{e_3,e_1} \right| \leqslant \prod_{e_3 \in \mathcal{N}_{e_1} - \{e_2\}} \mathrm{OUT}^{\phi_{e_3,e_1}} \leqslant \mathrm{OUT}^{\phi_{e_1,e_2}}$$

where the first inequality follows that $\bigcup_{e \in \mathcal{T}_{e_1,e_2}} (e \cap \mathbf{y}) = \bigcup_{e_3 \in \mathcal{N}_{e_1} - \{e_2\}} \bigcup_{e \in \mathcal{T}_{e_1,e_3}} (e \cap \mathbf{y})$ and the second inequality follows that $\sum_{e_3 \in \mathcal{N}_{e_1} - \{e_2\}} \phi_{e_3,e_1} = \phi_{e_1,e_2}$. By definition, (e_1, e_2) must be limited. □

Lemma 5.12 (Large-Reverse-Limited). *If edge* (e_1, e_2) *is large, then edge* (e_2, e_1) *must be limited.*

Proof of Lemma 5.12. Consider an arbitrary tuple $t \in \mathrm{dom}(e_1 \cap e_2)$. As (e_1, e_2) is large, t can be joined with at least $\mathrm{OUT}^{\phi_{e_1,e_2}}$ query result of Q_{e_1,e_2}. Without dangling tuples, every tuple $t' \in \bigoplus_{\mathcal{V}-\mathbf{y}} Q_{e_2,e_1}$ appears together with at least $\mathrm{OUT}^{\phi_{e_1,e_2}}$ tuples from $\bigoplus_{\mathcal{V}-\mathbf{y}} Q_{e_1,e_2}$ in the final query result. This way, $|\bigoplus_{\mathcal{V}-\mathbf{y}} Q_{e_2,e_1}| \leqslant \mathrm{OUT}^{1-\phi_{e_1,e_2}} = \mathrm{OUT}^{\phi_{e_2,e_1}}$. Hence, (e_2, e_1) is limited. □

Example 5.13. We continue the example in Figure 9. Initially, all edges are unlabeled in (9.1). We start with applying line 13 to label edge (e_1, e_5) since $\mathcal{N}_{e_1} - \{e_5\} = \varnothing$. In (9.2), the instance with large edge (e_1, e_5) and limited edge (e_5, e_1) already meets the optimal condition. The remaining instance has a small edge (e_5, e_1). We can apply a similar argument to edges (e_2, e_5), (e_3, e_6) and (e_4, e_6). In (9.3), we are left with the remaining instance with small edges (e_1, e_5), (e_2, e_5), (e_3, e_6) and (e_4, e_6). Then, we can apply line 13 to label edge (e_5, e_6).

In (9.4), for the instance with large edge (e_5, e_6) and limited edge (e_6, e_5), we can apply line 13 to partition both edges (e_5, e_1) and (e_5, e_2). Suppose we partition edge (e_5, e_1) wlog. In (9.6), the instance with small edge (e_5, e_1) already meets the optimal condition. In (9.7), we are left with an

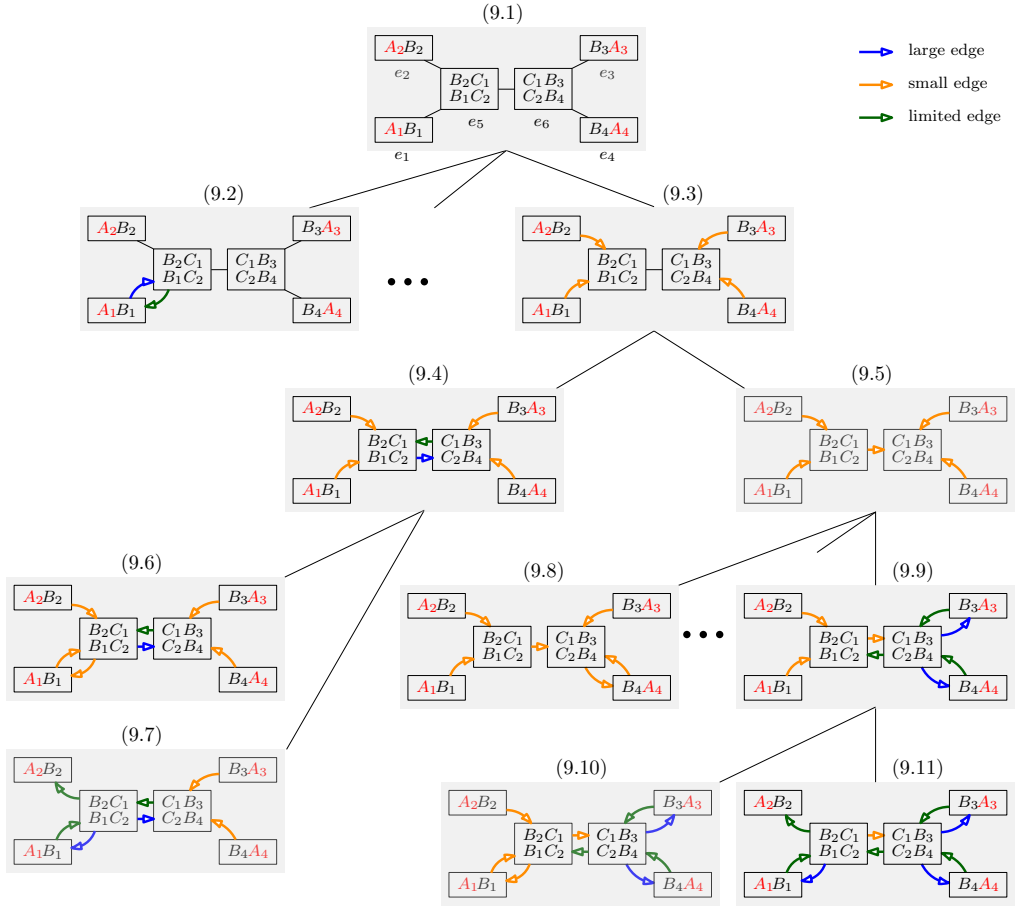

Fig. 10. An illustration of the partition procedure on the query in Figure 9.

instance with large edge (e_5, e_1) and limited edge (e_1, e_5). We can apply the limited-imply-limited rule to infer edge (e_5, e_2) as limited. This instance also meets the optimal condition.

