Output-Optimal Evaluation of Conjunctive Queries

Xiao Hu
University of Waterloo
Canada
xiaohu@waterloo.ca

Shaleen Deep
Microsoft
USA
shaleen.deep@microsoft.com

Austen Z. Fan
UW-Madison
Madison, WI, USA
afan@cs.wisc.edu

Paraschos Koutris
UW-Madison
Madison, WI, USA
paris@cs.wisc.edu

Hangdong Zhao
UW-Madison
Madison, WI, USA
hangdong@cs.wisc.edu

ABSTRACT

One of the most celebrated results for evaluating conjunctive queries (CQs) is the Yannakakis algorithm [24] proposed in 1981. It is known that free-connex CQs can be evaluated in $O(N + \mathsf{OUT})$ time, 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 $O(N \cdot \mathsf{OUT})$ on the runtime is known for the remaining acyclic but non-free-connex CQs. Alternatively, one can convert a non-free-connex CQ into a free-connex one using tree decomposition techniques, and then run the Yannakakis algorithm. However, none of them is known to be output-optimal.

In this work, we show a lower and upper bound matching $\Theta\left(N \cdot \mathsf{OUT}^{1-\frac{1}{\mathsf{fn\text{-}fhtw}}} + \mathsf{OUT}\right)$ for computing acyclic CQs, where $\mathsf{fn\text{-}fhtw}$ is the *free-connex fractional hypertree width* of the query. Although free-connex fractional hypertree width is a natural and well-established measure of how far a CQ is from being free-connex, we demonstrate that it precisely captures the output-optimal complexity of acyclic CQs. 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 CQs in an output-optimal way. Our output-optimal algorithm proposed for acyclic CQs also extends to cyclic CQs, as well as CQs with aggregations over arbitrary commutative semirings.

1. INTRODUCTION

Conjunctive query (CQ) evaluation is a fundamental task for relational databases. Finding efficient algorithms for evaluating CQs has been a holy grail in database theory since 1981. Previous results have achieved two flavors of runtimes: *worst-case optimal* [21, 18, 15, 24, 4] and *output-sensitive* [24, 15]. Worst-case optimal bounds are tight only on pathological instances with large outputs, which is uncommon 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 or projection operator may significantly reduce the output size. In addition, an output-optimal algorithm is also worst-case optimal (but not vice versa): the Yannakakis algorithm [24] is the best-known example, which achieves an output-optimal bound $O(N + \mathsf{OUT})$ for *free-connex* queries.

Although output-optimal algorithms are practically desirable, they are much more difficult to design. For the large class of *acyclic but non-free-connex* queries, prior works have yet to discover an output-optimal algorithm. Currently, there are two predominant approaches to 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 [18, 19], and then run the Yannakakis algorithm. For (1), Yannakakis only gave an upper bound $O\left(N \cdot \mathsf{OUT}\right)$ on its runtime. Later, this bound was tightened to $O\left(N \cdot \mathsf{OUT}^{1-\frac{1}{k}}\right)$ for star CQs with k relations, which is output-optimal [20]. For (2), Khamis et al. showed an upper bound $O\left(N^{\#\mathsf{fn\text{-}subw}} + \mathsf{OUT}\right)$ on its runtime, where $\#\mathsf{fn\text{-}subw}$ is the *#free-connex submodular width* of the query [15]. Both algorithms are worst-case optimal, but neither is known to be *output-optimal* for general acyclic CQs.

In this work, we identify the free-connex fractional hypertree width ($\mathsf{fn\text{-}fhtw}$) for CQs to characterize the output-optimal complexity. Specifically, we show how to evaluate acyclic CQs in $\Theta\left(N \cdot \mathsf{OUT}^{1-\frac{1}{\mathsf{fn\text{-}fhtw}}} + \mathsf{OUT}\right)$ time, together with a matching lower bound under the semiring circuit model. Note that $\mathsf{fn\text{-}fhtw} = 1$ for the free-connex queries and $\mathsf{fn\text{-}fhtw} = k$ for star queries with k relations, thus generalizing the previous results on these two special cases. As a by-product, our output-optimal algorithm for acyclic queries also yields new output-sensitive algorithms for cyclic queries. In ad-

This is an invitation of two independent works published in Proc. ACM Manag. Vol. 2, No. 5 (PODS), Article 220 (https://doi.org/10.1145/3695838), and Vol 3, No. 2 (PODS), Article No. 104 (https://doi.org/10.1145/3725241).

dition, all of our results can be extended to aggregation queries over commutative semirings.

2. PRELIMINARIES

We establish the minimal notation and concepts required to state our main result.

Conjunctive Query (CQ). We associate a CQ Q with a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where the set of vertices $\mathcal{V} = \{x_1, \ldots, x_n\}$ model the *attributes* and the set of hyperedges $\mathcal{E} = \{e_1, \ldots, e_k\} \subseteq 2^{\mathcal{V}}$ model the *relations*. Let $\mathsf{dom}(x)$ be the *domain* of attribute $x \in \mathcal{V}$. Let $\mathsf{dom}(X) = \prod_{x \in X} \mathsf{dom}(x)$ be the *domain* of a subset $X \subseteq \mathcal{V}$ of attributes. An *instance* of Q is associated a set of relations $\mathcal{D} = \{R_e(\mathbf{x}_e) : e \in \mathcal{E}\}$, where $\mathbf{x}_e \subseteq \mathcal{V}$ is the schema (or the set of variables) of R_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 $\mathsf{dom}(x)$ to x for every attribute $x \in \mathbf{x}_e$. The CQ is written as:

$$Q(\mathbf{x}_\mathcal{F}) \leftarrow \bigwedge_{e \in \mathcal{E}} R_e(\mathbf{x}_e)$$

The variables $\mathbf{x}_\mathcal{F}$ in the head of the query are the free variables (or the output variables). In this way, a CQ Q can be modeled as a tuple $(\mathcal{H}, \mathcal{F})$. The query result of Q on the instance $\mathcal{D}$, denoted as $Q(\mathcal{D})$, is defined as:

$$Q(\mathcal{D}) = \{t' \in \mathsf{dom}(\mathbf{x}_\mathcal{F}) : \exists t \in \mathsf{dom}(\mathcal{V}), \pi_{\mathbf{x}_\mathcal{F}} t = t', \\ \forall e \in \mathcal{E}, \pi_{\mathbf{x}_e} t \in R_e\}.$$

We use N to denote the input size of $\mathcal{D}$, and use $\mathsf{OUT} = |Q(\mathcal{D})|$ to denote the output size.

We study the data complexity of this problem by assuming the query size (i.e., $|\mathcal{V}|$ and $|\mathcal{E}|$) is constant. Our objective in this work is to design algorithms for evaluating CQs by taking both the input and output size into account. In other words, the running time we obtain should be a function of both N and OUT.

