Ranked Enumeration of Join Queries with Projections

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

Xiao Hu
Duke University
xh102@cs.duke.edu

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

ABSTRACT

Join query evaluation with ordering is a fundamental data processing task in relational database management systems. SQL and custom graph query languages such as Cypher offer this functionality by allowing users to specify the order via the ORDER BY clause. In many scenarios, the users also want to see the first k results quickly (expressed by the LIMIT clause), but the value of k is not predetermined as user queries are arriving in an online fashion. Recent work has made considerable progress in identifying optimal algorithms for ranked enumeration of join queries that do *not* contain any projections. In this paper, we initiate the study of the problem of enumerating results in ranked order for queries *with projections*. Our main result shows that for any acyclic query, it is possible to obtain a near-linear (in the size of the database) delay algorithm after only a linear time preprocessing step for two important ranking functions: sum and lexicographic ordering. For a practical subset of acyclic queries known as star queries, we show an even stronger result that allows a user to obtain a smooth tradeoff between faster answering time guarantees using more preprocessing time. Our results are also extensible to queries containing cycles and unions. We also perform a comprehensive experimental evaluation to demonstrate that our algorithms, which are simple to implement, improve up to three orders of magnitude in the running time over state-of-the-art algorithms implemented within open-source RDBMS and specialized graph databases.

PVLDB Reference Format:
Shaleen Deep, Xiao Hu, and Paraschos Koutris. Ranked Enumeration of Join Queries with Projections. PVLDB, 15(5): 1024 - 1037, 2022.
doi:10.14778/3510397.3510401

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/shaleen/rankedenumprojections.

1 INTRODUCTION

Join processing is one of the most fundamental problems in database research with applications in many areas such as anomaly and community detection in social media, fraud detection in finance, and health monitoring. In many data analytics tasks, it is also required to rank the query results in a specific order. This functionality is supported by the ORDER BY clause in SQL, Cypher and SPARQL. We demonstrate a practical example use-case.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 5 ISSN 2150-8097.
doi:10.14778/3510397.3510401

Example 1. *Consider the DBLP dataset as a single relation $R(A, B)$, indicating that A is an author of paper B. Given an author a, the function h-index(a) returns the h-index of a. A popular analytical task asks to find all co-authors who authored at least one paper together. Additionally, the pairs of authors should be returned in decreasing order of the sum of their h-indexes, since users are only interested in the top-100 results. The following SQL query captures this task.*

```
SELECT DISTINCT R1.A, R2.A FROM R AS R1, R AS R2
WHERE R1.B = R2.B
ORDER BY h_index(R1.A) + h_index(R2.A) LIMIT 100;
```

The above task is an example of a join query with projections (join-project queries) because attribute B has been projected out (i.e. it is not present in the selection clause). The DISTINCT clause ensures that there are no duplicate results.

Importance of joins with projections. Join queries containing projections appear in several practical applications such as recommendation systems [30, 47], similarity search [67], and network reachability analysis [13, 26]. In fact, as Manegold et al. [48] remarked, joins in real-life queries almost always come with projections over certain attributes. Matrix multiplication [7], path queries (equivalent to sparse matrix multiplication), and reachability queries [32] are all examples of join-project queries that have widespread applications in linear and relational algebra. Other data models such as SPARQL [54] also support the projection operator and evaluation of join-project queries has been a subject of research, both theoretically [8] and practically [20]. In fact, as SPARQL supports ORDER BY/LIMIT operator, ranked enumeration for queries (that include projections) and top-k over knowledge bases in the SPARQL model has also been explicitly studied recently [18, 43]. As many practical SPARQL evaluation systems [33, 59] evaluate queries using RDBMS, it is important to develop efficient algorithms for such queries in the relational model. Similarly, [64] argued that since a large fraction of the data of interest resides in RDBMS, efficient execution of graph queries (such as path and reachability queries that contain projections and ranking) using RDBMS as the backend is valuable. In the relational setting, join-project queries also appear in the context of probabilistic databases (see Section 2.3 in [21]). This motivates us to develop efficient algorithms, both in theory and practice, that address the challenge of incorporating the ranked enumeration paradigm for join-project queries.

Prior Work. Efficient evaluation of join queries in the presence of ranking functions has been a subject of intense research in the database community. Recent work [16, 24, 62, 63, 65] has made significant progress in identifying optimal algorithms for enumerating query results in ranked order. In each of these works, the key idea is to perform on-the-fly sorting of the output via the use of priority queues by taking into account the query structure. [16] considered

the problem of top-k tree matching in graphs and proposed optimal algorithms by combining Lawler's procedure [42] with the ranking function. [62] introduced multiple dynamic programming algorithms that lazily populate the priority queues. [65] took a different approach where all possible candidates were eagerly inserted into the priority queues and [24] generalized these ideas to present a unified theory of ranked enumeration for full join queries. Very recently, [63] was able to extend some of these results to non equi-joins as well. The performance metric for enumerating query results is the *delay* [9], defined as the time difference between any two consecutive answers. Prior work was able to obtain logarithmic delay guarantees, which were shown to be optimal. However, all prior work in this space suffer from one fundamental limitation: it assumes that the join query is full, i.e. there are no non-trivial projections involved. In fact, [63] explicitly remarks that in presence of projections, the strong guarantees obtained for full queries do not hold anymore. Their suggestion to handle this limitation is to convert the query with projections into a projection-free result, i.e. materialize the join query result, apply the projection filter, and then rank the resulting output. However, this conversion requires an expensive materialization step. An alternate approach is to modify the weights of the tuples/attribute values to allow re-use of existing algorithms (we describe this approach in Section 2). As we show later, this approach also does not fare any better and requires enumerating the full output of the join query, which can be polynomially slower than the optimal solution.

On the practical side, all RDBMS and graph processing engines evaluate join-project queries in the presence of ranking functions by performing three operations in serial order: (i) materializing the result of the full join query, (ii) de-duplicating the query result (since the query has `DISTINCT` clause), and (iii) sorting the de-duplicated result according to the ranking function. The first step in this process is a show-stopper. Indeed, the size of the full join query result can be orders of magnitude larger than the size of the final output after applying projections and de-duplicating it. Thus, the materialization and the de-duplication step introduces significant overhead since they are blocking operators. Further, if the user is interested in only a small fraction of the ordered output, the user still has to wait until the entire query completes even to see the top-ranked result.

1.1 Our Contribution and Key Ideas

In this paper, we initiate the study of ranked enumeration over join-project queries. We focus on two important ranking functions: `SUM` ($f(x,z) = x + z$) and `LEXICOGRAPHIC` ($f(x,z) = x, z$) for two reasons. First, both of these functions are very useful in practice [35]. Second, extending the algorithmic ideas to other functions, such as `MIN`, `MAX`, `AVG` and circuits that use sum and products, is quite straightforward. More specifically, we make three contributions.

1. Enumeration with Formal Delay Guarantees. Our first main result shows that for any *acyclic* query (the most common fragment of queries in practice [14]) with arbitrary projection attributes, it is possible to develop efficient enumeration algorithms (Section 3).

Theorem 1. *For an acyclic join-project query Q, an instance D, and a ranking function* rank $\in$ {`SUM`, `LEXICOGRAPHIC`}*, the query result $Q(D)$ can be enumerated according to* rank *with worst-case delay $O(|D| \log |D|)$, after $O(|D|)$ preprocessing time.*

This result implies that top-k results in $Q(D)$ can be enumerated in $O(k|D| \log |D|)$ time. Theorem 1 is able to recover the prior results for ranked enumeration of full queries as well [24]. The key idea of our algorithm is to develop multiway join plans [52] by exploiting the properties of join trees. Embedding the priority queues in the join tree strategically allows us to generate the sorted output on-the-fly and avoid the binary join plans that all state-of-the-art systems use. Further, since we formulate the problem in terms of delay guarantees, it allows our techniques to be limit-aware: for small k, the answering time is also small.