In (9.5), for the other instance with small edge (e_5, e_6), we can apply line 13 to label both edges (e_6, e_4) and (e_6, e_5). Suppose we label edge (e_6, e_4) wlog. In (9.8), the instance with small edge (e_6, e_4) meets the optimal condition. We can apply a similar argument to edge (e_6, e_3). In (9.9), the remaining instance has large edges (e_6, e_3) and (e_6, e_4), as well as limited edges (e_3, e_6) and (e_4, e_6). We can apply the limited-imply-limited rule to infer edge (e_6, e_5) as limited. Now, we can apply line 13 to label edges (e_5, e_1) and (e_5, e_2). Suppose we label edge (e_5, e_1). In (9.10), the instance with small edge (e_5, e_1) already meets the optimal condition. In (9.11), we are left with the instance with large edge (e_3, e_1) and limited edge (e_3, e_2). Then, we can apply the limited-imply-limited rule to infer edge (e_3, e_2) as limited. This instance also meets the optimal condition. Now, $\mathcal{P} = \varnothing$, and we are done with the partition procedure.

5.3.3 Analysis

To show the correctness of Algorithm 6,we first need to prove the following result:

Lemma 5.14. *In Algorithm 6, for any $(\mathcal{T}', \mathcal{R}') \in \mathcal{P}$, either line 8, line 10, or line 13 can be applied.*

Algorithm 7: IdentifyLeaf$(Q, \mathcal{R}', \mathcal{T}')$

```
1 while T' is not a single node do
2     Root T' at an arbitrary node r such that r ∩ y ≠ ∅;
3     if ∃ node e with its parent p_e such that edge (e, p_e) is large then T' ← T' − T'_{e,p_e};
4     else return r;
5 return the single node in T';
```

Proof of Lemma 5.14. Consider an arbitrary instance $\mathcal{R}'$ with a partially labeled separated width-1 TD $(\mathcal{T}', \chi')$. By contradiction, we assume that neither line 8, line 10, nor line 13 of Algorithm 6 can be applied. We next show how to identify a leaf node e of $\mathcal{T}'$ such that $(*, e)$ is small; hence, $\mathcal{R}'$ already meets the optimal condition, leading to a contradiction.

As shown in Algorithm 7, the high-level idea is to conceptually remove nodes in $\mathcal{T}'$, until a single node is left, which is exactly the leaf node as desired. We root $\mathcal{T}'$ at an arbitrary node $r \in \text{nodes}(\mathcal{T}')$ such that $r \cap \mathbf{y} \neq \emptyset$. Let p_e be the parent of e in this rooted tree. If there exists a node e such that (e, p_e) is large, we simply remove the whole subtree $\mathcal{T}'_{e,p_e}$ and continue the process. Otherwise, every edge (e, p_e) is small, including the edge $(*, r)$ incident to r, hence r is returned.

It remains to show that a node $r \in \text{nodes}(\mathcal{T}')$ with $r \cap \mathbf{y} \neq \emptyset$ can always be found at line 2. By contradiction, we assume that such a node does not exist. In the execution process above, $\mathcal{T}'$ is always a connected subtree. For clarity, let $\mathcal{T}_1'$ be the state of $\mathcal{T}'$ at some point and let $\mathcal{T} - \mathcal{T}_1'$ be the subtree(s) removed by the process above. As $\mathcal{T}_1'$ is connected and $\mathcal{T}_1'$ does not contain any leaf nodes of $\mathcal{T}$, $\mathcal{T} - \mathcal{T}_1'$ must contain all leaf nodes and hence be disconnected. Moreover, for each pair of nodes $e_1 \in \text{nodes}(\mathcal{T}_1')$, $e_2 \in \text{nodes}(\mathcal{T} - \mathcal{T}_1')$, the subtree $\mathcal{T}'_{e_2,e_1}$ has been removed due to the fact that edge (e_1, e_2) is large at line 3. This way, Lemma 5.15 is violated on $\mathcal{T}'$, leading to a contradiction. Hence, it is always feasible to find a node $r \in \text{nodes}(\mathcal{T}')$ with $r \cap \mathbf{y} \neq \emptyset$. □

Lemma 5.15 (Not-All-Large). *Consider a connected subtree $\mathcal{T}_1$ of $\mathcal{T}$ that does not contain any leaf node of $\mathcal{T}$. For any instance $\mathcal{R}'$ with non-empty query result, there must exist a pair of nodes $e_1 \in \text{nodes}(\mathcal{T}_1), e_2 \in \text{nodes}(\mathcal{T} - \mathcal{T}_1)$ such that (e_1, e_2) is an edge in $\mathcal{T}$ but not large.*

Proof of Lemma 5.15. Let U_1 be the set of nodes in $\mathcal{T}_1$ that are incident to some nodes in $\mathcal{T}_2$. Let U_2 be the set of nodes in $\mathcal{T} - \mathcal{T}_1$ that are incident to some nodes in $\mathcal{T}_1$. Note that there is a one-to-one correspondence between U_1 and U_2. Let $e_2 \in U_2$ be the corresponding node for $e_1 \in U_1$, i.e., (e_1, e_2) is an edge. By contradiction, suppose every such edge (e_1, e_2) is large for every $e_1 \in U_1$. Implied by Lemma 5.12, every such edge (e_2, e_1) is limited. Consider an arbitrary node $e_1^* \in U_1$ with its corresponding node $e_2^* \in U_2$. We note that for every tuple in $t \in \text{dom}(e_1^* \cap e_2^*)$,

$$\left|\sigma_{e_1^* \cap e_2^* = t} Q_{e_1^*, e_2^*}(\mathcal{R})\right| \leqslant \prod_{e_1 \in U_1} \left|\bigoplus_{\mathcal{V}-\mathbf{y}} Q_{e_2,e_1}(\mathcal{R})\right| \leqslant \prod_{e_1 \in U_1 - \{e_1^*\}} \tilde{\text{OUT}}^{\phi_{e_2,e_1}} \leqslant \tilde{\text{OUT}}^{\phi_{e_1^*,e_2^*}}$$

where the last inequality is implied by the fact that $\sum_{e_1 \in U_1 - \{e_1^*\}} |\mathcal{L}_{e_2,e_1}| = \left|\mathcal{L}_{e_1^*,e_2^*}\right|$. Hence, edge (e_1^*, e_2^*) is not large, leading to a contradiction. □

