Ranked Enumeration of Conjunctive Query Results

Shaleen Deep
University of Wisconsin-Madison
shaleen@cs.wisc.edu

Paraschos Koutris
University of Wisconsin-Madison
paris@cs.wisc.edu

ABSTRACT

We investigate the enumeration of top-k answers for conjunctive queries against relational databases according to a given ranking function. The task is to design data structures and algorithms that allow for efficient enumeration after a preprocessing phase. Our main contribution is a novel priority queue based algorithm with near-optimal delay and non-trivial space guarantees that are output sensitive and depend on structure of the query. In particular, we exploit certain desirable properties of ranking functions that frequently occur in practice and degree information in the database instance, allowing for efficient enumeration. We introduce the notion of *decomposable* and *compatible* ranking functions in conjunction with query decomposition, a property that allows for partial aggregation of tuple scores in order to efficiently enumerate the ranked output. We complement the algorithmic results with lower bounds justifying why certain assumptions about properties of ranking functions are necessary and discuss popular conjectures providing evidence for optimality of enumeration delay guarantees. Our results extend and improve upon a long line of work that has studied ranked enumeration from both theoretical and practical perspective.

ACM Reference Format:
Shaleen Deep and Paraschos Koutris. 2019. Ranked Enumeration of Conjunctive Query Results. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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, [7, 33] 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 [18, 27]. 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 results of a conjunctive query (CQ, for short) against a relational database where the tuples are enumerated in order according to a given ranking function.

The simplest way to enumerate the output is to materialize the join result (denoted as OUT) and sort the tuples based on the score of each tuple. Although the approach is conceptually simple, this requires that |OUT| tuples are materialized; moreover, the time from when the user submits the query to when she receives the first output tuples is also $\tilde{O}(|\text{OUT}|)$ [1] in the worst case. 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 have constructed optimizers that exploit properties such as monotonicity of the ranking function, allowing for join evaluation on a subset of the input relations (see [17] and references within). However, all of the known techniques suffer from large worst-case space requirement, no dependence on k and provide no formal guarantees on the delay during enumeration. Fagin et al [12] initiated a long line of study related to aggregation over *sorted lists*. However, [12] 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 enumeration 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* $\tilde{O}(1)$ during enumeration. Although we cannot hope that we can perform efficient ranked enumeration for an arbitrary ranking function, we show that for most ranking functions of practical interest (such as lexicographic ordering, sum of weights of input tuples, product, max etc.) it is possible to apply our techniques. Next, we give an example of a query and ranking function.

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 naive algorithm that computes and ranks all tuples needs $O(N^2)$ preprocessing time in the worst case. In this paper, we can show that we can design an algorithm with $O(N)$ preprocessing time, such that the delay during enumeration is only $\tilde{O}(1)$. This algorithm can output the first k tuples by materializing $O(N + k)$ data – even if the output is much larger. Further, even when $k = O(N^2)$, the space requirement of the algorithm is at most $O(N^{3/2})$.

[1] $\tilde{O}$ hides a polylogarithmic factor in the size of the database

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The problem of enumerating ranked CQ results has been studied both theoretically [8, 19] and practically [7, 33]. Theoretically, [19] establishes tractability of enumerating answers in sorted order with polynomial delay (under combined complexity), albeit with suboptimal space and delay factors for two classes of ranking functions. More recently, [33] presented a novel anytime algorithm for enumerating *tree patterns* with worst case delay and space guarantees for some simple ranking functions. In particular, their algorithm uses $O(|\texttt{OUT}|)$ space in the worst case, provides a worst case delay guarantee of $O(|D|)$ (where D is the database instance), and works only for acyclic queries on graphs. As we will see later, both these guarantees are sub-optimal and can be improved. Ranked enumeration has also been studied for the more restricted class of lexicographic orderings. In a key result [2], the authors showed that the class of *free-connex acyclic queries* can be enumerated in constant delay after only linear time preprocessing. Here the lexicographic order is chosen by the algorithm and not the user. Our results imply that for full acyclic queries it is possible to achieve $O(\log |D|)$ delay enumeration for any lexicographic ordering, after only linear time preprocessing, but at the cost of extra space during the enumeration phase.

Our Contribution. In this work, we show how to obtain logarithmic delay guarantees with small preprocessing time for ranking results of full (without projections) conjunctive queries. We achieve this by taking into account both the structure of the query, as well as the properties of the ranking function. We summarize our technical contributions below:

(1) Our main contribution (Theorem 2) consists of a novel algorithm that combines the use of priority queues and hash maps in conjunction with query decomposition techniques. 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 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 2 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 $\tilde{O}(1)$. We then discuss how to apply our main result to commonly used classes of ranking functions.

(2) Theorem 2 uses more space than is required during runtime – in the worst case, it will use $O(|\texttt{OUT}|)$ space. Our next result, Theorem 3 incorporates to the algorithm from Theorem 2 degree information of input tuples to reduce the space consumption during enumeration. Remarkably, for certain queries, this allows $\tilde{O}(1)$ delay enumeration after linear time preprocessing and sublinear space requirement $|\texttt{OUT}|^{1-\epsilon}, \epsilon > 0$ even when for worst-case output $|\texttt{OUT}| = |D|^{\rho^*}$. Here ρ^* is the AGM bound exponent [1].

(3) Finally, we show lower bounds (conditional and unconditional) for our algorithmic results. In particular, we show that subject to a popular algorithmic conjecture, the logarithmic factor in delay obtained by our algorithms cannot be removed. Perhaps surprisingly, we show that for *coordinate-monotone* functions, there are queries where the optimal strategy is to spend almost $|\texttt{OUT}|$ amount in the preprocessing phase, and queries where linear preprocessing is sufficient for $\tilde{O}(1)$ delay guarantee. This provides justification for introducing more restrictive assumption on the structure of ranking functions to find fragments that admit small preprocessing time (ideally linear in the size of database).

Organization. We present our framework in Section 2, along with the preliminaries and basic notation. Section 3 shows the first main result (Theorem 2) which is subsequently used as a building block in Section 4 for second main result (Theorem 3). Lower bounds are presented in Section 5 followed by related work in Section 6. Lastly, 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 in detail 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. An 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 of θ to U.

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 $|R_F|$.

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 [1] proved that for every fractional edge cover $\mathbf{u}$ of $\mathcal{V}$, the size of a natural join is bounded using the following inequality, known as the *AGM inequality*:

$$| \bowtie_{F\in\mathcal{E}} R_F| \leq \prod_{F\in\mathcal{E}} |R_F|^{u_F} \tag{1}$$

The above bound is constructive [23, 24]: 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 $\mathsf{p}(t)$ to denote the (unique) parent of node $t \in V(\mathcal{T})$. Then, we define $\mathsf{key}(t) = \mathcal{B}_t \cap \mathcal{B}_{\mathsf{p}(t)}$ to be the common variables that occur in the bag $\mathcal{B}_t$ and its parent, and $\mathsf{value}(t) = \mathcal{B}_t \setminus \mathsf{key}(t)$ the remaining variables of the bag. We also use $\mathcal{B}_t^{\prec}$ to denote the union of all bags in the the subtree rooted at t (including $\mathcal{B}_t$).

The *fractional hypertree width* of a tree 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 $\mathsf{fhw}(Q)$, is the minimum fractional hypertree width among all tree decompositions of its hypergraph. We say that a query is *acyclic* if $\mathsf{fhw}(Q) = 1$.

Computational Model. To measure the running time of our algorithms, we will use the uniform-cost RAM model [16], 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. We use the notation $\tilde{O}$ to hide a polylogarithmic factor $\log^k |D|$ for some constant k, where D is the input database.

2.2 Ranking Functions

Consider a full 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 a practical setting, this ordering could be specified, for instance, in the **ORDER BY** clause of a **SQL** query. For Example 1, the **SQL** query would be

```
SELECT R1.x, R1.y, R2.y, R3.y, R1.w1, R2.w2, R3.w3
FROM R AS R1, R AS R2, R AS R3
WHERE R1.y = R2.x AND R2.y = R3.x
ORDER BY 5*R1.w1 + 2*R2.w2 + 4*R3.w3 ASC
```

Formally, we assume that there exists a total order $\succeq$ of the valuations θ over the variables $\mathcal{V}_Q$ of the query Q. The total order is induced by a ranking function rank that maps each valuation θ to a number $\mathsf{rank}(\theta) \in \mathbb{R}$. In particular, for θ_1, θ_2, we have $\theta_1 \succeq \theta_2$ if and only if $\mathsf{rank}(\theta_1) \geq \mathsf{rank}(\theta_2)$. We present below two concrete examples of ranking functions.