Fractional Edge Covering Number. For a subset $S \subseteq \mathcal{V}$, we use $\mathcal{H}[S] = (S, \mathcal{E}[S])$ to denote the sub-hypergraph induced by S, where $\mathcal{E}[S] = \{e \cap S : e \in \mathcal{E}\}$. For any subset $S \subseteq \mathcal{V}$, a *fractional edge covering of* S is a function $\rho : \mathcal{E} \to [0, 1]$ such that $\sum_{e \in \mathcal{E}: x \in J} \rho(J) \geq 1$ for each variable $x \in S$. The *fractional edge covering number* of $\mathcal{H}$, denoted as $\rho^*(\mathcal{H})$, is defined as the minimum sum of weight over all possible fractional edge coverings ρ of $\mathcal{V}$, i.e., $\rho^*(\mathcal{H}) = \min_\rho \sum_{e \in \mathcal{E}} \rho(J)$.

Tree Decomposition. A *tree decomposition* (TD) of $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is a pair $(\mathcal{T}, \chi)$, where $\mathcal{T}$ is a tree and $\chi : \mathsf{nodes}(\mathcal{T}) \to 2^{\mathcal{V}}$ is a mapping from the nodes of $\mathcal{T}$ to subsets of $\mathcal{V}$, that satisfies the following properties: (1) For each relation $e \in \mathcal{E}$, there is a node $u \in \mathsf{nodes}(\mathcal{T})$ such that $\mathbf{x}_e \subseteq \chi(u)$. (2) For each attribute $A \in \mathcal{V}$, the set $\{u \in \mathsf{nodes}(\mathcal{T}) : A \in \chi(u)\}$ forms a connected sub-tree of $\mathcal{T}$.

Each set $\chi(u)$ is called a *bag* of the TD. The *width* of $(\mathcal{T}, \chi)$ noted as $\mathsf{width}(\mathcal{T}, \chi)$ is defined as

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

i.e., the maximum fractional edge covering number of the subqueries induced by bags in $\mathcal{T}$. A $\mathsf{TD}(\mathcal{T}, \chi)$ is *free-connex* for a CQ $(\mathcal{H}, \mathcal{F})$ if there is a connected subtree S of $\mathcal{T}$ such that $\bigcup_{u \in \mathsf{nodes}(S)} \chi(u) = \mathbf{x}_\mathcal{F}$, i.e., the union of attributes appearing in S is exactly the output variables. Figure 1 shows a tree decomposition rooted at two different nodes.

Classification of CQs. A CQ is acyclic [6, 14] if and only if it has a width-1 TD. A CQ is free-connex [5] if and only if it has a width-1 free-connex TD. As an example, both tree decompositions in Figure 1 have width 1.

Model of Computation. We use the standard RAM model with a uniform cost measure. For an instance of size N, every register has length $O(\log N)$. Any arithmetic operation (such as addition, subtraction, multiplication, and division) on the values of two registers can be done in $O(1)$ time. Sorting the values of N registers can be done in $O(N \log N)$ time.

3. CHAIN CQS

In this section, we first review the classic Yannakakis algorithm, then examine chain queries (a.k.a. chain matrix multiplication):

$$Q_{\text{chain}}(x_1, x_{k+1}) \leftarrow \bigwedge_{i \in [k]} R_i(x_i, x_{i+1}).$$

on which the Yannakakis algorithm is not optimal, and finally, show our optimal algorithm. We will use the 4-chain query Q_{chain4} ($k = 4$) as a running example.

3.1 Yannakakis Algorithm

Suppose we are given an acyclic CQ $Q = (\mathcal{H}, \mathcal{F})$ and an instance $\mathcal{D}$ 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 can simply replace $R_{e'}$ by $R_{e'} \bowtie R_e$ and remove R_e. Let $(\mathcal{T}, \chi)$ be a width-1 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. The first phase removes all *dangling tuples* via a bottom-up and top-down pass of semi-joins along $\mathcal{T}$, where a tuple is *dangling* if it does not participate in any full join result. This step can be performed in linear time $O(N)$.

Join-Project. The second phase roots $\mathcal{T}$ on an arbitrary root node r, and then performs joins and projections in a bottom-up way. Specifically, it takes two nodes R_u and R_{p_u} such that u is a leaf and p_u is the parent of u, projects away non-output variables that do not appear in $\chi(p_u)$ by replacing R_u with $\pi_{\chi(u) \cap (\chi(p_u) \cup \mathbf{x}_\mathcal{F})} R_u$, and replaces R_{p_u} with $R_{p_u} \bowtie R_u$. Then R_u is removed, and the step repeats until only one node remains, i.e., the root node r. Hence, it projects on output variables in r, and outputs $\pi_{\mathbf{x}_\mathcal{F}} R_r$ as the final result.

Figure 1 shows the query plans of running Yannakakis' algorithm on two rooted tree decompositions.

The runtime is proportional to the largest number of intermediate results materialized (after dangling tuples are removed). It can be shown that this number can always be upper bounded by $O(N \cdot \mathsf{OUT})$, hence the

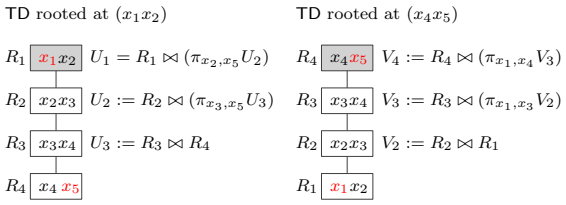

Figure 1: Two width-1 TDs for Q_{chain4}. For TD rooted at (x_1x_2), U_3, U_2, U_1 are the intermediate results materialized by the Yannakakis algorithm. For TD rooted at (x_4x_5), V_2, V_3, V_4 are the intermediate results materialized by the Yannakakis algorithm.

above algorithm takes time $O(N + N \cdot \mathsf{OUT})$. When the join query is free-connex, it can be shown that the runtime can be bounded instead by the tighter expression $O(N + \mathsf{OUT})$. However, this upper bound can sometimes be loose and dependent on the specific *query plan* chosen for a given input instance.

3.2 Why Yannakakis is suboptimal

We now construct an instance as shown in Figure 2, where we will show that Yannakakis' algorithm has a suboptimal behavior, independent of which plan we choose. This is a generalization of the analysis in [11].

Example 1. *Assume* $2 \le \mathsf{OUT} \le N$. *Let us first focus on the top half of Figure 2). Attributes* x_1, x_2, x_3, x_4 *have domain sizes* $\frac{\mathsf{OUT}}{2}, \frac{N}{\mathsf{OUT}}, \frac{N}{2}, 1$ *respectively.* R_1 *is a Cartesian product between* $\mathsf{dom}(x_1)$ *and* $\mathsf{dom}(x_2)$, R_2 *is a many-to-one mapping from* $\mathsf{dom}(x_2)$ *to* $\mathsf{dom}(x_3)$, *and* R_3 *is a one-to-many mapping between* $\mathsf{dom}(x_3)$ *and* $\mathsf{dom}(x_4)$. *It is easy to check that this instance has input size* $\Theta(N)$ *and the output size is* OUT. *The plan on the left in Figure 1 incurs a cost of* $\Theta(N \cdot \mathsf{OUT})$ *since* $|R_1 \bowtie R_2| = \frac{N \cdot \mathsf{OUT}}{2}$, *while the plan on the right incurs a cost of* $\Theta(N)$ *since* $|R_2 \bowtie R_3| = N$ *and* $|R_1 \times \mathsf{dom}(x_4)| = N$.

