Logical Methods in Computer Science
Volume 21, Issue 2, 2025, pp. 14:1–14:33
https://lmcs.episciences.org/

Submitted Nov. 02, 2021
Published May 16, 2025

RANKED ENUMERATION OF CONJUNCTIVE QUERY RESULTS

SHALEEN DEEP AND PARASCHOS KOUTRIS

Department of Computer Sciences, University of Wisconsin-Madison, Madison, Wisconsin, USA
e-mail address: shaleen@cs.wisc.edu, paris@cs.wisc.edu

Abstract. We study the problem of enumerating answers of Conjunctive Queries ranked according to a given ranking function. Our main contribution is a novel algorithm with small preprocessing time, logarithmic delay, and non-trivial space usage during execution. To allow for efficient enumeration, we exploit certain properties of ranking functions that frequently occur in practice. To this end, we introduce the notions of *decomposable* and *compatible* (w.r.t. a query decomposition) ranking functions, which allow for partial aggregation of tuple scores in order to efficiently enumerate the output. We complement the algorithmic results with lower bounds that justify why restrictions on the structure of ranking functions are necessary. Our results extend and improve upon a long line of work that has studied ranked enumeration from both a theoretical and practical perspective.

1. Introduction

For many data processing applications, enumerating query results according to an order given by a ranking function is a fundamental task. For example, [YAG$^+$18, CLZ$^+$15] consider a setting where users want to extract the top patterns from an edge-weighted graph, where the rank of each pattern is the sum of the weights of the edges in the pattern. Ranked enumeration also occurs in SQL queries with an ORDER BY clause [QCS07, ISA$^+$04]. In the above scenarios, the user often wants to see the first k results in the query as quickly as possible, but the value of k may not be predetermined. Hence, it is critical to construct algorithms that can output the first tuple of the result as fast as possible, and then output the next tuple in the order with a very small *delay*. In this article, we study the algorithmic problem of enumerating the result of a Conjunctive Query (CQ, for short) against a relational database where the tuples must be output in order given by a ranking function.

The simplest way to enumerate the output is to materialize the result OUT and sort the tuples based on the score of each tuple. Although this approach is conceptually simple, it requires that $|\texttt{OUT}|$ tuples are materialized; moreover, the time from when the user submits the query to when she receives the first output tuples is $\Omega(|\texttt{OUT}| \cdot \log |\texttt{OUT}|)$. Further, the space and delay guarantees do not depend on the number of tuples that the user wants to actually see. More sophisticated approaches to this problem construct optimizers that exploit properties such as the monotonicity of the ranking function, allowing for join evaluation on a subset of the input relations (see [IBS08] and references within). In spite of the significant progress,

We are grateful to the reviewers for a careful reading of the manuscript and their feedback.

DOI:10.46298/LMCS-21(2:14)2025

all of the known techniques suffer from large worst-case space requirements, no dependence on k, and provide no non-trivial guarantees on the delay during enumeration, with the exception of a few cases where the ranking function is of a special form. Fagin et al. [FLN03] initiated a long line of study related to aggregation over *sorted lists*. However, [FLN03] and subsequent works also suffer from the above mentioned limitations as we do not have the materialized output $Q(D)$ that can be used as sorted lists.

In this article, we construct algorithms that remedy some of these issues. Our algorithms are divided into two phases: the *preprocessing phase*, where the system constructs a data structure that can be used later and the *enumeration phase*, when the results are generated. All of our algorithms aim to minimize the time of the preprocessing phase, and guarantee a *logarithmic delay* $O(\log |D|)$ during enumeration. Although we cannot hope to perform efficient ranked enumeration for an arbitrary ranking function, we show that our techniques apply for most ranking functions of practical interest, including lexicographic ordering, and sum (also product or max) of weights of input tuples among others.

Example 1.1. Consider a weighted graph G, where an edge (a, b) with weight w is represented by the relation $R(a, b, w)$. Suppose that the user is interested in finding the (directed) paths of length 3 in the graph with the lowest score, where the score is a (weighted) sum of the weights of the edges. The user query in this case can be specified as: $Q(x, y, z, u, w_1, w_2, w_3,) = R(x, y, w_1), R(y, z, w_2), R(z, u, w_3)$ where the ranking of the output tuples is specified for example by the score $5w_1 + 2w_2 + 4w_3$. If the graph has N edges, the naïve algorithm that computes and ranks all tuples needs $\Omega(N^2 \log N)$ preprocessing time. We show that it is possible to design an algorithm with $O(N)$ preprocessing time, such that the delay during enumeration is $O(\log N)$. This algorithm outputs the first k tuples by materializing $O(N + k)$ data, even if the full output is much larger.

The problem of ranked enumeration for CQs has been studied both theoretically [KS06, CS07, OZ15, TAG$^+$20] and practically [YAG$^+$18, CLZ$^+$15, BOZ12]. Theoretically, [KS06] establishes the tractability of enumerating answers in sorted order with polynomial delay (combined complexity), albeit with suboptimal space and delay factors for two classes of ranking functions. [YAG$^+$18] presents an anytime enumeration algorithm restricted to acyclic queries on graphs that uses $\Theta(|\texttt{OUT}| + |D|)$ space in the worst case, has a $\Theta(|D|)$ delay guarantee, and supports only simple ranking functions. As we will see, both of these guarantees are suboptimal and can be improved upon.

Ranked enumeration has also been studied for the class of lexicographic orderings. In [BDG07], the authors show that *free-connex acyclic CQs* can be enumerated in constant delay after only linear time preprocessing. Here, the lexicographic order is chosen by the algorithm and not the user. Factorized databases [BOZ12, OZ15] can also support constant delay ranked enumeration, but only when the lexicographic ordering agrees with the order of the query decomposition. In contrast, our results imply that we can achieve a logarithmic delay with the same preprocessing time for *any* lexicographic order. Concurrent work [TAG$^+$20] has also considered ranked enumeration from a theoretical and practical perspective. Our work recovers some of the theoretical results presented in [TAG$^+$20]. Further, we consider a broader set of ranking functions compared to all prior works along with new lower bounds.

Our Contributions. In this work, we show how to obtain logarithmic delay guarantees with small preprocessing time for ranking results of full (projection-free) CQs. We summarize our technical contributions below:

(1) Our main contribution (Theorem 3.1) is a novel algorithm that uses query decomposition techniques in conjunction with structure of the ranking function. The preprocessing phase sets up priority queues that maintain partial tuples at each node of the decomposition. During the enumeration phase, the algorithm materializes the output of the subquery formed by the subtree rooted at each node of the decomposition *on-the-fly*, in sorted order according to the ranking function. In order to define the rank of the partial tuples, we require that the ranking function can be *decomposed* with respect to the particular decomposition at hand. Theorem 3.1 then shows that with $O(|D|^{\texttt{fhw}})$ preprocessing time, where $\texttt{fhw}$ is the *fractional hypertree width* of the decomposition, we can enumerate with delay $O(\log |D|)$. We then discuss how to apply our main result to commonly used classes of ranking functions. Our work thoroughly resolves an open problem stated at the Dagstuhl Seminar 19211 [BKPS19] on ranked enumeration (see Question 4.6).

(2) We propose two extensions of Theorem 3.1 that improve the preprocessing time to $O(|D|^{\texttt{subw}})$, a polynomial improvement over Theorem 3.1 where $\texttt{subw}$ is the *submodular width* of the query Q. The result is based on a simple but powerful application of the main result that can be applied to any full UCQ Q combined with the `PANDA` algorithm proposed by Abo Khamis et al. [AKNS17].

(3) Finally, we show lower bounds (conditional and unconditional) for our algorithmic results. In particular, we show that subject to a popular conjecture, the logarithmic factor in delay cannot be removed. Additionally, we show that for two particular classes of ranking functions, we provide simple properties over the hypergraph that characterize whether it is possible to achieve logarithmic delay with linear preprocessing time for a large class of fully acyclic CQs.

This article is the full version of a conference publication [DK21]. We have added all of the proofs and intermediate results that were excluded from the paper. In particular, we have added the full proof of our main result – ranked enumeration of full CQs (Theorem 3.1). Additionally, we have also added the full algorithm for the the extensions in section 4. We have reworked the example for the main result and added a more detailed discussion to improve the exposition. In section 5, we present dichotomy results for *graph* queries (i.e., queries over binary relations)[1]. Finally, we have added a brief discussion in the conclusion regarding the extension of our results to the dynamic setting based on discussion with other community members at ICDT 2021. The remainder of the article is organized as follows. In the next section, we present the preliminaries and basic notation. Section 3 shows the first main result (Theorem 3.1), which is subsequently used as a building block in section 4 for the second main result (Theorem 4.1 and Theorem 4.2). Lower bounds are presented in section 5 and the related work in section 6. We conclude with a list of open problems in section 7.

2. Problem Setting

In this section we present the basic notions and terminology, and then discuss our framework.

[1] We also amend an error in the conference publication version.

2.1. **Conjunctive Queries.**

We will focus on the class of *Conjunctive Queries (CQs)*, which are expressed as $Q(\mathbf{y}) = R_1(\mathbf{x}_1), R_2(\mathbf{x}_2), \ldots, R_n(\mathbf{x}_n)$ Here, the symbols $\mathbf{y}, \mathbf{x}_1, \ldots, \mathbf{x}_n$ are vectors that contain *variables* or *constants*, the atom $Q(\mathbf{y})$ is the *head* of the query, and the atoms $R_1(\mathbf{x}_1), R_2(\mathbf{x}_2), \ldots, R_n(\mathbf{x}_n)$ form the *body.* The variables in the head are a subset of the variables that appear in the body. We use $\mathsf{vars}(Q)$ to denote the set of all variables in Q, i.e., $\mathbf{x}_1 \cup \cdots \cup \mathbf{x}_n$. A CQ is *full* if every variable in the body appears also in the head, and it is *boolean* if the head contains no variables, *i.e.* it is of the form $Q()$. If $\mathbf{x}_i \subseteq \mathbf{x}_j$, we can join $R_i(\mathbf{x}_i)$ and $R_j(\mathbf{x}_j)$, followed by removing atom R_i from the query.

We will typically use the symbols $x, y, z, \ldots$ to denote variables, and $a, b, c, \ldots$ to denote constants. We use $Q(D)$ to denote the result of the full CQ Q over input database D. A *valuation* θ over a set V of variables is a total function that maps each variable $x \in V$ to a value $\theta(x) \in \mathbf{dom}$, where $\mathbf{dom}$ is a domain of constants. We will often use $\mathbf{dom}(x)$ to denote the constants that the valuations over variable x can take. It is implicitly understood that a valuation is the identity function on constants. If $U \subseteq V$, then $\theta[U]$ denotes the restriction of θ to U. An *answer* to a full CQ Q is a tuple $\theta(\mathsf{vars}(Q))$ which is a mapping from $\mathsf{vars}(Q)$ to $\mathbf{dom}$ such that $\theta[\mathbf{x}_i] \in R_i$. $Q(D)$ is defined as the set of all answers.

A *Union of Conjunctive Queries* $\varphi = \bigcup_{i \in \{1,\ldots,\ell\}} \varphi_i$ is a set of CQs where $\text{head}(\varphi_{i_1}) = \text{head}(\varphi_{i_2})$ for all $1 \leq i_1, i_2 \leq \ell$. Semantically, $\varphi(D) = \bigcup_{i \in \{1,\ldots,\ell\}} \varphi_i(D)$. A UCQ is said to be full if each φ_i is full.

Natural Joins. If a CQ is full, has no constants, and no repeated variables in the same atom, then we say it is a *natural join query.* For instance, the 3-path query $Q(x, y, z, w) = R(x, y), S(y, z), T(z, w)$ is a natural join query. A natural join can be represented equivalently as a *hypergraph* $\mathcal{H}_Q = (\mathcal{V}_Q, \mathcal{E}_Q)$, where $\mathcal{V}_Q$ is the set of variables, and for each hyperedge $F \in \mathcal{E}_Q$ there exists a relation R_F with variables F. We will write the join as $\bowtie_{F \in \mathcal{E}_Q} R_F$. We denote the size of relation R_F by $|R_F|$. Given two tuples t_1 and t_2 over a set of variables $\mathcal{V}_1$ and $\mathcal{V}_2$ where $\mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset$, we will use $t_1 \circ t_2$ to denote the tuple formed over the variables $\mathcal{V}_1 \cup \mathcal{V}_2$. If $\mathcal{V}_1 \cap \mathcal{V}_2 \neq \emptyset$, then $t_1 \circ t_2$ will perform a join over the common variables.

Join Size Bounds. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph, and $S \subseteq \mathcal{V}$. A weight assignment $\mathbf{u} = (u_F)_{F \in \mathcal{E}}$ is called a *fractional edge cover* of S if (i) for every $F \in \mathcal{E}, u_F \geq 0$ and (ii) for every $x \in S, \sum_{F:x \in F} u_F \geq 1$. The *fractional edge cover number* of S, denoted by $\rho^*_{\mathcal{H}}(S)$ is the minimum of $\sum_{F \in \mathcal{E}} u_F$ over all fractional edge covers of S. We write $\rho^*(\mathcal{H}) = \rho^*_{\mathcal{H}}(\mathcal{V})$.

In a celebrated result, Atserias, Grohe and Marx [AGM13] proved that for every fractional edge cover $\mathbf{u}$ of $\mathcal{V}$, the size of a natural join is bounded using the *AGM inequality*: $| \bowtie_{F \in \mathcal{E}} R_F| \leq \prod_{F \in \mathcal{E}} |R_F|^{u_F}$ The above bound is constructive [NRR13, NPRR12]: there exist worst-case algorithms that compute the join $\bowtie_{F \in \mathcal{E}} R_F$ in time $O(\prod_{F \in \mathcal{E}} |R_F|^{u_F})$ for every fractional edge cover $\mathbf{u}$ of $\mathcal{V}$.

Tree Decompositions. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph of a natural join query Q. A *tree decomposition* of $\mathcal{H}$ is a tuple $(\mathcal{T}, (\mathcal{B}_t)_{t \in V(\mathcal{T})})$ where $\mathcal{T}$ is a tree, and every $\mathcal{B}_t$ is a subset of $\mathcal{V}$, called the *bag* of t, such that

(1) each edge in $\mathcal{E}$ is contained in some bag; and
(2) for each variable $x \in \mathcal{V}$, the set of nodes $\{t \mid x \in \mathcal{B}_t\}$ is connected in $\mathcal{T}$.

Given a rooted tree decomposition, we use $\mathtt{p}(t)$ to denote the (unique) parent of node $t \in V(\mathcal{T})$. Then, we define $\mathtt{key}(t) = \mathcal{B}_t \cap \mathcal{B}_{\mathtt{p}(t)}$ to be the common variables that occur in the bag $\mathcal{B}_t$ and its parent, and $\texttt{value}(t) = \mathcal{B}_t \setminus \texttt{key}(t)$ the remaining variables of the bag. We also use $\mathcal{B}_t^{\prec}$ to denote the union of all bags in the subtree rooted at t (including $\mathcal{B}_t$).

The *fractional hypertree width* of a decomposition is defined as $\max_{t \in V(\mathcal{T})} \rho^*(\mathcal{B}_t)$, where $\rho^*(\mathcal{B}_t)$ is the minimum fractional edge cover of the vertices in $\mathcal{B}_t$. The fractional hypertree width of a query Q, denoted $\texttt{fhw}(Q)$, is the minimum fractional hypertree width among all tree decompositions of its hypergraph. We say that a query is *acyclic* if $\texttt{fhw}(Q) = 1$. If a query is acyclic, then there exists a tree decomposition such that the bags for the nodes of the decomposition correspond to the hyperedges $\mathcal{E}$. Such a decomposition is known as a join tree. The *depth* of a rooted tree decomposition is the largest distance over all root to leaf paths in $\mathcal{T}$.