2. Faster Enumeration with More Preprocessing. Our second contribution is an algorithm that allows for a smooth tradeoff between preprocessing time and delay guarantee for a subset of join-project queries known as *star* queries over binary relations of the form $R_i(A_i, B)$ (denoted as $Q^\star_m$) (Section 4):

`SELECT DISTINCT` $A_1, \ldots, A_m$ `FROM` $R_1, \ldots, R_m$
`WHERE` $R_1.B = \cdots = R_m.B$ `ORDER BY` $A_1 + \cdots + A_m$ `LIMIT` k;

Theorem 2. *For a star join-project query $Q^\star_m$, an instance D, and a ranking function* rank $\in$ {`SUM`, `LEXICOGRAPHIC`}*, the query result $Q(D)$ can be enumerated according to* rank *with worst-case delay $O\left(|D|^{1-\epsilon} \log |D|\right)$, using $O\left(|D|^{1+(m-1)\epsilon}\right)$ preprocessing time and $O\left(|D|^{m(1-\epsilon)}\right)$ space, for any $0 \leq \epsilon \leq 1$.*

Theorem 2 enables users to carefully control the space usage, preprocessing time and delay. For both Theorem 1 and Theorem 2, we can show that the delay guarantee is optimal subject to a conjecture about the running time of star join-project queries in Subsection 4.2.

3. Experimental Evaluation. Our final contribution is an extensive experimental evaluation for practical join-project queries on real-world datasets (Section 6). To the best of our knowledge, this is the first comprehensive evaluation of how existing state-of-the-art relational and graph engines execute join-project queries in the presence of ranking. We choose MariaDB, PostgreSQL, two popular open-source RDBMS, and Neo4j as our baselines. We highlight two key results. First, our experimental evaluation demonstrates the bottleneck of serially performing materialization, de-duplicating, and sorting. Even with `LIMIT 10` (i.e. return the top-10 ranked results), the engines are orders of magnitude slower than our algorithm. For some queries, they cannot finish the execution in a reasonable time since they run out of main memory. On the other hand, our algorithm has orders of magnitude smaller memory footprint that allows for faster execution. The second key result is that all baseline engines are agnostic of the ranking function. The execution time of the queries is identical for both the sum and lexicographic ranking function. However, our algorithm uses the additional structure of lexicographical ordering and can execute queries $2-3\times$ faster than the sum function. For queries with unions and cycles, our algorithm maintains its performance improvement over the baselines.

2 PROBLEM SETTING

In this section we present the basic notions and terminology, and then discuss our framework. We focus on the class of *join-project*

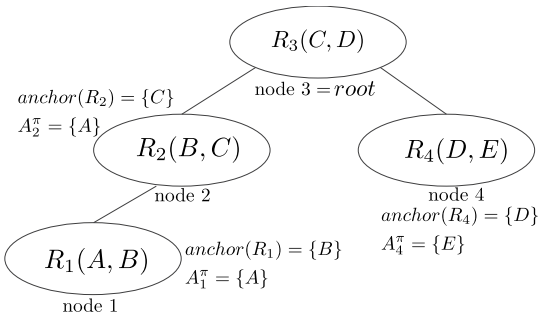


Figure 1: Illustration of join tree for a join-project query $Q = \pi_{A,E}(R_1(A,B) \bowtie R_2(B,C) \bowtie R_3(C,D) \bowtie R_4(D,E))$.

queries, which are defined as

$$Q = \pi_{\mathbf{A}}(R_1(\mathbf{A}_1) \bowtie R_2(\mathbf{A}_2) \bowtie \ldots \bowtie R_m(\mathbf{A}_m))$$

Here, each relation has schema $R_i(\mathbf{A}_i)$, where $\mathbf{A}_i$ is an ordered set of attributes. Let $\mathbb{A} = \mathbf{A}_1 \cup \mathbf{A}_2 \cup \cdots \cup \mathbf{A}_m$. The projection operator $\pi_{\mathbf{A}}$ only keeps a subset of the attributes from $\mathbb{A}$. The join we consider is *natural join*, where tuples from two relations can be joined if they share the same value on the common attributes. A join-project query is *full* if $\mathbf{A} = \mathbb{A}$. Unlike prior work on ranked enumeration, in this paper we place no restriction on the set of attributes in the projection operator. For simplicity of presentation, we do not consider selections; these can be easily incorporated into our algorithms. As an example, the SQL query in Example 1 corresponds to the following query: $\pi_{A,B}(R_1(A,C) \bowtie R_2(B,C))$.

A database D is a set of relations, whose size is defined as the total number of tuples in all relations denoted as $|D|$. For tuple t, we will use the shorthand $t[A]$ to denote $\pi_A(t)$. We use $\mathcal{E}$ to denote the set of all relations in the database.

Acyclic Queries and Join Trees. A join-project query Q is *acyclic* if and only if it admits a *join tree* $\mathcal{T}$. In a join tree, each relation is a node, and for each attribute A, all nodes in the tree containing A form a connected subtree. For simplicity, we will use node i to refer to the node corresponding to relation R_i in $\mathcal{T}$. Given a join tree $\mathcal{T}$, pick any node to be the root, and then orient each edge towards the root. Let $\mathcal{T}_i$ be the subtree rooted at node R_i. Let $\texttt{parent}(R_i)$ be the (unique) parent of R_i, and $\texttt{anchor}(R_i) = R_i \cap \texttt{parent}(R_i)$ to be the *anchor* attributes between R_i and its parent. Let $\texttt{child}(R_i)$ be the set of children nodes of R_i. Finally, we fix the ordering of the projection attributes in $\mathbf{A}$ to be the order of visiting them in the in-order traversal of $\mathcal{T}$. Finally, we define $\mathbf{A}_i^\pi$ as the ordered set of projection attributes in subtree rooted at node i (including projection attributes of node i). As a convention, we define $\texttt{anchor}(r) = \emptyset, A_r^\pi = \emptyset$ for the root r and $\texttt{child}(R_i) = \emptyset$ for a leaf node R_i.

Example 2. *Consider a join-project query $Q = \pi_{A,E}(R_1(A,B) \bowtie R_2(B,C) \bowtie R_3(C,D) \bowtie R_4(D,E))$ under the ranking function* SUM *defined over attributes A, E. In other words, for every output tuple t, the score of the tuple is $t[A] + t[E]$. Figure 1 shows the join tree for the query. We fix R_3 as the root with R_2 as the left child and R_4 (a leaf node) as the right child. R_1, as a leaf node, is also the only child of R_2.*

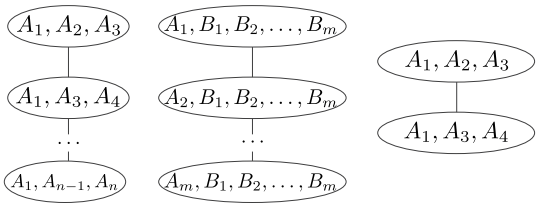


Figure 2: Examples of GHD and fhw. The leftmost is the minimal GHD of a cycle join $Q = R_1(A_1,A_2) \bowtie R_2(A_2,A_3) \bowtie \cdots \bowtie R_{n-1}(A_{n-1},A_n) \bowtie R_n(A_n,A_1)$ with $\texttt{fhw} = 2$. The middle is the minimal GHD of a bi-clique join $Q = \bowtie_{i\in[n],j\in[m]} R_{(i-1)m+j}(A_i,B_j)$ with $\texttt{fhw} = m$. The rightmost is the minimal GHD of a butterfly join $Q = R_1(A_1,A_2) \bowtie R_2(A_2,A_3) \bowtie R_3(A_1,A_4) \bowtie R_4(A_4,A_3)$ with $\texttt{fhw} = 2$.

