Ranked Enumeration of Conjunctive Query Results

Shaleen Deep
Department of Computer Sciences, University of Wisconsin-Madison, Madison, WI, USA

Paraschos Koutris
Department of Computer Sciences, University of Wisconsin-Madison, Madison, WI, USA

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.

2012 ACM Subject Classification Theory of computation → Database theory

Keywords and phrases Query result enumeration, joins, ranking

Digital Object Identifier 10.4230/LIPIcs.ICDT.2021.5

Related Version *Full Version*: `https://arxiv.org/pdf/1902.02698.pdf` [17]

Funding This research was supported in part by National Science Foundation grants CRII-1850348 and III-1910014.

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, [44, 10] 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 [37, 26]. 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 paper, 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 $Q(D)$ and sort the tuples based on the score of each tuple. Although this approach is conceptually simple, it requires that $|Q(D)|$ tuples are materialized; moreover, the time from when the user submits the query to when she receives the first output tuples is $\Omega(|Q(D)| \cdot \log |Q(D)|)$. 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 [25] and references within). In spite of the significant progress, all of the known techniques suffer from large worst-case space requirements, no dependence on k, and provide no formal 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. [21]

initiated a long line of study related to aggregation over *sorted lists*. However, [21] 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 paper, 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.** 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 [28, 12, 36] and practically [44, 10, 5]. Theoretically, [28] 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. [44] presents an anytime enumeration algorithm restricted to acyclic queries on graphs that uses $\Theta(|Q(D)| + |D|)$ 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 [3], 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 [5, 36] 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.

Our Contribution. 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 12) 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 12 then shows that with $O(|D|^{\texttt{fhw}})$ preprocessing time, where `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 [8] on ranked enumeration (see Question 4.6).

2. We propose two extensions of Theorem 12 that improve the preprocessing time to $O(|D|^{\texttt{subw}})$, polynomial improvement over Theorem 12 where `subw` is the *submodular width* of the query Q. The result is based on a simple but powerful corollary of the main result that can be applied to any full UCQ Q combined with the `PANDA` algorithm proposed by Abo Khamis et al. [1].
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 can characterize for which acyclic queries it is possible to achieve logarithmic delay with linear preprocessing time, and for which it is not.

2 Problem Setting

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

2.1 Conjunctive Queries

In this paper we will focus on the class of *Conjunctive Queries (CQs)*, which are expressed as $Q(\mathbf{y}) = R_1(\mathbf{x}_1), R_2(\mathbf{x}_2), \dots, R_n(\mathbf{x}_n)$ Here, the symbols $\mathbf{y}, \mathbf{x}_1, \dots, \mathbf{x}_n$ are vectors that contain *variables* or *constants*, the atom $Q(\mathbf{y})$ is the *head* of the query, and the atoms $R_1(\mathbf{x}_1), R_2(\mathbf{x}_2), \dots, R_n(\mathbf{x}_n)$ form the *body*. The variables in the head are a subset of the variables that appear in the body. 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 $Q()$. We will typically use the symbols $x, y, z, \dots$ to denote variables, and $a, b, c, \dots$ to denote constants. We use $Q(D)$ to denote the result of the query 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. A *Union of Conjunctive Queries* $\varphi = \bigcup_{i \in \{1,\dots,\ell\}} \varphi_i$ 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, $\varphi(D) = \bigcup_{i \in \{1,\dots,\ell\}} \varphi_i(D)$. 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 $\Join_{F \in \mathcal{E}_Q} R_F$. 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 [2] 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 [34, 33]: 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 $\mathtt{value}(t) = \mathcal{B}_t \setminus \mathtt{key}(t)$ 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 $\mathtt{fhw}(Q)$, is the minimum fractional hypertree width among all tree decompositions of its hypergraph. We say that a query is *acyclic* if $\mathtt{fhw}(Q) = 1$. 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 [24], where data values as well as pointers to databases are of constant size. Throughout the paper, all complexity results are with respect to data complexity (unless explicitly mentioned), where the query is assumed fixed.