Lemma 5.16. *Algorithm 6 will terminate after running the while-loop at most $O(2^{|\mathcal{E}|})$ iterations.*

Proof of Lemma 5.16. As each edge will be labeled once, and labeling each edge can lead to at most 2 sub-instances, the total number of sub-instances in $\mathcal{P}$ is $O(2^{|\mathcal{E}|})$, where the number of edges

in $\mathcal{T}$ is bounded by the number of relations in $\mathcal{Q}$. Each iteration removes at least one sub-instance from $\mathcal{P}$. Moreover, when a sub-instance is removed, it won't be added back to $\mathcal{P}$. Hence, $\mathcal{P}$ will become $\varnothing$ after at most $O(2^{|\mathcal{E}|})$ iterations. □

Next, we analyze the runtime of Algorithm 6. Let's first focus on line 14. For any node $e_3 \in \mathcal{N}_{e_1} - \{e_2\}$, as edge (e_3, e_1) is small, the query result $Q_{e_3,e_1}(\mathcal{R}')$ can be computed in $O\left(N \cdot \text{OUT}^{\phi_{e_3,e_1}}\right)$ time following Lemma 5.10. To compute $Q_{e_1,e_2}(\mathcal{R}')$, the Yannakakis algorithm needs to materialize the following intermediate join result: $R_{e_1} \bowtie \left(\bowtie_{e_3 \in \mathcal{N}_{e_1} - \{e_2\}} Q_{e_3,e_1}(\mathcal{R}')\right)$. As there are N tuples in R_{e_1}, and each tuple in R_{e_1} can be joined with at most $\prod_{e_3 \in \mathcal{N}_{e_1} - \{e_2\}} \tilde{\text{OUT}}^{\phi_{e_3,e_1}} = \tilde{\text{OUT}}^{\phi_{e_1,e_2}}$ tuples in this intermediate result, its size can be bounded by $O\left(N \cdot \text{OUT}^{\phi_{e_1,e_2}}\right)$. The cost of line 14 is bounded by Lemma 5.10. The cost of lines 15-18 is bounded by $O(N)$. All other lines take $O(1)$ time. As analyzed above, the number of the while-loop iterations is $O(2^{|\mathcal{E}|})$, still a constant. So, the partitioning procedure takes $O\left(N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}(\mathcal{Q})}}\right)$ time. Putting everything together, we obtain:

Lemma 5.17. *For a separated acyclic query $\mathcal{Q}$ with a separated width-1* TD *$(\mathcal{T}, \chi)$, an arbitrary instance $\mathcal{R}$ can be partitioned into $O(1)$ sub-instances $\mathcal{R}_1, \mathcal{R}_2, \cdots, \mathcal{R}_h$ within $O\left(N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}(\mathcal{Q})}}\right)$ time such that $\mathcal{Q}(\mathcal{R}) = \bigoplus_{i\in[h]} \mathcal{Q}(\mathcal{R}_i)$, and each $\mathcal{R}_i$ satisfies Corollary 5.10 together with $\mathcal{T}$.*

Combining Lemma 5.17 and Corollary 5.10, we complete the proof of Theorem 5.5.

6 CYCLIC QUERIES

In this section, we turn to cyclic queries. As mentioned, cyclic queries are usually tackled with tree decomposition techniques. Similarly, we can derive output-sensitive algorithms for cyclic queries by replacing the Yannakakis algorithm with our new output-optimal algorithm.

For an arbitrary query $\mathcal{Q} = (\mathcal{V}, \mathcal{E}, \mathbf{y})$ and a TD $(\mathcal{T}, \chi)$, we define another query $\mathcal{Q}_\mathcal{T} = (\mathcal{V}, \{\chi(u) : u \in \text{nodes}(\mathcal{T})\}, \mathbf{y})$, i.e., each node in the TD defines a relation. Given each instance $\mathcal{R}$ for $\mathcal{Q}$, we define another instance $\mathcal{R}_\mathcal{T}$ for $\mathcal{Q}_\mathcal{T}$ as follows. For each node $u \in \text{nodes}(\mathcal{T})$, we get a relation $R_u := \bowtie_{e\in\mathcal{E}} \pi_{\chi(u)\cap e} R_e$ as the result of the corresponding sub-query $q[\chi(u)]$. For each relation $e \in \mathcal{E}$, we assign it to one specific node $u \in \text{nodes}(\mathcal{T})$ such that $e \subseteq \chi(u)$. The annotations of tuples in R_e are defined as follows: if no relation is assigned to u, every tuple t has its annotation $w(t) = \mathbf{1}$; otherwise, every tuple t has its annotation $w(t) = \bigotimes_e w(\pi_e t)$ over all relations e assigned to u. After getting the induced query $\mathcal{Q}_\mathcal{T}$ and instance $\mathcal{R}_\mathcal{T}$, we simply invoke our new algorithm for acyclic queries in Section 5 to compute the query result $\mathcal{Q}_\mathcal{T}(\mathcal{R}_\mathcal{T})$, which is essentially $\mathcal{Q}(\mathcal{R})$.