Computational Model. To measure the running time of our algorithms, we use the uniform-cost RAM model [HUA75], where data values as well as pointers to databases are of constant size. Throughout the article, all complexity results are with respect to data complexity (unless explicitly mentioned), where the query is assumed fixed. We will also use the set data structure that supports insertion and lookup of an element in constant time [CLRS22]. In practice, hashing can only achieve amortized constant time for some of the operations. Therefore, all lookups, insertion times, and enumeration delays are amortized.

2.2. Ranking Functions.

Consider a natural join query Q and a database D. Our goal is to enumerate all the tuples of $Q(D)$ according to an order that is specified by a *ranking function.* In practice, this ordering could be specified, for instance, in the `ORDER BY` clause of a `SQL` query.

Formally, we assume a total order $\succeq$ of the valuations θ over the variables of Q. The total order is induced by a ranking function $\texttt{rank}$ that maps each valuation θ to a number $\texttt{rank}(\theta) \in \mathbb{R}$. In particular, for two valuations θ_1, θ_2, we have $\theta_1 \succeq \theta_2$ if and only if $\texttt{rank}(\theta_1) \geq \texttt{rank}(\theta_2)$. Throughout the article, we will assume that $\texttt{rank}$ is a computable function that takes times linear in the input size to the function. We present below two concrete examples of ranking functions.

Example 2.1. For every constant $c \in \mathbf{dom}$, we associate a weight $w(c) \in \mathbb{R}$. Then, for each valuation θ, we can define $\texttt{rank}(\theta) := \sum_{x \in \mathcal{V}} w(\theta(x))$. This ranking function sums the weights of each value in the tuple.

Example 2.2. For every input tuple $t \in R_F$, we associate a weight $w_F(t) \in \mathbb{R}$. Then, for each valuation θ, we can define $\texttt{rank}(\theta) = \sum_{F \in \mathcal{E}} w_F(\theta[x_F])$ where x_F is the set of variables in F. In this case, the ranking function sums the weights of each contributing input tuple to the output tuple t (we can extend the ranking function to all valuations by associating a weight of 0 to tuples that are not contained in a relation).

Decomposable Rankings. As we will see later, not all ranking functions are amenable to efficient evaluation. Intuitively, an arbitrary ranking function will require that we look across all tuples to even find the smallest or largest element. We next present several restrictions which are satisfied by ranking functions seen in practical settings.

Definition 2.3 (Decomposable Ranking)**.** Let $\texttt{rank}$ be a ranking function over $\mathcal{V}$ and $S \subseteq \mathcal{V}$. We will use φ to denote valuations over the set of variables $\mathcal{V} \setminus S$. We say that $\texttt{rank}$ is

S-decomposable if there exists a total order for all valuations over S, such that for any two valuations θ_1, θ_2 over S we have:

$$\theta_1 \succeq \theta_2 \Rightarrow \begin{cases} \forall \varphi, \texttt{rank}(\varphi \circ \theta_1) = \texttt{rank}(\varphi \circ \theta_2) \\ \text{or} \\ \forall \varphi, \texttt{rank}(\varphi \circ \theta_1) > \texttt{rank}(\varphi \circ \theta_2) \end{cases}$$

We say that a ranking function is *totally decomposable* if it is S-decomposable for every subset $S \subseteq \mathcal{V}$, and that it is *coordinate decomposable* if it is S-decomposable for any singleton set. Additionally, we say that it is *edge decomposable* for a query Q if it is S-decomposable for every set S that is a hyperedge in the query hypergraph. We point out here that totally decomposable functions are equivalent to monotonic orders as defined in [KS06].

Example 2.4. The ranking function $\texttt{rank}(\theta) = \sum_{x \in \mathcal{V}} w(\theta(x))$ defined in Example 2.1 is totally decomposable, and hence also coordinate decomposable. Indeed, pick any set $S \subseteq \mathcal{V}$. We construct a total order on valuations θ over S by using the value $\sum_{x \in S} w(\theta(x))$. Now, consider valuations θ_1, θ_2 over S such that $\sum_{x \in S} w(\theta_1(x)) \geq \sum_{x \in S} w(\theta_2(x))$. Then, for any valuation φ over $\mathcal{V} \setminus S$, if $\sum_{x \in S} w(\theta_1(x)) = \sum_{x \in S} w(\theta_2(x))$, we have:

$$\texttt{rank}(\varphi \circ \theta_1) = \sum_{x \in \mathcal{V} \setminus S} w(\varphi(x)) + \sum_{x \in S} w(\theta_1(x)) = \sum_{x \in \mathcal{V} \setminus S} w(\varphi(x)) + \sum_{x \in S} w(\theta_2(x)) = \texttt{rank}(\varphi \circ \theta_2)$$

Similarly, if $\sum_{x \in S} w(\theta_1(x)) > \sum_{x \in S} w(\theta_2(x))$, we have:

$$\texttt{rank}(\varphi \circ \theta_1) = \sum_{x \in \mathcal{V} \setminus S} w(\varphi(x)) + \sum_{x \in S} w(\theta_1(x)) > \sum_{x \in \mathcal{V} \setminus S} w(\varphi(x)) + \sum_{x \in S} w(\theta_2(x)) = \texttt{rank}(\varphi \circ \theta_2)$$

Next, we construct a function that is coordinate-decomposable but it is not totally decomposable. Consider the query

$$Q(x_1 \ldots, x_d, y_1, \ldots, y_d) = R(x_1, \ldots, x_d), S(y_1, \ldots, y_d)$$

where $\mathbf{dom} = \{-1, 1\}$, and define $\texttt{rank}(\theta) := \sum_{i=1}^{d} \theta(x_i) \cdot \theta(y_i)$. This ranking function corresponds to taking the inner product of the input tuples if viewed as vectors. The total order for $\mathbf{dom}$ is $-1 \prec 1$. It can be shown that for $d = 2$, the function is not $\{x_1, x_2\}$-decomposable. For instance, if we define $\theta_1(x_1, x_2) = (1, -1) \succeq \theta_2(x_2, x_2) = (1, 1)$, then for $\varphi(y_1, y_2) = (-1, -1)$ we get $\texttt{rank}(\theta_1 \circ \varphi) = \texttt{rank}(1, -1, -1, -1) = 1 \cdot (-1) + (-1) \cdot (-1) = 0 > \texttt{rank}(\theta_2 \circ \varphi) = \texttt{rank}(1, 1, -1, -1) = 1 \cdot (-1) + 1 \cdot (-1) = -2$ but if we define $\varphi = (1, 1)$, then we get $\texttt{rank}(\theta_1 \circ \varphi) = \texttt{rank}(1, -1, 1, 1) = 1 \cdot 1 + 1 \cdot (-1) = 0 < \texttt{rank}(\theta_2 \circ \varphi) = \texttt{rank}(1, 1, 1, 1) = 1 \cdot 1 + 1 \cdot 1 = 2$. This demonstrates that the ranking function is not independent of valuations over $\{y_1, y_2\}$ and thus, the function does not satisfy the definition of decomposability.

Definition 2.5. Let $\texttt{rank}$ be a ranking function over a set of variables $\mathcal{V}$, and $S, T \subseteq \mathcal{V}$ such that $S \cap T = \emptyset$. We say that $\texttt{rank}$ is *T-decomposable conditioned on S* if for every valuation θ over S, the function $\texttt{rank}_\theta(\varphi) := \texttt{rank}(\theta \circ \varphi)$ defined over $\mathcal{V} \setminus S$ is T-decomposable.

The next lemma connects the notion of conditioned decomposability with decomposability.

Proposition 2.6. *Let* $\texttt{rank}$ *be a ranking function over a set of variables $\mathcal{V}$, and $T \subseteq \mathcal{V}$. If* $\texttt{rank}$ *is T-decomposable, then it is also T-decomposable conditioned on S for any $S \subseteq \mathcal{V} \setminus T$.*

Proof. We need to show that for every valuation π over S, $\texttt{rank}(\pi \circ \Phi \circ \theta)$ is T-decomposable where Φ is defined over $U = \mathcal{V} \setminus (S \cup T)$ and θ is defined over T. We use the same total order for θ as used for T-decomposability. Let $\theta_1 \succeq \theta_2$, and consider any valuation Φ over U. Define the valuation φ over $\mathcal{V} \setminus T$ such that $\varphi[S] = \pi$ and $\varphi[U] = \Phi$. Then,

$$\theta_1 \succeq \theta_2 \Rightarrow \begin{cases} \forall \varphi, \texttt{rank}(\varphi \circ \theta_1) = \texttt{rank}(\varphi \circ \theta_2) \\ \text{or} \\ \forall \varphi, \texttt{rank}(\varphi \circ \theta_1) > \texttt{rank}(\varphi \circ \theta_2) \end{cases}$$

$$\Leftrightarrow \begin{cases} \forall \varphi, \texttt{rank}(\varphi[S] \circ \varphi[U] \circ \theta_1) = \texttt{rank}(\varphi[S] \circ \varphi[U] \circ \theta_2) \\ \text{or} \\ \forall \varphi, \texttt{rank}(\varphi[S] \circ \varphi[U] \circ \theta_1) > \texttt{rank}(\varphi[S] \circ \varphi[U] \circ \theta_2) \end{cases}$$

$$\Leftrightarrow \begin{cases} \forall \varphi, \texttt{rank}(\pi \circ \Phi \circ \theta_1) = \texttt{rank}(\pi \circ \Phi \circ \theta_2) \\ \text{or} \\ \forall \varphi, \texttt{rank}(\pi \circ \Phi \circ \theta_1) > \texttt{rank}(\pi \circ \Phi \circ \theta_2) \end{cases}$$

Step 1 follows from the definition of T-decomposable. Step 2 and 3 compute the restriction of φ to S and U. □

It is also easy to check that if a function is $(S \cup T)$-decomposable, then it is also T-decomposable conditioned on S.

Definition 2.7 (Compatible Ranking)**.** Let $\mathcal{T}$ be a rooted tree decomposition of hypergraph $\mathcal{H}$ of a natural join query. We say that a ranking function is *compatible with* $\mathcal{T}$ if for every node t it is $(\mathcal{B}_t^{\prec} \setminus \texttt{key}(t))$-decomposable conditioned on $\texttt{key}(t)$.

Example 2.8. Consider the join query $Q(x, y, z) = R(x, y), S(y, z)$, and the ranking function from Example 2.2, $\texttt{rank}(\theta) = w_R(\theta(x), \theta(y)) + w_S(\theta(y), \theta(z))$. This function is not $\{z\}$-decomposable, but it is $\{z\}$-decomposable conditioned on $\{y\}$.

Consider a decomposition of the hypergraph of Q that has two nodes: the root node r with $\mathcal{B}_r = \{x, y\}$, and its child t with $\mathcal{B}_t = \{y, z\}$. Since $\mathcal{B}_t^{\prec} = \{y, z\}$ and $\texttt{key}(t) = \{y\}$, the condition of compatibility holds for node t. Similarly, for the root node $\mathcal{B}_t^{\prec} = \{x, y, z\}$ and $\texttt{key}(t) = \{\}$, hence the condition is trivially true as well. Thus, the ranking function is compatible with the decomposition.

2.3. Problem Parameters.

Given a natural join query Q and a database D, we want to enumerate the tuples of $Q(D)$ according to the order specified by $\texttt{rank}$. We will study this problem in the enumeration framework similar to that of [Seg15a], where an algorithm can be decomposed into two phases:

- a **preprocessing phase** that takes time T_p and computes a data structure of size S_p,
- an **enumeration phase** that outputs $Q(D)$ with no repetitions. The enumeration phase has full access to any data structures constructed in the preprocessing phase and can also use additional space of size S_e. The *delay* δ is defined as the maximum time to output any two consecutive tuples (and also the time to output the first tuple, and the time to notify that the enumeration has completed).

It is straightforward to perform ranked enumeration for any ranking function by computing $Q(D)$, storing the tuples in an ordered list, and finally enumerating by scanning the ordered list with constant delay. This simple strategy implies the following result.

Proposition 2.9. *Let Q be a natural join query with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. Let $\mathcal{T}$ be a tree decomposition with fractional hypertree-width* `fhw`*, and* `rank` *be a ranking function. Then, for any input database D, we can preprocess D in time $T_p = O(\log |D| \cdot |D|^{\texttt{fhw}} + |Q(D)|)$ and space $S_p = O(|Q(D)|)$, such that for any k, we can enumerate the top-k results of $Q(D)$ with delay $\delta = O(1)$ and space $S_e = O(1)$*

The drawback of Proposition 2.9 is that the user will have to wait $\Omega(|Q(D)| \cdot \log |Q(D)|)$ time to even obtain the first tuple in the output. Moreover, even when we are interested in a few tuples, the whole output result will have to be materialized. Instead, we want to design algorithms that minimize the preprocessing time and space, while guaranteeing a small delay δ. Interestingly, as we will see in section 5, the above result is essentially the best we can do if the ranking function is completely arbitrary; thus, we need to consider reasonable restrictions of `rank`.

To see what it is possible to achieve in this framework, it will be useful to keep in mind what we can do in the case where there is no ordering of the output.

Theorem 2.10 (due to [OZ15])**.** *Let Q be a natural join query with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. Let $\mathcal{T}$ be a tree decomposition with fractional hypertree-width* `fhw`*. Then, for any input database D, we can pre-process D in time $T_p = O(|D|^{\texttt{fhw}})$ and space $S_p = O(|D|^{\texttt{fhw}})$ such that we can enumerate the results of $Q(D)$ with delay $\delta = O(1)$ and space $S_e = O(1)$*

For acyclic queries, $\texttt{fhw} = 1$, and hence the preprocessing phase takes only linear time and space in the size of the input.

3. Main Result

In this section, we present our first main result.

Theorem 3.1 (Main Theorem)**.** *Let Q be a natural join query with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. Let $\mathcal{T}$ be a fixed tree decomposition with fractional hypertree-width* `fhw`*, and* `rank` *be a ranking function that is compatible with $\mathcal{T}$. Then, for any database D, we can preprocess D with*

$$T_p = O(|D|^{\texttt{fhw}}) \qquad S_p = O(|D|^{\texttt{fhw}})$$

such that for any k, we can enumerate the top-k tuples of $Q(D)$ with

$$\textit{delay } \delta = O(\log |D|) \qquad \textit{space } S_e = O(\min\{k, |Q(D)|\})$$

In the above theorem, the preprocessing step is independent of the value of k: we perform exactly the same preprocessing if the user only wants to obtain the first tuple, or all tuples in the result. However, if the user decides to stop after having obtained the first k results, the space used during enumeration will be bound by $O(k)$. We should also note that all of our algorithms work in the case where the ordering of the tuples/valuations is instead expressed through a `comparable` function that, given two valuations, returns the larger.

It is instructive to compare Theorem 3.1 with Theorem 2.10, where no ranking is used when enumerating the results. There are two major differences. First, the delay δ has an additional logarithmic factor. As we will discuss later in section 5, this logarithmic factor is a result of doing ranked enumeration, and it is most likely unavoidable. The second difference is that the space S_e used during enumeration blows up from constant $O(1)$ to $O(|Q(D)|)$ in the worst case (when all results are enumerated). Let us now define some notation that we will use in the section. Given a set U, $\langle U, \oplus \rangle$ is said to be a *commutative monoid* if $\oplus$ is a binary operator that is commutative, associative, and has an identity element in U. We say that the operator $\oplus$ is strictly monotone if $a = b$ implies that $a \oplus c = b \oplus c$, and if $a > b$ then $a \oplus c > b \oplus c$ for every $a, b, c \in U$. $\langle \mathbb{R}, + \rangle$ and $\langle \mathbb{N}^+, * \rangle$ are examples of commutative monoids where $\oplus$ is strictly monotone.