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 $\mathtt{rank}$ that maps each valuation θ to a number $\mathtt{rank}(\theta) \in \mathbb{R}$. In particular, for two valuations θ_1, θ_2, we have $\theta_1 \succeq \theta_2$ if and only if $\mathtt{rank}(\theta_1) \geq \mathtt{rank}(\theta_2)$. Throughout the paper, we will assume that $\mathtt{rank}$ 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.** For every constant $c \in \mathbf{dom}$, we associate a weight $w(c) \in \mathbb{R}$. Then, for each valuation θ, we can define $\mathtt{rank}(\theta) := \sum_{x \in \mathcal{V}} w(\theta(x))$. This ranking function sums the weights of each value in the tuple.

▶ **Example 3.** 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 $\mathtt{rank}(\theta) = \sum_{F \in \mathcal{E}} w_F(\theta[x_F])$ 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 4** (Decomposable Ranking). *Let $\texttt{rank}$ be a ranking function over $\mathcal{V}$ and $S \subseteq \mathcal{V}$. We say that $\texttt{rank}$ is S-*decomposable *if there exists a total order for all valuations over S, such that for every valuation φ over $\mathcal{V} \setminus S$, and any two valuations θ_1, θ_2 over S we have:*

$$\theta_1 \succeq \theta_2 \Rightarrow \texttt{rank}(\varphi \circ \theta_1) \geq \texttt{rank}(\varphi \circ \theta_2).$$

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 [28].

▶ **Example 5.** The ranking function $\texttt{rank}(\theta) = \sum_{x \in \mathcal{V}} w(\theta(x))$ defined in Example 2 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 $\mathcal{V} \setminus S$ we have:

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

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

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

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 binary 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 $(1, 1) \succeq (1, -1)$, then inner product ranking function over x_1, x_2, y_1, y_2 for $\varphi = (1, -1)$ is $\texttt{rank}(1, 1, 1, -1) < \texttt{rank}(1, -1, 1, -1)$ but if we define $(1, -1) \succeq (1, 1)$, then for $\varphi = (-1, -1)$ we get $\texttt{rank}(1, -1, -1, -1) < \texttt{rank}(1, 1, -1, -1)$. This shows that there exists no total ordering over the valuations of variables $\{x_1, x_2\}$.

▶ **Definition 6.** *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.

▶ **Lemma 7.** *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 $S \subseteq \mathcal{V} \setminus T$.*

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 8** (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 9.** Consider the join query $Q(x, y, z) = R(x, y), S(y, z)$, and the ranking function from Example 3, $\texttt{rank}(\theta) = w_R(\theta(x), \theta(y)) + w_S(\theta(y), \theta(z))$. 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 `rank`. We will study this problem in the enumeration framework similar to that of [41], 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 10.** *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 10 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 11** (due to [36]). *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 12** (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*

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

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

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

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 largest one.

It is instructive to compare Theorem 12 with Theorem 11, 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).

In the remainder of this section, we will present a few applications of Theorem 12, and then sketch the construction for the proof of the theorem.

3.1 Applications

We show here how to apply Theorem 12 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*. Recall that this means that $\oplus$ is a binary operator that is commutative, associative, and has an identity element in U. We say that the function is monotone if $a \geq b$ implies that $a \oplus c \geq b \oplus c$ for every c. Such examples are $\langle \mathbb{R}, + \rangle$, $\langle \mathbb{R}, * \rangle$, and $\langle U, max \rangle$, where U is bounded.