Note that the input size of $\mathcal{R}_\mathcal{T}$ is $O\left(N^{\text{width}(\mathcal{T},\chi)}\right)$, and the output size is OUT. Plugging into Theorem 5.6, we obtain the time complexity as $O\left(N^{\text{width}(\mathcal{T},\chi)} \cdot \max\left\{1, \text{OUT}^{1-\frac{1}{\text{fn-fhtw}(\mathcal{Q}_\mathcal{T})}}\right\} + \text{OUT}\right)$. As the query size is constant, we can find within $O(1)$ time the tree decomposition that minimizes the complexity formula above. Putting everything together, we obtain:

Theorem 6.1. *For a query $\mathcal{Q} = (\mathcal{V}, \mathcal{E}, \mathbf{y})$, and an instance $\mathcal{R}$ of input size N and output size* OUT*, the query result $\mathcal{Q}(\mathcal{R})$ can be computed in $O\left(\min_{(\mathcal{T},\chi)} N^{\text{width}(\mathcal{T},\chi)} \cdot \max\left\{1, \text{OUT}^{1-\frac{1}{\text{fn-fhtw}(\mathcal{Q}_\mathcal{T})}}\right\} + \text{OUT}\right)$ time, where $(\mathcal{T}, \chi)$ is over all possible* TD*s of $\mathcal{Q}$.*

See Example 6.3. People have also exploited the power of hybrid TDs to speed up query processing in the literature. Building on the idea of [36], we first define the notion of *TD-coverage* as a set of TDs $(\mathcal{T}_1, \chi_1), (\mathcal{T}_2, \chi_2), \cdots, (\mathcal{T}_h, \chi_h)$ for $\mathcal{Q}$ such that $\mathcal{Q}(\mathcal{R}) = \bigoplus_{i\in[h]} \mathcal{Q}_{\mathcal{T}_i}(\mathcal{R}_{\mathcal{T}_i})$. Then, we can apply our new algorithm for acyclic queries to each TD $(\mathcal{T}_i, \chi_i)$ and aggregate the results over all TDs.

Theorem 6.2. *For a query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$, and an instance $\mathcal{R}$ of input size N and output size* OUT*, the query result $Q(\mathcal{R})$ can be computed in*

$$O\left(\min_{\{(\mathcal{T}_1,\chi_1),(\mathcal{T}_2,\chi_2),\cdots,(\mathcal{T}_h,\chi_h)\}} \max_{i\in[h]} N^{\mathsf{width}(\mathcal{T}_i,\chi_i)} \cdot \mathrm{OUT}^{1-\frac{1}{\mathsf{fn\text{-}fhtw}(Q_{\mathcal{T}_i})}} + \mathrm{OUT}\right)$$

time, where $\{(\mathcal{T}_1, \chi_1), (\mathcal{T}_2, \chi_2), \cdots, (\mathcal{T}_h, \chi_h)\}$ is over all possible TD-coverages of Q.

If we only consider the parameter $\mathsf{width}(\mathcal{T}_i, \chi_i)$, the minimum TD -coverage can be identified, and the corresponding cost of is captured by the #submodular width [36]. However, we need to consider both $\mathsf{width}(\mathcal{T}_i, \chi_i)$ and $\mathsf{fn\text{-}fhtw}(Q_{\mathcal{T}_i})$ to minimize the cost formula above. This question is interesting but very challenging, which we leave as future work.

Example 6.3. Consider a cyclic query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$, where $\mathcal{V} = \{A_1, A_2, A_3, B_1, B_2, B_3\}$, $\mathcal{E} = \{\{A_1, B_1\}, \{A_2, B_2\}, \{A_3, B_3\}, \{B_1, B_2\}, \{B_2, B_3\}, \{B_1, B_3\}\}$ and $\mathbf{y} = \{A_1, A_2, A_3\}$. Figure 11 shows a TD $(\mathcal{T}, \chi)$ for Q: $\chi(u_1) = \{A_1, B_1\}$, $\chi(u_2) = \{A_2, B_2\}$, $\chi(u_3) = \{A_3, B_2\}$, and $\chi(u_4) = \{B_1, B_2, B_3\}$. Moreover, $\mathsf{width}(\mathcal{T}, \chi) = \#\mathsf{subw}(Q) = \frac{3}{2}$ and $\#\mathsf{fn\text{-}subw}(Q) = \mathsf{fn\text{-}fhtw}(Q_{\mathcal{T}}) = 3$. Our new algorithm can compute it in $O\left(N^{\frac{3}{2}} \cdot \max\left\{1, \mathrm{OUT}^{\frac{2}{3}}\right\} + \mathrm{OUT}\right)$ time, strictly improving the previous result $O(N^3)$ or $O\left(N^{\frac{3}{2}} \cdot \max\{1, \mathrm{OUT}\}\right)$.

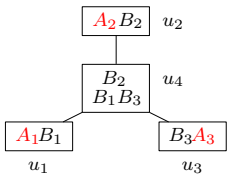

Fig. 11. A TD for a cyclic query.

7 LOWER BOUND

In this section, we will show our lower bound for semi-ring algorithms computing acyclic join-aggregate queries. Recall that semiring algorithms work with semiring elements as an abstract type and can only copy them from existing semiring elements or combine them using $\oplus$ or $\otimes$. No other operations on semi-ring elements are allowed, such as division, subtraction, or equality check.

Theorem 7.1. *For an arbitrary self-join-free acyclic query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$, given any parameters $1 \leqslant N$ and* $\mathrm{OUT} \leqslant \max_{\mathcal{R}'\in\mathfrak{R}(N)} |Q(\mathcal{R}')|$*, there exists an instance $\mathcal{R}$ of input size N and output size* OUT *such that any semi-ring algorithm computing $Q(\mathcal{R})$ requires at least $\Omega\left(N \cdot \mathrm{OUT}^{1-\frac{1}{\mathsf{fn\text{-}fhtw}(Q)}} + \mathrm{OUT}\right)$ time, where $\mathsf{fn\text{-}fhtw}(Q)$ is the free-connex fractional hypertree width of Q, and $\mathfrak{R}(N)$ is the set of all instances for Q of input size N.*

In [7], it has been shown that for the matrix query $Q_{\mathsf{matrix}} = \bigoplus_B R_1(A, B) \bowtie R_2(B, C)$, and parameters $1 \leqslant N$ and $\mathrm{OUT} \leqslant N^2$, there exists an instance $\mathcal{R}$ of input size N and output size OUT such that any semiring algorithm for computing $Q_{\mathsf{matrix}}(\mathcal{R})$ requires $\Omega\left(N \cdot \sqrt{\mathrm{OUT}}\right)$ time. A similar argument has been extended to star queries:

$$Q_{\mathsf{star}} = \bigoplus_B R_1(A_1, B) \bowtie R_2(A_2, B) \bowtie \cdots \bowtie R_k(A_k, B)$$

which will used to establish our new lower bound for general acyclic queries.