Next, we construct a symmetric instance (see the bottom half of Figure 2). Now, these two plans would have the opposite behavior. Finally, we consider the instance by combining these two sub-instances together. On this combined instance, both plans incur a cost of $\Theta(N \cdot \mathsf{OUT})$, *since* $|R_1 \bowtie R_2| = |R_2 \bowtie R_3| = \Theta(N \cdot \mathsf{OUT})$.

We can apply a similar argument to all other possible query plans for evaluating this instance, and any single query plan has to materialize $\Theta(N \cdot \mathsf{OUT})$ *intermediate join results. Hence, Yannakakis algorithm indeed requires* $\Theta(N \cdot \mathsf{OUT})$ *time on this instance.*

Inspired by this instance, we can leverage *the power of multiple query plans* to overcome the fundamental limitation of the Yannakakis' algorithm. In our example, we want to run the upper part of the input using the TD rooted at (x_4x_5), and then the lower part using the TD rooted at (x_1x_2); this leads to a running time of only $O(N + \mathsf{OUT})$. However, in general, it is not as straightforward to see which TDs should be chosen, how the data must be partitioned, and which part we should send to each TD. In the next section, we will show our novel strategy for the class of chain queries before we generalize it to all acyclic CQs.

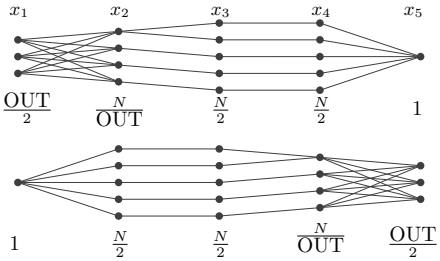

Figure 2: A database instance for 4-chain CQ.

Algorithm 1: ChainCQ$(Q_{\text{chain}}, \mathcal{D})$

foreach $i \in [k-1]$ **do**
  if $i = 1$ **then** $T_1(x_1, x_2) \leftarrow R_1$;
  else $T_i(x_1, x_{i+1}) \leftarrow S_{i-1} \wedge R_i$;
  $x_{i+1}^{\mathsf{heavy}}, x_{i+1}^{\mathsf{light}} \leftarrow$
    $\left\{a \in \mathsf{dom}(x_{i+1}) : \left|\sigma_{x_{i+1}=a} T_i\right| > \sqrt{\mathsf{OUT}}\right\}$,
    $\left\{a \in \mathsf{dom}(x_{i+1}) : 1 \le \left|\sigma_{x_{i+1}=a} T_i\right| \le \sqrt{\mathsf{OUT}}\right\}$;
  $R_i^{\mathsf{heavy}}, R_i^{\mathsf{light}} \leftarrow R_i \ltimes x_{i+1}^{\mathsf{heavy}},\ R_i \ltimes x_{i+1}^{\mathsf{light}}$;
  $S_i(x_1, x_{i+1}) \leftarrow T_i \ltimes x_{i+1}^{\mathsf{light}}$;
foreach $i \in [k-1]$ **do**
  $Q_i(x_1, x_{k+1}) \leftarrow \left(\wedge_{j \in [i-1]} R_j^{\mathsf{light}}\right) \wedge R_i^{\mathsf{heavy}} \wedge \left(\wedge_{j=i+1}^{k} R_j\right)$;
$Q_*(x_1, x_{k+1}) \leftarrow \left(\wedge_{j \in [k-1]} R_j^{\mathsf{light}}\right) \wedge R_k$;
return $Q_1 \cup Q_2 \cup \cdots \cup Q_{k-1} \cup Q_*$;

3.3 New Algorithm

Given the wide variety of TDs for a chain query, one natural question arises: which query plans should we select? For chain queries, we only consider two query plans that correspond to two width-1 TDs: (a) one is rooted at (x_1x_2); (b) one is rooted at (x_kx_{k+1}). See Figure 1 for Q_4. 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. For example, if the active domain of x_{k+1} is small, we choose the one rooted at (x_1x_2); and if the domain of x_1 is small, we choose the TD rooted at (x_kx_{k+1}).

As described in Algorithm 1, our new algorithm 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 union the results of all subqueries. See an example in Figure 3. We assume a constant-approximation of OUT is known; this assumption can be removed without increasing the complexity asymptotically [12]. In the following, we use OUT to denote a constant-approximation of the output size.

Stage I: Partition. For each value $a \in \mathsf{dom}(x_2)$, we define its *degree* as the number of tuples from R_1 that have value as a in x_2, i.e., $\triangle(a) = |\pi_{x_1}\sigma_{x_2=a}R_1|$. A value $a \in \mathsf{dom}(x_2)$ is *heavy* if $\triangle(a) > \sqrt{\mathsf{OUT}}$, and *light* otherwise. Let $A_2^{\mathsf{heavy}}, A_2^{\mathsf{light}}$ be the set of heavy and light values in x_2. Let $R_1^{\mathsf{heavy}} = R_1 \ltimes A_2^{\mathsf{heavy}}$ and $R_1^{\mathsf{light}} = R_1 \ltimes A_2^{\mathsf{light}}$ be the set of heavy and light tuples in R_1 respectively.

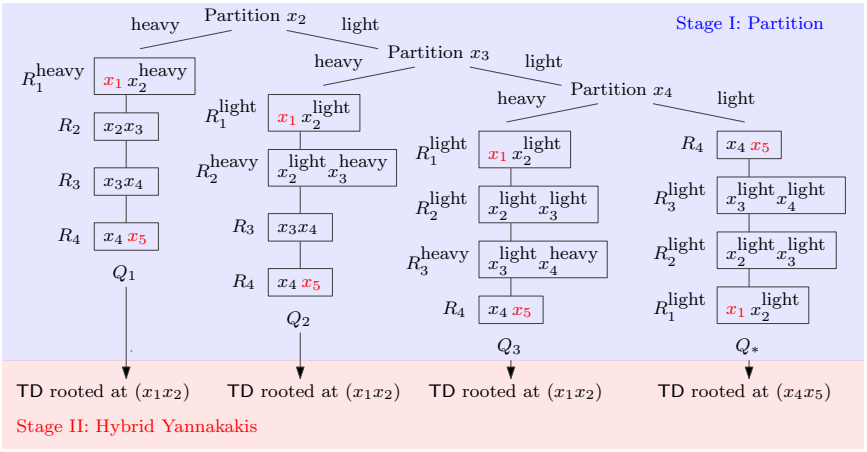

Figure 3: An illustration of Algorithm 1 for the 4-chain query.