Example 2. *For every constant $c \in$ **dom**, we associate a weight $w(c) \in \mathbb{R}$. Then, for each valuation θ, we can define*

$$\mathsf{rank}(v) := \sum_{x\in\mathcal{V}} w(\theta(x)).$$

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*

$$\mathsf{rank}(v) = \sum_{F\in\mathcal{E}} w_F(\theta[x_F])$$

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

Definition 1 (Decomposable Ranking). *Let rank be a ranking function over a set of variables $\mathcal{V}$. Let $S \subseteq \mathcal{V}$. We say that rank is S-decomposable if there exists a valuation $\varphi^\star$ over $\mathcal{V} \setminus S$, such that for every valuation φ over $\mathcal{V} \setminus S$, and any two valuations θ_1, θ_2 over S we have:*

$$\mathsf{rank}(\varphi^\star \circ \theta_1) \geq \mathsf{rank}(\varphi^\star \circ \theta_2) \Rightarrow \mathsf{rank}(\varphi \circ \theta_1) \geq \mathsf{rank}(\varphi \circ \theta_2).$$

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 $S \subseteq \mathcal{V}$. We point out to the reader that totally decomposable functions are equivalent to monotonic orders as defined in [19].

Example 4. *The ranking function $\mathsf{rank}(\theta) = \sum_{x\in\mathcal{V}} w(\theta(x))$ from Example 2 is totally decomposable, and hence also coordinate decomposable. Indeed, pick any set $S \subseteq \mathcal{V}$ and let $\varphi^\star$ be any valuation over $\mathcal{V} \setminus S$. Suppose that $\mathsf{rank}(\varphi^\star \circ \theta_1) \geq \mathsf{rank}(\varphi^\star \circ \theta_2)$. This implies 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:*

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

Definition 2. *Let 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 rank is T-decomposable conditioned on S if for every valuation θ over S, the function $\mathsf{rank}_\theta(\varphi) = \mathsf{rank}(\theta \circ \varphi)$ defined over $\mathcal{V} \setminus S$ is T-decomposable.*

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

Definition 3 (Compatible Ranking). *Let $\mathcal{T}$ be a rooted tree decomposition of $\mathcal{H}$. We say that a ranking function is* compatible with $\mathcal{T}$ *if for every node t it is $(\mathcal{B}_t^{\prec} \setminus \text{key}(t))$-decomposable conditioned on* $\text{key}(t)$.

Example 5. *Consider the join query $Q(x, y, z) = R(x, y), S(y, z)$, and the ranking function from Example 3,* $\text{rank}(\theta) = w_R(\theta(x), \theta(y)) + w_S(\theta(y), \theta(z))$. *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 $\text{key}(t) = \{y\}$, the condition of compatibility holds for node t. Similarly, for the root node $\mathcal{B}_t^{\prec} = \{x, y, z\}$ and $\text{key}(t) = \{\}$, hence the condition is trivially true as well. Thus, the ranking function is compatible with the decomposition.

2.3 Problem Parameters

Given a full natural join query Q and a database D, we want to enumerate the tuples of $Q(D)$ according to the order that is specified by rank. We will study this problem in the enumeration framework similar to that of [30], where an algorithm can be decomposed into two phases:

- a **preprocessing phase** that is performed in 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 the whole output $Q(D)$, then 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 1. *Let Q be a full 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 pre-process D in time and space,*

$$T_p = \tilde{O}(|D|^{\mathsf{fhw}} + |Q(D)|) \qquad S_p = O(|Q(D)|)$$

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

$$\textit{delay } \delta = O(1) \qquad \textit{space } S_e = O(1)$$

The drawback of Proposition 1 is that the user may have to wait in the worst case $\tilde{O}(|Q(D)|)$ 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. Hence, our goal is 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 1. *[26] Let Q be a full 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 and space,*

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

such that we can enumerate the results of $Q(D)$ with

$$\textit{delay } \delta = O(1) \qquad \textit{space } S_e = O(1)$$

For acyclic queries, $\mathsf{fhw} = 1$, 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 2 (Main Theorem)**.** *Let Q be a full 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 input database D, we can pre-process D in time and space,*

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

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

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

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 smallest tuple, or all tuples in the result. However, if the user decides to stop the enumeration after having obtained the first k results, the space used during enumeration will be bound by $O(k)$. We should also note that, although Theorem 2 works when given a ranking function, all of our algorithms work in the case where the ordering of the tuples/valuations is expressed through a `comparable` function that, given two valuations, returns which one is the largest one.

It is instructive to compare Theorem 2 with Theorem 1, 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 Section 4, we will discuss techniques that can reduce the space S_e for certain queries.

In the remaining of this section, we will first present a few applications of Theorem 2. Then, in Section 3 we will discuss the proof of the main theorem in detail, by presenting and analyzing our algorithmic construction.

3.1 Applications

In this section, we show how to apply Theorem 2 to obtain algorithms for different types of ranking functions.

Vertex-Based Ranking. A vertex-based ranking function over $\mathcal{V}$ is of the form:

$$\text{rank}(\theta) = \oplus_{x \in \mathcal{V}} f_x(\theta(x))$$

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. Moreover, we say that the function is monotone if $a \geq b$ implies that $a \oplus c \geq b \oplus c$. Such examples are $\langle \mathbb{R}, + \rangle$, $\langle \mathbb{R}, * \rangle$, and $\langle U, max \rangle$, where U is bounded.

Lemma 1. *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:

$$\mathsf{rank}(\theta) = \oplus_{F \in \mathcal{E}} w_F(\theta[x_F])$$

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 the binary operator $\oplus$.

Lemma 2. *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 2. *Let Q be a full 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 and space,*

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

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

$$\delta = \tilde{O}(1) \qquad S_e = O(\min\{k, |Q(D)|\})$$

For instance, if the query is acyclic, hence $\mathsf{fhw} = 1$, the above theorem gives an algorithm with linear preprocessing time $O(|D|)$ and $\tilde{O}(1)$ 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 $\langle x_{i_1}, \dots, x_{i_k} \rangle$ 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_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, and then defining $\mathsf{rank}(\theta) = \sum_x w_x f_x(\theta(x))$, where w_x are appropriately chosen constants. It is easy to see that this ranking function is actually a monotone vertex-based ranking, and hence Theorem 2 applies here as well. Interestingly, if the lexicographic order "agrees" with the rooted 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 possible to get an even better result than Theorem 2, by achieving constant delay $O(1)$, and constant space S_e.

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 [19]: c-determined implies c-bounded, but not vice versa. For c-bounded ranking functions, we can show the following result:

Proposition 3. *Let Q be a full 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 and space,*

$$T_p = O(|D|^{\mathsf{fhw}+c}) \qquad S_p = O(|D|^{\mathsf{fhw}+c})$$

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

$$\delta = \tilde{O}(1) \qquad S_e = O(\min\{k, |Q(D)|\})$$

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 $S \subseteq \mathsf{key}(t)$. Hence, if we fix a valuation over $\mathsf{key}(t)$, the ranking function will output exactly the same score, independent of what values the other variables take. □

3.2 Proof of Main Theorem

At a high level, each node t in the 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 v 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 $C(v) = \langle v, [], \perp \rangle$, where $\perp$ is used to denote a null pointer. Otherwise, suppose that t has k children $t_1, \dots, t_n$. Then, $C(v) = \langle v[\mathcal{B}_t], [p_1, \dots, p_n], q \rangle$, where p_i is a pointer to the cell $C(v[\mathcal{B}_{t_i}^{\prec}])$ 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 $C(v)$, one can reconstruct v 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 $\mathsf{key}(\mathcal{B}_t)$ to a *priority queue* $\mathfrak{Q}_t[u]$. The elements of $\mathfrak{Q}_t$ are cells $C(v)$, where v is a valuation over $\mathcal{B}_{\mathcal{B}_t}^{\prec}$ such that $u = v[\mathsf{key}(\mathcal{B}_t)]$. 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 [9]) 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 $\mathsf{key}(\mathcal{B}_t)$, we will order the valuations v over $\mathcal{B}_{\mathcal{B}_t}^{\prec}$ that agree with u according to the score: $\mathsf{rank}(w_{t,u}^{\star} \circ v)$ where $w_{t,u}^{\star}$ 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 $w^{\star}_{t,u}$ with any other valuation.