Generalized Hypertree Decompositions. A weight assignment $\mathbf{u} = (u_F)_{F\in\mathcal{E}}$ is said to be a fractional edge cover if (i) for every $F \in \mathcal{E}$, $u_F \geq 0$; and (ii) for every $X \in \mathbb{A}$, $\sum_{F:X\in F} u_F \geq 1$. The *fractional edge cover number* for $\mathbb{A}$, denoted $\rho^*(\mathbb{A})$ is the minimum of $\sum_{F\in\mathcal{E}} u_F$ over all possible edge covers. A generalized hypertree decomposition (GHD) of a query Q is a tuple $(\mathcal{T}, (\mathcal{B}_t)_{t\in V(\mathcal{T})})$ where $\mathcal{T}$ is a tree and every $\mathcal{B}_t$ (called the bag of t) is a subset of $\mathbb{A}$ for each node t of the tree such that: (i) each variables of each $F \in \mathcal{E}$ is contained in some bag; and (ii) for each $A \in \mathbb{A}$, the set of nodes $\{t \mid A \in \mathcal{B}_t\}$ is connected in $\mathcal{T}$. 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 fractional edge cover number of the attributes in $\mathcal{B}_t$. The fractional hypertree width of a query Q, denoted $\texttt{fhw}(Q)$, is the minimum fractional hypertree width over all possible GHDs of Q. Figure 2 gives examples of GHDs of popular queries and their width. For an acyclic query, it holds that $\texttt{fhw} = 1$ and any join tree is a valid GHD.

Computational Model. To measure the running time of our algorithms, we use the uniform-cost RAM model [34], 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, where the query is assumed fixed. It is important to note that we focus on the main memory setting. We further assume existence of perfect hashing that allows constant time lookups in hash tables.

2.1 Ranking Functions

The ordering of query results in $Q(D)$ can be specified by a *ranking function*, or through the ORDER BY clause of a SQL query in practice. Formally, a total order $\succeq$ on the tuples in $Q(D)$ defined over the attributes $\mathbf{A}$, is induced by a ranking function $\texttt{rank}$ that maps each tuple $t \in Q(D)$ to a real number $\texttt{rank}(t) \in \mathbb{R}$. In particular, for two tuples t_1, t_2, we have $t_1 \succeq t_2$ if and only if $\texttt{rank}(t_1) \geq \texttt{rank}(t_2)$. We assume that $\mathbf{dom}(A)$ for any $A \in \mathbf{A}$ is also equipped with a total order $\geq$. We present an example of a ranking function below.

Example 3. *Consider a function $w : \mathbf{dom}(A) \to \mathbb{R}$ for any attribute $A \in \mathbf{A}$. For each query result t, we define its rank as $\texttt{rank}(t) = \sum_{A\in\mathbf{A}} w(t[A])$, the total sum of the weights over all attributes in $\mathbf{A}$.*

We will focus on SUM and LEXICOGRAPHIC in this paper. We note that both functions are instantiations of a more general class of *decomposable functions* [24]. The ideas introduced for SUM and LEXICOGRAPHIC are readily applicable to more complicated functions including products, a combination of sum and products, etc.

2.2 Problem Parameters

Given a join-project query Q and a database D, an enumeration query asks to enumerate the tuples of $Q(D)$ according to some specific ordering defined by rank. We study this problem in a similar framework as [58], where an algorithm is decomposed into:

- a **preprocessing phase** that takes time T_p and computes a data structure of size S_p
- an **enumeration phase** (i.e. the online query phase) that outputs $Q(D)$ without duplicates under the specified ordering whenever a user query is issued. This phase has full access to any data structures constructed in the preprocessing phase. The time between outputting any two consecutive tuples (and also the time to output the first tuple, and the time to notify that the enumeration has completed after the last tuple) is at most δ.

Prior work [24] has shown that for acyclic joins without projections, there exists an algorithm with $T_p = S_p = O(|D|)$ that can achieve $\delta = O(\log |D|)$ delay under ranking. However, the problem of ranked enumeration when projections are involved is wide open.

Using Existing Algorithms. One possible solution to the problem is to set the weights of non-projection attributes to 0. This will ensure that for SUM function, only the projection attributes are considered in the ranking and existing algorithms for full join queries could be used. However, this proposal gives poor delay guarantees and is as expensive as enumerating the full join result. For example, for the four path query in Example 2, the output of the query could be constant in size but the full join can be as large as $\Omega(|D|^2)$ which is prohibitively expensive, but our algorithm would only require $O(|D|)$ in this case. In general, a join with ℓ relations may require as much as $\Omega(|D|^{\ell-1})$ time to output the smallest tuple. We describe more details and the formal proof in [23].

3 GENERAL ACYCLIC QUERIES

We first describe the main algorithm of enumerating acyclic join-project queries for SUM ordering in Subsection 3.1, followed by a specialized algorithm for LEXICOGRAPHIC ordering in Subsection 3.2. Before we describe the algorithm, we introduce two key data structures that will be used: *cell* and *priority queues*.

Definition 1. *A cell, denoted as $c = \langle t, [p_1, \ldots, p_k], q\rangle$, is a vector consisting of three values: (i) a tuple $t \in R_i$ for node i in the join tree $\mathcal{T}$, (ii) an array of pointers $[p_1, \ldots, p_k]$ where the ℓ^{th} pointer points to a cell defined for ℓ^{th} child of node i in $\mathcal{T}$, (iii) a pointer q that can only point to another cell defined for node i.*

Given a cell c defined for node i, one can reconstruct the tuple over $\mathbf{A}_i^\pi$ in constant time (dependent only on the query size, which is a constant) by traversing the pointers recursively. We will use output(c) to denote the utility method that performs this task. Note that the time and space complexity of creating a cell is $O(1)$ since the size of the query and the database schema is assumed to be a constant. This implies that we only need to insert/access a constant number of entries in the vector representing a cell. Similarly, output(c) also takes $O(1)$ time since the join tree size is a constant.

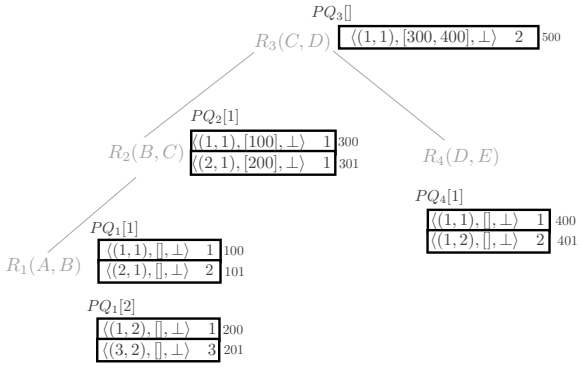
Priority queue. A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. The space complexity of a priority queue containing $|S|$ elements is $O(|S|)$. We will use an implementation of a priority queue (e.g., a Fibonacci heap [31]) with the following properties: (i) an element can be inserted in $O(1)$ time, (ii) the min element can be obtained in $O(1)$ time, and (iii) the min element can be popped and deleted in $O(\log |D|)$ time. We will use the priority queue in conjunction with a cell in the following way: for two cells c_1 and c_2, the priority queue uses $\texttt{rank}(\texttt{output}(c_1))$ and $\texttt{rank}(\texttt{output}(c_2))$ in the comparator function to determine the relative ordering of c_1 and c_2. If $\texttt{rank}(\texttt{output}(c_1)) = \texttt{rank}(\texttt{output}(c_2))$, then we break ties according to the lexicographic order of $\texttt{output}(c_1)$ and $\texttt{output}(c_2)$. The choice of lexicographic ordering is not driven by any specific consideration; as long as the ties are broken consistently, we can use other tie-breaking criteria too. Once again, the comparator function only takes a $O(1)$ time to compare since the ranking function $\texttt{rank}(\texttt{output}(c))$ can be evaluated in constant time.