▶ **Lemma 13.** *Let* `rank` *be a monotone vertex-based ranking function over $\mathcal{V}$. Then,* `rank` *is totally decomposable, and hence compatible with any tree decomposition of a hypergraph with vertices $\mathcal{V}$.*

Tuple-Based Ranking. Given a query hypergraph $\mathcal{H}$, a tuple-based ranking function assigns for every valuation θ over the variables x_F of relation R_F a weight $w_F(\theta) \in U \subseteq \mathbb{R}$. Then, it 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 14.** *Let* `rank` *be a monotone tuple-based ranking function over $\mathcal{V}$. Then,* `rank` *is compatible with any tree decomposition of a hypergraph with vertices $\mathcal{V}$.*

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

▶ **Proposition 15.** *Let Q be a natural join query with optimal fractional hypertree-width* `fhw`*. Let* `rank` *be a ranking function that can be either (i) monotone vertex-based, (ii) monotone tuple-based. Then, for any input D, we can pre-process D in time $T_p = O(|D|^{\mathtt{fhw}})$ and space $S_p = O(|D|^{\mathtt{fhw}})$ 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 $\mathtt{fhw} = 1$, 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 $\mathcal{V} = \{x_1, \ldots, x_k\}$, a lexicographic order $\langle x_{i_1}, \ldots, x_{i_\ell}\rangle$ 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 $\mathtt{rank}(\theta) := \sum_x w_x f_x(\theta(x))$, where w_x are appropriately chosen constants. Since this ranking function is a monotone vertex-based ranking, Proposition 15 applies here as well.

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

▶ **Proposition 16** (due to [36, 5])**.** *Let Q be a natural join query with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, and $\langle x_{i_1}, \ldots, x_{i_\ell}\rangle$ a lexicographic ordering of the variables in $\mathcal{V}$.*

Let $\mathcal{T}$ be a tree decomposition with fractional hypertree-width `fhw-lex` *such that $\langle x_{i_1}, \ldots, x_{i_\ell}\rangle$ 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 $T_p = O(|D|^{\mathtt{fhw\text{-}lex}})$ and $S_p = O(|D|^{\mathtt{fhw\text{-}lex}})$ 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 12, by achieving constant delay $O(1)$, and constant space S_e. However, given a tree decomposition, Theorem 12 applies for any lexicographic ordering - in contrast to Proposition 16. 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 $\mathtt{fhw} = 1$, 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 $\mathtt{fhw\text{-}lex} = 2$; hence, Proposition 16 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 16 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 [28]: c-determined implies c-bounded, but not vice versa. For c-bounded ranking functions, we can show the following result:

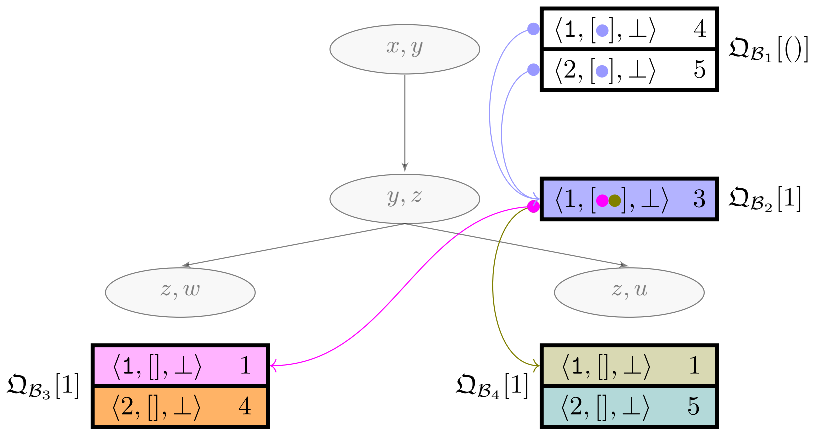
▶ **Proposition 17.** *Let Q be a natural join query with optimal fractional hypertree-width $\mathtt{fhw}$. If $\mathtt{rank}$ is a c-bounded ranking function, then for any input D, we can pre-process D in time $T_p = O(|D|^{\mathtt{fhw}+c})$ and space $S_p = O(|D|^{\mathtt{fhw}+c})$ such that for any k, we can enumerate the top-k results of $Q(D)$ with $\delta = O(\log |D|)$ and $S_e = O(\min\{k, |Q(D)|\})$*