We next discuss in detail the *preprocessing* and *enumeration* phase of the algorithm.

Preprocessing. The preprocessing phase is presented in Algorithm 1.

Algorithm 1: Preprocessing Phase

```
1  foreach t ∈ V(T) do
2  |   materialize the bag B_t
3  perform full reducer pass on materialized bags in T
4  forall t ∈ V(T) in post-order traversal do
5  |   foreach valuation v in bag B_t do
6  |   |   u ← v[key(B_t)]
7  |   |   if Q_t[u] is NULL then
8  |   |   |   Q_t[u] ← new priority queue
9  |   |   ℓ ← []
10 |   |   foreach child s of t do
11 |   |   |   ℓ.append(&Q_s[v[key(B_s)]].top())
12 |   |   Q_t[u].insert(⟨v, ℓ, ⊥⟩)
```

The algorithm consists of two steps. The first step works exactly as the preprocessing phase 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 v over $\mathcal{B}_t$ is added to the corresponding queue as the triple $\langle v, [], \perp\rangle$. If 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 $\mathsf{key}(r) = \{\}$.

Example 6. *As a running example for this section, we consider the following 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.

id	w_1	x	y
1	1	1	1
2	2	2	1

R_1

id	w_2	y	z
1	1	1	1
2	1	3	1

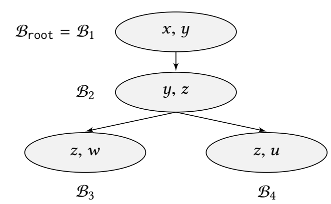
R_2

id	w_3	z	w
1	1	1	1
2	4	1	2

R_3

id	w_3	z	u
1	1	1	1
2	5	1	2

R_4



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, v in each cell $\langle v, [p_1, \ldots, p_k], next\rangle$ 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 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 $\langle 1, [\bullet\bullet], \perp\rangle$ 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.

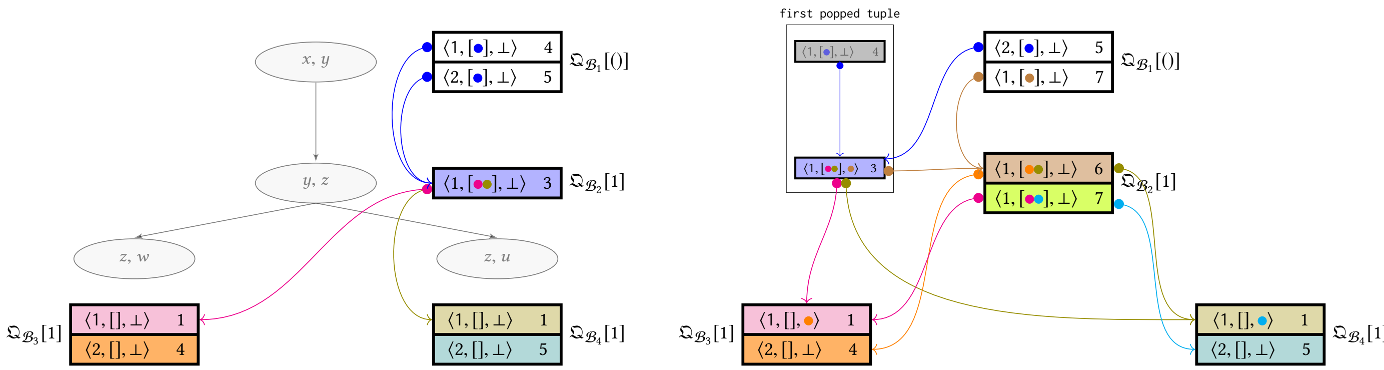
The next lemma analyzes the runtime of the preprocessing phase, as well as the space of the data structure at the end of preprocessing.

Lemma 3. *The runtime of Algorithm 1 is $O(|D|^{\mathsf{fhw}})$. Moreover, at the end of the algorithm, the resulting data structure has size $O(|D|^{\mathsf{fhw}})$.*

Enumeration. We next describe how enumeration is performed. Algorithm 2 gives the algorithm in detail.

The key idea of the algorithm 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.

Example 7. *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 $\langle 1, [\bullet], \perp\rangle$ 4 is popped from $\mathfrak{Q}_{\mathcal{B}_1}[()]$ (after satisfying If condition 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 $\langle 1, [\bullet\bullet], \perp\rangle$ 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 $\langle 1, [], \perp\rangle$ 1. The least ranked tuple but larger than $\langle 1, [], \perp\rangle$ 1 in $\mathfrak{Q}_{\mathcal{B}_3}[1]$ is cell at address ●. Thus, next for $\langle 1, [], \perp\rangle$ 1 is updated to ● and cell at ● is returned which leads to*



(a) Priority queue state (mirroring the decomposition) after pre-processing phase.

(b) Priority queue state after one iteration of loop in procedure `enum()`**.**

Figure 1: Pre-processing 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.

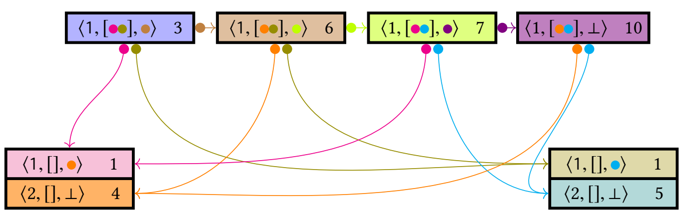


Figure 2: Ranked materialized output of subtree rooted at $\mathcal{B}_2$ as a sequence of pointers after full enumeration is complete

Algorithm 2: Enumeration Phase

PROCEDURE `enum()`
 while $\mathcal{Q}_r[()]$ *is not empty* **do**
  output $\mathcal{Q}_r[()].top()$
  `topdown`$(\mathcal{Q}_r[()].top(), r)$

PROCEDURE `topdown`(c, t)
 /* $c = \langle v, [p_1, \ldots, p_k], next\rangle$ */
 $u \leftarrow v[\mathsf{key}(\mathcal{B}_t)]$
 if $next = \perp$ **then**
  $\mathcal{Q}_t[u].pop()$
  foreach *child* t_i *of* t **do**
   $p'_i \leftarrow$ `topdown`$(*p_i, t_i)$
   if $p'_i \neq \perp$ **then**
    $\mathcal{Q}_t[u].insert(\langle v, [p_1, \ldots, p'_i, \ldots p_k], \perp\rangle)$
  if t *is not the root* **then**
   $next \leftarrow \&\mathcal{Q}_t[u].top()$
 return $next$

creation and insertion of ⟨1, [••], ⊥⟩ 6 *cell in* $\mathcal{Q}_{\mathcal{B}_2}[1]$. *Similarly, we get the other cell in* $\mathcal{Q}_{\mathcal{B}_2}[1]$ *by recursive call for* $\mathcal{B}_4$. *After both the calls are over for node* $\mathcal{B}_2$, *the topmost cell at* $\mathcal{Q}_{\mathcal{B}_2}[1]$ *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* $Q_{\mathcal{B}_1}[()]$ ⟨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.* ⟨2, [•], ⊥⟩ 5 *is garbage collected. As the enumeration continues, we are materializing the output of each subtree on-the-fly. Figure 2 shows the eventual sequence of pointers at node* $\mathcal{B}_2$ *which is the ranked materialized output of subtree rooted at* $\mathcal{B}_2$.

We next analyze the behavior of the algorithm in terms of delay, space and correctness.

Lemma 4. *Algorithm 2 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 $\perp$, the processing takes time $O(1)$. If $next = \perp$, it will perform one pop – with cost $O(\log |D|)$ – and a number of inserts equal to the number of children. 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 5. *After Algorithm 2 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 $|Q(D)|$. [2]

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). 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 $k \cdot O(1) = O(k)$. □

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

Lemma 6. *Algorithm 2 enumerates $Q(D)$ in order according to* `rank`.