3.1 General Algorithm

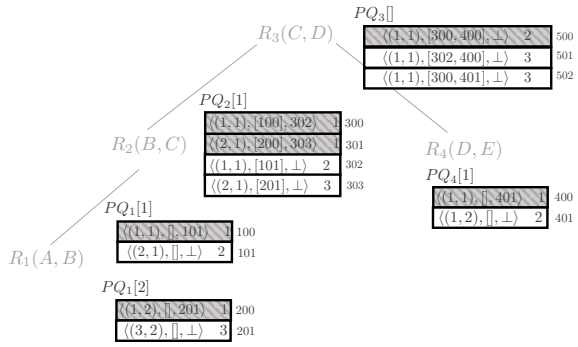
In this section, we present the algorithm for Theorem 1. At a high level, each node i in the join tree will materialize, in an incremental fashion, all tuples over the attributes $\mathbf{A}_i^\pi \cup \texttt{anchor}(R_i)$ in sorted order. In order to efficiently store the materialized output, we will use the cell data structure. Since we need to sort the materialized output, each node in the join tree maintains a set of priority queues indexed by $\pi_{\texttt{anchor}(R_i)}(u), u \in R_i$. The values of the priority queue are the cells of node i. For example, given the join tree from Example 2, node 2 containing R_2 will incrementally materialize the sorted result of the subquery $\pi_{C,A}(R_2(B,C) \bowtie R_1(A,B))$ that is indexed by the values $\pi_C(R_2(B,C))$ since $\mathbf{A}_2^\pi = \{A\}$ and $\texttt{anchor}(R_2) = \{C\}$. Note that there may be multiple possible join trees for a given acyclic query. Our algorithm is applicable to all join trees. In fact, any node in the join tree can be chosen as the root without any impact on the time and space complexity.

Preprocessing Phase. We begin by describing the algorithm for preprocessing in Algorithm 1. We assume that a join tree has been fixed and the input instance D does not contain any dangling tuples, i.e., tuples that will not contribute in the join; otherwise, we can invoke the Yannakakis algorithm [66] to remove all dangling tuples. We initialize a set of empty priority queues for every node in the join tree. We proceed in a bottom up fashion and perform the following steps. For each leaf relation $R_i \in \mathcal{T}$, we create a cell $\langle t, [], \perp\rangle$ for each tuple $t \in R_i$ and insert it into $\texttt{PQ}_i[\pi_{\texttt{anchor}(R_i)}(t)]$. For each non-leaf relation $R_j \in \mathcal{T}$, we create a cell for $t \in R_j$, which points to the top of the priority queue in each child node of R_j that can be joined with t. This cell is then added to the priority queue $\texttt{PQ}_i[\pi_{\texttt{anchor}(R_i)}(t)]$. Note that we only have one priority queue for the root relation r since $\texttt{anchor}(r) = \emptyset$ by definition.

Example 4. *Continuing with the 4-path query running example, consider the following instance D as shown below.*



(a) Data structure state after the preprocessing phase. Each memory location has a cell and the partial score of the partial answer



(b) Data structure after one iteration of procedure Enum()

Figure 3: Example to demonstrate the preprocessing and enumeration phase of the general algorithm

Algorithm 1: PreprocessAcyclic

Input : Input query Q, database instance D; join tree $\mathcal{T}$; ranking function rank.
Output : Priority Queues PQ

foreach $R_i \in \mathcal{T}$ *in post order traversal* **do**
  foreach $t \in R_i$ **do**
    $u \leftarrow \pi_{\text{anchor}(R_i)}(t)$;
    if $\text{PQ}_i[u]$ *does not exist* **then**
      $\text{PQ}_i[u] \leftarrow \emptyset$; /* Initialize a priority queue */
    $L \leftarrow \emptyset$;
    foreach R_j *is the child of* R_i **do**
      L.insert($\text{PQ}_j[\pi_{\text{anchor}(R_j)}(u)]$.top());
  $\text{PQ}_i[u]$.insert($\langle t, L, \perp\rangle$);

R_1

A	B
1	1
2	1
1	2
3	2

R_2

B	C
1	1
2	1

R_3

C	D
1	1
1	2

R_4

D	E
1	1
1	2

As we saw before, Figure 1 shows the join tree along with the anchor attributes in each relation. Figure 3a shows the state of priority queues after the preprocessing step. After the full reducer pass, tuple $(1,2)$ *is removed from* R_3 *because there is no join tuple that can be formed using it. Then, we start constructing the cells for each node starting with the leaf nodes. Since B is the anchor for relation* R_1*, we create two priority queues* $PQ_1[1]$ *and* $PQ_1[2]$*. For* $PQ_1[1]$*, we create the cells for tuples* $(1,1)$ *and* $(2,1)$*. For convenience, the cells are followed by the partially aggregated score. Consider relation* $R_2(B,C)$*. The cell for tuple* $(1,1)$ *in* $PQ_2[1]$ *points to the top of* $PQ_1[1]$ *(shown as pointer with address* 100*). The root bag consists of a single tuple entry which points to the cells at locations* 300 *and* 400*. The output tuple that can be formed by the root bag is* $(A=1, E=1)$*.*

Enumeration Phase. We describe the enumeration procedure in Algorithm 2. The high-level idea is to output answers by repeatedly popping elements from the root priority queue. It may be

Algorithm 2: EnumAcyclic

Input : Input query Q, database instance D; join tree $\mathcal{T}$; ranking function rank; Priority queues PQ
Output : $Q(D)$ in ranked order

procedure Enum()
  last $\leftarrow \emptyset$;
  while $\text{PQ}_r[\emptyset] \neq \emptyset$ **do**
    $o \leftarrow \text{PQ}_r[\emptyset]$.top();
    if is_equal(o, last) = *false* **then**
      print output(o), last $\leftarrow o$; /* new output */
    Topdown (o, r);
procedure Topdown(c, j) /* $c = \langle t, [p_1, \ldots, p_k], \text{next}\rangle$ */
  $u \leftarrow \pi_{\text{anchor}(R_j)}(c.t)$;
  if c.next $= \perp$ **then**
    while *true* **do**
      temp $\leftarrow$ **pop**($\text{PQ}_j[u]$);
      foreach R_i *is a child of* R_j **do**
        $p_i' \leftarrow$ Topdown($c.p_i, i$) ;
        if $p_i' \neq \perp$ **then**
          $\text{PQ}_j[u]$.insert($\langle t, [c.p_1, \ldots, p_i', \ldots c.p_k], \perp\rangle$)
      if R_j *is not the root* **then**
        c.next $\leftarrow$ addressof($\text{PQ}_j[u]$.top());
      if is_equal(temp, $\text{PQ}_j[u]$.top()) = *false* **then** break;
  return c.next;
procedure is_equal(c_1, c_2)
  if rank(output(c_1))$\neq$ rank(output(c_2)) **then** return false
  foreach $A \in$ A **do**
    if output(c_1)[A] $<$ output(c_2)[A] **then** return false
  return true;

possible that multiple tuples of the root priority queue output the same final result. In order to deduplicate answers, we compare the answer at the current top of the priority queue with the previous answer (line 5), and output it only if they are different. Then, we invoke the procedure Topdown to insert new candidates into the priority queue. This procedure will be recursively propagated over the join tree until it reaches the leaf nodes. Observe that once the new candidates have been inserted, the next pointer of a cell is updated by pointing to the topmost element in the priority queue. This chaining materializes the answers for a particular node that can be reused and is key to avoiding repeated computation.