In the remainder of this section, we will present a few applications of Theorem 3.1, and then prove the theorem.

3.1. Applications.

We show here how to apply Theorem 3.1 to obtain algorithms for different ranking functions.

Vertex-Based Ranking. A vertex-based ranking function over $\mathcal{V}$ is of the form: $\texttt{rank}(\theta) := \bigoplus_{x \in \mathcal{V}} f_x(\theta(x))$ where f_x maps values from **dom** to some set $U \subseteq \mathbb{R}$ and $\langle U, \oplus \rangle$ forms a commutative monoid.

Lemma 3.2. *Let* $\texttt{rank}$ *be a strictly monotone vertex-based ranking function over* $\mathcal{V}$*. Then,* $\texttt{rank}$ *is totally decomposable, and hence compatible with any tree decomposition of a hypergraph with vertices* $\mathcal{V}$*.*

Proof. Pick any set $S \subseteq \mathcal{V}$ and let $\theta^\star$ be the valuation over $\mathcal{V} \setminus S$ such that for every x, $f_x(\theta^\star(x)) = e$, where e is the identity element of the monoid. We will define a total order over S in the following way. Since f_x maps values from **dom** to U, it holds that $\texttt{rank}(\theta^\star \circ \theta_1)$ and $\texttt{rank}(\theta^\star \circ \theta_2)$ are comparable for any two valuations θ_1, θ_2 that are defined over S. Therefore, if $\texttt{rank}(\theta^\star \circ \theta_1) \geq \texttt{rank}(\theta^\star \circ \theta_2)$, then $\theta_1 \succeq \theta_2$ and vice-versa. This establishes a total order.

Therefore, if $\theta_1 \succeq \theta_2$, it holds that $\oplus_{x \in S} f_x(\theta_1(x)) = \oplus_{x \in S} f_x(\theta_2(x))$ or $\oplus_{x \in S} f_x(\theta_1(x)) > \oplus_{x \in S} f_x(\theta_2(x))$.

If $\oplus_{x \in S} f_x(\theta_1(x)) = \oplus_{x \in S} f_x(\theta_2(x))$, for any valuation θ over $\mathcal{V} \setminus S$ we have:

$$\begin{aligned}\texttt{rank}(\theta \circ \theta_1) &= \oplus_{x \in \mathcal{V} \setminus S} f_x(\theta(x)) \bigoplus \oplus_{x \in S} f_x(\theta_1(x)) \\ &= \oplus_{x \in \mathcal{V} \setminus S} f_x(\theta(x)) \bigoplus \oplus_{x \in S} f_x(\theta_2(x)) \\ &= \texttt{rank}(\theta \circ \theta_2)\end{aligned}$$

Similarly, if $\oplus_{x \in S} f_x(\theta_1(x)) > \oplus_{x \in S} f_x(\theta_2(x))$, for any valuation θ over $\mathcal{V} \setminus S$ we have:

$$\begin{aligned}\texttt{rank}(\theta \circ \theta_1) &= \oplus_{x \in \mathcal{V} \setminus S} f_x(\theta(x)) \bigoplus \oplus_{x \in S} f_x(\theta_1(x)) \\ &> \oplus_{x \in \mathcal{V} \setminus S} f_x(\theta(x)) \bigoplus \oplus_{x \in S} f_x(\theta_2(x)) \\ &= \texttt{rank}(\theta \circ \theta_2)\end{aligned}$$

The inequalities holds because of the strict monotonicity of the binary operator. □

Tuple-Based Ranking. Given a query hypergraph $\mathcal{H}$, suppose we assign for every valuation θ over the variables x_F of relation R_F a weight $w_F(\theta) \in U \subseteq \mathbb{R}$. Then, a tuple-based ranking function takes the following form: $\texttt{rank}(\theta) := \bigoplus_{F \in \mathcal{E}} w_F(\theta[x_F])$ where $\langle U, \oplus \rangle$ forms a commutative monoid. In other words, a tuple-based ranking function assigns a weight to each input tuple, and then combines the weights through $\oplus$.

Lemma 3.3. *Let* $\texttt{rank}$ *be a strictly monotone tuple-based ranking function over the hypergraph* $\mathcal{H} = (\mathcal{V}, \mathcal{E})$*. Then,* $\texttt{rank}$ *is compatible with any tree decomposition of the hypergraph.*

Proof. Pick some node t in the decomposition, and fix a valuation θ_0 over $\texttt{key}(t)$. Let $E \subseteq \mathcal{E}$ be the hyperedges that correspond to bags in the subtree rooted at t, and $\bar{E}$ the remaining hyperedges. Let $\theta^\star$ be the valuation over $\mathcal{V} \setminus \mathcal{B}_t^{\prec}$ such that for every $F \in \bar{E}$ we have $w_F((\theta_0 \circ \theta^\star)[x_F]) = e$, where e is the identity element. Notice that the latter is well-defined, since the hyperedges in $\bar{E}$ can not contain any variables in $\mathcal{B}_t^{\prec} \setminus \texttt{key}(t)$.

We will define a total order over $\mathcal{B}_t^{\prec} \setminus \texttt{key}(t)$ in the following way. Since w_F maps values to U, it holds that $\texttt{rank}(\theta_0 \circ \theta^\star \circ \theta_1)$ and $\texttt{rank}(\theta_0 \circ \theta^\star \circ \theta_2)$ are comparable for any two valuations θ_1, θ_2 over S. Therefore, if $\texttt{rank}(\theta_0 \circ \theta^\star \circ \theta_1) \geq \texttt{rank}(\theta_0 \circ \theta^\star \circ \theta_2)$, then $\theta_1 \succeq \theta_2$ and vice-versa. This establishes a total order. Given θ_1 and θ_2 such that $\theta_1 \succeq \theta_2$, we have that either $\oplus_{F \in E} w_F((\theta_0 \circ \theta_1)[x_F]) = \oplus_{F \in E} w_F((\theta_0 \circ \theta_2)[x_F])$ or $\oplus_{F \in E} w_F((\theta_0 \circ \theta_1)[x_F]) > \oplus_{F \in E} w_F((\theta_0 \circ \theta_2)[x_F])$.

If the former is true, then for any valuation θ over $\mathcal{V} \setminus \mathcal{B}_t^{\prec}$, we have:

$$\begin{aligned}
&\texttt{rank}(\theta_0 \circ \theta \circ \theta_1) = \\
&\qquad = \oplus_{F \in E} w_F((\theta_0 \circ \theta_1)[x_F]) \bigoplus \oplus_{F \in \bar{E}} w_F((\theta \circ \theta_0)[x_F]) \\
&\qquad = \oplus_{F \in E} w_F((\theta_0 \circ \theta_2)[x_F]) \bigoplus \oplus_{F \in \bar{E}} w_F((\theta \circ \theta_0)[x_F]) \\
&\qquad = \texttt{rank}(\theta_0 \circ \theta \circ \theta_2)
\end{aligned}$$

For the latter, we have:

$$\begin{aligned}
&\texttt{rank}(\theta_0 \circ \theta \circ \theta_1) = \\
&\qquad = \oplus_{F \in E} w_F((\theta_0 \circ \theta_1)[x_F]) \bigoplus \oplus_{F \in \bar{E}} w_F((\theta \circ \theta_0)[x_F]) \\
&\qquad > \oplus_{F \in E} w_F((\theta_0 \circ \theta_2)[x_F]) \bigoplus \oplus_{F \in \bar{E}} w_F((\theta \circ \theta_0)[x_F]) \\
&\qquad = \texttt{rank}(\theta_0 \circ \theta \circ \theta_2)
\end{aligned}$$

The inequality hold because of the strict monotonicity of the binary operator. □

Since both monotone tuple-based and vertex-based ranking functions are compatible with any tree decomposition we choose, the following result is immediate.

Proposition 3.4. *Let Q be a natural join query with optimal fractional hypertree-width $\texttt{fhw}$. Let $\texttt{rank}$ be a ranking function that can be either (i) strictly monotone vertex-based, (ii) strictly monotone tuple-based. Then, for any input D, we can pre-process D in time $T_p = O(|D|^{\texttt{fhw}})$ and space $S_p = O(|D|^{\texttt{fhw}})$ such that for any k, we can enumerate the top-k results of $Q(D)$ with $\delta = O(\log |D|)$ and $S_e = O(\min\{k, |Q(D)|\})$*

For instance, if the query is acyclic, hence $\texttt{fhw} = 1$, the above theorem gives an algorithm with linear preprocessing time $O(|D|)$ and $O(\log |D|)$ delay.

Lexicographic Ranking. A typical ordering of the output valuations is according to a *lexicographic order*. In this case, each $\mathbf{dom}(x)$ is equipped with a total order. If $\mathcal{V} = \{x_1, \dots, x_k\}$, a lexicographic order $\langle x_{i_1}, \dots, x_{i_\ell} \rangle$ for $\ell \leq k$ means that two valuations θ_1, θ_2 are first ranked on x_{i_1}, and if they have the same rank on x_{i_1}, then they are ranked on x_{i_2}, and so on. This ordering can be naturally encoded by first taking a function $f_x : \mathbf{dom}(x) \to \mathbb{R}$ that captures the total order for variable x, and then defining $\texttt{rank}(\theta) := \sum_x w_x f_x(\theta(x))$, where w_x are appropriately chosen values. Suppose the size of the domain is n. Then, the weight for variable x_i can we set as $w_x = n^i$. For example, if $\mathbf{dom}(x) = \{0, 1, \dots, 9\}$, then fixing $w_{x_i} = 10^i$ allows us to compare any two tuples by looking at their score computed using the ranking function. Since this example ranking function is a monotone vertex-based ranking, Proposition 3.4 applies here as well.

We should note here that lexicographic ordering has been previously considered in the context of factorized databases.

Proposition 3.5 (due to [OZ15, BOZ12])**.** *Let Q be a natural join query with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, and $\langle x_{i_1}, \dots, x_{i_\ell} \rangle$ a lexicographic ordering of the variables in $\mathcal{V}$.*

Let $\mathcal{T}$ be a tree decomposition with fractional hypertree-width `fhw-lex` *such that $\langle x_{i_1}, \dots, x_{i_\ell} \rangle$ forms a prefix in the topological ordering of the variables in the decomposition. Then, for any input database D, we can pre-process D with $T_p = O(|D|^{\texttt{fhw-lex}})$ and $S_p = O(|D|^{\texttt{fhw-lex}})$ such that results of $Q(D)$ can be enumerated with delay $\delta = O(1)$ and space $S_e = O(1)$.*

In other words, if the lexicographic order "agrees" with the tree decomposition (in the sense that whenever x_i is before x_j in the lexicographic order, x_j can never be in a bag higher than the bag where x_i is), then it is possible to get an even better result than Theorem 3.1, by achieving constant delay $O(1)$, and constant space S_e. However, given a tree decomposition, Theorem 3.1 applies for any lexicographic ordering - in contrast to Proposition 3.5. As an example, consider the join query $Q(x, y, z) = R(x, y), S(y, z)$ and the lexicographic ordering $\langle z, x, y \rangle$. Since $\texttt{fhw} = 1$, our result implies that we can achieve $O(|D|)$ time preprocessing with delay $O(\log |D|)$. On the other hand, the optimal width of a tree decomposition that agrees with $\langle z, x, y \rangle$ is $\texttt{fhw-lex} = 2$; hence, Proposition 3.5 implies $O(|D|^2)$ preprocessing time and space. Thus, variable orderings in a decomposition fail to capture the additional challenge of user chosen lexicographic orderings. It is also not clear whether further restrictions on variable orderings in Proposition 3.5 are sufficient to capture ordered enumeration for other ranking functions (such as sum).

Bounded Ranking. A ranking function is *c-bounded* if there exists a subset $S \subseteq \mathcal{V}$ of size $|S| = c$, such that the value of `rank` depends only on the variables from S. A c-bounded ranking is related to c-determined ranking functions [KS06]: c-determined implies c-bounded, but not vice versa. For c-bounded ranking functions, we can show the following result:

Proposition 3.6. *Let Q be a natural join query with optimal fractional hypertree-width* `fhw`*. If* `rank` *is a c-bounded ranking function, then for any input D, we can pre-process D in time $T_p = O(|D|^{\texttt{fhw}+c})$ and space $S_p = O(|D|^{\texttt{fhw}+c})$ such that for any k, we can enumerate the top-k results of $Q(D)$ with $\delta = O(\log |D|)$ and $S_e = O(\min\{k, |Q(D)|\})$*

Proof. Let $\mathcal{T}$ by the optimal decomposition of Q with fractional hypertree-width `fhw`. We create a new decomposition $\mathcal{T}'$ by simply adding the variables S that determine the ranking functions in all the bags of $\mathcal{T}$. By doing this, the width of the decomposition will grow by at most an additive factor of c. To complete the proof, we need to show that `rank` is compatible with the new decomposition.

Indeed, for any node in $\mathcal{T}'$ (with the exception of the root node) we have that $S \subseteq \texttt{key}(t)$. Hence, if we fix a valuation over $\texttt{key}(t)$, the ranking function will output exactly the same score, independent of what values the other variables take. □

3.2. **The Algorithm for the Main Theorem.** At a high level, each node t in the tree decomposition will materialize, in an incremental fashion, all valuations over $\mathcal{B}_t^{\prec}$ that satisfy the join query corresponding to the subtree rooted at t. We will not store explicitly each valuation θ over $\mathcal{B}_t^{\prec}$ at every node t, but instead we use a simple recursive structure $c(v)$ that we call a *cell*. If t is a leaf, then $c(\theta) = \langle \theta, [], \bot \rangle$, where $\bot$ is used to denote a null pointer. Otherwise, suppose that t has n children $t_1, \dots, t_n$. Then, $c(\theta) = \langle \theta[\mathcal{B}_t], [p_1, \dots, p_n], q \rangle$, where p_i is a pointer to the cell $c(\theta[\mathcal{B}_{t_i}^{\prec}])$ stored at node t_i, and q is a pointer to a cell stored at node t (intuitively representing the "next" valuation in the order). We will use the notation $c(\theta)$.`FIRST()` and $c(\theta)$.`MID()` to refer to the first and middle values of the triple denoted by the cell. Given a cell $c(\theta)$, Algorithm 8 shows how to reconstruct the full tuple in constant time (dependent only on the query) by traversing the subtree rooted at node t.

```
Algorithm 1: Constructing tuple represented by a cell $c$
  input   : Cell $c$ for node $t$
  output  : Tuple output($c$)
  $L \leftarrow [c]$        /* List $L$ supports popping from the front and pushing to the back. */
  while $L$ is not empty do
      $s \leftarrow L$.POPFRONT()
      $\gamma_s \leftarrow s$.FIRST()
      foreach $p \in s$.MID() do
          $L$.PUSHBACK($*p$)        /* Push the cell at address $p$ at the end of $L$. */
  $\theta \leftarrow \Join_{s \in \mathcal{B}_t^{\prec}} \gamma_s$
  return $\theta$
```

Given a cell c, we will refer to the tuple constructed by Algorithm 8 as output(c). Next, each node t maintains one hash map $\mathfrak{Q}_t$, which maps each valuation u over `key`(t) to a *priority queue* $\mathfrak{Q}_t[u]$. The elements of $\mathfrak{Q}_t[u]$ is a pair $\langle score, c(\theta) \rangle$, where θ is a valuation over $\mathcal{B}_t^{\prec}$ such that $u = \theta[\texttt{key}(t)]$ and *score* is the value assigned to the cell by the ranking function and it is used by the priority queue. The priority queues will be the data structure that performs the comparison and ordering between different tuples. We will use an implementation of a priority queue (e.g., a Fibonacci heap [CLRS09]) with the following properties: (*i*) we can insert an element in constant time $O(1)$, (*ii*) we can obtain the min element (top) in time $O(1)$, and (*iii*) we can delete the min element (pop) in time $O(\log n)$.