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 $\mathsf{dom}(x_{i+1})$ that can be joined with any value in $\mathsf{dom}(x_1)$ via $R_1^{\mathsf{light}}, R_2^{\mathsf{light}}, \cdots, R_{i-1}^{\mathsf{light}}$. Let

$$\triangle(a) = \left|\pi_{x_1}\left\{\left(\bowtie_{j\in[i-1]} R_j^{\mathsf{light}}\right) \bowtie (\sigma_{x_{i+1}=a} R_i)\right\}\right|$$

be the *degree* of each value $a \in \mathsf{dom}(x_{i+1})$. A value $a \in \mathsf{dom}(x_{i+1})$ is *heavy* if $\triangle(a) > \sqrt{\mathsf{OUT}}$, and *light* otherwise. Let $x_{i+1}^{\mathsf{heavy}}, x_{i+1}^{\mathsf{light}}$ be the set of heavy, light values in x_i. Then, $R_i^{\mathsf{heavy}} = R_i \ltimes x_{i+1}^{\mathsf{heavy}}$ and $R_i^{\mathsf{light}} = R_i \ltimes x_{i+1}^{\mathsf{light}}$. Note that some values in x_{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:

$$T_i(x_1, x_{i+1}) = \pi_{x_1,x_{i+1}}\left(\bowtie_{j\in[i-1]} R_j^{\mathsf{light}}\right) \bowtie R_i,$$
$$S_i(x_1, x_{i+1}) = \pi_{x_1,x_{i+1}}\left(\bowtie_{j\in[i]} R_j^{\mathsf{light}}\right)$$

and recursively compute them by ($T_1 = R_1$ in line 2):

$$T_i = \pi_{x_1,x_{i+1}} S_i \bowtie R_i; \quad \text{(line 3)}$$
$$S_i = \pi_{x_1,x_{i+1}} T_i \ltimes x_{i+1}^{\mathsf{light}}; \quad \text{(line 6)}$$

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

$$Q_i(x_1, x_{k+1}) \leftarrow \left(\wedge_{j\in[i-1]} R_j^{\mathsf{light}}\right) \wedge R_i^{\mathsf{heavy}} \wedge \left(\wedge_{j=i+1}^{k} R_j\right)$$

for $i \in [k-1]$ and $Q_*(x_1, x_{k+1}) \leftarrow \left(\wedge_{j\in[k-1]} R_j^{\mathsf{light}}\right) \wedge R_k$.

Stage II: Hybrid Yannakakis. We invoke the Yannakakis algorithm to compute Q_i for each $i \in [k-1]$ using the TD rooted at (x_1x_2), and Q_* using the TD rooted at (x_4x_5). Finally, we union 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 $\sqrt{\mathsf{OUT}}$ tuples in S_{i-1}, T_i can be computed in $O\left(N \cdot \sqrt{\mathsf{OUT}}\right)$ time. Also, $|T_i| = O\left(N \cdot \sqrt{\mathsf{OUT}}\right)$. The cost of computing S_i is $O(|T_i|) = O\left(N \cdot \sqrt{\mathsf{OUT}}\right)$. In **Stage II**, for Q_i, each active value in x_{k+1} can be joined with at least $\sqrt{\mathsf{OUT}}$ values in x_1, implied by x_{i+1}^{heavy}. As there are OUT results in total, the active domain size of x_{k+1} is $O\left(\sqrt{\mathsf{OUT}}\right)$. Hence, the number of intermediate join results materialized for each node is at most $O\left(N \cdot \sqrt{\mathsf{OUT}}\right)$. For Q_*, the total number of intermediate join results is $O\left(N \cdot \sqrt{\mathsf{OUT}}\right)$, as there are N tuples in R_k and each of them can be joined with at most $\sqrt{\mathsf{OUT}}$ tuples in S_{k-1}. Hence, this step takes $O\left(N \cdot \sqrt{\mathsf{OUT}}\right)$ time. Finally, as each subquery produces at most OUT results, and there are $O(1)$ subqueries, the aggregation step takes $O(\mathsf{OUT})$ time. Putting everything together, we obtain:

Proposition 1. *For Q_{chain}, and an arbitrary instance $\mathcal{D}$ of input size N and output size OUT, the query result $Q(\mathcal{D})$ can be computed in $O\left(N + N \cdot \sqrt{\mathsf{OUT}}\right)$ time.*

4. MAIN RESULTS

In this section, we present our main result. We start with the formal definition of free-connex fractional hypertree width, state the main result, and then show the general algorithm for acyclic CQs. Finally, we extend our result to cyclic CQs and CQs with aggregation. Due to the page limit, all missing details can be found at [12].

4.1 Main Theorem

One of our main results is to identify that free-connex fractional hypertree width is the correct quantity for capturing the output-optimality of acyclic CQs:

Definition 1 (Free-connex Fractional Hypertree Width)**.** For a CQ $Q = (\mathcal{H}, \mathcal{F})$, let $\mathsf{FTD}(Q)$ be the set of all free-connex TDs for Q. The free-connex fractional hypertree

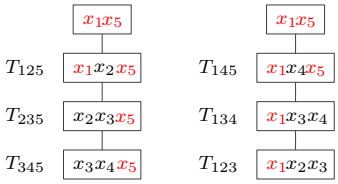

Figure 4: Two free-connex tree decompositions of the 4-chain CQ Q_{chain4}. $T_{\chi(u)}$ shows the attributes in the node of the tree decomposition.

width of Q, denoted as $\mathsf{fn\text{-}fhtw}(Q)$, is defined as:

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

In other words, it is the minimum width of all possible free-connex TDs. Contrast this with the traditional notion of fractional hypertree width, which instead considers all possible TDs. By definition, $\mathsf{fn\text{-}fhtw}(Q) \geq 1$ for all CQs and $\mathsf{fn\text{-}fhtw}(Q) = 1$ for free-connex CQs.

Example 2. *Figure 4 shows two free-connex tree decompositions for Q_4. Each of the two* TD*s has width 2, and one can show that* $\mathsf{fn\text{-}fhtw}(Q_{chain4}) = 2$. *In fact, for the general chain query Q_{chain},* $\mathsf{fn\text{-}fhtw}(Q_{chain}) = 2$.

We are now ready to state our main theorem.

Theorem 1. *For any acyclic CQ Q and a database $\mathcal{D}$ with input size N and output size* OUT*, there is an algorithm that can compute $Q(\mathcal{D})$ in time*

$$O\left(N + \mathsf{OUT} + N \cdot \mathsf{OUT}^{1-\frac{1}{\mathsf{fn\text{-}fhtw}(Q)}}\right).$$

4.2 Structual Analysis

To achieve Theorem 1, we point out a helper lemma that can significantly simplify the query structure.

Definition 2 (Separated CQ)**.** A CQ $Q = (\mathcal{H}, \mathcal{F})$ is separated if there is a one-to-one correspondence between the set of output attributes and the set of attributes that only appear in one relation, and for each $e \in \mathcal{E}$ that contain some output attributes, there exists some relation $e' \in \mathcal{E} - \{e\}$ that contains all non-output attributes in e, i.e., $\mathbf{x}_e - \mathbf{x}_{\mathcal{F}} \subseteq \mathbf{x}_{e'}$.