Example 5. *Continuing our running example, Figure 3b shows the state of the priority queues after one complete iteration of procedure Enum(). We first pop the only element in root priority queue and note that the output tuple* $(A = 1, E = 1)$ *is enumerated. Then we call Topdown with cell at memory* 500 *and root (node* 3*) as arguments (denoted as* Topdown$(*500, 3)$*). The next for the cell is* $\perp$ *so we pop the cell at* 500 *from the priority queue (shown as greyed out in the figure) and recursively call* Topdown$(*300, 2)$*. The cell at memory location* 300 *has next* $= \perp$*. Therefore, we enter the while loop, pop the cell and recursively call* Topdown$(*100, 1)$*. We have now reached the leaf node. The anchor attribute value for cell at* 100 *is* $u = 1$*, so we pop the current cell from* $PQ_1[1]$ *(greyed out cell at* 100*), find the next candidate at the top of* $PQ_1[1]$ *(which is cell at* 101*), chain it to the cell at* 100 *by assigning next* $=$ 101 *and return the cell at* 101 *to the parent. When the program control returns from the recursive call back to node* 2*, we create a new cell (at memory address* 302*) that points to* 101 *and insert it into the priority queue. However, observe that the cell at memory location* 301 *also generates* $A = 1$*, a duplicate since cell at* 300 *also generated it. This is where the equality check at line 19 comes in. Since both cells at* 300 *and* 301 *generate the same value, we also pop off the cell at* 301 *in the subsequent while loop iteration, find its next candidate and create the cell at* 303*, and insert into the priority queue. This ensures that all elements in* $PQ_2[1]$ *generating the same* A *value are removed, ensuring no duplicates at the root level. Finally, the control returns to the root level Topdown call. The recursive call to the right child (node* 4*) create a new cell* 401 *and we insert two cells at the root priority queue, cell* 501 *and* 502 *that correspond to output tuple* $(A = 2, E = 1)$ *and* $(A = 1, E = 2)$ *respectively.*

We are now ready to formally prove Theorem 1.

Lemma 1. *The delay guarantee of EnumAcyclic is at most* $O(|D| \log |D|)$.

Lemma 2. PreprocessAcyclic *running in* $O(|D| \log |D|)$ *time, generates a data structure of size* $O(|D|)$.

Lemma 3. *EnumAcyclic enumerates the query result* $Q(D)$ *in ranked order correctly.*

Together, the above lemmas establish Theorem 1. We defer the full proofs to [23]. We also show how we can recover logarithmic delay guarantee for full queries from [24, 62].

3.2 Improvement for Lexicographic Ranking

The algorithm from last section is also applicable to LEXICOGRAPHIC ranking function. In fact, we can transform LEXICOGRAPHIC with an attribute ordering of $A_1, A_2, \cdots, A_m$, into SUM by defining a ranking function $\mathsf{rank}(t) = \sum_{i=1}^{m} 10^{m-i} \cdot w(\pi_{A_i}(t))$ for tuple t, while preserving the LEXICOGRAPHIC ordering. In this section, we present an alternative algorithm by exploiting the special structural properties of LEXICOGRAPHIC, that the global ranking also implies local ranking over every output attribute. Moreover, it admits to enumerate query results not only in lexicographic order as given by ORDER BY $A_1, A_2, \cdots, A_m$ but also arbitrary ordering on each attribute (for instance, ORDER BY A_1 ASC, A_2 DESC ...).

Preprocessing Phase. In this phase, we perform the full reducer pass to remove all dangling tuples and create hash indexes for the base relations in sorted order. We also sort $\mathbf{dom}(A_i)$.

Enumeration Phase. Given an attribute order of output attributes $\mathbf{A} = \{A_1, A_2, \cdots, A_m\}$, we start by fixing the minimum value in $\mathbf{dom}(A_1)$ as a_1. Then, we perform the two-phase semi-joins to remove tuples that cannot be joined with value a_1, and find the values in $\mathbf{dom}(A_2)$ that survive after semi-joins, denoted as $\mathcal{L}_{A_2}(a_1)$. Similarly, we fix the minimum value in $\mathcal{L}_{A_2}(a_1)$ as a_2, and perform the two-phase semi-joins for finding the values in $\mathbf{dom}(A_3)$ that can be joined with both a_1, a_2. We continue this process until all attributes in $\mathbf{A}$ have been fixed, and end up with enumerating such a query result (with fixed values). Then, we backtrack and continue the process until all values in attribute A_1 are exhausted.

Algorithm 3: EnumAcyclicLexi$(t, \mathcal{L}, i)$

Input : Input query Q, database D
Output: $Q(D) \ltimes t$ in lexicographic order of $A_i, \cdots, A_m$

1 **if** $i = m$ **then output** t and **return**;
2 **foreach** $a \in \mathcal{L}$ **do**
3 $\quad \mathcal{L}' \leftarrow \pi_{A_{i+1}}(\sigma_{A_i=a}(R_{i+1} \ltimes t))$; /* by semi-joins */
4 $\quad t' \leftarrow (t, a)$; /* create new tuple */
5 $\quad$ EnumAcyclicLexi$(t', \mathcal{L}', i+1)$;

Algorithm 3 takes as input an acyclic query Q, an database D, an integer $i \in \{1, \cdots, m\}$, a tuple t defined over attributes $A_1, \cdots, A_{i-1}$, and a set of values $\mathcal{L} \subseteq \mathbf{dom}(A_i)$ that can be joined with t in D. The original problem can be solved by invokingbreak EnumAcyclicLexi$(\emptyset, \mathbf{dom}(A_1), 1)$ for sorted $\mathbf{dom}(A_1)$.

Lemma 4. *EnumAcyclicLexi enumerates* $Q(D)$ *correctly in lexicographic order with delay guarantee* $O(|D|)$ *after preprocessing time* $T_p = O(|D| \log |D|)$ *and space complexity* $O(|D|)$.

4 STAR QUERIES

In this section, we present a specialized data structure for the *star query*, which is represented as: $Q_m^\star = \pi_{\mathbf{A}}(R_1(A_1, B) \bowtie R(A_2, B) \bowtie \cdots \bowtie R_m(A_m, B))$. where $\mathbf{A} = \{A_1, \cdots, A_m\}$. All relations in a star query join on exactly the same attribute(s). In this following, we present a specialized data structure on ranked enumeration for $Q_m^\star$ in Section 4.1, and prove the optimality in Section 4.2.

4.1 The Algorithm

Consider the star query $Q_m^\star$, a database D and a ranking function rank. Now we present a data structure for Theorem 2.

Algorithm 4: PreprocessStar

Input : Input star query $Q_m^\star$, ranking function rank and database D; degree threshold $\delta \geq 1$
Output : Heavy output O^H and priority queue PQ

foreach $i \in \{1, 2, \cdots, m\}$ **do**
  $R_i^H \leftarrow \{t \in R_i : |\sigma_{A_i=\pi_{A_i}(t)}| \geq \delta\}$;
  $R_i^L \leftarrow \{t \in R_i : |\sigma_{A_i=\pi_{A_i}(t)}| < \delta\}$;
Compute $O^H \leftarrow \pi_{\mathbf{A}}\left(R_1^H \bowtie \cdots \bowtie R_m^H\right)$;
Sort O^H by rank;
for $i \in \{0, 1, \ldots, m-1\}$ **do**
  $Q_i \leftarrow R_1^H \bowtie \cdots \bowtie R_{m-1}^H \bowtie R_i^L \bowtie R_{i+1} \bowtie \cdots \bowtie R_m$;
  $\mathcal{T}_i \leftarrow$ a join tree for Q with R_i as root and all other relations as children of R_i;
  PreprocessAcyclic$(Q_i, \mathcal{T}_i)$;
  next $\leftarrow$EnumAcyclic$(Q_i, \mathcal{T}_i)$;
  PQ.**insert**(next); /* insert the smallest tuple into PQ */