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 query that corresponds to the subtree rooted at t. We do 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 $C(\theta) = \langle \theta, [], \perp \rangle$, where $\perp$ 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). It is easy to see that, given a cell $C(\theta)$, one can reconstruct θ in constant time (dependent only on the query). Additionally, each node t maintains one hash map $\mathfrak{Q}_t$, which maps each valuation u over $\mathtt{key}(\mathcal{B}_t)$ to a *priority queue* $\mathfrak{Q}_t[u]$. The elements of $\mathfrak{Q}_t$ are cells $C(\theta)$, where θ is a valuation over $\mathcal{B}_t^{\prec}$ such that $u = \theta[\mathtt{key}(\mathcal{B}_t)]$. 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 [13]) 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 $O(\log n)$.

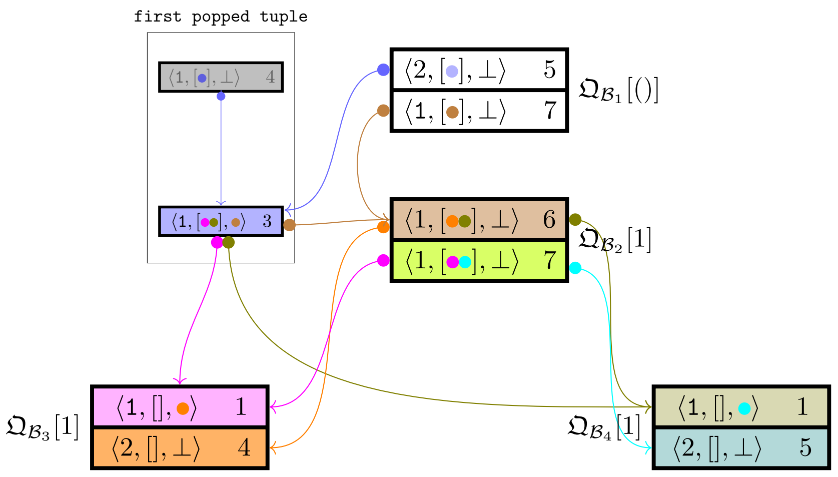
Notice that it is not straightforward to rank the cells according to the valuations, since the ranking function is defined over all variables $\mathcal{V}$. However, here we can use the fact that the ranking function is compatible with the decomposition at hand. Indeed, given a fixed valuation u over $\mathtt{key}(\mathcal{B}_t)$, we will order the valuations θ over $\mathcal{B}_t^{\prec}$ that agree with u according to the score: $\mathtt{rank}(v_t^\star \circ \theta)$ where $v_t^\star$ is a valuation over $\mathcal{V} \setminus \mathcal{B}_t^{\prec}$ chosen according to the definition of decomposability. The key intuition is that the compatibility of the ranking function with the decomposition implies that the ordering of the tuples 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.

Preprocessing. Algorithm 1 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 10, so each valuation θ over $\mathcal{B}_t$ is added to the corresponding queue as the triple $\langle \theta, [], \perp \rangle$. 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 10). 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 $\mathtt{key}(r) = \{\}$.



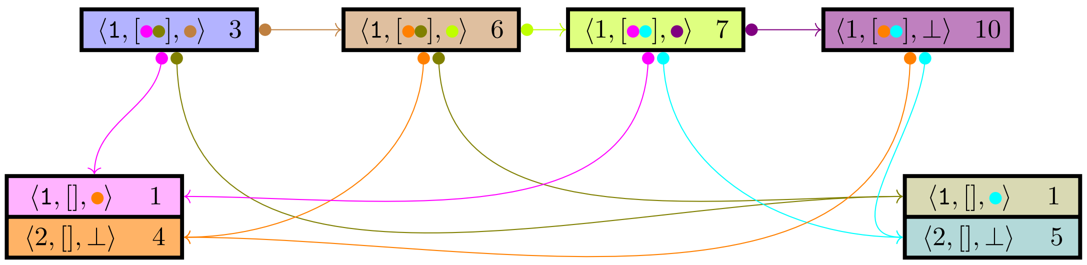
x, y
y, z
z, w
z, u
$\langle 1, [\bullet], \perp \rangle$ 4
$\langle 2, [\bullet], \perp \rangle$ 5
$\mathfrak{Q}_{\mathcal{B}_1}[()]$
$\langle 1, [\bullet\bullet], \perp \rangle$ 3
$\mathfrak{Q}_{\mathcal{B}_2}[1]$
$\mathfrak{Q}_{\mathcal{B}_3}[1]$
$\langle 1, [], \perp \rangle$ 1
$\langle 2, [], \perp \rangle$ 4
$\mathfrak{Q}_{\mathcal{B}_4}[1]$
$\langle 1, [], \perp \rangle$ 1
$\langle 2, [], \perp \rangle$ 5