Lemma 1. *An acyclic CQ Q with an instance $\mathcal{D}$ of input size N can be transformed into a set of k separated acyclic CQs $Q_1, Q_2, \cdots, Q_k$ and k sub-instances $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_k$ each of input size $O(N)$, such that $Q(\mathcal{D}) = \wedge_{i\in[k]} Q(R_i)$, $|Q_i(\mathcal{D}_i)| \leq |Q(\mathcal{D})|$ for any $i \in [k]$, and* $\mathsf{fn\text{-}fhtw}(Q) = \max_{i\in[k]} \mathsf{fn\text{-}fhtw}(Q_i)$*, in $O(N)$ time.*

With Lemma 1, we can focus on evaluating each separated CQ first, and then combining their query result via join, since $Q(\mathcal{D}) = \wedge_{i\in[k]} Q(R_i)$, which is also the final query result. This join step will only take $O(N + \mathsf{OUT})$ time. In Section 4.3, we present an output-optimal algorithm for separated acyclic CQs.

We start with a nice structural property of separated acyclic CQs regarding $\mathsf{fn\text{-}fhtw}$. In a TD $(\mathcal{T}, \chi)$, for a node e, let $\mathcal{N}_e$ denote the set of nodes incident to it. Node e is in the *boundary* of $\mathcal{T}$ if $|\mathcal{N}_e| = 1$. Throughout this subsection, we will use a more complicated CQ $Q_{\text{separated}}(x_1, x_2, x_3, x_4) \leftarrow R_1(x_1, y_1) \wedge R_2(x_2, y_2) \wedge R_3(x_3, y_3) \wedge R_4(x_4, y_4) \wedge R_5(y_1, y_2, z_1, z_2) \wedge R_6(y_3, y_4, z_3, z_4)$ as the running example.

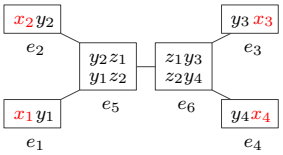

Figure 5: A separated TD $(\mathcal{T}, \chi)$ for $Q_{\text{separated}}$. The set of output attributes, or the set of attributes appearing in only one relation, is $\{x_1, x_2, x_3, x_4\}$ (highlighted in red). The set of boundary nodes, or the set of relations containing output attributes, is $\{e_1, e_2, e_3, e_4\}$.

Lemma 2. *Any separated acyclic CQ $Q = (\mathcal{H}, \mathcal{F})$ has a width-1* $\mathsf{TD}(\mathcal{T}, \chi)$ *such that there is a one-to-one correspondence between the set of relations containing output attribute(s) of Q and the boundary of $\mathcal{T}$. $(\mathcal{T}, \chi)$ is called a* separated TD *for Q.*

For a pair of incident nodes e_1, e_2 in $\mathcal{T}$, 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 $\mathcal{T}_{e_2,e_1}$, containing e_1 and e_2 separately. Let $\mathcal{L}_{e_1,e_2}$ be the set of boundary nodes in $\mathcal{T}_{e_1,e_2}$. We define $\phi_{e_1,e_2} = \frac{|\mathcal{L}_{e_1,e_2}|}{\mathsf{fn\text{-}fhtw}(Q)}$ as the *fraction* of boundary nodes in $\mathcal{T}_{e_1,e_2}$. Each edge (e_1, e_2) derives a sub-query as:

$$Q_{e_1,e_2}(\mathbf{x}_{\mathcal{F}} \cup (\mathbf{x}_{e_1} \cap \mathbf{x}_{e_2})) \leftarrow \bigwedge_{u\in\mathsf{nodes}(\mathcal{T}_{e_1,e_2})} R_e$$

i.e., projecting away all non-output attributes except the join attributes between e_1 and e_2.

Example 3. *Continue with the separated* TD *for $Q_{separated}$ in Figure 5. Removing edge $\{e_5, e_6\}$ leads to two sub-trees: $\mathcal{T}_{e_5,e_6}$ contains nodes e_1, e_2, e_5 and $\mathcal{T}_{e_6,e_5}$ contains nodes e_3, e_4, e_6. Edge (e_5, e_6) derives a CQ $Q_{e_5,e_6}(z_1, z_2,$ $x_1, x_2) \leftarrow R_1(x_1, y_1) \wedge R_2(x_2, y_2) \wedge R_5(y_1, y_2, z_1, z_2)$. And, $\phi(e_5, e_6) = \phi(e_6, e_5) = \frac{1}{2}$.*

Definition 3 (Edge Label)**.** For a separated acyclic query Q and a separated $\mathsf{TD}(\mathcal{T}, \chi)$, an instance $\mathcal{D}$ and parameter OUT, an edge (e_1, e_2) in $\mathcal{T}$ is

- *large* if $|Q_{e_1,e_2}(\mathcal{D}) \ltimes t| > \mathsf{OUT}^{\phi_{e_1,e_2}}$ holds for every tuple $t \in \pi_{\mathbf{x}_{e_1}\cap\mathbf{x}_{e_2}} R_{e_1}$; and
- *small* if $|Q_{e_1,e_2}(\mathcal{D}) \ltimes t| \leq \mathsf{OUT}^{\phi_{e_1,e_2}}$ holds for every tuple $t \in \pi_{\mathbf{x}_{e_1}\cap\mathbf{x}_{e_2}} R_{e_1}$; and
- *unlabeled* otherwise.

Furthermore, we distinguish a small edge (e_1, e_2) as *limited* if $|\pi_{\mathbf{x}_{\mathcal{F}}} Q_{e_1,e_2}(\mathcal{D})| \leq \mathsf{OUT}^{\phi_{e_1,e_2}}$.

Algorithm 2: HybridYannakakis$(Q, \mathcal{D})$

1 $(\mathcal{T}, \chi) \leftarrow$ a separated TD of Q with all edges non-labeled;
2 $\mathcal{P} \leftarrow \{(\mathcal{T}, \mathcal{D})\}$;
3 **while** $\mathcal{P} \neq \emptyset$ **do**
4 $(\mathcal{T}', \mathcal{D}') \leftarrow$ an arbitrary pair in $\mathcal{P}$;
5 **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 Label edge (e_1, e_2) as limited;
7 **if** $(\mathcal{T}', \mathcal{D}')$ *meets Lemma 5* **then**
8 Computing $Q(\mathcal{D}')$;
9 $\mathcal{S} \leftarrow \mathcal{S} \cup Q(\mathcal{D}')$;
10 **else**
11 $(e_1, e_2) \leftarrow$ a non-labeled edge in $\mathcal{T}'$ such that (e, e_1) is small for any $e \in \mathcal{N}_{e_1} - \{e_2\}$;
12 Compute $Q_{e_1,e_2}(\mathcal{D}')$;
13 $\mathcal{H} \leftarrow$ the set of tuples t in $\mathsf{dom}(\mathbf{x}_{e_1} \cap \mathbf{x}_{e_2})$ such that $\left|\sigma_{\mathbf{x}_{e_1} \cap \mathbf{x}_{e_2} = t} Q_{e_1,e_2}(\mathcal{D}')\right| > \mathsf{OUT}^{\phi_{e_1,e_2}}$;
14 $\mathcal{T}_1' \leftarrow \mathcal{T}'$ with (e_1, e_2), (e_2, e_1) labeled as large, limited separately;
15 $\mathcal{T}_2' \leftarrow \mathcal{T}'$ with (e_1, e_2) labeled as small;
16 Add $\left(\mathcal{T}_1', \mathcal{D}' - \{R_{e_1}\} \cup \{R_{e_1} \ltimes \mathcal{H}\}\right)$ to $\mathcal{P}$;
17 Add $\left(\mathcal{T}_2', \mathcal{D}' - \{R_{e_1}\} \cup \{R_{e_1} \triangleright \mathcal{H}\}\right)\}$ to $\mathcal{P}$;
18 Remove $(\mathcal{T}', \mathcal{D}')$ from $\mathcal{P}$;
19 **return** $\mathcal{S}$;