4 REDUCING MATERIALIZATION SPACE

A direct application of Theorem 2 can lead to cases where, if enumeration is carried to the end, almost all the tuples in the output are materialized. In this section, we show how to incorporate instance specific properties to reduce this space requirement.

Before we state the main result, we introduce some notation. For a node t of a tree decomposition, we denote by $\mathsf{OUT}(\mathcal{B}_t)$ the materialized output of its bag, and by $\mathsf{OUT}(\mathcal{B}_t^{\prec})$ the output of the subtree rooted at t. For a valuation θ, we use $R \ltimes \theta$ to denote the restriction of the set R on valuations that agree with θ.

Theorem 3. *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, *such that the root node r has two children b, c. Then, for any input database D and ranking function compatible with $\mathcal{T}$, we can preprocess D in time and space,*

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

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

$$\begin{aligned} &\textit{delay } \delta = \tilde{O}(1) \\ &\textit{space } S_e = O(|D|^{\mathsf{fhw}} + \min\{k, |\mathsf{OUT}(\mathcal{B}_b^{\prec})| + |\mathsf{OUT}(\mathcal{B}_c^{\prec})| + \\ &\qquad \sum_{\theta \in \mathsf{OUT}(\mathcal{B}_r)} \min\{|\mathsf{OUT}(\mathcal{B}_b^{\prec}) \ltimes \theta|, |\mathsf{OUT}(\mathcal{B}_c^{\prec}) \ltimes \theta|\}\}) \end{aligned}$$

Before we show how to prove the theorem, we demonstrate how to apply Theorem 3 to different cases.

Example 8. *Consider the 3-path query $Q(x, y, z) = R(x, y), S(y, z), T(z, w)$. Fix the decomposition where $\{y, z\}$ is at the root with the one child $\mathcal{B}_b = \{x, y\}$ and the other child $\mathcal{B}_c = \{z, w\}$. For simplicity, we fix the size of each relation as N. This gives $|\mathsf{OUT}(\mathcal{B}_b^{\prec})| = |\mathsf{OUT}(\mathcal{B}_c^{\prec})| = N$. Further, we have the following bound:*

$$\sum_{\theta \in \mathsf{OUT}(\mathcal{B}_r)} \min\{|\mathsf{OUT}(\mathcal{B}_b^{\prec}) \ltimes \theta|, |\mathsf{OUT}(\mathcal{B}_c^{\prec}) \ltimes \theta|\} \leq N^{1/2}|\mathsf{OUT}|^{1/2}$$

Then, Theorem 3 gives space $S_e = O(N + \min\{k, \sqrt{N \cdot |\mathsf{OUT}|}\})$. In the worst-case $|\mathsf{OUT}| = N^2$ and $S_e = O(N^{3/2})$, while Theorem 2 requires $\Omega(N^2)$ space.

[2]Here we must make sure that the priority queue is implemented such that any duplicate valuations are rejected.

Generalizing the 3-path example to t-path queries (each of fixed size N), we can achieve linear time preprocessing with $\tilde{O}(1)$ delay using space $S_e = O(N + \min\{k, N^{(\rho^*+1)/2}\})$.

If for a specific tree decomposition $\mathcal{T}$ the root has more than two children, we can still apply the theorem by modifying the decomposition as follows: we partition the children into two subsets S_b, S_c, and create two new nodes b, c. The bag of node b contains all the variables from S_b that occur in the root, and similarly for c. Node b is connected to all nodes in S_b, as well as the root node (similarly for c). Note that this transformation does not change the fractional hypertree width of the decomposition.

Example 9. *Consider the cartesian product query $Q(x, y, z, w) = R(x), S(y), T(z), U(w)$. Let the decomposition for Q have $\{x\}$ at the root and $\{y\}, \{z\}, \{w\}$ as its children. We create two nodes b, c, with empty bags, where b is connected to $\{y\}, \{z\}, \{x\}$, and c to $\{w\}, \{x\}$. One can see that the resulting space guarantee during enumeration becomes $S_e = O(N^2)$, where N is the size of each relation.*

The above idea can be generalized as follows.

Proposition 4. *Consider a cartesian product query $Q(x_1, \ldots, x_t) = R_1(x_1), \ldots, R_t(x_t)$. For any totally decomposable ranking function, we can enumerate the top-k tuples after linear time preprocessing with $\tilde{O}(1)$ delay using space $S_e = O(N + \min\{k, O(N^{\lceil \frac{t}{2} \rceil})\}$*

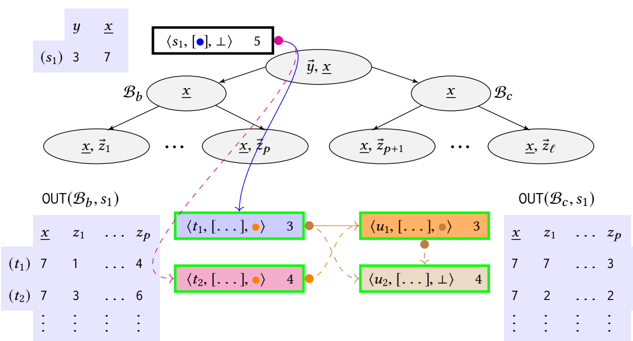
In the case where the root node of the decomposition has only one child (say t), we can again transform it accordingly. Indeed, we can create a new decomposition $\mathcal{T}'$ (with the same width) where $\mathcal{B}_r \cap \mathcal{B}_t$ is the new root, $\mathcal{B}_r$ is the left child and $\mathcal{B}_t$ is the right child. By doing this transformation, Theorem 3 implies that the space guarantee reduces from $|\mathsf{OUT}|$ to $|\mathsf{OUT}(\mathcal{B}_t^{\prec})|$, which can be much smaller.

Example 10. *Consider the 2-path query $Q(x, y, z) = R(x, y), S(y, z)$. For any totally decomposable ranking function, we can enumerate the top-k tuples after linear time preprocessing with $\tilde{O}(1)$ delay using only $O(N)$ space in the worst case.*

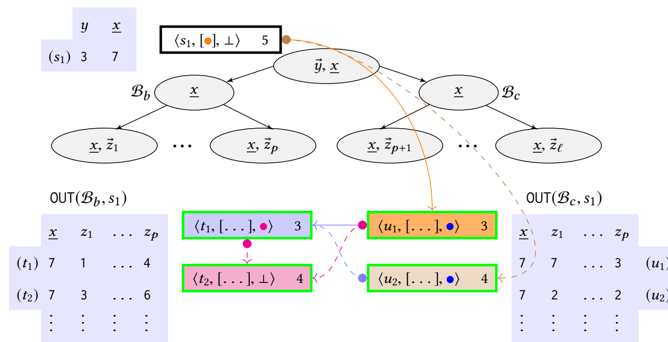
It is an interesting question to find the best instance-specific decomposition according to the data, query structure and ranking function; we leave this as a problem for future research.

4.1 Modified algorithm

Before we describe the main algorithm, we present some intuition. Figure 4 shows the conceptual difference between Theorem 2 and the modified algorithm. For a fixed root valuation, algorithm from previous section would generate next candidates by incrementally enumerating the cartesian product. This is inefficient as it inevitably takes $|\mathsf{OUT}_b| \cdot |\mathsf{OUT}_c|$ space in the worst-case. The proposed change in this section enumerates the next candidates by keeping $|\mathsf{OUT}_b|$ pointers to OUT_c which is enough to get space savings. We now describe the modifications. To simplify the description, we assume that for each $\theta \in \mathsf{OUT}(\mathcal{B}_r)$, $d^+(\mathcal{B}_b^{\prec}, \theta) = |\mathsf{OUT}(\mathcal{B}_b^{\prec}) \ltimes \theta|$ (and similarly for c) is already computed. Further we will create a modified cell structure only for node b and c, $S(v) = \langle v, [p_1, \ldots p_i], next\rangle$. $[p_1, \ldots p_i]$ are pointers to cells of children nodes of b and c. Additionally, it also includes a pointer *next* that will only point to an object of type modified cell. We will modify our algorithm in the following way:



(a) Pointers from OUT($\mathcal{B}_b$) to OUT($\mathcal{B}_c$)