Lemma 7.2. *For Q_{star} of k relations, any parameters $1 \leqslant N$ and* $\mathrm{OUT} \leqslant N^k$*, there exists an instance $\mathcal{R}_{\mathsf{star}}$ of input size N and output size* OUT *such that any semiring algorithm for computing $Q_{\mathsf{star}}(\mathcal{R}_{\mathsf{star}})$ requires at least $\Omega\left(N \cdot \mathrm{OUT}^{1-\frac{1}{k}}\right)$ time.*

Note that Lemma 7.2 has been proved for Q_{matrix} when $k = 2$ by Pagh and Stockel [48]. Below, we simply generalized their proof to Q_{star}.

Proof of Lemma 7.2. We construct a hard instance $\mathcal{R}_{\text{star}}$ of input size N and output size OUT for Q_{star} as follows. There are $\text{OUT}^{\frac{1}{k}}$ distinct values in each attribute A_i for $i \in [k]$. There are $\frac{N}{k \cdot \text{OUT}^{1/k}}$ distinct values in attribute B. Each relation R_i is a Cartesian product between A_i and B. It can be checked that each relation contains exactly $\frac{N}{k}$ tuples, so the input size of $\mathcal{R}$ is exactly N. The query result of $Q_{\text{star}}(\mathcal{R}_{\text{star}})$ is all combinations of values in attributes $A_1, A_2, \cdots, A_k$, and the output size is exactly OUT. On this hard instance $\mathcal{R}_{\text{star}}$, for each query result $(a_1, a_2, \cdots, a_k) \in \text{dom}(A_1) \times \text{dom}(A_2) \times \cdots \times \text{dom}(A_k)$, its annotation is defined as $\bigoplus_{b \in \text{dom}(B)} \bigotimes_{i \in [k]} w(a_i, b)$, which takes at least $|\text{dom}(B)|$ operations to compute $w(a_1, a_2, \cdots, w_k, b) = \bigotimes_{i \in [k]} w(a_i, b)$ for each $b \in \text{dom}(B)$. Note that every pair of query results do not share any common operations.

Similar to [48] we require the algorithm to work over fields of infinite size such as real numbers. We consider each output value as a polynomial over nonzero entries of the input matrices. Again, by the Schwartz-Zippel theorem [41], two polynomials agree on all inputs if and only if they are identical. Since we are working in the semiring model, the only way to get the annotation for the query result $(a_1, a_2, \cdots, a_k)$ in an output polynomial is to directly multiply these input entries. That means that to compute a query result $(a_1, a_2, \cdots, a_k)$, one needs to compute a polynomial that is identical to the sum of elementary products $\bigoplus_{b \in \text{dom}(B)} \bigotimes_{i \in [k]} w(a_i, b)$. Summing over all query results, the number of operations required is at least $N \cdot \text{OUT}^{1-\frac{1}{k}}$. Hence, any semiring algorithm requires at least $\Omega\left(N \cdot \text{OUT}^{1-\frac{1}{k}}\right)$ time for computing $Q_{\text{star}}(\mathcal{R}_{\text{star}})$. □

Lemma 7.3. *For any $\exists$-connected acyclic query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$, there exists a subset $S \subseteq \mathbf{y}$ of $\mathsf{fn\text{-}fhtw}(Q)$ attributes such that no pair of them appear in the same relation from $\mathcal{E}$.*

Proof of Lemma 7.3. Note that $q[\mathbf{y}]$ is also acyclic. Initially, we set $S = \emptyset$. As shown in [26], the following greedy strategy leads to an optimal fractional edge covering for $q[\mathbf{y}]$ that is also integral. It iteratively performs the following two procedures: (i) removes a relation e if there exists another relation e' such that $e \subseteq e'$; (ii) if there exists a relation e containing some unique attribute, we remove relation e as well as all attributes in e, and add an arbitrary attribute in e to S. It can be easily checked that $|S| = \mathsf{fn\text{-}fhtw}$, and no pair of them appears in the same relation from $\mathcal{E}$. □

Proof of Theorem 1.2. First, $\Omega(N + \text{OUT})$ is a trivial lower bound. Hence, it suffices to show the lower bound $\Omega\left(N \cdot \text{OUT}^{1-\frac{1}{\mathsf{fn\text{-}fhtw}(Q)}}\right)$ when $\text{OUT} < N^{\mathsf{fn\text{-}fhtw}(Q)}$. We will prove it via three steps:

Step 1. Consider any $\exists$-connected and cleansed query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$. Suppose we are given parameters $1 \leqslant N$ and $\text{OUT} \leqslant N^{\mathsf{fn\text{-}fhtw}(Q)}$. We show a reduction from Q to Q_{star} with $\mathsf{fn\text{-}fhtw}(Q)$ relations. Implied by Lemma 7.3, let $S \subseteq \mathbf{y}$ be the set of output attributes such that no pair appears in the same relation, and $|S| = \mathsf{fn\text{-}fhtw}(Q)$. Suppose we are given a hard instance $\mathcal{R}_{\text{star}}$ for Q_{star} with $\mathsf{fn\text{-}fhtw}(Q)$ relations, which has input size N and output size OUT. We next construct an instance $\mathcal{R}$ for Q as follows.

Each output attribute $A \in \mathbf{y} - S$ contains one distinct value $\{*\}$. Each output attribute $A \in S$ contains $\text{OUT}^{\frac{1}{\mathsf{fn\text{-}fhtw}(Q)}}$ distinct values. Each non-output attribute $B \in (\mathcal{V} - \mathbf{y})$ contains $\frac{N}{|\mathcal{E}| \cdot \text{OUT}^{\frac{1}{\mathsf{fn\text{-}fhtw}(Q)}}}$ distinct values. For each relation R_e, $|e \cap S| \leqslant 1$. The projection of R_e onto all non-output attributes contains tuples in a form of $(b_i, b_i, \cdots, b_i)$, for $i \in \left[N \cdot \text{OUT}^{1-\frac{1}{\mathsf{fn\text{-}fhtw}(Q)}}\right]$. The projection of R_e onto all output attributes is the full Cartesian product. Moreover, $R_e = \left(\pi_{\mathbf{y}} R_e\right) \times \left(\pi_{\mathcal{V}-\mathbf{y}} R_e\right)$. For each output attribute $A \in S$, we choose an arbitrary relation $e \in \mathcal{E}$ such that $A \in e$. Note that all chosen relations are also distinct. Let $\mathcal{E}_S$ be the set of chosen relations. We specify an arbitrary one-to-one

mapping from relations in Q_{star} and relations in $\mathcal{E}_S$, say R_i corresponds to S_i. From our construction above, there is also a one-to-one mapping between tuples in R_i and S_i. We just set the annotation of a tuple $t \in S_i$ as the same as $t' \in R_i$, if t corresponds to t'. For every remaining relation R_e in $\mathcal{R}$, we simply set the annotation of each tuple as $\mathbf{1}$. As $|e \cap S| \leqslant 1$ for each relation $e \in \mathcal{E}$, it can be checked that each relation contains at most N tuples. The input size of the constructed instance is N, and the output size is exactly OUT.