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



first popped tuple
$\langle 1, [\bullet], \perp \rangle$ 4
$\langle 1, [\bullet\bullet], \bullet \rangle$ 3
$\langle 2, [\bullet], \perp \rangle$ 5
$\langle 1, [\bullet], \perp \rangle$ 7
$\mathfrak{Q}_{\mathcal{B}_1}[()]$
$\langle 1, [\bullet\bullet], \perp \rangle$ 6
$\langle 1, [\bullet\bullet], \perp \rangle$ 7
$\mathfrak{Q}_{\mathcal{B}_2}[1]$
$\mathfrak{Q}_{\mathcal{B}_3}[1]$
$\langle 1, [], \bullet \rangle$ 1
$\langle 2, [], \perp \rangle$ 4
$\mathfrak{Q}_{\mathcal{B}_4}[1]$
$\langle 1, [], \bullet \rangle$ 1
$\langle 2, [], \perp \rangle$ 5

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



$\langle 1, [\bullet\bullet], \bullet \rangle$ 3
$\langle 1, [\bullet\bullet], \bullet \rangle$ 6
$\langle 1, [\bullet\bullet], \bullet \rangle$ 7
$\langle 1, [\bullet\bullet], \perp \rangle$ 10
$\langle 1, [], \bullet \rangle$ 1
$\langle 2, [], \perp \rangle$ 4
$\langle 1, [], \bullet \rangle$ 1
$\langle 2, [], \perp \rangle$ 5

(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. Each memory location is shown with a different color. Pointers in cells are denoted using ● which means that the it points to a memory location with the corresponding color (shown using pointed arrows). Root bag priority queue cells are not color coded as nobody points to them.

Algorithm 1 Preprocessing Phase.

```
1  foreach $t \in V(\mathcal{T})$ do
2      materialize the bag $\mathcal{B}_t$
3  full reducer pass on materialized bags in $\mathcal{T}$
4  forall $t \in V(\mathcal{T})$ in post-order traversal do
5      foreach valuation $\theta$ in bag $\mathcal{B}_t$ do
6          $u \leftarrow \theta[\mathtt{key}(\mathcal{B}_t)]$
7          if $\mathfrak{Q}_t[u]$ is NULL then
8              $\mathfrak{Q}_t[u] \leftarrow$ new priority queue
9          $\ell \leftarrow []$
           /* $\ell$ is a list of pointers */
10         foreach child $s$ of $t$ do
11             $\ell$.INSERT($\mathfrak{Q}_s[\theta[\mathtt{key}(\mathcal{B}_s)]]$.TOP())
12         $\mathfrak{Q}_t[u]$.INSERT($\langle \theta, \ell, \perp \rangle$) /* ranking function uses $\theta, \ell, v_t^\star$ to calculate score used
               by priority queue */
```