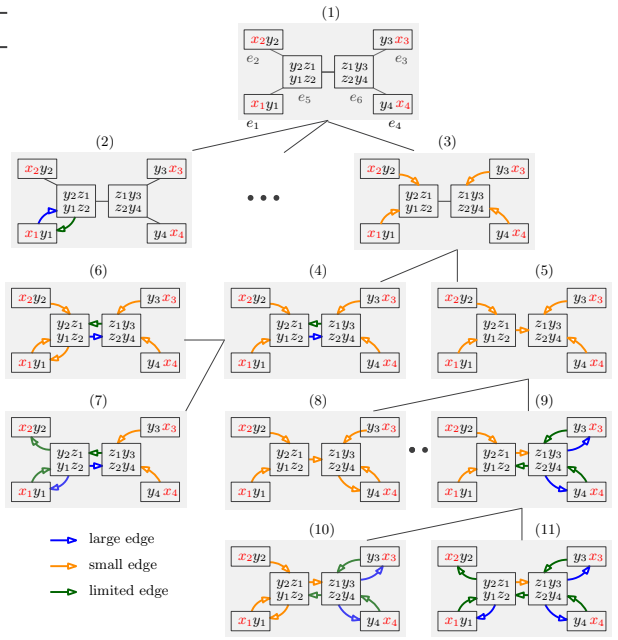

Figure 6: An illustration of the partition procedure on the query in Figure 5.

Lemma 3 (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.*

Lemma 4 (Large-Reverse-Limited)**.** *If edge* (e_1, e_2) *is large, then edge* (e_2, e_1) *must be limited.*

4.3 Algorithm

Base Case. We start by characterizing an *optimal condition* on the input instances for which the runtime of the Yannakakis algorithm can be bounded as stated in Theorem 1. We call such an instance a base case.

Lemma 5 (Base Case)**.** *For a separated acyclic CQ* $Q = (\mathcal{H}, \mathcal{F})$ *with a separated* TD $(\mathcal{T}, \chi)$*, and an instance* $\mathcal{D}$ *of input size* N *and output size* OUT*, if there is a boundary node* e *of* $\mathcal{T}$ *such that the only edge* $(*, e)$ *is small, then Yannakakis algorithm can compute* $Q(\mathcal{D})$ *in* $O\left(N + \mathsf{OUT} + N \cdot \mathsf{OUT}^{1 - \frac{1}{\mathsf{fn\text{-}fhtw}(Q)}}\right)$ *time.*

Partition into Base Cases. The Yannakakis algorithm falls short of optimality when input instances fail to meet the required condition in Lemma 5. Even worse, it is unknown how to efficiently *decide* the label of each edge because the definitions rely on sub-queries that are expensive to compute. Therefore, the core technical challenge is defining an efficient ordering for edge labeling and isolating sub-instances once they fulfill the criteria of Lemma 5.

As described in Algorithm 2, let $(\mathcal{T}, \chi)$ be a separated TD for Q (line 1), with all edges unlabeled initially. We put $(\mathcal{T}, \mathcal{D})$ into a candidate set $\mathcal{P}$ of instances to be partitioned (line 2). In general, consider an arbitrary pair $(\mathcal{T}', \mathcal{D}') \in \mathcal{P}$. From Lemma 3, 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 7-9). Otherwise, we further partition it (lines 9-17). 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 11). We compute $Q_{e_1,e_2}(\mathcal{D}')$ using the Yannakakis algorithm along $\mathcal{T}_{e_1,e_2}$ rooted at e_1 (line 12). A tuple $t \in \mathsf{dom}(\mathbf{x}_{e_1} \cap \mathbf{x}_{e_2})$ is *heavy* if $|Q_{e_1,e_2}(\mathcal{D}') \ltimes t| > \hat{\mathsf{OUT}}^{\phi_{e_1,e_2}}$, and *light* otherwise. Now, we construct two sub-instances for $\mathcal{D}'$, which contain heavy and light tuples in R_{e_1} separately (line 16-17), and two copies of $\mathcal{T}'$ in which edge (e_1, e_2) is further labeled as large and small separately (line 14-15). By Lemma 4, 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 16-17). In either case, we will remove $(\mathcal{T}', \mathcal{D}')$ from $\mathcal{P}$ (line 18). We continue applying this procedure to every remaining pair in $\mathcal{P}$ until $\mathcal{P}$ becomes empty (line 3).

Example 4. *We continue the example in Figure 5. Initially, all edges are unlabeled in (1). We start with applying line 13 to label edge* (e_1, e_5) *since* $\mathcal{N}_{e_1} - \{e_5\} = \emptyset$*. In (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 (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 (4), for 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 (6), the instance with small edge* (e_5, e_1) *already meets the optimal condition. In (7), we are left with an instance with a large edge* (e_5, e_1) *and a 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 (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 (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), 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 (10), the instance with small edge* (e_5, e_1) *already meets the optimal condition. In (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,* $\mathcal{P} = \emptyset$*, and we are done with the partition procedure.*

4.4 Extensions

Below, we discuss two nice extensions of our output-optimal algorithm for acyclic CQs:

Cyclic CQs. Cyclic queries are usually tackled with tree decomposition techniques. Given a tree decomposition of the cyclic CQ, we can first evaluate the derived query for each bag, materialize the query results as a relation, and then apply our new output-optimal algorithm to the transformed acyclic CQ. [3, 16] showed an algorithm that can decompose any cyclic CQ with a database instance $\mathcal{D}$ of input size N into a set of tree decompositions with sub-instances, such that the input size of each sub-instance is bounded by at most $O(N^{\mathsf{subw}(Q)})$, where $\mathsf{subw}(Q)$ is the *s*ub-modular width of Q [17].

Theorem 2. *For any CQ* Q *and a database* $\mathcal{D}$*, there is an algorithm that can compute* $Q(\mathcal{D})$ *in*

$$O\left(N + \mathsf{OUT} + N^{\mathsf{subw}(Q)} \cdot \mathsf{OUT}^{1 - \frac{1}{\max_i \mathsf{fn\text{-}fhtw}(Q_i)}}\right)$$

time, where $\{Q_i\}_i$ *is the set of acyclic queries that correspond to* TD*s of* Q*.*

CQs with Aggregation. Let $(\mathbf{D}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ be a commutative semiring.[1] Some commonly used commutative semiring include Boolean semiring $\mathbb{B} = (\{\mathsf{false}, \mathsf{true}\}, \vee, \wedge, \mathsf{false}, \mathsf{true})$, counting semiring $\mathbb{C} = (\mathbb{N}, +, \cdot, 0, 1)$, and tropical semiring $\mathsf{Trop}^+ = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$. Abo Khamis et al. [2] introduced the functional aggregate queries (FAQ) that express CQs with aggregation via semiring annotations.