It is not hard to see that $\bigoplus_{\mathcal{V}-S} Q(\mathcal{R}) = Q_{\text{star}}(\mathcal{R}_{\text{star}})$. Any semiring algorithm that compute $Q(\mathcal{R})$ within $O\left(N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}(Q)}}\right)$ time, can also compute $Q_{\text{star}}(\mathcal{R}_{\text{star}})$ within $O\left(N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}(Q)}}\right)$ time. Hence, this automatically follows Lemma 7.2.

Step 2. Consider any $\exists$-connected but not cleansed query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$. Let $Q' = (\mathcal{V}', \mathcal{E}', \mathbf{y})$ be the cleansed version of Q, and $\mathcal{R}'$ be the hard instance for Q' as constructed above. Each non-output attribute $B \in \mathcal{V} - \mathcal{V}'$ contains one distinct value $\{*\}$. For each $e' \in \mathcal{E} - \mathcal{E}'$, there must exist a relation $e \in \mathcal{E}'$ such that $e' \subseteq e$. The relation $R_{e'}$ is just a projection of R_e onto attribute e', where each tuple has its annotation as $\mathbf{1}$. It is not hard to see $\bigoplus_{\mathcal{V}-\mathbf{y}'} Q'(\mathcal{R}') = \bigoplus_{\mathcal{V}-\mathbf{y}} Q(\mathcal{R})$. Any semiring algorithm that can compute $Q(\mathcal{R})$ within $O\left(N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}(Q)}}\right)$ time, can also compute $Q'(\mathcal{R}')$ within $O\left(N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}(Q)}}\right)$ time. Hence, this automatically follows **Step 1**.

Step 3. Consider any $\exists$-disconnected acyclic query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$. Let $Q_1, Q_2, \cdots, Q_h$ be the $\exists$-connected subqueries of Q. Wlog, assume $\text{fn-fhtw}(Q_1) = \text{fn-fhtw}(Q)$. Suppose we are given any parameters $1 \leqslant N$ and $\text{OUT} \leqslant N^{\text{fn-fhtw}(Q)}$. There exists an instance $\mathcal{R}_1$ of input size N and output size OUT such that any semiring algorithm computing $Q_1(\mathcal{R}_1)$ requires $\Omega\left(N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}(Q)}} + \text{OUT}\right)$ time. We can extend the sub-instance $\mathcal{R}_1$ to an instance $\mathcal{R}$ to Q as follows. Each attribute in $\mathcal{V} - \mathcal{V}_1$ contains one distinct value $\{*\}$. Consider an arbitrary relation $e \in \mathcal{E} - \mathcal{E}_1$. If $e \cap \mathcal{V}_1 = \emptyset$, relation R_e only contains one tuple in a form of $(*, *, \cdots, *)$. Otherwise, $e \cap \mathcal{V}_1 \subseteq \mathbf{y}$. In this case, $R_e = (\pi_{\mathbf{y}} R_e) \times (\pi_{\mathcal{V}-\mathbf{y}} R_e)$. Note that each tuple in R_e has its annotation as $\mathbf{1}$. It is not hard to see $\bigoplus_{\mathcal{V}_1-\mathbf{y}_1} Q_1(\mathcal{R}_1) = \bigoplus_{\mathcal{V}-\mathbf{y}} Q(\mathcal{R})$. Any semiring algorithm that can compute $Q(\mathcal{R})$ within $O\left(N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}(Q)}} + \text{OUT}\right)$ time, also computes $Q_1(\mathcal{R}_1)$ within $O\left(N \cdot \text{OUT}^{1-\frac{1}{\text{fn-fhtw}(Q)}} + \text{OUT}\right)$ time. Hence, this automatically follows **Step 2**. □

8 RELATED WORK

Efficient evaluation of join-aggregate queries. Olteanu and Závodnỳ [46] investigate the problem of representing and computing results of conjunctive queries in a factorized form. They introduce the width $s^{\uparrow}(Q)$, which captures both the time complexity for computation and the size of factorized representations for conjunctive queries. They then establish a mapping between this notion of width and the free-connex fractional hypertree width. Subsequent work further explores the computation of aggregates over factorized databases [50]. In particular, aggregate functions such as `SUM`, `PROD`, `MIN`, and `MAX` can be evaluated on these factorized representations within $O(N^{s^{\uparrow}(Q)})$ time. Later, Abo Khamis et al. [3] extended these results by generalizing the approach to arbitrary semirings. Specifically, they define the FAQ query that considers multiple semirings simultaneously, and show that it can be computed in $O(N^{\text{faqw}(Q)} + \text{OUT})$ time, where $\text{faqw}(Q)$ denotes the FAQ-width of the query.

Extension of the Yannakakis Algorithm. Recent studies extend the Yannakakis algorithm to support a variety of operators and scenarios, including projections [9], unions [16], differences [29],

comparisons [54], dynamic query processing [32, 55], massively parallel query processing [4, 30] and secure query processing [56, 57]. Moreover, several works focus on efficiently implementing the Yannakakis algorithm within practical data systems [12, 13, 59, 61].