Preprocessing Phase. Without loss of generality, assume that there is no dangling tuples in D. Moreover, if $\mathbf{A}$ does not include an attribute A, we can remove efficiently R_i using a semi-join. We first fix a degree threshold $\delta \geq 1$ (whose value will be determined later). For each $i \in \{1, 2, \cdots, m\}$, a value $a_i \in \mathbf{dom}(A)$ is *heavy* if it has degree larger than δ in R_i, i.e., $|\sigma_{A=a_i}(R_i)| \geq \delta$, and *light* otherwise. A tuple $t = (a_i, b) \in R_i$ is *heavy* if a_i is heavy. For R_i, let R_i^H, R_i^L be the set of heavy and light tuples in R_i. An output $t = (a_1, a_2, \ldots, a_m) \in Q_m^\star(D)$ is *heavy* if a_i is heavy in R_i for each $i \in \{1, 2, \cdots, m\}$, and *light* otherwise. In this way, we can divide the output $Q_m^\star(D)$ into O^H and O^L, containing all heavy and light output tuples separately. In the preprocessing phase, our goal is to materialize all heavy output tuples (O^H) ordered by rank. Details are described in Algorithm 4. We compute $O^H = \pi_{\mathbf{A}}\left(R_1^H \bowtie R_2^H \bowtie \cdots \bowtie R_m^H\right)$ by invoking the Yannakakis algorithm [66], and then sort O^H by rank. Next, we insert the smallest query result from O^H into the priority queue. Then, we define m different subqueries as $Q_i = \pi_{\mathbf{A}}\left(R_1^H \bowtie \cdots \bowtie R_{i-1}^H \bowtie R_i^L \bowtie R_{i+1} \bowtie \cdots \bowtie R_m\right)$ where tuples in relation R_j are heavy for any $j < i$ and tuples in relation R_i are light. For such Q_i, we consider a join tree $\mathcal{T}_i$ in which R_i is the root and all other relations are children of R_i. We preprocess a data structure for Q_i with $\mathcal{T}_i$, by invoking Algorithm 1.

Enumeration Phase. As described in Algorithm 5, the high-level idea in the enumeration is to perform a $(m+1)$-way merge over O^H and Q_i's. Specifically, we maintain a priority queue PQ with one entry for each subquery Q_i and one entry for O^H. Once the smallest element is extracted from PQ (say t generated by Q_i), we extract the next smallest candidate from Q_i (if there is any) and insert it into PQ. Moreover, finding the smallest candidate output result from O^H is trivial since O^H have been materialized in a sorted way in the preprocessing phase. We conclude this subsection with the formal statement of the result.

Algorithm 5: EnumStar

Input : Star query $Q_m^\star$, ranking function rank and database D; Output of O^H and priority queue PQ
Output : $Q_m^\star(D)$ in ranked order

while PQ $\neq \emptyset$ **do**
  $t \leftarrow$ PQ.**pop()**;
  output t; /* enumerate the result */
  if $t \notin O^H$ **then**
    $i \leftarrow$ smallest positive index such that $\pi_{A_j}(t)$ is heavy for all $j < i$ and $\pi_{A_i}(t)$ is light;
    next $\leftarrow$ EnumAcyclic$(Q_i, \mathcal{T}_i)$;
    PQ.**insert**(next);
  else PQ.**insert**(O^H.**pop()**);

Lemma 5. *Algorithm 4 runs in time $T = O(|D| \cdot (|D|/\delta)^{m-1})$ and requires space $S = O((|D|/\delta)^m)$. Algorithm 5 correctly enumerates the result of the query in ranked order with delay $O(|D| \log |D|/\delta)$.*

4.2 Tradeoff Optimality

We next present conditional optimality for our tradeoff achieved in Theorem 2. Before showing the proof, we first revisit a result on unranked evaluation for $Q_m^\star$ in [7]:

Lemma 6 ([7]). *There exists a combinatorial[1] algorithm that can evaluate $Q_m^\star$ on any database D in time $O\left(|D| \cdot |Q_m^\star(D)|^{1-\frac{1}{m}}\right)$.*

This result was presented over a decade ago without any improvement since then. Thus, it is not unreasonable to conjecture that Lemma 6 is optimal. Based on its conjectured optimality, we can show the following result for unranked enumeration.

Lemma 7. *Consider star query $Q_m^\star$, database D and some constant $\epsilon \in [0, 1]$. If there exists an algorithm that supports $O(|D|^{1-\epsilon} \log |D|)$-delay enumeration after $O(|D|^{1+(m-1)\epsilon-\epsilon'})$ preprocessing time for some constant $\epsilon' > 0$, the optimality of Lemma 6 will be broken.*

The lower bound holds for any ranking function. Lemma 7 implies that for star queries, both Theorem 1 and Theorem 2 are optimal. Before concluding this section, we also remark on the question of whether the logarithmic factor that we obtain in the delay guarantee is removable. Prior work [24] showed that for the following simple join query $Q = R(x) \bowtie S(y)$ over SUM, there exists no algorithm supporting constant-delay enumeration after linear preprocessing time. Note that this does not rule out a sub-logarithmic delay guarantee, which remains an open problem.

5 GENERAL QUERIES

In this section, we will describe how to extend the algorithm for acyclic queries to handle cyclic queries. The key idea is to transform the cyclic query into an acyclic one, by constructing a GHD as defined in Section 2. A GHD automatically implies an algorithm for cyclic joins. After materializing the results of the subquery induced by each node in the decomposition, the residual query

[1] An algorithm is called combinatorial if it does not use algebraic techniques such as fast matrix multiplication.

becomes acyclic. Hence, we can apply our algorithm for acyclic queries directly obtaining the following:

Theorem 3. *For a join-project query Q, a database instance D and a ranking function* rank $\in$ {SUM, LEXICOGRAPHIC}*, the query results $Q(D)$ can be enumerated according to* rank *with $O(|D|^{\mathsf{fhw}} \log |D|)$ delay, after $O(|D|^{\mathsf{fhw}} \log |D|)$ preprocessing time.*

We now go one step further and extend our algorithm to queries that are *unions* of join-project queries (UCQs) using an idea introduced by [24, 62]. A UCQ query is of the form $Q = Q_1 \cup Q_2 \cup \cdots \cup Q_m$, where each Q_i is a join-project query defined over the same projection attributes $\mathbf{A}$. Semantically, $Q(D) = \bigcup_i Q_i(D)$. Recent work by Abo Khamis et al. [4] presents an improved algorithm (called PANDA) that constructs multiple GHDs by partitioning the input database into disjoint pieces and build a GHD for each piece. In this way, the size of materialized subquery can be bounded by $O(|D|^{\mathsf{subw}})$, where subw is the *submodular width* [49] of input query Q. Moreover, $\mathsf{subw} \leq \mathsf{fhw}$ holds generally for query Q, thus improving the previous result on fhw. By using Theorem 1 in conjunction with data-dependent decompositions from PANDA we can immediately obtain the following result:

Theorem 4. *For a join-project query Q, a database instance D and a ranking function* rank $\in$ {SUM, LEXICOGRAPHIC}*, the query results $Q(D)$ can be enumerated according to* rank *with $O(|D|^{\mathsf{subw}} \log |D|)$ delay, after $O(|D|^{\mathsf{subw}} \log |D|)$ preprocessing time.*

Example 6. *Consider the 4-cycle (butterfly) query $\pi_{A,C}(R_1(A,B) \bowtie R_2(B,C) \bowtie R_3(C,D) \bowtie R_4(D,A))$ with ranking function $\mathsf{rank}(t) = \pi_A(t) + \pi_C(t)$. With $\mathsf{fhw} = 2$, Theorem 4 implies that the query results can be enumerated according to* rank *with $O(|D|^2 \log |D|)$ delay, after $O(|D|^2)$ preprocessing time. With $\mathsf{subw} = \frac{3}{2}$, Theorem 4 implies that query results can be enumerated according to* rank *with delay $O(|D|^{3/2} \log |D|)$, after $O(|D|^{3/2} \log |D|)$ preprocessing time.*

A note on optimality. The reader may wonder whether the exponent of fhw and subw in Theorem 3 and Theorem 4 are truly necessary. For the triangle query $Q_\triangle(x,y) = R(x,y) \bowtie S(y,z) \bowtie T(z,x)$ which is the simplest cyclic query, $\mathsf{fhw} = \mathsf{subw} = 3/2$ and even after 30 years, the original AYZ algorithm [6] that detects the existence of a triangle in $O(|D|^{3/2})$ time still remains the best known combinatorial algorithm. It is widely conjectured [1, 2, 40, 50] that there exists no better algorithm. As noted in [4], the notion of submodular width was suggested as the yardstick for optimality. Indeed, the groundbreaking results by Marx [49] rules out algorithms with better dependence than subw in the exponent for a small of class of queries but a general unconditional lower bound still remains out of reach. Thus, any improvement in the exponent would automatically imply a better algorithm for cycle detection since ranked enumeration is at least as hard. In [23], we formally show that the exponential dependence of subw in Theorem 4 is unavoidable subject to popular conjectures.