Notice that it is not straightforward to rank the cells according to the valuations, since the ranking function is defined over all variables $\mathcal{V}$. However, here we can use the fact that the ranking function is compatible with the decomposition at hand. For each variable $x \in \mathcal{V}$, we designate some value from $\mathbf{dom}(x)$ as $v^\star(x)$. Given a fixed valuation u over `key`(t), we will order the valuations θ over $\mathcal{B}_t^{\prec}$ that agree with u according to the score: $\texttt{rank}(v_t^\star \circ \theta)$ where $v_t^\star = (v^\star(x_1), v^\star(x_2), \dots, v^\star(x_p))$ is a valuation over the p variables $S = \mathcal{V} \setminus \mathcal{B}_t^{\prec}$. The key intuition is that the compatibility of the ranking function with the decomposition implies that the ordering of the tuples implied by the cells in the priority queue $\mathfrak{Q}_t[u]$ will not change if we replace $v_t^\star$ with any other valuation. Thus, the comparator can use $v_t^\star$ to calculate the score which is used by the priority queue internally. We next discuss the *preprocessing* and *enumeration* phase of the algorithm.

Algorithm 2: Preprocessing Phase

```
   input  : CQ $Q$, Tree decomposition $(\mathcal{T}, \mathcal{B}_{t \in V(\mathcal{T})})$, Database $D$, Ranking
            function rank
   output : Initialized priority queues $\mathfrak{Q}_t$ and a set $H[t]$ for each node
            $t \in V(\mathcal{T})$
1  foreach $t \in V(\mathcal{T})$ do
2      materialize the bag $\mathcal{B}_t$        /* $R_t$ denotes the materialized relation for node $t$ */
3  full reducer pass on materialized bags in $\mathcal{T}$
4  forall $t \in V(\mathcal{T})$ in post-order traversal do
5      $H[t] \leftarrow$empty set
6      foreach valuation $\theta$ in the relation corresponding to $t$ do
7          $u \leftarrow \theta[\texttt{key}(t)]$
8          if $\mathfrak{Q}_t[u]$ is NULL then
9              $\mathfrak{Q}_t[u] \leftarrow$ new priority queue
10         $\ell \leftarrow []$                                  /* $\ell$ is a list of pointers */
11         foreach child $s$ of $t$ do
12             $\ell.\texttt{INSERT}(\mathfrak{Q}_s[\theta[\texttt{key}(s)]].\texttt{TOP}())$
13         $c \leftarrow \langle \theta, \ell, \bot \rangle\rangle$
14         $H[t].\texttt{INSERT}(\text{output}(c))$
15         $\mathfrak{Q}_t[u].\texttt{INSERT}(\langle \texttt{rank}(v_t^\star \circ \theta), c)$
```

Preprocessing. Algorithm 2 consists of two steps. The first step works exactly as in the case where there is no ranking function: each bag $\mathcal{B}_t$ is computed and materialized, and then we apply a full reducer pass to remove all tuples from the materialized bags that will not join in the final result. The second step initializes the hash map with the priority queues for every bag in the tree. We traverse the decomposition in a bottom up fashion (post-order traversal), and do the following. For a leaf node t, notice that the algorithm does not enter the loop in line 11, so each valuation θ over $\mathcal{B}_t$ is added to the corresponding queue as the triple $\langle \theta, [], \bot \rangle$. For each non-leaf node t, we take each valuation v over $\mathcal{B}_t$ and form a valuation (in the form of a cell) over $\mathcal{B}_t^{\prec}$ by using the valuations with the largest rank from its children (we do this by accessing the top of the corresponding queues in line 11). The cell is then added to the corresponding priority queue of the bag. Observe that the root node r has only one priority queue, since $\texttt{key}(r) = \{\}$.

Example 3.7. As a running example, we consider the natural join query $Q(x, y, z, p) = R_1(x, y), R_2(y, z), R_3(z, p), R_4(z, u)$ where the ranking function is the sum of the weights of each input tuple. Consider the following instance D and decomposition $\mathcal{T}$ for our running example.

Algorithm 3: Enumeration Phase

```
input  : CQ $Q$, Tree decomposition $(\mathcal{T}, \mathcal{B}_{t \in V(\mathcal{T})})$, Database $D$, Ranking function
         rank, Initialized priority queues $\mathfrak{Q}_t$ and a set $H[t]$ for each node $t \in V(\mathcal{T})$
output : Enumerates $Q(D)$ sorted according to rank
procedure ENUM()
    while $\mathfrak{Q}_r[()]$ is not empty do
        print output($\mathfrak{Q}_r[()]$.TOP())
        TOPDOWN($\mathfrak{Q}_r[()]$.TOP(), $r$)
procedure TOPDOWN($c, t$)
    /* $c = \langle \theta, [p_1, \dots, p_k], \mathsf{next} \rangle$ */
    $u \leftarrow \theta[\mathtt{key}(t)]$                                /* $\theta = c$.FIRST() */
    if next $== \bot$ then
        $b \leftarrow \mathfrak{Q}_t[u]$.POP()
        foreach child $t_i$ of $t$ do
            $p'_i \leftarrow$TOPDOWN($*p_i, t_i$)                     /* $p_i = c$.MID()$[i]$ */
            if $p'_i \neq \bot$ then
                $c' \leftarrow \langle \theta, [p_1, \dots, p'_i, \dots p_k], \bot \rangle\rangle$
                if output$(c') \notin H[t]$          /* avoiding addition of duplicate cells */
                 then
                    $H[t]$.INSERT(output($c'$))
                $\mathfrak{Q}_t[u]$.INSERT(($\langle$rank$(v_t^\star \circ \theta), c'$)
        if $t$ is not the root then
            next $\leftarrow$ ADDRESS_OF($\mathfrak{Q}_t[u]$.TOP())      /* next is an alias for $c$.LAST() */
    return next
```

R_1

id	$\mathbf{w_1}$	$\mathbf{x}$	$\mathbf{y}$
1	1	1	1
2	2	2	1

R_2

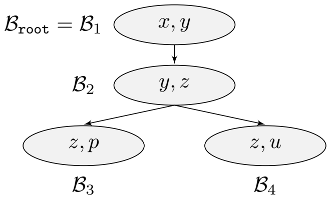
id	$\mathbf{w_2}$	$\mathbf{y}$	$\mathbf{z}$
1	1	1	1
2	1	3	1

R_3

id	$\mathbf{w_3}$	$\mathbf{z}$	$\mathbf{p}$
1	1	1	1
2	4	1	2

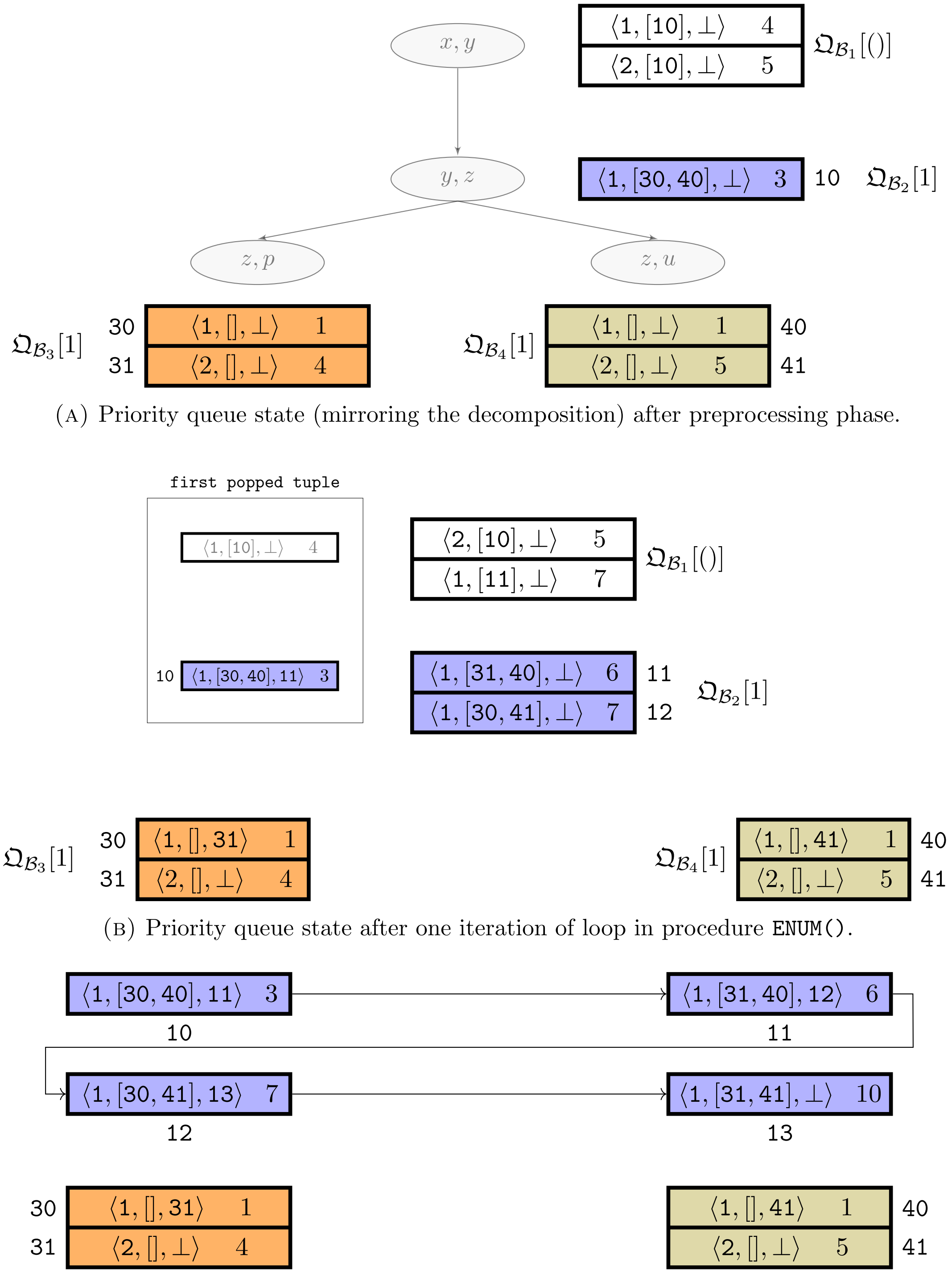
R_4

id	$\mathbf{w_3}$	$\mathbf{z}$	$\mathbf{u}$
1	1	1	1
2	5	1	2



For the instance shown above and the query decomposition that we have fixed, relation R_i covers bag $\mathcal{B}_i, i \in [4]$. Each relation has size $N = 2$. Since the relations are already materialized, we only need to perform a full reducer pass, which can be done in linear time. This step removes tuple $(3, 1)$ from relation R_2 as it does not join with any tuple in R_1.

Figure 1(A) shows the state of priority queues after the pre-processing step. For convenience, θ in each cell $\langle \theta, [p_1, \dots, p_k], \mathsf{next} \rangle$ is shown using the primary key of the tuple and pointers p_i and next are shown using the address of the cell it points to. For example, the tuple $(id = 1, \mathbf{w_2} = 1, \mathbf{y} = 1, \mathbf{z} = 1)$ in relation R_2 is shown as cell $\langle 1, [30, 40], \bot \rangle$ 3 with


x, y
y, z
z, p
z, u
$\langle 1, [10], \bot\rangle$ 4
$\langle 2, [10], \bot\rangle$ 5
$\mathfrak{Q}_{\mathcal{B}_1}[()]$
$\langle 1, [30, 40], \bot\rangle$ 3
10
$\mathfrak{Q}_{\mathcal{B}_2}[1]$
$\mathfrak{Q}_{\mathcal{B}_3}[1]$
30
31
$\langle 1, [], \bot\rangle$ 1
$\langle 2, [], \bot\rangle$ 4
$\mathfrak{Q}_{\mathcal{B}_4}[1]$
$\langle 1, [], \bot\rangle$ 1
$\langle 2, [], \bot\rangle$ 5
40
41
first popped tuple
$\langle 1, [10], \bot\rangle$ 4
10
$\langle 1, [30, 40], 11\rangle$ 3
$\langle 2, [10], \bot\rangle$ 5
$\langle 1, [11], \bot\rangle$ 7
$\mathfrak{Q}_{\mathcal{B}_1}[()]$
$\langle 1, [31, 40], \bot\rangle$ 6
$\langle 1, [30, 41], \bot\rangle$ 7
11
12
$\mathfrak{Q}_{\mathcal{B}_2}[1]$
$\mathfrak{Q}_{\mathcal{B}_3}[1]$
30
31
$\langle 1, [], 31\rangle$ 1
$\langle 2, [], \bot\rangle$ 4
$\mathfrak{Q}_{\mathcal{B}_4}[1]$
$\langle 1, [], 41\rangle$ 1
$\langle 2, [], \bot\rangle$ 5
40
41
$\langle 1, [30, 40], 11\rangle$ 3
10
$\langle 1, [31, 40], 12\rangle$ 6
11
$\langle 1, [30, 41], 13\rangle$ 7
12
$\langle 1, [31, 41], \bot\rangle$ 10
13
30
31
$\langle 1, [], 31\rangle$ 1
$\langle 2, [], \bot\rangle$ 4
$\langle 1, [], 41\rangle$ 1
$\langle 2, [], \bot\rangle$ 5
40
41

(A) Priority queue state (mirroring the decomposition) after preprocessing phase.

(B) Priority queue state after one iteration of loop in procedure `ENUM()`.

(C) The materialized output stored at subtree rooted at $\mathcal{B}_2$ after enumeration is complete.

FIGURE 1. Preprocessing and enumeration phase for Example 1.1. Each cell is assigned a memory addressed (written next to the cell). Pointers in cells are populated with the memory address of the cell they are pointing to. Cells are color coded according to the bag (white for root bag, blue for $\mathcal{B}_2$, orange for $\mathcal{B}_3$ and olive for $\mathcal{B}_4$.)

$\theta = id = 1$ and an empty next. The cell in a memory location is followed by the score of the tuple formed by creating the tuple from the pointers in the cell recursively. For instance, the score of the tuple formed by joining $(\mathbf{y} = 1, \mathbf{z} = 1) \in R_2$ with $(\mathbf{z} = 1, \mathbf{p} = 1)$ from R_3 and $(\mathbf{z} = 1, \mathbf{1} = 1)$ in R_4 is $1 + 1 + 1 = 3$ (shown as $\boxed{\langle 1, [30, 40], \bot\rangle \quad 3}$ in the figure)[2]. The pointer addresses (which have been chosen arbitrarily) 30 and 40 refer to the topmost cell in the priority queue for $\mathcal{B}_3$ and $\mathcal{B}_4$. Each cell in every priority queue points to the top element of the priority queue of child nodes that it joins with. Note that since both tuples in R_1 join with the sole tuple from R_2, they point to the same cell.

Lemma 3.8. *The runtime of Algorithm 2 is $O(|D|^{\mathtt{fhw}})$. Moreover, at the end of the algorithm, the resulting data structure has size $O(|D|^{\mathtt{fhw}})$.*

Proof. It is known that the materialization of each bag can be done in time $O(|D|^{\mathtt{fhw}})$, and the full reducer pass is linear in the size of the bags [Yan81]. For the second step of the preprocessing algorithm, observe that for each valuation in a bag, the algorithm performs only a constant number of operations (the number of children in the tree plus one), where each operation takes a constant time (since insert and top can be done in $O(1)$ time for the priority queue). Hence, the second step needs $O(|D|^{\mathtt{fhw}})$ time as well.