Fast matrix multiplication for conjunctive queries. As mentioned in Section 1, the semi-ring lower bound can be surpassed on specific semirings when certain conditions are met, as is the case for conjunctive queries defined over the Boolean ring. Recently, fast matrix multiplication has been widely adopted to accelerate the processing of conjunctive queries. Suppose that computing two rectangular matrices of size $n^a \times n^b$ and $n^b \times n^c$ can be done in $O(n^{\omega(a,b,c)+o(1)})$ time. For simplicity, we use ω to denote $\omega(1,1,1)$. There are some important constants related to rectangular matrix multiplication, such as, $\alpha \leqslant 1$ defined as the largest constant such that $\omega(1,\alpha,1) = 2$, and μ is the (unique) solution to the equation $\omega(\mu,1,1) = 2\mu + 1$. Note that $\alpha = 1$ if and only if $\omega = 2$, and the current best bounds on α are $0.321334 < \alpha \leqslant 1$ [58]. Moreover, $\mu = \frac{1}{2}$ if $\omega = 2$, and the current best bounds on μ are $\frac{1}{2} \leqslant \mu < 0.527661$ [58].

Amossen and Pagh [7] first proposed an algorithm for the Boolean matrix multiplication by using fast matrix multiplication, which runs in $\tilde{O}\left(N^{\frac{2}{3}} \cdot \mathrm{OUT}^{\frac{2}{3}} + N^{\frac{(2-\alpha)\omega-2}{(1+\omega)(1-\alpha)}} \cdot \mathrm{OUT}^{\frac{2-\alpha\omega}{(1+\omega)(1-\alpha)}} + \mathrm{OUT}\right)$ time when $\mathrm{OUT} \geqslant N$, and $\tilde{O}\left(N \cdot \mathrm{OUT}^{\frac{\omega-1}{\omega+1}}\right)$ time when $\mathrm{OUT} < N$. Very recently, Abboud et al. [1] have completely improved this to $\tilde{O}\left(N \cdot \mathrm{OUT}^{\frac{\mu}{1+\mu}} + \mathrm{OUT} + N^{\frac{(2+\alpha)\mu}{1+\mu}} \cdot \mathrm{OUT}^{\frac{1-\alpha\mu}{1+\mu}}\right)$. Improving these results further for any value of OUT remains a challenging task, given the hardness of the *all-edge-triangle* problem [1]. Several algorithms for Boolean matrix multiplication also measure complexity by the domain size of attributes; readers are referred to [1] for additional details. Deep et al. [21] and Huang and Chen [31] have further investigated the efficient practical implementation of these algorithms for sparse matrix multiplication.

Finally, Hu [28] applied fast matrix multiplication to speed up acyclic join-project queries and showed a polynomial improvement over the combinatorial Yannakakis algorithm. Independently, the algorithm community has extensively utilized fast matrix multiplication to speed up detecting, counting, and listing subgraph patterns, such as cliques [6, 19, 49] and cycles [33]. In addition, this technique has been used to approximately count cycles [17, 52], k-centering in graphs [34], etc.

9 CONCLUSION

In this paper, we established matching lower and upper bounds for computing general acyclic join-aggregate queries in an output-optimal manner, characterized by the free-connex fractional hypertree width of the queries. This result generalizes and improves upon all previously known upper and lower bounds. As a by-product, it also implies new output-sensitive algorithms for certain cyclic queries, but their optimality is still unknown. Hence, achieving output-optimal algorithms for cyclic join-aggregate queries remains an interesting open question. In addition, it is interesting to investigate join-aggregate queries defined over multiple semi-rings [3] and to explore if one can apply our new output-optimal algorithm to improve processing multiple semi-rings.

ACKNOWLEDGEMENT

This work was supported by the Natural Sciences and Engineering Research Council of Canada Discovery Grant. We thank Ke Yi, Mahmoud Abo Khamis, and Victor Zhong for their invaluable feedback. We also thank the constructive comments provided by all anonymous reviewers.

REFERENCES

[1] A. Abboud, K. Bringmann, N. Fischer, and M. Künnemann. The time complexity of fully sparse matrix multiplication. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 4670–4703. SIAM, 2024.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[3] M. Abo Khamis, H. Q. Ngo, and A. Rudra. Faq: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 13–28, 2016.

[4] F. Afrati, M. Joglekar, C. Ré, S. Salihoglu, and J. D. Ullman. GYM: A multiround join algorithm in MapReduce. In *Proc. International Conference on Database Theory*, 2017.

[5] S. M. Aji and R. J. McEliece. The generalized distributive law. *IEEE transactions on Information Theory*, 46(2):325–343, 2000.

[6] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997. doi: 10.1007/BF02523189. URL https://doi.org/10.1007/BF02523189.

[7] R. R. Amossen and R. Pagh. Faster join-projects and sparse matrix multiplications. In *Proceedings of the 12th International Conference on Database Theory*, pages 121–126. ACM, 2009.

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

[9] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*, pages 208–222. Springer, 2007.

[10] H. Barthels, M. Copik, and P. Bientinesi. The generalized matrix chain algorithm. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 138–148, 2018.

[11] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM (JACM)*, 30(3):479–513, 1983.

[12] L. Bekkers, F. Neven, S. Vansummeren, and Y. R. Wang. Instance-optimal acyclic join processing without regret: Engineering the yannakakis algorithm in column stores. *arXiv preprint arXiv:2411.04042*, 2024.

[13] A. Birler, A. Kemper, and T. Neumann. Robust join processing with diamond hardened joins. *Proceedings of the VLDB Endowment*, 17(11):3215–3228, 2024.

[14] A. Björklund, R. Pagh, V. V. Williams, and U. Zwick. Listing triangles. In *International Colloquium on Automata, Languages, and Programming*, pages 223–234. Springer, 2014.

[15] J.-Y. Cai, P. Lu, and M. Xia. The complexity of complex weighted boolean# csp. *Journal of Computer and System Sciences*, 80(1):217–236, 2014.

[16] N. Carmeli and M. Kröll. On the enumeration complexity of unions of conjunctive queries. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 134–148. ACM, 2019.

[17] K. Censor-Hillel, T. Even, and V. Vassilevska Williams. Fast approximate counting of cycles. In *51st International Colloquium on Automata, Languages, and Programming*, pages 37–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[18] M. Dalirrooyfard, S. Mathialagan, V. V. Williams, and Y. Xu. Listing cliques from smaller cliques. *arXiv preprint arXiv:2307.15871*, 2023.