6 EXPERIMENTAL EVALUATION

In this section, we perform an extensive evaluation of our proposed algorithm. Our goal is to evaluate three aspects: (a) how fast our algorithm is compared to state-of-the-art implementations for both SUM and LEXICOGRAPHIC ranking functions on various queries and datasets, (b) test the empirical performance of the space-time tradeoff in Theorem 2, (c) investigate the performance of our algorithm on various cyclic queries based on different shapes and (d) test the scalability behavior of our algorithm.

6.1 Experimental Setup

We use Neo4j 4.2.3 community edition, MariaDB 10.1.47[2] and PostgreSQL 11.12 for our experiments. All experiments are performed on a Cloudlab machine [25] running Ubuntu 18.04 equipped with two Intel E5-2630 v3 8-core CPUs@2.40 GHz and 128 GB RAM. We focus only on the main memory setting and all experiments run on a single core. Since the join queries are memory intensive, we take special care to ensure that only one DBMS engine is running at a time, restart the session for each query to ensure temp tables in main memory are flushed out to avoid any interference, and also monitor that no temp tables are created on the disk. We only keep one database containing a single relation when performing experiments. We switch off all logging to avoid any performance impact. For PostgreSQL and MariaDB, we allow the engines to use the full main memory to ensure all temp tables are resident in the RAM and sorting (if any) happens without any disk IOs by increasing the sort buffer limit. For Neo4j, we allow the JVM heap to use the full main memory at the time of start up. We also build bidirectional B-tree indexes for each relation ahead of time and create named indexes in Neo4j. All of our algorithms are implemented in C++ and compiled using the GNU C++ 7.5.0 compiler that ships with Ubuntu 18.04. Each experiment is run 5 times and we report the median after removing the slowest and the fastest run.

6.1.1 Small-Scale Datasets. We use two real world small scale datasets for our experiments: the DBLP dataset, containing relationship between authors and papers, and the IMDB dataset, containing relationship between actors, directors, and movies. We use these datasets for two reasons: (i) both datasets have been found to be useful and studied extensively in practical problems such as similarity search [67], citation graph analysis [56], and network analysis [13, 26]. (ii) small-scale datasets allow experiments to finish for all systems allowing us to make a fair comparison and develop a fine-grained understanding. In line with prior work [38], for each tuple we assign the weight attribute (and add it to the table schema) in two ways: first, we assign a randomly chosen value, and second, logarithmic weights in which the weight of the entity (author and paper in DBLP) v is $\log_2(1 + deg_v)$, where deg_v denotes its degree in the relation. The schema for both datasets is as shown below (underlined attributes are primary keys for the relation):

(1) DBLP: AuthorPapers($\underline{aid}, pid$), Author($\underline{aid}, name, weight$), Paper($\underline{aid}, title, venue, year, weight, is\_research$).
(2) IMDB: PersonMovie($\underline{pid}, \underline{mid}$), Company($\underline{cid}, name, nation$), Person($\underline{pid}, name, role, weight$), Movie($\underline{mid}, name, year, genre, cid, weight$)

Queries. We consider 4 acyclic join queries as shown in Figure 4 for the small-scale datasets, which are commonly seen in practice [14, 60]. Intuitively, the first three queries find all the top-k weighted 2-hops, 3-hops and 4-hops reachable attribute pairs within the

[2]Compared with MySQL, MariaDB performed better in our experiments, hence we report the results for MariaDB

```
DBLP_2hop = SELECT DISTINCT A_1.name, A_2.name FROM Author AS A_1, Author AS A_2, AuthorPapers AS AP_1, AuthorPapers as AP_2,
    Paper AS P WHERE AP_1.pid = AP_2.pid AND AP_1.aid = A_1.aid AND AP_2.aid = A_2.aid AND P.is_research =true ORDER BY
    A_1.weight + A_2.weight LIMIT k;

DBLP_3hop = SELECT DISTINCT A.name, P.name FROM Author AS A, Paper AS P, AuthorPapers AS AP_1, AuthorPapers as AP_2,
    AuthorPapers as AP_3 WHERE AP_1.pid = AP_2.pid AND AP_2.aid = AP_3.aid AND AP_1.aid = A.aid AND AP_3.pid = P.pid AND P.is_research =
    true ORDER BY A.weight + P.weight LIMIT k;

DBLP_4hop = SELECT DISTINCT A_1.name, A_2.name FROM Author AS A_1, Author AS A_2, AuthorPapers AS AP_1, AuthorPapers as AP_2,
    AuthorPapers as AP_3, AuthorPapers as AP_4, Paper AS P_1, Paper AS P_2 WHERE AP_1.pid = AP_2.pid AND AP_2.aid = AP_3.aid AND
    AP_3.pid = AP_4.pid AND AP_3.pid = P_2.pid AND AP_1.pid = P_1.pid AND AP_1.aid = A_1.aid AND AP_4.aid = A_2.aid AND P_1.is_research =true
    AND P_2.is_research =true ORDER BY A_1.weight + A2.weight LIMIT k;

DBLP_3star = SELECT DISTINCT A_1.name, A_2.name, A_3.name FROM Author AS A_1, Author AS A_2, Author AS A_3, AuthorPapers AS AP_1,
    AuthorPapers as AP_2, AuthorPapers as AP_3, Paper AS P WHERE AP_1.pid = AP_2.pid = AP_3.pid AND AP_1.aid = A_1.aid AND
    AP_2.aid = A_2.aid AND AP_3.aid = A_3.aid AND AP_3.pid = P.pid AND P.is_research =true ORDER BY A_1.weight + A_2.weight + A_3.weight
    LIMIT k;
```

Figure 4: Network analysis queries for DBLP. Queries for IMDB are defined similarly (see [23]).

DBLP network. As remarked by in [17, 41, 60], these queries are of immense practical interest (e.g., see Table 4 in [60]). Queries for IMDB dataset are defined similarly in [23]. In Subsection 6.2.2, we also investigate the performance for cyclic queries.

6.1.2 Large-Scale Datasets. We also perform experiments on two real-world large scale relational datasets and one relational benchmark. The first dataset is from the Friendster [44] online social network that contains $1.8B$ tuples. In the social network each, user is associated with multiple groups. The second dataset is the Memetracker [44] dataset which describes user generated memes and which users have interacted with the meme. The dataset contains $418M$ tuples. For both Friendster and Memetracker, we use weights for users as the number of groups they belong to and the number of memes they create respectively. Finally, we also use the queries containing a ranking function from the LDBC Social Network Benchmark [27] with scale factor SF = 10, a publicly available benchmark, to perform scalability experiments.

Queries. For Friendster and Memetracker, we use two popular queries that are used in network analysis. Similar to the DBLP queries, we identify the ranked user pairs in the two hop and three hop neighborhoods for all users. The ranking is the sum of weights of the user pair. These queries have widespread application in understanding information flow in a network [51] and are used in recommendation systems [30, 47]. For LDBC benchmark, we use the multi-source version of **Q3, Q10** and **Q11**. Each of these queries are variants of the neighborhood analysis and contains UNION.

6.2 Small Scale Experiments

In this section, we compare the empirical performance of the algorithm given by Theorem 1 (labeled as LinDelay in all figures) against the baselines for each query. In order to perform a fair comparison, we materialize the top-k answers in-memory since other engines also do it. However, a strength of our system is that if a downstream task only requires the output as a stream, we are able to enumerate the result instead of materializing it, which is not possible with other engines.