Algorithm 2 Enumeration Phase.

```
1  procedure ENUM()
2      while $\mathfrak{Q}_r[()]$ is not empty do
3          output $\mathfrak{Q}_r[()]$.TOP()
4          TOPDOWN($\mathfrak{Q}_r[()]$.TOP(), $r$)
5  procedure TOPDOWN($c, t$)
6      /* $c = \langle \theta, [p_1, \ldots, p_k], \mathsf{next} \rangle$ */
7      $u \leftarrow \theta[\mathtt{key}(\mathcal{B}_t)]$
8      if next $= \perp$ then
9          $\mathfrak{Q}_t[u]$.POP()
10         foreach child $t_i$ of $t$ do
11             $p'_i \leftarrow$ TOPDOWN($*p_i, t_i$)
12             if $p'_i \neq \perp$ then
13                 $\mathfrak{Q}_t[u]$.INSERT($\langle \theta, [p_1, \ldots, p'_i, \ldots p_k], \perp \rangle$) /* insert new candidate(s) */
14         if $t$ is not the root then
15             next $\leftarrow \mathfrak{Q}_t[u]$.TOP()
16     return next
```

▶ **Example 18.** As a running example, we consider the natural join query $Q(x, y, z, w) = R_1(x, y), R_2(y, z), R_3(z, w), R_4(z, u)$ 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.

R_1

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

R_2

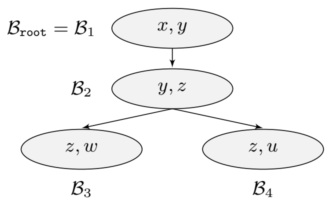
id	$\mathbf{w_2}$	**y**	**z**
1	1	1	1
2	1	3	1

R_3

id	$\mathbf{w_3}$	**z**	**w**
1	1	1	1
2	4	1	2

R_4

id	**w₃**	**z**	**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 1a shows the state of priority queues after the pre-processing step. For convenience, θ in each cell $\langle\theta, [p_1, \ldots, p_k], \mathsf{next}\rangle$ is shown using the primary key of the tuple and pointers p_i and next are shown using colored dots ● representing the memory location it points to. The cell in a memory location is followed by the partial aggregated 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 $(y = 1, z = 1) \in R_2$ with $(z = 1, w = 1)$ from R_3 and $(z = 1, u = 1)$ in R_4 is $1 + 1 + 1 = 3$ (shown as ⟨1, [●●], ⊥⟩ 3 in the figure). Each cell in every priority queue points to the top element of the priority queue of child nodes that are joinable. Note that since both tuples in R_1 join with the sole tuple from R_2, they point to the same cell.

Enumeration. Algorithm 2 presents the algorithm for the enumeration phase. The heart of the algorithm is the procedure `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 once the new candidates have been inserted, the next pointer of cell c is updated by pointing to the topmost element in the priority queue. This chaining materializes the answers for the particular bag that can be reused.

▶ **Example 19.** 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 top left of the figure). Cell ⟨1, [●], ⊥⟩ 4 is popped from $\mathfrak{Q}_{\mathcal{B}_1}[()]$ (after satisfying if condition on line 8 as next is $\perp$). Since nothing is pointing to this cell, it is garbage collected (denoted by greying out the cell). We recursively call `TOPDOWN` for child node $\mathcal{B}_2$ and cell ⟨1, [●●], ⊥⟩ 3 . The next for this cell is also $\perp$ 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 ⟨1, [], ⊥⟩ 1 . The least ranked tuple but larger than ⟨1, [], ⊥⟩ 1 in $\mathfrak{Q}_{\mathcal{B}_3}[1]$ is the cell at address ● . Thus, next for ⟨1, [], ⊥⟩ 1 is updated to ● and cell at ● is returned which leads to creation and insertion of ⟨1, [●●], ⊥⟩ 6 cell in $\mathfrak{Q}_{\mathcal{B}_2}[1]$. Similarly, we get the other cell in $\mathfrak{Q}_{\mathcal{B}_2}[1]$ by recursive call for $\mathcal{B}_4$. After both the calls are over for node $\mathcal{B}_2$, the topmost cell at $\mathfrak{Q}_{\mathcal{B}_2}[1]$ is ● ,which is set as the next for ⟨1, [●●], ⊥⟩ 3 (changing into ⟨1, [●●], ●⟩ 3), terminating one full iteration. ⟨1, [●●], ●⟩ 3 is not garbage collected as ⟨2, [●], ⊥⟩ 5 is pointing to it.