(b) Pointers from OUT($\mathcal{B}_c$) to OUT($\mathcal{B}_b$)

Figure 3: Modified algorithm example

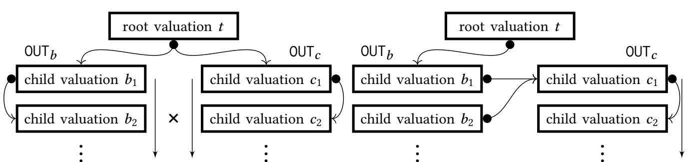


Figure 4: (left) Theorem 2 enumerates result by cartesian product; (right) modified algorithm that enumerates result via $|\texttt{OUT}_b|$ pointers

(1) *Initialization of* $\mathfrak{Q}_r[()]$: The preprocessing phase remains exactly the same as algorithm 1 for all bags except the root and its children. Once all other nodes have been processed, for each root node valuation θ, we create modified cells for nodes b and c (if they don't already exist). Let $\theta(e) = \theta[\mathsf{key}(\mathcal{B}_e)]$ for some node e.

$$\theta_c(1) = \langle \theta(c), [q_1, \dots q_\ell], \perp \rangle$$
$$\theta_b(1) = \langle \theta(b), [p_1, \dots p_m], \&\theta_c(1) \rangle$$

where $p_i = \&\mathfrak{Q}_s[\theta(s)].top(), i \in [m]$ for each child s of $\mathcal{B}_b$ and $q_i = \&\mathfrak{Q}_s[\theta(s)].top(), i \in [\ell]$ for each child s of $\mathcal{B}_c$. Intuitively, $\theta_b(1)$ and $\theta_c(1)$ are the smallest ranked valuation over subtree rooted at node b and c. We insert them into priority queue $\mathfrak{Q}_b[\theta(b)]$ and $\mathfrak{Q}_c[\theta(c)]$ respectively. Finally, we push $C(\theta) = \langle \theta, [\&\theta_b(1)], \perp \rangle$ into $\mathfrak{Q}_r$.

(2) *Enumeration phase* : Suppose that output valuation $C(\theta)$ is popped from $\mathfrak{Q}_r[()]$ pointing to modified cell $\theta_b(i)$ that in turn points to $\theta_c(j)$. Then, we will push two new cells into root priority queue. First,
- find the next ranked valuation after $\theta_c(j)$ (say $\theta_c(j+1)$)
- set $\theta_b(i).next = \&\theta_c(j+1)$
- chain $\theta_c(j).next = \&\theta_c(j+1)$
- insert $C'(\theta) = \langle \theta, [\&\theta_b(i)], \perp \rangle$ into root priority queue.

We will discuss in a bit how to find $\theta_c(j+1)$. Second, if $\pi_{\mathcal{B}_b}(\theta)$ is not the last valuation in $\texttt{OUT}(\mathcal{B}_b^{\prec}) \ltimes \theta$,
- find next ranked valuation after $\theta_b(i)$ (say $\theta_b(i+1)$)
- set $\theta_b(i+1).next = \&\theta_c(i)$
- insert $C'(\theta) = \langle \theta, [\&\theta_b(i+1)], \perp \rangle$.

In other words, in the worst-case, for each valuation $\theta \in \texttt{OUT}(\mathcal{B}_r)$ there will be at most $d^+(\mathcal{B}_b^{\prec}, \theta)$ pointers from each valuation in $\texttt{OUT}(\mathcal{B}_b^{\prec}) \ltimes \theta$ to $\texttt{OUT}(\mathcal{B}_c^{\prec}) \ltimes \theta$. The first observation is that we can use procedure topdown from Theorem 2 to enumerate $\texttt{OUT}(\mathcal{B}_b^{\prec})$ and $\texttt{OUT}(\mathcal{B}_c^{\prec})$ for a given key valuation in ranked order. This is possible because any compatible ranking function over the decomposition $\mathcal{T}$ is also compatible for any subtree of the decomposition. Thus, we can use topdown from algorithm 2 to find $\theta_b(i+1)$ and $\theta_c(j+1)$.

Example 11. *Figure 3a shows an example instance to illustrate the idea. Green bordered cells are instances of modified cells $S(v)$, i.e the cells that store valuations over subtree $\mathcal{B}_b$ or $\mathcal{B}_c$. After the preprocessing stage, cell with s_1 points to $b(s_1, 1)$ (modified cell t_1) shown in blue arrow. $b(s_1, 1)$ in turn points to $c(s_1, 1)$ (modified cell u_1). The first answer is formed by $s_1 \to t_1 \to u_1$. The next smallest valuation after u_1 is u_2. We update $u_1.next = u_2$ and insert $s_1 \to t_1 \to u_2$ (shown by dashed brown arrow from t_1 to u_2). Additionally, we insert $s_1 \to t_2 \to u_1$ (dashed magenta and orange arrow) into $\mathfrak{Q}_r$. t_1, t_2 and u_1, u_2 are constructed using* topdown *procedure from algorithm 2.*

Let us now analyze the space requirement of this algorithm,

$$S_1 = |\texttt{OUT}(\mathcal{B}_c^{\prec})| + \sum_{\theta \in \texttt{OUT}(\mathcal{B}_r)} d^+(\mathcal{B}_b^{\prec}, \theta)$$

However, we could have also chosen to have pointers from $\texttt{OUT}(\mathcal{B}_c^{\prec}) \ltimes \theta$ to $\texttt{OUT}(\mathcal{B}_b^{\prec}) \ltimes \theta$ as shown in Figure 3b. In this case, the space requirement would be,

$$S_2 = |\texttt{OUT}(\mathcal{B}_b^{\prec})| + \sum_{\theta \in \texttt{OUT}(\mathcal{B}_r)} d^+(\mathcal{B}_c^{\prec}, \theta)$$

Our second key observation is that we can use the power of two choices. In particular, for each root valuation t, we can choose which side we keep pointers from based on the smaller of $d^+(\mathcal{B}_b^{\prec}, \theta)$ and $d^+(\mathcal{B}_c^{\prec}, t)$. This will lead to the following space bound,

$$S = \sum_{e \in \{b,c\}} |\texttt{OUT}(\mathcal{B}_e^{\prec})| + \sum_{\theta \in \texttt{OUT}(\mathcal{B}_r)} \min\{d^+(\mathcal{B}_c^{\prec}, \theta), d^+(\mathcal{B}_b^{\prec}, \theta)\}$$

We conclude this section by removing the assumption about $d^+(\mathcal{B}_e^{\prec}, \theta)$. The key observation is that once the dangling tuples have been removed in the preprocessing phase, $d^+(\mathcal{B}_e^{\prec}, \theta)$ can be computed in bottom-up fashion by counting the number of tuples for a particular valuation of $t[\mathsf{key}(\mathcal{B}')]$ in the children bags $\mathcal{B}'$ and taking their product. For the base case of leaf nodes, this can simply be done by counting $|\mathsf{OUT}(\mathcal{B}) \ltimes \theta[\mathsf{key}(\mathcal{B})]|$ in a linear pass.

5 LOWER BOUNDS

In this section, we provide evidence for the (near) optimality of some 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 the following 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* ranking function. Note that all of our algorithms work under the blackbox assumption.

Proposition 5. *Let Q be a natural join query, and* rank *a blackbox ranking function. Then, any enumeration algorithm on a database D needs $\Omega(|Q(D)|)$ calls to* 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 1 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 a *coordinate monotone* function.

Definition 4. *Let* rank *be a ranking function over a set of variables $\mathcal{V}$. We say that* rank *is* coordinate monotone *if for every $x \in \mathcal{V}$ there exists a total order on $\mathbf{dom}(x)$ such that for every two valuations θ_1, θ_2 where $\theta_1(x) \geq \theta_2(x)$ for every x, we have $\mathsf{rank}(\theta_1) \geq \mathsf{rank}(\theta_2)$.*

In other words, θ_1 *dominates* θ_2. All coordinate decomposable functions are coordinate monotone but not vice-versa.

Example 12. *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 $\mathbf{dom} = \{0, 1\}$, and define the ranking function to be $\mathsf{rank}(\theta) = \sum_{i=1}^{d} \theta(x_i) \cdot \theta(y_i)$: this corresponds to taking the inner product of the input tuples if viewed as binary vectors. This ranking function is coordinate monotone (the total order is $1 > 0$), but it is not coordinate decomposable.