Regarding the space requirements, it is easy to see that the data structure uses only constant space for every valuation in each bag, hence the space is bounded by $O(|D|^{\mathtt{fhw}})$. □

Enumeration. Algorithm 3 presents the algorithm for the enumeration phase. The heart of the algorithm is the procedure $\texttt{TOPDOWN}(c, t)$. The key idea of the procedure is that whenever we want to output a new tuple, we can simply obtain it from the top of the priority queue in the root node (node r is the root node of the tree decomposition). Once we do that, we need to update the priority queue by popping the top, and inserting (if necessary) new valuations in the priority queue. This will be recursively propagated in the tree until it reaches the leaf nodes. Observe that as the new candidates are being inserted, the next pointer of cells c at some node of the decomposition are being updated by pointing to the topmost element in the priority queue of its children. This chaining materializes the answers for the particular bag that can be reused.

Example 3.9. Figure 1b shows the state of the data structure after one iteration in ENUM(). The first answer returned to the user is the topmost tuple from $\mathfrak{Q}_{\mathcal{B}_1}[()]$ (shown in the box labeled first popped tuple). Cell $\boxed{\langle 1, [10], \bot\rangle \quad 4}$ is popped from $\mathfrak{Q}_{\mathcal{B}_1}[()]$ (after satisfying if condition on line 8 since next is $\bot$). We recursively call TOPDOWN for child node $\mathcal{B}_2$ with cell $\boxed{\langle 1, [30, 40], \bot\rangle \quad 3}$ as the function argument (since that is the cell at memory address 10). Recall that $\boxed{\langle 1, [30, 40], \bot\rangle \quad 3}$ was created and pushed into the priority queue $\mathfrak{Q}_{\mathcal{B}_2}[1]$ during the preprocessing phase. The next for this cell is also $\bot$ and we pop it from $\mathfrak{Q}_{\mathcal{B}_2}[1]$. At this point, $\mathfrak{Q}_{\mathcal{B}_2}[1]$ is empty. The next recursive call is for $\mathcal{B}_3$ with $\boxed{\langle 1, [], \bot\rangle \quad 1}$ (cell at memory adress 30). The least ranked tuple but larger than $\boxed{\langle 1, [], \bot\rangle \quad 1}$ in $\mathfrak{Q}_{\mathcal{B}_3}[1]$ is the cell at address 31 . Thus, next for $\boxed{\langle 1, [], \bot\rangle \quad 1}$ is updated to 31 (on line 19) and cell at memory address 31 (which is $\boxed{\langle 2, [], \bot\rangle \quad 4}$) is returned, leading to the creation and insertion of $\boxed{\langle 1, [31, 40], \bot\rangle \quad 6}$ cell in $\mathfrak{Q}_{\mathcal{B}_2}[1]$ on line 17. Similarly, we get the other cell in $\mathfrak{Q}_{\mathcal{B}_2}[1]$ after making a recursive call to $\mathcal{B}_4$. After both the calls are over for node $\mathcal{B}_2$, the topmost cell at

[2] Note that since our ranking function is sum, we use $v^\star(x) = 0$ for each variable. This allows us to only look at the "partial" score of tuples that join in a particular subtree.

$\mathfrak{Q}_{\mathcal{B}_2}[1]$ is cell at memory address 11 ,which is set as the next (on line 19) for $\boxed{\langle 1, [30, 40], \bot\rangle \quad 3}$ (changing it into $\boxed{\langle 1, [30, 40], 11\rangle \quad 3}$), terminating one full iteration.

Let us now look at the second iteration of ENUM(). The tuple returned is top element of $Q_{\mathcal{B}_1}[()]$ which is $\boxed{\langle 2, [10], \bot\rangle \quad 5}$. However, the function TOPDOWN() with $\boxed{\langle 2, [10], \bot\rangle \quad 5}$ does not recursively go all the way down to leaf nodes. Since $\boxed{\langle 1, [30, 40], 11\rangle \quad 3}$ already has next populated, we insert $\boxed{\langle 2, [11], \bot\rangle \quad 5}$ in $Q_{\mathcal{B}_1}[()]$ completing the iteration. This demonstrates the benefit of materializing ranked answers at each node in the tree. As the enumeration continues, we are materializing the output of each subtree on-the-fly that can be reused by other tuples in the root bag.

New candidates are inserted to the priority queue using the logic on line 17 of Algorithm 3. Given a bag s with k children $s_1, \ldots, s_k$ and a cell c, the algorithm increments the pointers $p_1, \ldots p_k$ one at a time while keeping the remaining pointers fixed. Indeed as Figure 1(B) shows, initially, only $\boxed{\langle 1, [30, 40], \bot\rangle \quad 3}$ was present in $\mathfrak{Q}_{\mathcal{B}_2}[1]$ but it generated two cells $\boxed{\langle 1, [31, 40], \bot\rangle \quad 6}$ (observe that 30 was incremented to 30 but the second pointer still remains 40) and $\boxed{\langle 1, [30, 41], \bot\rangle \quad 7}$ (observe that 40 was incremented to 41 but the first pointer still remains 30). When these two cells are popped, they will increment pointers and both of them will generate $\boxed{\langle 1, [31, 41], \bot\rangle \quad 10}$. This where the set H helps us by ensuring that duplicates cells are not inserted into the priority queue (via the check on line 14. While each cell can generate k new candidates, in the worst-case, each cell can be generated at most k times and inserted into the priority queue. These duplicates are removed by the check on line line 14. Since the query size is a constant, we may only need to pop a constant number of times in the worst-case and thus affects the delay guarantee only by a constant factor.

Lemma 3.10. *Algorithm 3 enumerates $Q(D)$ with delay $\delta = O(\log |D|)$.*

Proof. In order to show the delay guarantee, it suffices to prove that procedure TOPDOWN takes $O(\log |D|)$ time when called from the root node, since getting the top element from the priority queue at the root node takes only $O(1)$ time.

Indeed, TOPDOWN traverses the tree decomposition recursively. The key observation is that it visits each node in $\mathcal{T}$ exactly once. For each node, if next is not $\bot$, the processing takes time $O(1)$. If next $== \bot$, it will perform a constant number of pops – with cost $O(\log |D|)$ – and a number of inserts equal to the number of children of the node in the tree $\mathcal{T}$. Thus, in either case the total time per node is $O(\log |D|)$. Summing up over all nodes in the tree, the total time until the next element is output will be $O(\log |D|) \cdot |V(\mathcal{T})| = O(\log |D|)$. □

We next bound the space S_e needed by the algorithm during the enumeration phase.

Lemma 3.11. *After Algorithm 3 has enumerated k tuples, the additional space used by the algorithm is $S_e = O(\min\{k, |Q(D)|\})$.*

Proof. The space requirement of the algorithm during enumeration comes from the size of the priority queues at every bag in the decomposition. Since we have performed a full reducer pass over all bags during the preprocessing phase, and each bag t stores in its priority queues all valuations over $\mathcal{B}_t^{\prec}$, it is straightforward to see that the sum of the sizes of the priorities queues in each bag is bounded by $O(|Q(D)|)$.

To obtain the bound of $O(k)$, we observe that for each tuple that we output, the TOPDOWN procedure adds at every node in the decomposition a constant number of new tuples in one of the priority queues in this node (equal to the number of children). Similarly, for the set H that ensures no duplicate cells are added, we also add a constant number of tuples at most. Hence, at most $O(1)$ amount of data will be added in the data structure between two consecutive tuples are output. Thus, if we enumerate k tuples from $Q(D)$, the increase in space will be $k \cdot O(1) = O(k)$. □

Chaining of cells. Observe that as `TOPDOWN` is called recursively, the `next` of the cells is continuously being updated. This chaining is critical to achieving good delay guarantees. Intuitively, chaining of cells at a bag allows materialization of the the join result of the subquery rooted at that bag in sorted order. Thus, repeated computation is not being performed and cells at the parent of a bag can re-use the sorted materialization. For example, Figure 1(C) shows the eventual sequence of pointers at node $\mathcal{B}_2$ which is the ranked materialized output of the subtree rooted at $\mathcal{B}_2$. The pointers between cells are added to emphasize the chained order. The reader can observe that the score for the cells highlighted in blue are also in increasing order.

Finally, we show that the algorithm correctly enumerates all tuples in $Q(D)$ in sorted order according to the ranking function.

Lemma 3.12. *Algorithm 3 enumerates $Q(D)$ in sorted order according to* `rank`*.*

Proof. We will prove our claim by induction on post-order traversal of the decomposition and use the compatibility property of the ranking function with the decomposition at hand. We use R_s to denote the relation corresponding to a node s and $\texttt{OUT}(\mathcal{B}_s^\prec, u)$ for any non-root node[3] to denote the ranked materialized output of $(\bowtie_{t \in \mathcal{B}_s^\prec} (R_t \ltimes u))$, where u is a tuple defined over $\texttt{key}(s)$ and $\ltimes$ is the standard semijoin operator [BG81]. The ranking is done according to the function $\texttt{rank}_{v^\star_S}$ where $S = \mathcal{V} \setminus \mathcal{B}_s^\prec$.

We will show that for each node s, the algorithm generates $\texttt{OUT}(\mathcal{B}_s^\prec, u)$ in sorted order according to the function $\texttt{rank}_{v^\star_S}$. Since $\texttt{OUT}(\mathcal{B}_s^\prec, u)$ is a list, we will frequently use the notation $\texttt{OUT}(\mathcal{B}_s^\prec, u)[\ell]$ to denote the cell at ℓ^{th} location in the list. First, we prove the following claim.

Claim 3.13. *For any node s and tuple u defined over $\texttt{key}(s)$, Algorithm 3 materializes $\texttt{OUT}(\mathcal{B}_s^\prec, u)$ in the sorted order according to $\texttt{rank}_{v^\star_S}$ where $S = \mathcal{V} \setminus \mathcal{B}_s^\prec$.*

Base Case. Let $v^\star$ be the valuation over $\mathcal{V} \setminus \mathcal{B}_s$ according to definition of decomposability. We insert each valuation θ in the relation R_s with score $\texttt{rank}(v^\star \circ \theta)$ (as shown in line 15 of Algorithm 2). We now argue that the valuations from the priority queue are popped in the sorted order. Consider two valuations θ_1 and θ_2 that are popped successively. Note that $\theta_1[\texttt{key}(s)] = \theta_2[\texttt{key}(s)]$ There are two cases to consider: either $\texttt{rank}(v^\star \circ \theta_2) > \texttt{rank}(v^\star \circ \theta_1)$ or $\texttt{rank}(v^\star \circ \theta_2) = \texttt{rank}(v^\star \circ \theta_1)$. The first case unambiguously guarantees that $\theta_2 \succeq \theta_1$ since the score for θ_1 is strictly smaller. However, if $\texttt{rank}(v^\star \circ \theta_2) = \texttt{rank}(v^\star \circ \theta_1)$, it is not immediately clear whether $\theta_2 \succeq \theta_1$ or $\theta_2 \succeq \theta_1$ because there could be a different valuation $v^\#$ for which $\texttt{rank}(v^\# \circ \theta_2) \neq \texttt{rank}(v^\# \circ \theta_1)$. We argue that such a $v^\#$ cannot exist. Indeed, Definition 2.3 guarantees that if $\texttt{rank}(v^\star \circ \theta_2) = \texttt{rank}(v^\star \circ \theta_1)$, then it must also be equal for any other valuation over $\mathcal{V} \setminus \mathcal{B}_s$. In other words, all output tuples $t \in Q(D)$, such that $t[\mathcal{B}_s] = \theta_1$ or $t[\mathcal{B}_s] = \theta_2$ are guaranteed to have the score, and thus, we can safely use the ordering $\theta_2 \succeq \theta_1$ for θ_1 and θ_2. Since the preprocessing phase already initializes the priority

[3] For the root node r, since $\texttt{key}(r) = \{\}$, we define $\texttt{OUT}(\mathcal{B}_r^\prec, \{\}) = \bowtie_{t \in \mathcal{B}_r^\prec} R_t = Q(D)$

queue, the pop operation will insert the tuple in $\texttt{OUT}(\mathcal{B}_s^{\prec}, \theta[\texttt{key}(s)])$ by populating the $\texttt{next}$ of the cell corresponding to θ correctly.

Inductive Case. Consider some node s in the post-order traversal with children $s_1, \dots s_m$. By the induction hypothesis, the ordering of $\texttt{OUT}(\mathcal{B}_{s_i}^{\prec}, b)$ for each valuation b over $\texttt{key}(s_i)$ is generated in sorted order for ranking function $\texttt{rank}_{v^\star_{S_i}}$ where $S_i = \mathcal{V} \setminus \mathcal{B}_{s_i}^{\prec}$. Let θ be a tuple in R_s and let $u = \theta[\texttt{key}(s)]$. Observe that the preprocessing phase creates a cell for θ whose pointer list $[p_1, \dots p_m]$ is the address of the cell at location 0 of the materialized list of $\texttt{OUT}(\mathcal{B}_{s_i}^{\prec}, \theta[\texttt{key}(s_i)])$. We claim that this is the least ranked tuple that can be formed over $\mathcal{B}_s^{\prec}$. Let c_i^0 denote the cell at location 0 for list $\texttt{OUT}(\mathcal{B}_{s_i}^{\prec}, \theta[\texttt{key}(s_i)])$. If any pointer p_i points to any other cell (say d_i) present at a different location in the list $\texttt{OUT}(\mathcal{B}_{s_i}^{\prec}, \theta[\texttt{key}(s_i)])$, we can create a smaller ranked tuple by changing p_i to point to the first cell in the list. In other words, since $\textsf{output}(d_i) \succeq \textsf{output}(c_i^0)$ (i.e., the first cell is the least ranked, which follows from the correctness of $\texttt{OUT}(\mathcal{B}_{s_i}^{\prec}, \theta[\texttt{key}(s_i)])$), it holds that

$$\texttt{rank}(\theta \circ v^\star \circ \textsf{output}(d_1) \circ \dots \textsf{output}(d_i) \cdots \circ \textsf{output}(d_m)) \geq$$
$$\texttt{rank}(\theta \circ v^\star \circ \textsf{output}(d_1) \circ \dots \textsf{output}(c_i^0) \cdots \circ \textsf{output}(d_m))$$

Here, we use the fact that $\texttt{rank}$ is $\mathcal{B}_{s_i}^{\prec} \setminus \texttt{key}(s_i)$-decomposable conditioned on $\texttt{key}(s_i)$. Note that no sibling of s_i can have any common variables with s_i other than the $\texttt{key}(s)$ which have already been fixed. This proves that the first tuple returned by the pop operation on the priority queue for key $\theta[\texttt{key}(s)]$ at node s will be correct, which is then added to the list $\texttt{OUT}(\mathcal{B}_s^{\prec}, \theta[\texttt{key}(s)])$.