[1] A commutative semiring $(\mathbf{D}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ satisfies the following properties: (1) $(\mathbf{D}, \oplus, \mathbf{0})$ is a commutative monoid with additive identity $\mathbf{0}$ (i.e., $\oplus$ is associative and commutative, and $a \oplus \mathbf{0} = a$ for all $a \in \mathbf{D}$); (2) $(\mathbf{D}, \otimes, \mathbf{1})$ is a (commutative) monoid with multiplicative identity $\mathbf{1}$ for $\otimes$; (3) $\otimes$ distributes over $\oplus$, i.e., $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ for $a, b, c \in \mathbf{D}$; and (4) $a \otimes \mathbf{0} = \mathbf{0}$ for all $a \in \mathbf{D}$.

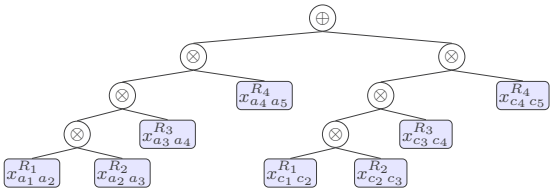

Figure 7: Arithmetic circuit for the 4-chain full CQ. Leaves are monomials representing the tuples; internal nodes are $\oplus$ and $\otimes$.

Let $Q = (\mathcal{H}, \mathcal{F})$ be a CQ where $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. A FAQ φ over a CQ Q and a commutative semiring $(\mathbf{D}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is defined as follows:

$$\varphi(\mathbf{x}_\mathcal{F}) \leftarrow \bigoplus_{\mathcal{V} \setminus \mathbf{x}_\mathcal{F}} \bigotimes_{e \in \mathcal{E}} R_e(\mathbf{x}_e),$$

Let Q_{full} be the full version of Q: $Q_{\mathsf{full}}(\mathcal{V}) \leftarrow \bigwedge_{e \in \mathcal{E}} R_e(\mathbf{x}_e)$. For a database instance $\mathcal{D}$, each relation $R_e(\mathbf{x}_e)$ is enriched into a σ-relation, where each tuple is associated with a semi-ring element from $\mathbf{D}$. The query result of ψ on $\mathcal{D}$, denoted as $\psi(\mathcal{D})$ is a σ-relation of schema $\mathbf{x}_\mathcal{F}$, where each query result $s_{\mathbf{x}_\mathcal{F}} \in Q(\mathcal{D})$ is associated with the following semi-ring element from $\mathbf{D}$:

$$p^{\mathcal{H}}_{\mathcal{D}[s_{\mathbf{x}_\mathcal{F}}]} := \bigoplus_{t \in Q_{\mathsf{full}}(\mathcal{D}) : \pi_{\mathbf{x}_\mathcal{F}} t = s_{\mathbf{x}_\mathcal{F}}} \bigotimes_{J \in \mathcal{E}} x^e_{(\pi_{\mathbf{x}_J} t)} \qquad (1)$$

where $x^e_{(\pi_{\mathbf{x}_e} t)}$ is the semi-ring element from $\mathbf{D}$ associated with tuple $(\pi_{\mathbf{x}_e} t) \in R_e$. Our output-sensitive algorithm can be similarly extended to acyclic FAQs with the following result:

Theorem 3. *For an* FAQ *query* ψ *over an acyclic CQ* Q*, a commutative semiring* $(\mathbf{D}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$*, and a database* $\mathcal{D}$*, there is an algorithm that can compute the query result in* $O\left(N + N \cdot \mathsf{OUT}^{1 - \frac{1}{\mathsf{fn\text{-}fhtw}(Q)}} + \mathsf{OUT}\right)$ *time.*

5. LOWER BOUND

We now present a matching lower bound by showing the smallest circuit needed for computing the provenance polynomial of all query results over a given semiring. The *sum-product polynomial* for a CQ $Q = (\mathcal{H}, \mathcal{F})$, is parameterized by an underlying semiring, a hypergraph $\mathcal{H}$, an instance $\mathcal{D}$ and a tuple $s_{\mathbf{x}_\mathcal{F}} \in Q(\mathcal{D})$, is essentially $p^{\mathcal{H}}_{\mathcal{D}[s_{\mathbf{x}_\mathcal{F}}]}$ defined in Equation 1.

Circuits over Semirings. A *circuit* F over a semiring is a *directed acyclic graph* with input nodes (with fan-in 0) variables in a set S_x containing $x^e_{(\pi_e t)}$'s in the right-hand-side of Equation 1 and the constants $\mathbf{0}, \mathbf{1}$. Every other node is labeled by $\oplus$ or $\otimes$ and has fan-in 2; these nodes are called $\oplus$*-gates* and $\otimes$*-gates*, respectively. An input gate of F is any gate with fan-in 0, and an output gate of F is any gate with fan-out 0. The size of the circuit F, denoted as $|F|$, is the number of gates in F.

Example 5. *Consider the full version of 4-chain CQ:* $Q_{full}(x_1, x_2, x_3, x_4) \leftarrow R_1(x_1, x_2) \wedge R_2(x_2, x_3) \wedge R_3(x_3, x_4) \wedge R_4(x_4, x_5)$ *with a database instance* $\mathcal{D}$*:* $R_1 = \{(a_1, a_2), (c_1, c_2)\}$*;* $R_2 = \{(a_2, a_3), (c_2, c_3)\}$*;*

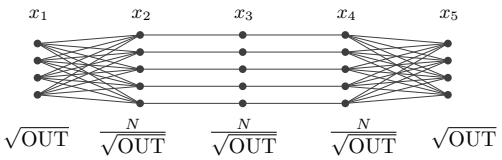

Figure 8: A database instance of 4-chain query.

$R_3 = \{(a_3, a_4), (c_3, c_4)\}$; *and* $R_4 = \{(a_4, a_5), (c_4, c_5)\}$. *Figure 7 shows the circuit that computes the polynomial shown below*

$$p_{\mathcal{D}}^{Q_{full}} = x_{a_1a_2}^{R_1} \otimes x_{a_2a_3}^{R_2} \otimes x_{a_3a_4}^{R_3} \otimes x_{a_4a_5}^{R_4} \oplus x_{c_1c_2}^{R_1} \otimes x_{c_2c_3}^{R_2} \otimes x_{c_3c_4}^{R_3} \otimes x_{c_4c_5}^{R_4}$$

The circuit has size 7 as there are three $\otimes$ gates in each term and one $\oplus$ gate that aggregates the final result.

Disjunctive datalog Rules. A *disjunctive datalog rule* [3] is an expression

$$P : \bigvee_{B \in \mathcal{B}} T_B(\mathbf{x}_B) \leftarrow \bigwedge_{J \in \mathcal{E}} R_J(\mathbf{x}_J)$$

where $B \subseteq [n]$ and $\mathcal{B} \subseteq 2^{[n]}$, where $2^{[n]}$ is the power set of $[n]$. The body is similar to that of a CQ, while the head is a disjunction of output relations T_B which are called *targets*. Given an instance $\mathcal{D}$, a *model* of a disjunctive datalog rule is a tuple $\mathbf{T} = (T_B)_{B \in \mathcal{B}}$ of relations, denoted as $\mathbf{T} \models P$, such that for any tuple t, if $(\pi_e t) \in R_e$ for all $e \in \mathcal{E}$, then there exists a target $T_B \in \mathbf{T}$ such that $t[B] \in T_B$.