For coordinate monotone functions, we can show the following lower bound result:

Lemma 7. *Consider the query*

$$Q(x_1, y_1, \dots, x_\ell, y_\ell, z) = R_1(x_1, y_1), S_1(y_1, z), \dots, R_\ell(x_\ell, y_\ell), S_\ell(y_\ell, z)$$

and let rank *be a blackbox coordinate monotone ranking function. Then, there exists an instance of size N such that the time required to find the smallest tuple is $\Omega(N^\ell)$.*

Proof. We construct an instance D of size N, as shown in Figure 5 for $\ell = 2$. Variable z takes exactly one value (c in the figure) and has a weight of 0. Each tuple in relation $R_i(x_i, y_i)$ joins with exactly one tuple from $S_i(y_i, z)$. The invariant for weight assignment is that $w_1 + w_2 = n + 1$ where w_1 is the weight assigned to valuation of x_i and w_2 is the weight of valuation y_i.

Let $t_1 \in Q(D)$ have the following weight vector according to weights of vertices $s_1 = \langle u_1, n - u_1 - 1, \dots, u_\ell, n - u_\ell - 1, 0\rangle$. Let $s_2 = \langle v_1, n - v_1 - 1, \dots, v_\ell, n - v_\ell - 1, 0\rangle$ be the weight vector for any other tuple $t_2 \neq t_1$. Note that each $u_i, v_i \in [n]$. We will show that under coordinate monotonicity property, s_1 and s_2 are *incomparable*, i.e., neither vector dominates the other.

Suppose that s_1 dominates s_2. Then, it must hold for each $i \in [\ell]$ that $u_i \geq v_i$ and $n - u_i - 1 \geq n - v_i - 1$, giving $u_i = v_i$. But this contradicts our assumption that $t_1 \neq t_2$ as no two tuples have the same weight vector. Therefore, no two weight vector corresponding to output tuples dominate each other and rank can assign an arbitrary score to tuples without violating coordinate monotonicity. Thus, any algorithm that does not examine all N^ℓ output tuples can miss the smallest tuple t, since we can always assign in the blackbox model the smallest value to $\mathsf{rank}(t)$. □

Lemma 7 shows that for coordinate monotone functions, there exist queries where obtaining constant (or almost constant) delay guarantee requires the algorithm to spend significant time during the preprocessing step. For the lower bound instance, $\rho^* = \ell + 1$. Thus, the preprocessing step must spend time $T_P = O(N^{\rho^*-1})$, effectively materializing the join output [3]. Given this result, the next immediate question is to see if we can extend the lower bound to other CQs. To this end, we show in the example below that for coordinate monotone functions, there also exist queries where linear time preprocessing is sufficient for $\tilde{O}(1)$ delay enumeration.

Example 13. *Consider the cartesian product query $Q(x_1, x_2, \dots, x_\ell) = R_1(x_1), \dots, R_\ell(x_\ell)$. We will construct a rooted tree decomposition such that any coordinate monotone function* rank *is compatible with it. Let the root node r have $\mathcal{B}_r = \{x_1\}$ and $\ell - 1$ children $t_1, \dots, t_{\ell-1}$ where child $\mathcal{B}_{t_i} = \{x_i\}$. For each t_i, $\mathsf{key}(t_i) = \{\}$ and $\mathcal{B}_{t_i}^{\prec} = \{x_i\}$. The key observation is that coordinate monotonicity implies that* rank *is $\{x_i\}$-decomposable. Similarly, for the root node $\mathcal{B}_r^{\prec} = \{x_1, x_2, \dots, x_\ell\}$ and $\mathsf{key}(r) = \{\}$, hence $\mathcal{B}_r^{\prec} \setminus \mathsf{key}(r)$-decomposability is trivially true.*

Then, Theorem 2 implies that for any coordinate monotone ranking function, we can enumerate the ranked result of Q with $\tilde{O}(1)$ delay after linear time preprocessing.

The above result shows that enumeration for coordinate monotone functions is dependent on query structure and whether there exists a compatible decomposition for rank.

[3] assuming we can afford N^ℓ space; we leave the study of space time tradeoffs as future work.

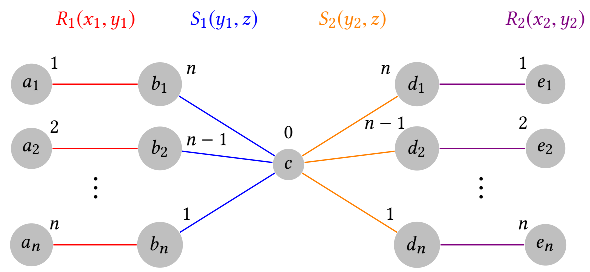


Figure 5: Database instance D for Lemma 7. Each edge is color coded by the relation it belongs to. Values over vertices denote the weight assigned to the vertex.

5.2 Beyond Logarithmic Delay

Next, we examine whether the logarithmic factor that we obtain in the delay of Theorem 2 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 [13] 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 [32] gave a simple algorithm that takes $O(n^2 \log n)$ time while using only $O(n^2)$ comparisons.

Conjecture 1. *The* $X + Y$ sorting *problem does not admit an* $O(n^2)$ *time algorithm.*

In our setting, $X + Y$ sorting can 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 $\mathtt{rank}(x, y) = x + y$. Conjecture 1 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 [18, 20, 21, 27] and uncertain databases [28, 34]. 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 number of operations required per answer tuple. We refer the reader to [17] for a comprehensive survey of top-k processing techniques in relational databases. More recent work [7, 15] 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 [19] and [33]. Algorithm in [19] 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. [33] 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.

Rank aggregation algorithms. Top-k processing over ranked lists of objects has a rich history. The problem was first studied by Fagin et al. [11, 12] where the database consists of N objects and m ranked streams, each contain 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 which then act as database objects. More importantly, TA can only give $O(N)$ delay guarantee using $O(N)$ space.

[22] extended the problem setting to the case where we want to enumerate top-k answers for join of 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 $\Omega(|\mathtt{OUT}|)$ with space requirement $S = \Omega(|\mathtt{OUT}|)$. 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 [2]. 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 [29, 31]. 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 [5, 6].

Factorized databases. Following the landmark result of [26] 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 [3, 4, 10, 25]. The core idea of factorized databases is to convert an arbitrary query into an acyclic query by finding a query

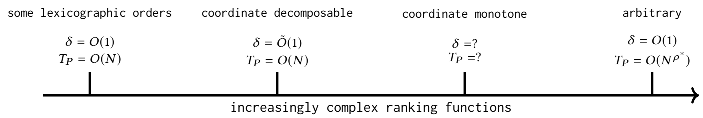


Figure 6: Algorithmic performance on various classes of ranking functions on acyclic CQs.

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 directly 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 (almost) constant delay enumeration with a small preprocessing time. Our techniques use on-the-fly materialization and instance-specific properties to achieve non-trivial guarantees. We view this as a fundamental building block to answer several more interesting questions of both theoretical and practical relevance.

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 some compatible ranking function with the 4-path query and $T_P = O(N)$, we can achieve the following at two extremes of the tradeoff,

$$S_e = O(N), \quad \delta = \tilde{O}(N^{3/2}) \quad [\text{using } 3 - \texttt{SUM} \text{ algorithm}]$$
$$S_e = O(N^2), \quad \delta = \tilde{O}(1) \quad [\text{due to Theorem 2}]$$

The precise characterization and its generalization to arbitrary CQs between these two points is unknown. Extension of our techniques to a broader class of queries (such as CQs with projections) is also an interesting problem. There also remain several open question regarding the properties of ranking functions. Figure 6 shows some key results for different choices of ranking functions.

It would be interesting to find fine-grained classes of ranking functions which are more expressive than coordinate decomposable, but less expressive than coordinate monotone. For instance, the ranking function $f(x, y) = |x - y|$ is not coordinate decomposable, but it is *piecewise* coordinate monotone on either side of the global minimum critical point for each x valuation. Lastly, it remains on our research agenda to test the proposed ideas in a practical setting to see if the theoretical gains translate into practice.