Next, we proceed to show the correctness for an arbitrary step in the execution. Suppose c is the last cell popped at line 9. From line 10-17, one may observe that a new candidate is pushed into priority queue for key by incrementing pointers to $\texttt{OUT}(\mathcal{B}_{s_i}^{\prec}, \theta[\texttt{key}(s_i)])$ one at a time for each child bag $\mathcal{B}_{s_i}$, while keeping the remainder of the cell content fixed (line 13). Let $c.\texttt{MID()}[i]$ point to the cell $\texttt{OUT}(\mathcal{B}_{s_i}^{\prec}, \theta[\texttt{key}(s_i)])[\ell_i]$. We will use $c_i = \ell_i$ as a shorthand to denote this information. Then, the m candidates generated by the logic will contain pointers that point to the following index location

$$\begin{aligned}\mathcal{L} = \ &\ell_1 + 1, \ell_2, \ell_3, \dots, \ell_m,\\ &\ell_1, \ell_2 + 1, \ell_3, \dots, \ell_m,\\ &\ell_1, \ell_2, \ell_3 + 1, \dots, \ell_m,\\ &\dots\\ &\ell_1, \ell_2, \ell_3, \dots, \ell_m + 1\end{aligned}$$

Suppose that up until now, the algorithm has generated the ranked output $\texttt{OUT}(\mathcal{B}_s^{\prec}, \theta[\texttt{key}(s)])$ in sorted order and the next smallest cell that must be popped on line 9 of Algorithm 3 is $c^{\succ}$. Let the pointer at location i in $c^{\succ}.\texttt{MID()}$ point to index $\ell_i^{\succ}$ in the list $\texttt{OUT}(\mathcal{B}_{s_i}^{\prec}, \theta[\texttt{key}(s_i)])$ (i.e., $c_i^{\succ} = \ell_i^{\succ}$). We need to show that $c^{\succ}$ is a cell with $c^{\succ}.\texttt{MID()}$ as one of the candidates in $\mathcal{L}$ or is already in the priority queue. For the sake of contradiction, suppose there is a cell c' with $c'.\mathtt{MID()}[i] = \mathsf{address\_of}(\mathtt{OUT}(\mathcal{B}^{\prec}_{s_i}, \theta[\mathtt{key}(s_i))[\ell'_i]), i \in [m]$, that is the next smallest after c but is neither present in $\mathcal{L}$ and nor present in the priority queue. In other words, we are assuming that $\mathtt{rank}_{v^\star_S}(\mathsf{output}(c')) < \mathtt{rank}_{v^\star_S}(\mathsf{output}(c^{\succ}))$.[4] We will show that such a scenario will violate the compatibility of the ranking function. There are three possible scenarios regarding the values of $\ell^{\succ}_i$ and ℓ'_i.

(1) $\ell^{\succ}_i \leq \ell'_i$ for each $i \in [m]$. This scenario implies that $\mathtt{rank}(v^\star \circ \mathsf{output}(c')) \geq \mathtt{rank}(v^\star \circ \mathsf{output}(c^{\succ}))$. Indeed, we have that

$$\begin{aligned}
&\mathtt{rank}(v^\star \circ \theta \circ \mathsf{output}(c^{\succ}_1) \circ \ldots \mathsf{output}(c^{\succ}_m)) \\
&\qquad\leq \mathtt{rank}(v^\star \circ \theta \circ \mathsf{output}(c'_1) \circ \mathsf{output}(c^{\succ}_2) \circ \ldots \mathsf{output}(c^{\succ}_m)) \\
&\qquad\leq \mathtt{rank}(v^\star \circ \theta \circ \mathsf{output}(c'_1) \circ \mathsf{output}(c'_2) \circ \ldots \mathsf{output}(c^{\succ}_m)) \\
&\qquad\ldots \\
&\qquad\leq \mathtt{rank}(v^\star \circ \theta \circ \mathsf{output}(c'_1) \circ \mathsf{output}(c'_2) \circ \ldots \mathsf{output}(c'_m))
\end{aligned}$$

Each inequality is a successive application of $(\mathcal{B}^{\prec}_{s_i} \setminus \mathtt{key}(s_i))$-decomposability since $\mathsf{output}(c'_i) \succeq \mathsf{output}(c^{\succ}_i)$, which follows from the assumed ordering correctness of $\mathtt{OUT}(\mathcal{B}^{\prec}_{s_i}, \theta[\mathtt{key}(s_i)])$.
Thus, it cannot be the case that $\mathtt{rank}_{v^\star_S}(\mathsf{output}(c')) < \mathtt{rank}_{v^\star_S}(\mathsf{output}(c^{\succ}))$ without violating the compatibility of the ranking function.

(2) $\ell^{\succ}_i > \ell'_i$ for each $i \in [m]$. This scenario implies that $\mathtt{rank}_{v^\star_S}(\mathsf{output}(c')) < \mathtt{rank}_{v^\star_S}(\mathsf{output}(c^{\succ}))$ and thus, violates our assumption that all cells ranked smaller than $\mathsf{output}(c^{\succ})$ have been generated correctly in sorted order.

(3) ℓ' and $\ell^{\succ}$ are incomparable. It is easy to see that all candidates in $\mathcal{L}$ dominate[5] pointer locations of c (recall that c is last cell popped from the priority queue) but are incomparable to each other. Also, the only way to generate new candidate tuples is through the logic in line 10-17. Thus, if c' is not in the priority queue, there are two possibilities. Either there is some cell c'' in the priority queue such that $\mathsf{output}(c'')$ dominates $\mathsf{output}(c')$ and thus, $\mathtt{rank}_{v^\star_S}(\mathsf{output}(c'')) \leq \mathtt{rank}_{v^\star_S}(\mathsf{output}(c'))$. c'' will eventually generate c' via a chain of cells that successively dominate each other. As c was popped before c'', it follows that $\mathtt{rank}_{v^\star_S}(\mathsf{output}(c)) \leq \mathtt{rank}_{v^\star_S}(\mathsf{output}(c'')) \leq \mathtt{rank}_{v^\star_S}(\mathsf{output}(c'))$, a contradiction to our assumption that c' is the next tuple that must be popped after c, which cannot happen until c'' is popped. The second possibility is that there is no such c'', which will mean that $c^{\succ}$ and c' are generated in the same for loop on line 13. But this would imply that c' is in the priority queue. Thus, both these cases violate one of our assumptions made.

Therefore, it cannot be the case that $\mathtt{rank}_{v^\star_S}(\mathsf{output}(c')) < \mathtt{rank}_{v^\star_S}(\mathsf{output}(c^{\succ}))$ which proves the ordering correctness for node s. Recall that for the root node r, we have $\mathtt{key}(r) = \{\}$ and thus, we have only a single priority queue. Since Claim 3.13 holds for all nodes in the

[4] Note that $\mathtt{rank}_{v^\star_S}(\mathsf{output}(c'))$ cannot be equal to $\mathtt{rank}_{v^\star_S}(\mathsf{output}(c^{\succ}))$ because otherwise, the order in which the cells are popped from the priority queue can be used to establish the total order (similar to the base case).

[5] Given two tuples t_1 and t_2 defined over the same set of variables V, we say that t_1 *dominates* t_2 if $t_1[v] \geq t_2[v]$ for all $v \in V$.

decomposition, we have that $\texttt{OUT}(\mathcal{B}_r^{\prec}, \{\}) = (\Join_{t \in \mathcal{B}_r} R_t)$ (which is nothing but $Q(D)$) will store the sorted output according to $\texttt{rank}_{v_S^\star}$ where $S = \mathcal{V} \setminus \mathcal{B}_r^{\prec}$. However, since $\mathcal{V} = \mathcal{B}_r^{\prec}$, we have that for the root node $\texttt{rank}_{v_S^\star} = \texttt{rank}$, as desired. Thus, $Q(D)$ is enumerated in sorted order according to $\texttt{rank}$. □

4. Extensions

In this section, we describe two extensions of Theorem 3.1 and how it can be used to further improve the main result.

4.1. Ranked Enumeration of UCQs.

We begin by discussing how ranked enumeration of full UCQs can be done. The first observation is that given a full UCQ $\varphi = \varphi_1 \cup \dots \varphi_\ell$, if the ranked enumeration of each φ_i can be performed efficiently, then we can perform ranked enumeration for the union of query results. This can be achieved by applying Theorem 3.1 to each φ_i and introducing another priority queue that compares the score of the answer tuples of each φ_i, pops the smallest result, and fetches the next smallest tuple from the data structure of φ_i accordingly. Although each $\varphi_i(D)$ does not contain duplicates, it may be the case that the same tuple is generated by multiple φ_i. Thus, we need to introduce a mechanism to ensure that all tuples with the same weight are enumerated in a specific order. Fortunately, this is easy to accomplish by modifying Algorithm 3 to enumerate all tuples with the same score in lexicographic increasing order. The choice of lexicographic ordering as a tie-breaking criteria is not the only valid choice. As long as the ties are broken consistently, other ranking functions can also be used. This ensures that tuples from each φ_i also arrive in the same order. Since each φ_i is enumerable in ranked order with delay $O(\log |D|)$ and the overhead of the priority queue is $O(\ell)$ (priority queue contains at most one tuple from each φ_i), the total delay guarantee is bounded by $O(\ell \cdot \log |D|) = O(\log |D|)$ as the query size is a constant. The space usage is determined by the largest fractional hypertree-width across all decompositions of subqueries in φ. This immediately leads to the following result.

Theorem 4.1. *Let $\varphi = \varphi_1 \cup \dots \varphi_\ell$ be a full UCQ. Let $\texttt{fhw}$ denote the fractional hypertree-width of all decompositions across all CQs φ_i, and $\texttt{rank}$ be a ranking function that is compatible with the decomposition of each φ_i. Then, for any input database D, we can pre-process D in time and space,*

$$T_p = O(|D|^{\texttt{fhw}}) \qquad S_p = O(|D|^{\texttt{fhw}})$$

such that for any k, we can enumerate the top-k tuples of $\varphi(D)$ with

$$\text{delay } \delta = O(\log |D|) \qquad \text{space } S_e = O(\min\{k, |\varphi(D)|\})$$

Algorithm 4 shows the enumeration algorithm. It outputs one output tuple t in every iteration and line 12-13 pop out all duplicates of t in the queue. Recall that since $Q = Q_1 \cup \dots Q_\ell$, there can be at most ℓ duplicates for some constant ℓ. $\texttt{ENUM}(Q_i)$ is the invocation of $\texttt{ENUM}()$ procedure from Algorithm 3 on query Q_i.

The comparison function for priority queues in Algorithm 3 for each subquery Q_i of Q is modified in the following way. Consider two tuples t_1 and t_2 with schema $(x_1, x_2, \dots, x_n)$ and scores $\texttt{rank}(t_1)$ and $\texttt{rank}(t_2)$ respectively.

Algorithm 4: Preprocessing and Enumeration Phase

```
input  : Full UCQ $Q = Q_1 \cup \cdots \cup Q_\ell$, Tree decomposition $(\mathcal{T}_i, \mathcal{B}^i_{t \in V(\mathcal{T}_i)})$ for each
         CQ $Q_i$, Database $D$, Ranking function rank
output : Enumerate $Q(D)$ in sorted order according to rank
procedure PREPROCESS($Q, D,$ rank$, (\mathcal{T}_i, \mathcal{B}^i_{t \in V(\mathcal{T}_i)})$)
    Apply Algorithm 2 to all $Q_i$
    QUEUE ← $\emptyset$
    for $i \in \{1, \dots, \ell\}$ do
        QUEUE.PUSH(ENUM($Q_i$))  /* Initialize QUEUE with smallest candidate for each $Q_i$
         */
procedure ENUM($Q, D,$ rank$, (\mathcal{T}_i, \mathcal{B}^i_{t \in V(\mathcal{T}_i)})$)
    while QUEUE is not empty do
        $t$ ← QUEUE.POP()
        output $t$                                  /* Suppose $t$ came from subquery $Q_i$ */
        QUEUE.PUSH(ENUM($Q_i$))                        /* Push the next candidate for $Q_i$ */
        while QUEUE.TOP() == $t$ do
            QUEUE.POP()                                /* drain the queue of duplicate $t$ */
            /* Suppose duplicate $t$ came from calling ENUM($Q_j$) on subquery $Q_j$       */
            QUEUE.PUSH(ENUM($Q_j$))                    /* Push the next candidate for $Q_j$ */
```

Algorithm 5: Comparison function for priority queues

```
input  : Tuples $t_1$ and $t_2$, Ranking function rank
output : Returns the smaller ranked tuple of $t_1$ and $t_2$; break ties in lexicographic
         ordering
procedure COMPARE ($t_1, t_2$)
    if rank($t_1$) < rank($t_2$) then
        return $t_1$
    if rank($t_2$) < rank($t_1$) then
        return $t_2$
    foreach $i \in \{n, n-1, \dots, 1\}$ do
        if $\pi_{x_i}(t_1) < \pi_{x_i}(t_2)$ then
            return $t_1$
        if $\pi_{x_i}(t_2) < \pi_{x_i}(t_1)$ then
            return $t_2$
```

Comparison function in Algorithm 5 compares t_1 and t_2 based on the ranking function and tie breaks by using the lexicographic ordering of the two tuples. This ensures that all tuples with the same score arrive in a fixed from ENUM(Q_i) procedure of each subquery Q_i.

4.2. **Improving The Main Result.** Although Theorem 4.1 is a straightforward extension of Theorem 3.1, it is powerful enough to improve the pre-processing time and space of Theorem 3.1 by using Theorem 4.1 in conjunction with *data-dependent* tree decompositions. It is well known that the query result for any CQ can be answered in time $O(|D|^{\texttt{fhw}} + |Q(D)|)$ time and this is asymptotically tight [AGM13]. However, there exists another notion of width known as the *submodular width* (denoted $\texttt{subw}$) [Mar13]. It is also known that for any CQ, it holds that $\texttt{subw} \leq \texttt{fhw}$. Recent work by Abo Khamis et al. [AKNS17] presented an elegant algorithm called PANDA that constructs multiple decompositions by partitioning the input database to minimize the intermediate join size result. PANDA computes the output of any full CQ in time $O(|D|^{\texttt{subw}} \cdot \log |D| + |\texttt{OUT}|)$. In other words, PANDA takes a CQ query Q and a database D as input and produces multiple tree decompositions in time $O(|D|^{\texttt{subw}} \cdot \log |D|)$ such that each answer tuple is generated by at least one decomposition. The number of decompositions depends only on size of the query and not on D. Thus, when the query size is a constant, the number of decompositions constructed is also a constant. We can now apply Theorem 4.1 by setting φ_i as the tree decompositions produced by PANDA to get the following result.

Theorem 4.2. *Let φ be a natural join query with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, submodular width* $\texttt{subw}$*, and* $\texttt{rank}$ *be a ranking function that is compatible with each tree decomposition of φ. Then, for any input database D, we can pre-process D in time and space,*

$$T_p = O(|D|^{\texttt{subw}} \cdot \log |D|) \qquad S_p = O(|D|^{\texttt{subw}})$$

such that for any k, we can enumerate the top-k tuples of $\varphi(D)$ with

$$\textit{delay } \delta = O(\log |D|) \qquad \textit{space } S_e = O(\min\{k, |\varphi(D)|\})$$

5. Lower Bounds

In this section, we provide evidence for the near optimality of our results.

5.1. The Choice of Ranking Function.

We first consider the impact of the ranking function on the performance of ranked enumeration. We start with a simple observation that deals with the case where $\texttt{rank}$ can be accessed only through a blackbox that, given a tuple/valuation, returns its score: we call this a *blackbox* [6] ranking function. Note that all of our algorithms work under the blackbox assumption.

Proposition 5.1. *Let Q be a natural join query, and* $\texttt{rank}$ *be an arbitrary blackbox ranking function. Then, any enumeration algorithm on a database D needs $\Omega(|Q(D)|)$ calls to* $\texttt{rank}$ *in order to output the smallest tuple.*

Proof. Suppose that some algorithm returns $t \in Q(D)$ as the least ranked tuple without examining the rank of tuple $t^\star \in Q(D)$. Then, the ranking function can assign a rank to $t^\star$ such that $\texttt{rank}(t^\star) < \texttt{rank}(t)$. Therefore, any algorithm must examine the rank of each tuple in the output of the query before returning the smallest tuple. □

The above proposition shows that without any additional restrictions on the ranking function, the simple result in Proposition 2.9 that materializes and sorts the output is essentially optimal. Thus, it is necessary to exploit properties of the ranking function in order to construct better algorithms. Unfortunately, even for certain natural restrictions of ranking functions, it is not possible to do much better than the $|D|^{\rho^*}$ bound for certain queries.