To find the lower bound, we seek to find the size of the smallest circuit that contains one output gate, each computing $p_{\mathcal{D}}^{\mathcal{H}}[s_U]$ for all $s_U \in Q(\mathcal{D})$, for a semiring $\mathbb{S}$. We begin by stating the matching lower bounds for acyclic queries under the output-sensitive setting.

Theorem 4. *For any acyclic but non-free-connex CQ $Q = (\mathcal{H}, \mathcal{F})$, and parameters $N, \mathsf{OUT} \in \mathbb{N}$ with $\mathsf{OUT} \leq N^{\rho^*}$, there is an instance $\mathcal{D}$ of input size $\Theta(N)$ and output size $\Theta(\mathsf{OUT})$ such that the size of the smallest circuit that computes $p_{\mathcal{D}}^{\mathcal{H}}[s_{\mathbf{x}_\mathcal{F}}]$ for all query result $s_{\mathbf{x}_\mathcal{F}} \in Q(\mathcal{D})$ is $\Omega\left(N \cdot \mathsf{OUT}^{1 - \frac{1}{\mathsf{fn\text{-}fhtw}(Q)}}\right)$, which holds for each semiring in $\{\mathsf{Trop}^+, \mathbb{C}, \mathbb{B}_{\mathsf{lin}}\}$.*

This lower bound is *unconditional*, i.e., it does not depend on any conjecture. The key technique is to consider the hypergraph with an extra hyperedge $\{\mathbf{x}_\mathcal{F}\}$ added, i.e., $Q' = (\mathcal{H}, \mathcal{F} \cup \{\mathbf{x}_\mathcal{F}\})$, and then consider the size of the largest bag that can be formed across all disjunctive datalog rules that can be formed for any given set of tree decompositions. We present the intuition of the lower bound result via the following example.

Example 6. *We construct a database instance for 4-chain CQ as shown in Figure 8. Attributes x_1, x_2, x_3, x_4 and x_5 have domain sizes $\sqrt{\mathsf{OUT}}$, $\frac{N}{\sqrt{\mathsf{OUT}}}$, $\frac{N}{\sqrt{\mathsf{OUT}}}$, $\frac{N}{\sqrt{\mathsf{OUT}}}$, $\sqrt{\mathsf{OUT}}$. R_1 is a Cartesian product between x_1 and x_2, R_2 is a many-to-one mapping from x_2 to x_3, R_3 is a one-to-many mapping from x_3 to x_4 and R_4 is a Cartesian product between x_4 and x_5. It can be easily checked that the instance has input size $\Theta(N)$ and the output size is $\Theta(\mathsf{OUT})$. We add another relation R_{OUT} containing $\{x_1, x_5\}$ to the body of the query, making it cyclic.*

Consider the two free-connex tree decompositions as shown in Figure 4 and the disjunctive rule below:

$$T_{125} \vee T_{345} \leftarrow R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_{\mathsf{OUT}}$$

Then, the size of the bag T_{125} is at least $O(N \cdot \sqrt{\mathsf{OUT}})$ since every tuple in relation R_1 has a join path to every attribute value of x_5.

Any free-connex tree decomposition will contain a bag that contains variables x_1, x_5 along with either x_2 or x_4. Thus, the worst-case size of such a bag is $\Theta(N \cdot \sqrt{\mathsf{OUT}})$.

6. CONCLUSION

In this paper, we establish matching lower and upper bounds for evaluating general acyclic CQs 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 cyclic CQs as well as CQs with aggregation over commutative semirings.

Our findings open several promising avenues for future investigation:

- *Output Optimality for Cyclic Queries:* The pursuit of output-optimal algorithms for cyclic CQs remains a significant open challenge. Specifically, it is yet to be determined if the free-connex fractional hypertree width serves as the definitive quantity to characterize the output optimality in these contexts.

- *Practical Systems Integration:* Beyond theoretical bounds, there is a clear opportunity to integrate this hybrid Yannakakis framework into practical data systems. This would build upon a growing body of implementation-focused research [25, 22, 23, 7] of the Yannakakis algorithm, bridging the gap between database theory and practical query processing.

- *Non-equi-join Queries:* It is an open problem to design output-sensitive algorithms for set similarity [9] and set containment joins [10]. We posit that some of our techniques can be applied to those queries to obtain novel algorithms with non-trivial guarantees.

- *Fast Matrix Multiplication:* A particularly compelling direction involves incorporating fast matrix multiplication to speed up combinatorial algorithms. Recent breakthroughs [8, 13, 11, 1] demonstrate that the use of algebraic techniques can effectively bypass traditional barriers and accelerate the evaluation of CQs, but it remains to be investigated for general CQs.

7. REFERENCES

[1] M. Abo Khamis, X. Hu, and D. Suciu. Fast matrix multiplication meets the submodular width. *Proceedings of the ACM on Management of Data*, 3(2):1–26, 2025.

[2] 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.

[3] M. Abo Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 429–444, 2017.

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

[5] 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.

[6] 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.

[7] L. Bekkers, F. Neven, S. Vansummeren, and Y. R. Wang. Instance-optimal acyclic join processing without regret: Engineering the yannakakis algorithm in column stores. *Proc. VLDB Endow.*, 18(8):2413–2426, Apr. 2025.

[8] 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*, SIGMOD '20, page 1213–1223, New York, NY, USA, 2020. Association for Computing Machinery.

[9] D. Deng, Y. Tao, and G. Li. Overlap set similarity joins with theoretical guarantees. In *Proceedings of the 2018 International Conference on Management of Data*, pages 905–920, 2018.

[10] D. Deng, C. Yang, S. Shang, F. Zhu, L. Liu, and L. Shao. Lcjoin: Set containment join via list crosscutting. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 362–373. IEEE, 2019.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] M. A. Khamis, H. Q. Ngo, and D. Suciu. Pandaexpress: a simpler and faster panda algorithm. *arXiv preprint arXiv:2512.10217*, 2025.

[17] D. Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *Journal of the ACM*, 2013.

[18] 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.

[19] 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.

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

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

[22] Q. Wang, B. Chen, B. Dai, K. Yi, F. Li, and L. Lin. Yannakakis+: Practical acyclic query evaluation with theoretical guarantees. *Proceedings of the ACM on Management of Data*, 3(3):1–28, 2025.

[23] Y. Yang, H. Zhao, X. Yu, and P. Koutris. Predicate transfer: Efficient pre-filtering on multi-join queries. *CIDR*, 2024.

[24] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, pages 82–94, 1981.

[25] J. Zhao, K. Su, Y. Yang, X. Yu, P. Koutris, and H. Zhang. Debunking the myth of join ordering: Toward robust sql analytics. *Proceedings of the ACM on Management of Data*, 3(3):1–28, 2025.