REFERENCES

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

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

[3] N. Bakibayev, T. Kočiskỳ, D. Olteanu, and J. Závodnỳ. Aggregation and ordering in factorised databases. *Proceedings of the VLDB Endowment*, 6(14):1990–2001, 2013.

[4] N. Bakibayev, D. Olteanu, and J. Závodnỳ. Fdb: A query engine for factorised relational databases. *Proceedings of the VLDB Endowment*, 5(11):1232–1243, 2012.

[5] C. Berkholz, J. Keppeler, and N. 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.

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

[7] L. Chang, X. Lin, W. Zhang, J. X. Yu, Y. Zhang, and L. Qin. Optimal enumeration: Efficient top-k tree matching. *Proceedings of the VLDB Endowment*, 8(5):533–544, 2015.

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

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[10] S. Deep and P. 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.

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

[12] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences*, 66(4):614–656, 2003.

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

[14] G. Gottlob, G. Greco, and F. Scarcello. Treewidth and hypertree width. *Tractability: Practical Approaches to Hard Problems*, 1, 2014.

[15] M. Gupta, J. Gao, X. Yan, H. Cam, and J. Han. Top-k interesting subgraph discovery in information networks. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 820–831. IEEE, 2014.

[16] J. E. Hopcroft, J. D. Ullman, and A. Aho. The design and analysis of computer algorithms, 1975.

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

[18] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid. Rank-aware query optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 203–214. ACM, 2004.

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

[20] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. 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.

[21] C. Li, M. A. Soliman, K. C.-C. Chang, and I. 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.

[22] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, volume 1, pages 281–290, 2001.

[23] H. Q. Ngo, E. Porat, C. Ré, and A. 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.

[24] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2013.

[25] D. Olteanu and M. Schleich. Factorized databases. *ACM SIGMOD Record*, 45(2):5–16, 2016.

[26] D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2, 2015.

[27] Y. Qi, K. S. Candan, and M. L. 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.

[28] C. Re, N. Dalvi, and D. 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.

[29] L. 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.

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

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

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

[33] X. Yang, D. Ajwani, W. Gatterbauer, P. K. Nicholson, M. Riedewald, and A. 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.

[34] Z. Zou, J. Li, H. Gao, and S. Zhang. Finding top-k maximal cliques in an uncertain graph. 2010.

A PROOFS

A.1 Proofs for Section 3.1

Proof of Lemma 1. 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. Suppose that $\mathsf{rank}(\theta^\star \circ \theta_1) \geq \mathsf{rank}(\theta^\star \circ \theta_2)$ for valuations θ_1, θ_2 over S. This implies that

$$\oplus_{x \in S} f_x(\theta_1(x)) \geq \oplus_{x \in S} f_x(\theta_2(x)).$$

Then, for any valuation θ over $\mathcal{V} \setminus S$ we have:

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

The inequality holds because of the monotonicity of the binary operator. □

Proof of Lemma 2. Pick some node t in the decomposition, and fix a valuation θ_0 over $\mathsf{key}(t)$. 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 $\mathcal{B}_t^\prec \setminus \mathsf{key}(t)$.

Suppose now that $\mathsf{rank}(\theta_0 \circ \theta^\star \circ \theta_1) \geq \mathsf{rank}(\theta_0 \circ \theta^\star \circ \theta_2)$ for valuations θ_1, θ_2 over $\mathcal{B}_t^\prec \setminus \mathsf{key}(t)$. This implies that

$$\oplus_{F \in E} w_F((\theta_0 \circ \theta_1)[x_F]) \geq \oplus_{F \in E} w_F((\theta_0 \circ \theta_2)[x_F]).$$

Then, for any valuation θ over $\mathcal{V} \setminus \mathcal{B}_t^\prec$ we have:

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

The inequality holds because of the monotonicity of the binary operator. □

Proof of Lemma 3. It is known that the materialization of each bag can be done in time $O(|D|^{\mathsf{fhw}})$, and the full reducer pass is linear in the size of the bags. 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 $O(|D|^{\mathsf{fhw}})$ 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 $O(|D|^{\mathsf{fhw}})$. □

A.2 Proofs for Section 3.2

Proof of Lemma 6. We will prove our claim by induction on post-order traversal of the decomposition. We will show that the priority queue for each node s gives the output in correct order which in turn populates $\mathsf{OUT}(\mathcal{B}_s^\prec)$ correctly. Here, $\mathsf{OUT}(\mathcal{B}_s^\prec)$ is the ranked materialized output of subtree rooted at $\mathcal{B}_s$.

Base Case. Correctness for ranked output of $\mathsf{OUT}(\mathcal{B}_s)$ for leaf node s is trivial as the leaf node tuples are popped from priority queues in order. Let $\phi^\star$ be the valuation over $\mathcal{V} \setminus \mathcal{B}_s$ according to definition of decomposability. We insert each valuation θ over node s with score $\mathsf{rank}(\phi^\star \circ \theta)$. Since rank is compatible with the decomposition, it follows that if $\mathsf{rank}(\phi^\star \circ \theta_2) \geq \mathsf{rank}(\phi^\star \circ \theta_1)$ such that $\theta_1[\mathsf{key}(\mathcal{S}_s)] = \theta_1[\mathsf{key}(\mathcal{S}_s)]$, then $\theta_2 \geq \theta_1$, thus recovering the correct ordering for tuple in s.

Let s be a non-leaf node whose children are leaf nodes $s_1, \ldots s_m$. Suppose θ is a valuation over $\mathcal{B}_s^\prec$ popped at line 9. Let $u = \theta[\mathsf{key}(\mathcal{B}_s)]$ and $u_i = \theta[\mathsf{key}(\mathcal{B}_{s_i})]$. From line 10-13, one may observe that a new candidate is pushed into priority queue for key by incrementing pointers to materialized output one at a time for each child bag $\mathcal{B}$, while keeping the remainder of tuple (including $\mathsf{key}(\mathcal{B})$) fixed (line 13). Let the notation $\phi^\succ$ denotes the smallest tuple in a bag that has rank greater than tuple ϕ such that $\phi^\succ[\mathsf{key}(\mathcal{B})] = \theta[\mathsf{key}(\mathcal{B})]$. Then, $\theta = \theta_1 \circ \theta_2 \circ \cdots \circ \theta_m$ will *generate* the following candidates:

$$\begin{aligned} \mathcal{L} = w &\circ \theta_1^\succ \circ \theta_2 \circ \cdots \circ \theta_m, \\ w &\circ \theta_1 \circ \theta_2^\succ \circ \cdots \circ \theta_m, \\ &\ldots \\ w &\circ \theta_1 \circ \theta_2 \circ \cdots \circ \theta_m^\succ \end{aligned}$$

Here w is the projection of θ over variables in $\mathcal{B}_s$ but not in any child node. However, we also need to argue that the next smallest valuation after θ that agrees with $\theta[\mathcal{B}_s]$ is one of the tuples in $\mathcal{L}$. Suppose there is a tuple $\theta' = w \circ \varphi_1 \circ \varphi_2 \circ \cdots \circ \varphi_m$ such that $\theta'[\mathcal{B}_s] = \theta[\mathcal{B}_s]$ [4]. For the sake of contradiction, suppose θ' has strictly smaller score than any tuple in $\mathcal{L}$ or θ but has not been enumerated yet. In other words, $\mathsf{rank}(\phi \circ \theta') < \mathsf{rank}(\phi \circ \theta)$ for any valuation ϕ over $\mathcal{V} \setminus \mathcal{B}_s^\prec$. Clearly, since θ' has smaller rank than θ but has not been enumerated, it follows that θ' has not been inserted into the priority queue. We will show that such a scenario will violate compatibility of ranking function. Recall that θ_1 is said to dominate θ_2 whenever $\theta_1(x) \geq \theta_2(x)$ for every variable $x \in S$ implies $\mathsf{rank}(\theta_1) \geq \mathsf{rank}(\theta_2)$. There are three possible scenarios regarding the tuples θ and θ':

(1) θ dominates θ' over each variable in $S_i = \mathcal{B}_{s_i}^\prec \setminus \mathsf{key}(\mathcal{B}_{s_i})$, $i \in [m]$. Note that this scenario would mean that θ' was generated before θ given our candidate generation logic line 10-9, violating our assumption that θ' has not been generated yet.