One such a natural restriction is that of coordinate-decomposable functions, where we can show the following lower bound result:

[6] Blackbox implies that the score $\texttt{rank}(\theta)$ is revealed only upon querying the function.

Lemma 5.2. *Consider the query $Q(x_1, y_1, x_2, y_2) = R(x_1, y_1), S(x_2, y_2)$ and suppose* `rank` *is a blackbox ranking function that is also known to be coordinate-decomposable. Then, there exists an instance of size N such that the time required to find the smallest tuple is $\Omega(N^2)$.*

Proof. We construct an instance D as follows. For every variable we use the domain $\{a_1, \dots, a_N\}$, which we equip with the order $a_1 < a_2 < \dots < a_N$. Then, every tuple in R and S is of the form (a_i, a_{N-i+1}) for $i = 1, \dots, N$. Similarly, every tuple in S is of the form (a_i, a_{N-i+1}).

To construct a family of coordinate-decomposable ranking functions, we consider all ranking functions that are monotone w.r.t. the order of the domain $\{a_1, \dots, a_N\}$ for every variable.

We will show that any two tuples in $Q(D)$ are incomparable, in the sense that neither tuple dominates the other in all variables. Indeed, consider two distinct tuples $t_1 = (a_i, a_{N-i+1}, a_j, a_{N-j+1})$, and $t_2 = (a_k, a_{N-k+1}, a_\ell, a_{N-\ell+1})$. For the sake of contradiction, suppose t_1 dominates t_2. Then, we must have $i \geq k$ and $N - i + 1 \geq N - k + 1$, giving $i = k$. Similarly, $j = \ell$. But this contradicts our assumption that $t_1 \neq t_2$.

Therefore, a ranking function from our family can assign an arbitrary score to the N^2 tuples without violating the coordinate decomposability. Indeed, coordinate decomposability tells us that for any two $\theta_1, \theta_2 \in Q(D)$ that agree on the three variables $\{x_1, y_1, x_2, y_2\} \setminus z$ for any $z \in \{x_1, y_1, x_2, y_2\}$, if $\theta_1(z) \succeq \theta_2(z)$, then $\texttt{rank}(\theta_1) \geq \texttt{rank}(\theta_2)$. In other words, the ranking function imposes a constraint on the score only if θ_1 dominates θ_2 or vice-versa. Thus, for any non-dominating tuple pair, the ranking function is free to assign any value as the score. Applying Lemma 5.2 gives us the desired lower bound. □

Lemma 5.2 shows that for coordinate-decomposable functions, there exist queries where obtaining constant (or almost constant) delay requires the algorithm to spend superlinear time during the preprocessing step. Given this result, the immediate question is to see whether we can extend the lower bound to other CQs. We first show a simple but powerful result for coordinate-decomposable functions. Before we present the result, we need to formally define the notion of path and diameter in a hypergraph.

Definition 5.3. Given a connected hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, a path P in $\mathcal{H}$ from vertex x_1 to x_{s+1} is a vertex-edge alternate set $x_1 E_1 x_2 E_2 \dots x_s E_s x_{s+1}$ such that $\{x_i, x_{i+1}\} \subseteq E_i (i \in [s])$ and $x_i \neq x_j, E_i \neq E_j$ for $i \neq j$. Here, s is the length of the path P. The distance between any two vertices u and v, denoted $d(u, v)$, is the length of the shortest path connecting u and v. The *diameter* of a hypergraph, $\mathsf{dia}(\mathcal{H})$, is the maximum distance between all pairs of vertices.

Lemma 5.4. *Consider a full acyclic connected query Q over binary relations and* `rank` *a blackbox ranking function that is also known to be coordinate-decomposable. Then, there exists an algorithm that enumerates the result of Q in ranked order with $O(\log |D|)$ delay guarantee and $T_p = O(|D|)$ preprocessing time if and only if $\mathsf{dia}(Q) \leq 3$.*

Proof. First, note that if $\mathsf{dia}(Q) \geq 4$, then there exists a path of the form $x_1 R y_1 T z_1 U x_2 S y_2$. We can embed the hard instance from Lemma 5.2 in the following way. We use the same relations as defined in Lemma 5.2 to create relations R and S. Let the domain of z_1 be $z^\star$. We define $T(y_1, z_1) = \{(a_i, z^\star) \mid i \in [N]\}$ and $U(z_1, x_1) = \{(z^\star, a_i) \mid i \in [N]\}$. For all other relations (let us use W to denote such a relation) in the query other than R, S, T and U, we create an instance in the following way. Fix the domain of all variables other than x_1, y_1, x_2, y_2 as $a^\star$. Then, $W(p, q) = \{\mathbf{dom}(p) \times \mathbf{dom}(q)\}$. It is easy to see that size of the relation is at most N since only one of the variables can be x_1, y_1, x_2, y_2 (otherwise the query becomes cyclic) and the output of the query will be non-empty. Using the same argument as before, we get $\Omega(N^2) = \Omega(|D|^2)$ incomparable tuples, giving us a lower bound of $\Omega(|D|^2)$ to find the least ranked tuple.

If $\mathsf{dia}(Q) \leq 3$, we will show that there exists a join tree of depth one. Indeed, if there exists no join tree of depth one, then there exists a root to leaf path (say root node r, its child s, and child of s as t) of length two. The root node must also have at least two children because otherwise, one could make s as the root with r and t as its children. Let the other child of the root node be u. We also note that each leaf node bag contains exactly one variable in common with the parent and one variable that is unique to the bag of the leaf node (i.e., it does not appear in any other bag). Thus, the distance from the unique variable in $\mathcal{B}_t$ to the unique variable in $\mathcal{B}_u$ requires traversing all of the three intermediate nodes, which leads to a shortest path of length four, a contradiction. Thus, there must exists a join tree of depth one. Next, we show the compatibility of the ranking function with the join tree. The root bag is $\mathcal{B}_\mathtt{r}$-decomposable by definition since $\mathtt{key}(r) = \{\}$. Let u_i be the unique variable in bag $\mathcal{B}_i$ for node s_i. Then, $\mathcal{T}$ is u_i-decomposable conditioned on $\mathtt{key}(s_i)$. Indeed, since the ranking functions is u_i-decomposable, we can apply Proposition 2.6 by fixing $T = \{u_i\}$ and $S = \mathtt{key}(s_i) \subseteq \mathcal{V} \setminus \{u_i\}$ to obtain the desired result. Thus, Theorem 3.1 is applicable. □

Our next result characterizes a class of queries which admit efficient ranked enumeration for edge-decomposable ranking functions: these are functions that are S-decomposable for any S that is a hyperedge in the query hypergraph.

Lemma 5.5. *Consider a full acyclic query Q and a blackbox ranking function* `rank` *that is also known to be edge-decomposable. Then, if Q admits a join tree of depth one, then there exists an algorithm that enumerates the result of Q in ranked order with $O(\log|D|)$ delay and $T_p = O(|D|)$ preprocessing time.*

Proof. Consider a join tree of depth one for Q. We will show that any such decomposition is compatible with an edge-decomposable function. First, note that the for the root node r, any ranking function is $\mathcal{B}_r^{\prec}$-decomposable since $\mathtt{key}(r) = \{\}$. Consider a child node s of the root. Since `rank` is edge-decomposable, it implies that for node s, the decomposition is $\mathcal{B}_s$-decomposable. Recall that if a ranking function is $(S \cup T)$-decomposable, then it is also T-decomposable conditioned on S. Since $\mathcal{B}_s = (\mathcal{B}_s \setminus \mathtt{key}(s)) \cup \mathtt{key}(s)$ and $\mathcal{B}_s^{\prec} = \mathcal{B}_s$ as node s is a leaf, we get that for any leaf node s, the ranking function is $(\mathcal{B}_s^{\prec} \setminus \mathtt{key}(s))$-decomposable conditioned on $\mathtt{key}(s)$. Since all nodes in the decomposition are either leaf or the root, we get the compatability of the ranking function with the decomposition at hand. □

As an example, $Q(x, y, z, w) = R(x, y), S(y, z), T(z, w)$ has a decomposition of depth one where $\{y, z\}$ is the root and $\{x, y\}$ and $\{z, w\}$ are the leaves, and thus we can enumerate the result with linear preprocessing time and logarithmic delay for any edge-decomposable ranking function.

On the other hand, for the 4-path query $Q(x, y, z, w, t) = R(x, y), S(y, z), T(z, w), U(w, t)$, it is not possible to achieve this. Figure 2 shows a database instance with n tuples for the 4-path query. For the family of ranking functions, we consider all functions that are monotone with respect to the order of the tuples as depicted in the figure. Using the same argument as in Lemma 5.2, it is easy to see that any algorithm must examine the rank of

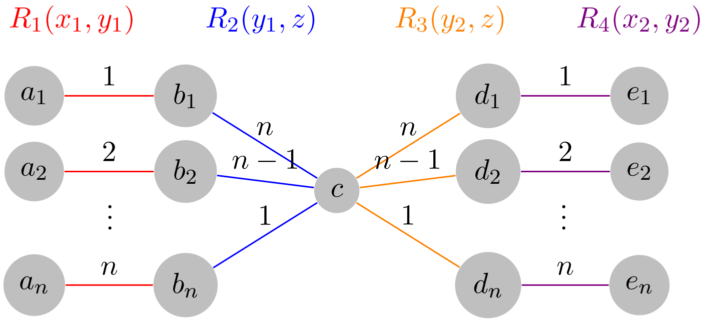


FIGURE 2. Database instance D for the 4-path query. Each edge is color coded by the relation it belongs to. Values over the edges denote the weight assigned to each tuple.

$\Omega(n^2)$ tuples in order to find the smallest one. Our last result of this section extends the idea to show a dichotomy for queries over binary relations with edge-decomposable functions.

Lemma 5.6. *Consider a full acyclic query Q over binary relations and a blackbox ranking function* `rank` *that is also known to be edge-decomposable. Then, there exists an algorithm that enumerates the result of Q in ranked order with $O(\log |D|)$ delay and $T_p = O(|D|)$ preprocessing time if and only if Q admits a join tree of depth one.*

Proof. For the one direction, Lemma 5.5 already shows the desired result (for all full acyclic CQs, and not just for binary relations) if there exists a join tree of depth one.

For the other direction, suppose that Q does not have a join tree of depth one. Then, we claim that there is a connected component in the hypergraph of Q with diameter at least four or there exist at least two connected components, each with diameter two or more. Indeed, if the connected components all have diameter one (i.e. each component only has one relation), we can pick any relation as the root and all other relations can become the leaf. Similarly, if there exists a component C with diameter two or three three, and all other components have diameter one, then the isolated relations can directly be made as the children of the root node in the join tree of C (which is guaranteed to be of depth one).

Consider the connected component Q' with diameter at least four. Then, there must exist a path of the form $x_1 R_1 y_1 R_2 z R_3 y_2 R_4 x_2$ and we can use the database instance as shown Figure 3. For all other nodes (if any) in the join tree of Q', as well as join tree of other connected components, we can create a relation per node with exactly one tuple such that $Q(D)$ is not empty and assign a uniform weight (say) 1. Using the same argument as in Lemma 5.2, it is easy to see that any correct algorithm must examine the rank of $\Omega(n^2)$ tuples in order to find the smallest one since the tuple formed by the weights of the edges for any two output tuples will be incomparable. If there are at least two connected components, each with diameter two or more, then the join tree is of the form as shown in Figure 3 (with possibly more nodes in the join tree) with a root to leaf path of length at least two. Suppose $\mathcal{B}_3$ and $\mathcal{B}_2$ belong to one component and thus have a variable in common. Similarly, $\mathcal{B}_1$ and $\mathcal{B}_4$ also have a variable in common. Now, we can modify the database instance from Figure 2 to have the schema for relations R_2 and R_3 as $R_2(y_1, z_1)$ and $R_3(y_2, z_2)$, and the domain of both z_1 and z_2 is $\{c\}$. Relations R_1 and R_2 correspond to $\mathcal{B}_3$ and $\mathcal{B}_2$, and R_3 and R_4 correspond to $\mathcal{B}_1$ and $\mathcal{B}_4$. The tuples and weights from Figure 2 remain the same. For all other nodes that may be in the join tree, we again create a relation with a single tuple,

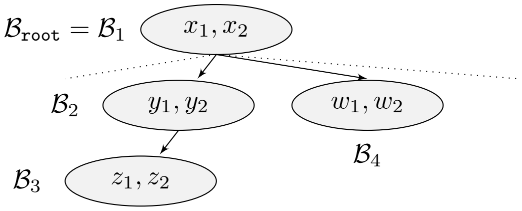


FIGURE 3. Query decomposition example with depth more than one.

such that the output of the query is non-empty. Once again, we get $\Omega(n^2)$ tuples that are incomparable when looking at the weights of the edges that form the tuple. This completes the proof. □

The results presented in this section demonstrate that small changes in the property of the ranking functions can lead to very different enumeration guarantees for the same query. For instance, for the cartesian product query $Q(x_1, y_1, x_2, y_2) = R(x_1, y_1), S(x_2, y_2)$, Lemma 5.2 showed that no linear preprocessing time and logarithmic delay algorithm can exist for coordinate-decomposable functions. However, Lemma 5.5 tells us that for edge-decomposable functions and the same query, there exists a linear preprocessing time and logarithmic delay algorithm.

While we show dichotomies for full acyclic CQs over binary relations, a complete syntactic characterization for arbitrary full acyclic CQs remains an open problem. Our results for edge-decomposable and coordinate-decomposable ranking functions show that depending on the properties of the query hypergraph, a query may or may not admit efficient algorithms. However, no other ranking functions with reasonable restrictions are known that are *intrinsically* hard. The problem of finding such natural families of ranking function that are hard intrinsically and do not admit efficient enumeration algorithms any acyclic CQs is also interesting.

5.2. **Beyond Logarithmic Delay.** Next, we examine whether the logarithmic factor that we obtain in the delay of Theorem 3.1 can be removed for ranked enumeration. In other words, is it possible to achieve constant delay enumeration while keeping the preprocessing time small, even for simple ranking functions? To reason about this, we need to describe the $X + Y$ *sorting* problem.

Given two lists of n numbers, $X = \langle x_1, x_2, \ldots, x_n \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, we want to enumerate all n^2 pairs (x_i, y_j) in ascending order of their sum $x_i + y_j$. This classic problem has a trivial $O(n^2 \log n)$ algorithm that materializes all n^2 pairs and sorts them. However, it remains an open problem whether the pairs can be enumerated faster in the RAM model. Fredman [Fre76] showed that $O(n^2)$ comparisons suffice in the nonuniform linear decision tree model, but it remains open whether this can be converted into an $O(n^2)$-time algorithm in the real RAM model. Steiger and Streinu [SS95] gave a simple algorithm that takes $O(n^2 \log n)$ time while using only $O(n^2)$ comparisons.

Conjecture 5.7 ([BCD$^+$06, DO05])**.** $X + Y$ *sorting* does not admit an $O(n^2)$ time algorithm.

In our setting, $X+Y$ sorting can be expressed as enumerating the output of the cartesian product $Q(x,y) = R(x), S(y)$, where relations R and S correspond to the sets X and Y respectively. The ranking function is $\texttt{rank}(x,y) = x+y$. Conjecture 5.7 implies that it is not possible to achieve constant delay for the cartesian product query and the sum ranking function; otherwise, a full enumeration would produce a sorted order in time $O(n^2)$.