Let us now look at the second iteration of `ENUM()`. The tuple returned is top element of $Q_{\mathcal{B}_1}[()]$ which is ⟨2, [●], ⊥⟩ 5 . However, the function `TOPDOWN()` with ⟨2, [●], ⊥⟩ 5 does not recursively go all the way down to leaf nodes. Since ⟨1, [●●], ●⟩ 3 already has next populated, we insert ⟨2, [●], ⊥⟩ 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. Figure 1c shows the eventual sequence of pointers at node $\mathcal{B}_2$ which is the ranked materialized output of the subtree rooted at $\mathcal{B}_2$. $\boxed{\langle 2, [\bullet], \perp\rangle \quad 5}$ is garbage collected.

4 Extensions

In this section, we describe two extensions of Theorem 12 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 12 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 2 to enumerate all tuples with the same score in lexicographic increasing order. 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 corollary of the main result.

▶ **Corollary 20.** *Let $\varphi = \varphi_1 \cup \dots \varphi_\ell$ be a full UCQ. Let* fhw *denote the fractional hypertree-width of all decompositions across all CQs φ_i, and* `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,*

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

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

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

4.2 Improving The Main Result

Although Corollary 20 is a straightforward extension of Theorem 12, it is powerful enough to improve the pre-processing time and space of Theorem 12 by using Corollary 20 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 [2]. However, there exists another notion of width known as the *submodular width* (denoted `subw`) [31]. It is also known that for any CQ, it holds that `subw` $\leq$ `fhw`. Recent work by Abo Khamis et al. [1] 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 $O(|D|^{\texttt{subw}} \cdot \log |D| + |Q(D)|)$. 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 Corollary 20 by setting φ_i as the tree decompositions produced by `PANDA` to get the following result. [17] describes the details of the enumeration algorithm and the tie-breaking comparison function.

▶ **Theorem 21.** *Let φ be a natural join query with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, submodular width $\mathtt{subw}$, and $\mathtt{rank}$ 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,*

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

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 `rank` has no structure, and can be accessed only through a blackbox that, given a tuple/valuation, returns its score: we call this a *blackbox* [1] ranking function. Note that all of our algorithms work under the blackbox assumption.

▶ **Proposition 22.** *Let Q be a natural join query, and $\mathtt{rank}$ a blackbox ranking function. Then, any enumeration algorithm on a database D needs $\Omega(|Q(D)|)$ calls to $\mathtt{rank}$– and worst case $\Omega(|D|^{\rho^*})$ calls – in order to output the smallest tuple.*

Indeed, if the algorithm does not examine the rank of an output tuple, then we can always assign a value to the ranking function such that the tuple is the smallest one. Hence, in the case where there is no restriction on the ranking function, the simple result in Proposition 10 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 natural restrictions of ranking functions, it is not possible to do much better than the $|D|^{\rho^*}$ bound for certain queries.

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

▶ **Lemma 23.** *Consider the query $Q(x_1, y_1, x_2, y_2) = R(x_1, y_1), S(x_2, y_2)$ and let $\mathtt{rank}$ be a blackbox coordinate decomposable ranking function. Then, there exists an instance of size N such that the time required to find the smallest tuple is $\Omega(N^2)$.*

Lemma 23 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.

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

Dichotomy for coordinate decomposable. We first show a dichotomy result for coordinate decomposable functions.

▶ **Theorem 24.** *Consider a full acyclic query Q and a coordinate decomposable blackbox ranking function. 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 there are no atoms R and S in Q such that $\mathsf{vars}(R) \setminus \mathsf{vars}(S) \geq 2$ and $\mathsf{vars}(S) \setminus \mathsf{vars}(R) \geq 2$.*

For example, the query $Q(x, y, z) = R(x, y), S(y, z)$ satisfies the condition of Theorem 24, while the Cartesian product query defined in Lemma 23 does not.