(2) θ' dominates θ over each variable in S_i. This scenario implies that $\mathsf{rank}(\phi \circ \theta) < \mathsf{rank}(\phi \circ \theta')$. Indeed, since rank is compatible with decomposition for each bag $\mathcal{B}_{s_i}$, it holds that $\mathsf{rank}(\phi' \circ \theta[S_i]) \leq \mathsf{rank}(\phi' \circ \theta'[S_i])$ which is implied from the leaf node ordering correctness. More formally,

[4] If $\theta'[\mathcal{B}_s] \neq \theta[\mathcal{B}_s]$, then smallest candidates of $\theta'[\mathcal{B}_s]$ will be compared with that of $\theta[\mathcal{B}_s]$ in $\mathfrak{Q}_s$

$$\begin{aligned}
\mathsf{rank}(\phi \circ w \circ \theta_1 \circ \ldots \theta_m) &\leq \mathsf{rank}(\phi \circ w \circ \varphi_1 \circ \theta_2 \ \ldots \theta_m) \\
&\leq \mathsf{rank}(\phi \circ w \circ \varphi_1 \circ \varphi_2 \ldots \theta_m) \\
&\ldots \\
&\leq \mathsf{rank}(\phi \circ w \circ \varphi_1 \circ \varphi_2 \ldots \varphi_m)
\end{aligned}$$

Each inequality is a successive application of $(\mathcal{B}^{\prec}_{s_i} \setminus \mathsf{key}(s_i))$-decomposability since $\varphi_i \geq \theta_i$ by domination assumption of θ' over θ.

(3) θ' and θ are incomparable. It is easy to see that all candidates in $\mathcal{L}$ dominate θ but are incomparable to each other. Also, the only way to generate new candidate tuples is line 10-13. Thus, if θ' is not in the priority queue, there are two possibilities. Either there is some tuple θ'' in the priority queue that is dominated by θ' and thus, $\mathsf{rank}(\theta'') \leq \mathsf{rank}(\theta')$. θ'' will eventually generate θ' via a chain of tuples that successively dominate each other. As θ was popped before θ'', it follows that $\mathsf{rank}(\theta) \leq \mathsf{rank}(\theta'') \leq \mathsf{rank}(\theta')$. The second possibility is that there is no such θ'', which will mean that θ and θ' are generated in the same for loop line 10. But this would again mean that θ' is in the priority queue. Both these cases violate our assumption that $\mathsf{rank}(\theta') < \mathsf{rank}(\theta)$.

Therefore, it cannot be the case that $\mathsf{rank}(\phi \circ \theta') < \mathsf{rank}(\phi \circ \theta)$ which proves the ordering correctness for node s. Since the output $\mathtt{OUT}(\mathcal{B}^{\prec}_s)$ is populated using this ordering form priority queue, it is also materialized (chaining of cells at line 15) in ranked order.

Inductive Case. Consider some node s in post-order traversal with children $s_1, \ldots s_m$. By induction hypothesis, the ordering of $\mathtt{OUT}(\mathcal{B}_{s_i})$ and correctness of $\mathfrak{Q}_{s_i}$ is guaranteed. Applying the same argument as in the base case, it is straightforward to show the correctness for bag s. This completes the proof.

It is easy to see that the algorithm indeed enumerates all tuples in $Q(D)$ since the full reducer pass removes all dangling tuples. □

B MODIFIED ALGORITHM

This section details algorithm 3 and algorithm 4 of Theorem 3 for the case when we choose to have pointers from $\mathtt{OUT}(\mathcal{B}^{\prec}_b) \ltimes \theta$ to $\mathtt{OUT}(\mathcal{B}^{\prec}_c) \ltimes$ for each $\theta \in \mathtt{OUT}(\mathcal{B}_r)$. Extension to the case where we choose the min degree is straightforward.

Algorithm 3: Preprocessing Phase

1 **foreach** $t \in V(\mathcal{T})$ **do**
2   *materialize the bag* $\mathcal{B}_t$
3 *perform full reducer pass on materialized bags in* $\mathcal{T}$
4 **forall** $t \in V(\mathcal{T})$ *in post-order traversal except root* **do**
5   **foreach** *valuation* v *in bag* $\mathcal{B}_t$ **do**
6         $u \leftarrow v[\mathsf{key}(\mathcal{B}_t)]$
7   **if** $\mathfrak{Q}_t[u]$ *is* NULL **then**
8             $\mathfrak{Q}_t[u] \leftarrow$ new priority queue
9         $\ell \leftarrow []$
10   **foreach** *child* s *of* t **do**
11             $\ell.append(\&\mathfrak{Q}_s[v[\mathsf{key}(\mathcal{B}_s)]].top())$
12         $\mathfrak{Q}_t[u].insert(\langle v, \ell, \perp \rangle)$
13 $\mathfrak{Q}_r \leftarrow$ new priority queue
14 **foreach** *valuation* θ *in bag* $\mathcal{B}_r$ **do**
15     $\theta(b) \leftarrow v[\mathsf{key}(\mathcal{B}_b)]$, $\theta(c) \leftarrow v[\mathsf{key}(\mathcal{B}_c)]$
16   **if** $\mathfrak{Q}_b[\theta_b]$ *is* NULL **then**
17         $\mathfrak{Q}_b[\theta_b] \leftarrow$ new priority queue
18   **if** $\mathfrak{Q}_c[\theta_c]$ *is* NULL **then**
19         $\mathfrak{Q}_c[\theta_c] \leftarrow$ new priority queue
20     $\ell \leftarrow [], w \leftarrow \emptyset$
21   **foreach** *child bag* s *of* c **do**
22         $\ell.append(\&\mathfrak{Q}_s[\theta(s)].top())$
23   /* $\circ$ operator joins tuples */
24         $w \leftarrow w \circ \mathfrak{Q}_s[\theta(s)].top().v$
25     $\mathfrak{Q}_c[\theta(c)].insert(\langle w, \ell, \perp \rangle)$
26     $\theta_c(1) \leftarrow \&\mathfrak{Q}_c[\theta(c)].top()$
27     $\ell \leftarrow [], w \leftarrow \emptyset$
28   **foreach** *child bag* s *of* b **do**
29         $\ell.append(\&\mathfrak{Q}_s[\theta(s)].top())$
30   /* $\circ$ operator joins tuples */
31         $w \leftarrow w \circ \mathfrak{Q}_s[\theta(s)].top().v$
32     $\mathfrak{Q}_b[\theta(b)].insert(\langle w, \ell, \perp \rangle)$
33     $\mathfrak{Q}_r[()].insert(\langle v, [\&\mathfrak{Q}_b[\theta(b)].top()], \perp \rangle)$

Algorithm 4: Enumeration Phase

PROCEDURE `enum()`
    while $\mathfrak{Q}_r[()]$ *is not empty* **do**
        output $\mathfrak{Q}_r[()].top()$
        `modifiedtopdown`$(\mathfrak{Q}_r[()].top(), r)$

PROCEDURE `modifiedtopdown`(c, t)
    /* $c = \langle \theta, [l], \perp \rangle$ for root node */
    $\theta(b) \leftarrow \theta[\mathsf{key}(\mathcal{B}_b)], \theta(c) \leftarrow \theta[\mathsf{key}(\mathcal{B}_c)]$
    if $*(*l.next).next = \perp$ **then**
        $\mathfrak{Q}_c[\theta(c)].pop()$
        /* $*l = \langle \phi, [c_1, \dots, c_\ell], next \rangle$ */
        `topdown`$(*(*l.next), c)$
    if $*(*l.next).next \neq \perp$ **then**
        $*l.next = *(*l.next).next$
        $\mathfrak{Q}_r[()].insert(\langle \theta, [l], \perp \rangle)$
    if *tuples left in* $\mathsf{OUT}(\mathcal{B}_b^{\prec})$ **then**
        $\mathfrak{Q}_b[\theta(b)].pop()$
        foreach *child bag* t_i *of* b **do**
            `topdown`$(*l, b)$
        if $\mathfrak{Q}_b[\theta(b)]$ *is not empty* **then**
            ref $\leftarrow \mathfrak{Q}_b[\theta(b)].top()$
            ref.$next \leftarrow \theta_c(1)$
            $\mathfrak{Q}_r[()].insert(\langle \theta, [\&\text{ref}], \perp \rangle)$