[19] M. Dalirrooyfard, S. Mathialagan, V. V. Williams, and Y. Xu. Towards optimal output-sensitive clique listing or: Listing cliques from smaller cliques. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, pages 923–934, 2024.

[20] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.

[21] S. Deep, X. Hu, and P. Koutris. Fast join project query evaluation using matrix multiplication. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1213–1223, 2020.

[22] S. Deep, H. Zhao, A. Z. Fan, and P. Koutris. Output-sensitive conjunctive query evaluation. *Proceedings of the ACM on Management of Data*, 2(5):1–24, 2024.

[23] S. S. Godbole. On efficient computation of matrix chain products. *IEEE Transactions on Computers*, 100(9):864–866, 1973.

[24] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proc. ACM Symposium on Principles of Database Systems*, 2007.

[25] V. Guruswami et al. Algorithmic results in list decoding. *Foundations and Trends® in Theoretical Computer Science*, 2 (2):107–195, 2007.

[26] X. Hu. Cover or pack: New upper and lower bounds for massively parallel joins. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 181–198, 2021.

[27] X. Hu. Output-optimal algorithms for join-aggregate queries. *arXiv preprint arXiv:2406.05536*, 2024.

[28] X. Hu. Fast matrix multiplication for query processing. *Proceedings of the ACM on Management of Data*, 2(2):1–25, 2024.

[29] X. Hu and Q. Wang. Computing the difference of conjunctive queries efficiently. *Proceedings of the ACM on Management of Data*, 1(2):1–26, 2023.

[30] X. Hu, K. Yi, and Y. Tao. Output-optimal massively parallel algorithms for similarity joins. *ACM Transactions on Database Systems (TODS)*, 44(2):6, 2019.

[31] Z. Huang and S. Chen. Density-optimized intersection-free mapping and matrix multiplication for join-project operations. *Proceedings of the VLDB Endowment*, 15(10):2244–2256, 2022.

[32] M. Idris, M. Ugarte, and S. Vansummeren. The dynamic Yannakakis algorithm: Compact and efficient query processing under updates. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2017.

[33] C. Jin, V. V. Williams, and R. Zhou. Listing 6-cycles. In *2024 Symposium on Simplicity in Algorithms (SOSA)*, pages 19–27. SIAM, 2024.

[34] C. Jin, Y. Kirkpatrick, V. V. Williams, and N. Wein. Beyond 2-approximation for k-center in graphs. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 175–211. SIAM, 2025.

[35] M. R. Joglekar, R. Puttagunta, and C. Ré. Ajar: Aggregations and joins over annotated relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 91–106, 2016.

[36] M. A. Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proc. ACM Symposium on Principles of Database Systems*, 2017.

[37] M. A. Khamis, R. R. Curtin, B. Moseley, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. Functional aggregate queries with additive inequalities. *ACM Transactions on Database Systems (TODS)*, 45(4):1–41, 2020.

[38] J. Kohlas and N. Wilson. Semiring induced valuation algebras: Exact and approximate local computation algorithms. *Artificial Intelligence*, 172(11):1360–1399, 2008.

[39] C. Lin, W. Luo, Y. Fang, C. Ma, X. Liu, and Y. Ma. On efficient large sparse matrix chain multiplication. *Proceedings of the ACM on Management of Data*, 2(3):1–27, 2024.

[40] D. Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *Journal of the ACM (JACM)*, 60(6):1–51, 2013.

[41] R. Motwani and P. Raghavan. Randomized algorithms. *ACM Computing Surveys (CSUR)*, 28(1):33–37, 1996.

[42] H. Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 111–124. ACM, 2018.

[43] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *Proc. ACM Symposium on Principles of Database Systems*, pages 37–48, 2012.

[44] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16, 2014.

[45] K. Nishida, Y. Ito, and K. Nakano. Accelerating the dynamic programming for the matrix chain product on the gpu. In *2011 Second International Conference on Networking and Computing*, pages 320–326. IEEE, 2011.

[46] D. Olteanu and J. Závodnỳ. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)*, 40(1):1–44, 2015.

[47] R. Pagh and F. Silvestri. The input/output complexity of triangle enumeration. In *Proc. ACM Symposium on Principles of Database Systems*, 2014.

[48] R. Pagh and M. Stöckel. The input/output complexity of sparse matrix multiplication. In *Proc. European Symposium on Algorithms*, 2014.

[49] M. Patrascu. Towards polynomial lower bounds for dynamic problems. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 603–610, 2010.

[50] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In F. Özcan, G. Koutrika, and S. 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 3–18. ACM, 2016. doi: 10.1145/2882903.2882939. URL https://doi.org/10.1145/2882903.2882939.

[51] D. Suciu, D. Olteanu, C. Ré, and C. Koch. Probabilistic databases, synthesis lectures on data management. *Morgan & Claypool*, 2011.

[52] J. Tětek. Approximate triangle counting via sampling and fast matrix multiplication. In *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, pages 107–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2022.

[53] T. L. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*, 2014.

[54] Q. Wang and K. Yi. Conjunctive queries with comparisons. In *Proceedings of the 2022 International Conference on Management of Data*, pages 108–121, 2022.

[55] Q. Wang, X. Hu, B. Dai, and K. Yi. Change propagation without joins. *arXiv preprint arXiv:2301.04003*, 2023.

[56] Y. Wang and K. Yi. Secure yannakakis: Join-aggregate queries over private data. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1969–1981, 2021.

[57] Y. Wang and K. Yi. Query evaluation by circuits. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 67–78, 2022.
[58] V. V. Williams, Y. Xu, Z. Xu, and R. Zhou. New bounds for matrix multiplication: from alpha to omega. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3792–3835. SIAM, 2024.
[59] Y. Yang, H. Zhao, X. Yu, and P. Koutris. Predicate transfer: Efficient pre-filtering on multi-join queries. *arXiv preprint arXiv:2307.15255*, 2023.
[60] M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. International Conference on Very Large Data Bases*, pages 82–94, 1981.
[61] J. Zhao, K. Su, Y. Yang, X. Yu, P. Koutris, and H. Zhang. Debunking the myth of join ordering: Toward robust sql analytics. *arXiv preprint arXiv:2502.15181*, 2025.

Received December 2024; revised February 2025; accepted March 2025