Dichotomy for edge decomposable. We will show a dichotomy result for edge decomposable ranking functions: these are functions that are S-decomposable for any S that is a hyperedge in the query hypergraph. Before we present the result, we need to formally define the notion of path and diameter in a hypergraph.

▶ **Definition 25.** *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 $x_1E_1x_2E_2 \dots x_sE_sx_{s+1}$ 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.*

▶ **Theorem 26.** *Consider a full connected acyclic join query Q and a blackbox edge decomposable ranking function. 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 $\mathsf{dia}(Q) \leq 3$.*

For example, $Q(x, y, z, w) = R(x, y), S(y, z), T(z, w)$ has diameter 3, 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.

When the query is not connected, the characterization must be slightly modified: an acyclic query can be enumerated with $O(\log|D|)$ delay and $T_p = O(|D|)$ if and only if each connected subquery has diameter at most 3.

5.2 Beyond Logarithmic Delay

Next, we examine whether the logarithmic factor that we obtain in the delay of Theorem 12 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, $X = \langle x_1, x_2, \dots, x_n\rangle$ and $Y = \langle y_1, y_2, \dots, y_n\rangle$, 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 [22] 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 [43] gave a simple algorithm that takes $O(n^2 \log n)$ time while using only $O(n^2)$ comparisons.

▶ **Conjecture 27** ([9, 18]). $X + Y$ 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 27 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 [29, 37, 26, 30] and uncertain databases [38, 45]. 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 [25] for a comprehensive survey of top-k processing techniques. More recent work [10, 23] 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 are [28] and [44]. Algorithm in [28] is fundamentally different from ours. It uses an adaptation of Lawler-Murty's procedure to generate candidate output tuples which is also a source of inefficiency given that it ignores query structure. [44] presented a novel anytime algorithm 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. Their algorithm also generates candidate output tuples with different scores and sorts them via a priority queue. However, the candidate generation phase is expensive and can be improved substantially, as we show in this paper. Our algorithm also generalizes the approach of prior work that showed how to find k shortest paths in a graph [19] and may be useful to other problems in DP [39] and DGM [11] where ranked enumeration is useful.

Rank aggregation algorithms. Top-k processing over ranked lists of objects has a rich history. The problem was first studied by Fagin et al. [20, 21] where the database consists of N objects and m ranked streams, each containing a ranking of the N objects with the goal of finding 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. More importantly, TA can only give $O(N)$ delay guarantee using $O(N)$ space. [32] 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 [3]. 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 [40, 42]. 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 [7, 6].

Factorized databases. Following the landmark result of [36] 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 [5, 4, 35, 16, 27, 14, 15]. 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.

References

1 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.

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

3 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.

4 Nurzhan Bakibayev, Tomáš Kočiský, Dan Olteanu, and Jakub Závodný. Aggregation and ordering in factorised databases. *Proceedings of the VLDB Endowment*, 6(14):1990–2001, 2013.

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

6 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.

7 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.

8 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`.

9 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.

10 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.

11 Amrita Roy Chowdhury, Theodoros Rekatsinas, and Somesh Jha. Data-dependent differentially private parameter learning for directed graphical models. In *International Conference on Machine Learning*, pages 1939–1951. PMLR, 2020.

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

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

14 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.

15 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.

16 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.

17 Shaleen Deep and Paraschos Koutris. Ranked enumeration of conjunctive query results. *arXiv preprint arXiv:1902.02698*, 2019.

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

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

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

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

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

23 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.

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

25 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.

26 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.

27 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.

28 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.

29 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.

30 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.

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

32 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.

33 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.

34 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`.

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

36 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`.

37 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.

38 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.

39 Amrita Roy Chowdhury, Chenghong Wang, Xi He, Ashwin Machanavajjhala, and Somesh Jha. Crypt?: Crypto-assisted differential privacy on untrusted servers. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 603–619, 2020.

40 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.

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

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

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

44 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.

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