6. Related Work

Top-k ranked enumeration of join queries has been studied extensively by the database community for both certain [LCIS05, QCS07, ISA$^+$04, LSCI05] and uncertain databases [RDS07, ZLGZ10]. Most of these works exploit the monotonicity property of scoring functions, building offline indexes and integrate the function into the cost model of the query optimizer in order to bound the number of operations required per answer tuple. We refer the reader to [IBS08] for a comprehensive survey of top-k processing techniques discovered prior to 2008. More recent work [CLZ$^+$15, GGY$^+$14] has focused on enumerating *twig-pattern* queries over graphs. Our work departs from this line of work in two aspects: *(i)* use of novel techniques that use query decompositions and clever tricks to achieve strictly better space requirement and formal delay guarantees; *(ii)* our algorithms are applicable to arbitrary hypergraphs as compared to simple graph patterns over binary relations. Most closely related to our setting is [KS06] and a line of work initiated by [YAG$^+$18]. [KS06] uses an adaptation of Lawler-Murty's procedure to incrementally computing ordered answers of full acyclic CQs. However, that work was mainly focused on studying the combined complexity of the problem. Further, since the goal was to obtain polynomial delay guarantees, the authors did not attempt to obtain the best possible delay guarantees. This line of work was further extended to parallel setting [GKS11] and also when the data is incomplete [KS07].

The other line of work was initiated by Yang et al. [YAG$^+$18] who presented a novel anytime algorithm, called KARPET, for enumerating *homomorphic tree patterns* with worst case delay and space guarantees where the ranking function is sum of weights of input tuples that contribute to an output tuple. KARPET is an any-time algorithm that generates candidate output tuples with different scores and sorts them incremental via a priority queue. However, the candidate generation phase is expensive (which translates to linear delay guarantees) and can be improved substantially, as we show in this article. [YRLG18] made the further connection that KARPET can be extended to arbitrary full CQs (including cycles) by considering different tree decompositions. This connection was concretely established in concurrent work [TAG$^+$20] that built upon [YAG$^+$18, YRLG18] to obtain logarithmic delay guarantees using a dynamic programming approach combined with Lawler's procedure [Law72]. Both our work and prior work [TAG$^+$20] are generalizations of known algorithms [JM99, Epp98] from paths to CQs. In comparison with [TAG$^+$20], we (1) present a framework that considers defines general properties of ranking functions and how to combine it with tree decompositions via the notion of compatibility, (2) we consider ranking functions beyond the sum of tuple weights as considered in [TAG$^+$20], and (3) present conditional and unconditional lower bounds. On the other hand, [TAG$^+$20] considers CQs with projections (i.e., non-full CQs), conducts a thorough experimental evaluation on real-world datasets, and considers other measures of success such as *time-to-k* $TT(k)$ which is defined as time required until the k^{th} answer is returned. Note that a low delay is sufficient but not necessary to achieve low $TT(k)$. More recently, the authors were also able to extend their results to theta-joins as well [TGR21]. For a more detailed overview of the prior work on the topic of ranked enumeration, we refer the reader to [TGR20, TAG+20].

Rank aggregation algorithms. Top-k processing over ranked lists of objects has a rich history. The problem was first studied by Fagin et al. [Fag02, FLN03] where the database consists of a *single* relation $R(x_1, \dots, x_m)$ containing N rows (referred to as objects) and m attributes (referred to as ranked streams). The ranking function is defined over the the m attributes and the goal is to find the top-k results for coordinate monotone functions. The authors proposed Fagin's algorithm (FA) and Threshold algorithm (TA), both of which were shown to be instance optimal for database access cost under sorted list access and random access model. This model would be applicable to our setting only if $Q(D)$ is already computed and materialized (so as to obtain a single relation, which would be of size $|Q(D)|$). More importantly, TA can only give $O(N)$ delay guarantee using $O(N)$ space. [NCS+01] extended the problem setting to the case where we want to enumerate top-k answers for t-path query. The first proposed algorithm J^* uses an iterative deepening mechanism that pushes the most promising candidates into a priority queue. Unfortunately, even though the algorithm is instance optimal with respect to number of sorted access over each list, the delay guarantee is $\Omega(|Q(D)|)$ with space requirement $S = \Omega(|Q(D)|)$. A second proposed algorithm J^*_{PA} allows random access over each sorted list. J^*_{PA} uses a dynamic threshold to decide when to use random access over other lists to find joining tuples versus sorted access but does not improve formal guarantees.

Query enumeration. The notion of constant delay query enumeration was introduced by Bagan, Durand and Grandjean in [BDG07]. In this setting, preprocessing time is supposed to be much smaller than the time needed to evaluate the query (usually, linear in the size of the database), and the delay between two output tuples may depend on the query, but not on the database. This notion captures the *intrinsic hardness* of query structure. For an introduction to this topic and an overview of the state-of-the-art we refer the reader to the survey [Seg13, Seg15b]. Most of the results in existing works focus only on lexicographic enumeration of query results where the ordering of variables cannot be arbitrarily chosen. Transferring the static setting enumeration results to under updates has also been a subject of recent interest [BKS18, BKS17].

Factorized databases. Following the landmark result of [OZ15] which introduced the notion of using the logical structure of the query for efficient join evaluation, a long line of research has benefited from its application to learning problems and broader classes of queries [BOZ12, BKOZ13, OS16, DK18, KNOZ20, DHK20, DHK21]. The core idea of factorized databases is to convert an arbitrary query into an acyclic query by finding a query decomposition of small width. This width parameter controls the space and pre-processing time required in order to build indexes allowing for constant delay enumeration. We build on top of factorized representations and integrate ranking functions in the framework to enable enumeration beyond lexicographic orders.

7. Conclusion

In this paper, we study the problem of CQ result enumeration in ranked order. We combine the notion of query decompositions with certain desirable properties of ranking functions to enable logarithmic delay enumeration with small preprocessing time. The most natural open problem is to prove space lower bounds to see if our algorithms are optimal at least for certain classes of CQs. An intriguing question is to explore the full continuum of time-space tradeoffs. For instance, for any compatible ranking function with the 4-path query and $T_P = O(N)$, we can achieve $\delta = O(N^{3/2})$ with space $S_e = O(N)$ and $\delta = O(\log N)$ with space $S_e = O(N^2)$. The precise tradeoff between these two points and its generalization to arbitrary CQs is unknown. There also remain several open question regarding how the structure of ranking functions influences the efficiency of the algorithms. In particular, it would be interesting to find fine-grained classes of ranking functions which are more expressive than totally decomposable, but less expressive than coordinate decomposable. For instance, the ranking function $f(x, y) = |x - y|$ is not coordinate decomposable, but it is *piecewise* coordinate decomposable on either side of the global minimum critical point for each x valuation. Finally, recent work has made considerable progress in query evaluation under updates. In this setting, the goal is to minimize the update time of the data structure as well as minimize the delay. A simple application of our algorithm is useful here. For any full acyclic query, one can maintain the relations under updates in constant time by updating the hash maps and then apply the preprocessing and enumeration phase of our algorithm. This algorithm gives a linear delay guarantee since the preprocessing phase takes linear time. One could also apply the preprocessing phase of our algorithm after each update to reset all priority queues which makes the update time linear but the enumeration delay can now be $O(\log |D|)$. Both of these guarantees can be improved upon for the class of hierarchical queries [BKS17, KNOZ20]. We leave the precise construction, algorithms, and empirical evaluation as a topic for future research.

References

[AGM13] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

[AKNS17] Mahmoud Abo Khamis, Hung Q Ngo, and Dan Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 429–444. ACM, 2017.

[BCD$^+$06] David Bremner, Timothy M Chan, Erik D Demaine, Jeff Erickson, Ferran Hurtado, John Iacono, Stefan Langerman, and Perouz Taslakian. Necklaces, convolutions, and x+ y. In *European Symposium on Algorithms*, pages 160–171. Springer, 2006.

[BDG07] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*, pages 208–222. Springer, 2007.

[BG81] Philip A Bernstein and Nathan Goodman. Power of natural semijoins. *SIAM Journal on Computing*, 10(4):751–771, 1981.

[BKOZ13] Nurzhan Bakibayev, Tomáš Kočiskỳ, Dan Olteanu, and Jakub Závodnỳ. Aggregation and ordering in factorised databases. *Proceedings of the VLDB Endowment*, 6(14):1990–2001, 2013.

[BKPS19] Endre Boros, Benny Kimelfeld, Reinhard Pichler, and Nicole Schweikardt. Enumeration in Data Management (Dagstuhl Seminar 19211). *Dagstuhl Reports*, 9(5):89–109, 2019. `doi:10.4230/DagRep.9.5.89`.

[BKS17] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 303–318. ACM, 2017.

[BKS18] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering fo+ mod queries under updates on bounded degree databases. *ACM Transactions on Database Systems (TODS)*, 43(2):7, 2018.

[BOZ12] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodnỳ. Fdb: A query engine for factorised relational databases. *Proceedings of the VLDB Endowment*, 5(11):1232–1243, 2012.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[CLRS22] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms.* MIT press, 2022.

[CLZ$^+$15] Lijun Chang, Xuemin Lin, Wenjie Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. Optimal enumeration: Efficient top-k tree matching. *Proceedings of the VLDB Endowment*, 8(5):533–544, 2015.

[CS07] Sara Cohen and Yehoshua Sagiv. An incremental algorithm for computing ranked full disjunctions. *Journal of Computer and System Sciences*, 73(4):648–668, 2007.

[DHK20] Shaleen Deep, Xiao Hu, and Paraschos 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.

[DHK21] Shaleen Deep, Xiao Hu, and Paraschos Koutris. Enumeration algorithms for conjunctive queries with projection. In *To appear the the proceedings of ICDT '21 Proceedings*, 2021.

[DK18] Shaleen Deep and Paraschos Koutris. Compressed representations of conjunctive query results. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 307–322. ACM, 2018.

[DK21] Shaleen Deep and Paraschos Koutris. Ranked enumeration of conjunctive query results. In *24th International Conference on Database Theory (ICDT 2021).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[DO05] Erik D Demaine and Joseph O'Rourke. Open problems from cccg 2005. In *Canadian Conference on Computational Geometry*, pages 75–80, 2005.

[Epp98] David Eppstein. Finding the k shortest paths. *SIAM Journal on computing*, 28(2):652–673, 1998.

[Fag02] Ronald Fagin. Combining fuzzy information: an overview. *ACM SIGMOD Record*, 31(2):109–118, 2002.

[FLN03] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences*, 66(4):614–656, 2003.

[Fre76] Michael L Fredman. How good is the information theory bound in sorting? *Theoretical Computer Science*, 1(4):355–361, 1976.

[GGY$^+$14] Manish Gupta, Jing Gao, Xifeng Yan, Hasan Cam, and Jiawei Han. Top-k interesting subgraph discovery in information networks. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 820–831. IEEE, 2014.

[GKS11] Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. Optimizing and parallelizing ranked enumeration. *Proceedings of the VLDB Endowment*, 4(11):1028–1039, 2011.

[HUA75] John E Hopcroft, Jeffrey D Ullman, and AV Aho. The design and analysis of computer algorithms, 1975.

[IBS08] Ihab F Ilyas, George Beskales, and Mohamed A Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.

[ISA$^+$04] Ihab F Ilyas, Rahul Shah, Walid G Aref, Jeffrey Scott Vitter, and Ahmed K Elmagarmid. Rank-aware query optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 203–214. ACM, 2004.

[JM99] Víctor M Jiménez and Andrés Marzal. Computing the k shortest paths: A new algorithm and an experimental comparison. In *Algorithm Engineering: 3rd International Workshop, WAE'99 London, UK, July 19–21, 1999 Proceedings 3*, pages 15–29. Springer, 1999.

[KNOZ20] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 375–392, 2020.

[KS06] Benny Kimelfeld and Yehoshua Sagiv. Incrementally computing ordered answers of acyclic conjunctive queries. In *International Workshop on Next Generation Information Technologies and Systems*, pages 141–152. Springer, 2006.

[KS07] Benny Kimelfeld and Yehoshua Sagiv. Combining incompleteness and ranking in tree queries. In *International Conference on Database Theory*, pages 329–343. Springer, 2007.

[Law72] Eugene L Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management science*, 18(7):401–405, 1972.

[LCIS05] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F Ilyas, and Sumin Song. Ranksql: query algebra and optimization for relational top-k queries. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 131–142. ACM, 2005.

[LSCI05] Chengkai Li, Mohamed A Soliman, Kevin Chen-Chuan Chang, and Ihab F Ilyas. Ranksql: supporting ranking queries in relational database management systems. In *Proceedings of the 31st international conference on Very large data bases*, pages 1342–1345. VLDB Endowment, 2005.

[Mar13] Dániel Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *Journal of the ACM (JACM)*, 60(6):42, 2013.

[NCS$^+$01] Apostol Natsev, Yuan-Chi Chang, John R Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, volume 1, pages 281–290, 2001.

[NPRR12] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 37–48. ACM, 2012.

[NRR13] Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2013. `doi:10.1145/2590989.2590991`.

[OS16] Dan Olteanu and Maximilian Schleich. Factorized databases. *ACM SIGMOD Record*, 45(2):5–16, 2016.

[OZ15] Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2, 2015. `doi:10.1145/2656335`.

[QCS07] Yan Qi, K Selçuk Candan, and Maria Luisa Sapino. Sum-max monotonic ranked joins for evaluating top-k twig queries on weighted data graphs. In *Proceedings of the 33rd international conference on Very large data bases*, pages 507–518. VLDB Endowment, 2007.

[RDS07] Christopher Re, Nilesh Dalvi, and Dan Suciu. Efficient top-k query evaluation on probabilistic data. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 886–895. IEEE, 2007.

[Seg13] Luc Segoufin. Enumerating with constant delay the answers to a query. In *Proceedings of the 16th International Conference on Database Theory*, pages 10–20. ACM, 2013.

[Seg15a] Luc Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Record*, 44(1):10–17, 2015. `doi:10.1145/2783888.2783894`.

[Seg15b] Luc Segoufin. Constant delay enumeration for conjunctive queries. *ACM SIGMOD Record*, 44(1):10–17, 2015.

[SS95] William L Steiger and Ileana Streinu. A pseudo-algorithmic separation of lines from pseudo-lines. *Inf. Process. Lett.*, 53(5):295–299, 1995.

[TAG$^+$20] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, volume 13, page 1582. NIH Public Access, 2020.

[TGR20] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Optimal join algorithms meet top-k. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2659–2665, 2020.

[TGR21] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Beyond equi-joins: Ranking, enumeration and factorization. *Proc. VLDB Endow.*, 14(11):2599–2612, 2021. URL: `http://www.vldb.org/pvldb/vol14/p2599-tziavelis.pdf`.

[YAG$^+$18] Xiaofeng Yang, Deepak Ajwani, Wolfgang Gatterbauer, Patrick K Nicholson, Mirek Riedewald, and Alessandra Sala. Any-k: Anytime top-k tree pattern retrieval in labeled graphs. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 489–498. International World Wide Web Conferences Steering Committee, 2018.

[Yan81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, pages 82–94, 1981.

[YRLG18] Xiaofeng Yang, Mirek Riedewald, Rundong Li, and Wolfgang Gatterbauer. Any-k algorithms for exploratory analysis with conjunctive queries. In *Proceedings of the 5th International Workshop on Exploratory Search in Databases and the Web*, pages 1–3, 2018.

[ZLGZ10] Zhaonian Zou, Jianzhong Li, Hong Gao, and Shuo Zhang. Finding top-k maximal cliques in an uncertain graph. 2010.