Sum ordering. Figure 5 shows the main results for the DBLP and IMDB datasets when the ranking function is the sum function and the weights are chosen randomly. Let us first review the results for the DBLP dataset. Figure 5a shows the running time for different values of k in the limit clause. The first observation is that all engines materialize the join result, followed by deduplicating and sorting according to the ranking function which leads to poor performance for all baselines. This is because all engines treat sorting and distinct clause as blocking operators, verified by examining the query plan. On the other hand, our approach is limit-aware. For small values of k, we are up to two orders of magnitude faster and as the value of k increases, the total running time of our algorithm increases linearly. Even when our algorithm has to enumerate and materialize the entire result, it is still faster than asking the engines for the top-10 results. This is a direct benefit of generating the output in deduplicated and ranked order. As the path length increases from two to three and four path (Figure 5b and Figure 5c), the performance gap between existing engines and our approach also becomes larger. We also point out that all engines require a large amount of main memory for query execution. For example, MariaDB requires about 40GB of memory for executing $\text{DBLP}_{4\text{hop}}$. In contrast, the space overhead of our algorithm is dominated by the size of the priority queue. For DBLP dataset, our approach requires a measly 1.3GB, 4GB, 3GB and 2.7GB total space for $\text{DBLP}_{2\text{hop}}$, $\text{DBLP}_{3\text{hop}}$, $\text{DBLP}_{4\text{hop}}$ and $\text{DBLP}_{3\text{star}}$ respectively. For $\text{DBLP}_{3\text{hop}}$, $\text{DBLP}_{4\text{hop}}$ and $\text{DBLP}_{3\text{star}}$, we also implement breadth first search (BFS) followed by a sorting step using the idea of Algorithm 3. As it can be seen from the figures, BFS and sort provides an intermediate strategy which is faster than our algorithm for large values of k but at the cost of expensive materialization of the entire result, which may not be always possible (and is the case for IMDB dataset). However, deciding to use BFS and sort requires knowledge of the output result size, which is unknown apriori and difficult to estimate. For the IMDB dataset, we observe a similar trend of our algorithm displaying superior performance compared to all other baselines. In this case, BFS and sorting even for $\text{DBLP}_{4\text{hop}}$ is not possible since the result is almost 0.5 trillion items. For $\text{DBLP}_{3\text{star}}$, none of the engines were able to compute

the result after running for 5 hours when main memory ran out. BFS and sort also failed due to the size being larger than the main memory limit. Lastly, Neo4j was consistently the best performing (albeit marginally) engine among all baselines. While there is little scope for rewriting the SQL queries to try to obtain better performance, Neo4j has graph-specific operators such as variable length expansion. We tested multiple rewritings of the query to obtain the best performance (although this is the job of the query optimizer), which is finally reported in the figures. Regardless of the rewritings, Neo4j still treats materializing and sorting as a blocking operator which is a fundamental bottleneck.

Lexicographic ordering. Figures 6a,6b,6c and 6d show the running time for different values of k in the limit clause for lexicographic ranking function on DBLP (i.e. we replace A_1.weight + A_2.weight with A_1.weight, A_2.weight in the `ORDER BY` clause) for random weights. The first striking observation here is that the running time for all baseline engines is identical to that of sum function. This demonstrates that existing engines are also agnostic to the ranking function in the query and fail to take advantage of the additional structure. However, lexicographic functions are easier to handle in practice than sum because we can avoid the use of a priority queue altogether. This in turn leads to faster running time since push and pops from the priority queue are expensive due to the logarithmic overhead and need for re-balancing of the tree structure. Thus, we obtain a 2× improvement for lexicographic ordering as compared to the sum function.

Join ordering. At this point, the reader may wonder what is the impact of different join orderings on the query execution time for DBMS engines in the presence of `ORDER BY`. To investigate this, we supply join order hints to each of the engines. We run the queries on all possible join order hints to find the best possible running time. We found that the join order hints had virtually no impact on execution time. For instance, $\text{DBLP}_{4\text{hop}}$ on Neo4J takes 5521.61s without any join hints and the best possible join ordering reduces the time to 5418.23s, a mere 1.8% reduction. This is not surprising since the bottleneck for all engines is the materialization of the unsorted output, which is orders of magnitude larger than the final output and ends up being the dominant cost. In fact, for queries containing only self-joins, join order hints do not have any impact on the query plan because all relations are identical. Further, the number of possible join orderings that may need to be explored is exponential in the number of relations. On the other hand, our algorithm has the advantage of bypassing the materialization due to the delay based problem formulation and use of multi-way joins.

Logarithmic weights. Instead of choosing the weights randomly, we also investigate the behavior when the weights scale logarithmically w.r.t. to the degree. We observed that all systems as well as our algorithm had identical execution times. This is not surprising considering that no algorithm takes into account the actual distribution of the weights. This observation points to an additional opportunity for optimization where one could use the weight distribution to allow for fine-grained, data-dependent processing. We leave the study of this problem for future work.

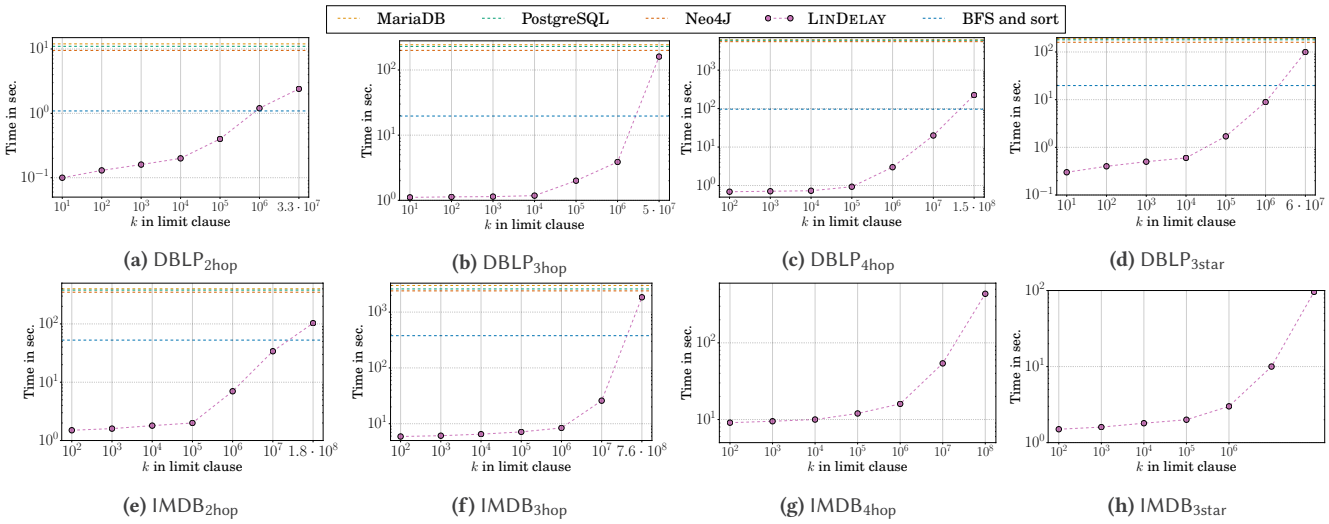
6.2.1 Enumeration with Preprocessing. We next investigate the empirical performance of the preprocessing step and its impact on the result enumeration as described by Theorem 2. For all experiments in this section, we fix k to be large enough to enumerate the entire result (which is equivalent to having no limit clause at all).

Sum ordering. Figure 7a and Figure 7b show the tradeoff between space used by the data structure constructed in the preprocessing phase and running time of the enumeration algorithm for $\text{DBLP}_{2\text{hop}}$ and $\text{IMDB}_{2\text{hop}}$ repectively. We show the tradeoff for 6 different space budgets but the user is free to choose any space budget in the entire spectrum. As expected, the time required to enumerate the result is large when there is no preprocessing and it gradually drops as more and more results are materialized in the preprocessing phase. The sum of preprocessing time and enumeration time is not a flat line: this is because as an optimization, we do not use priority queues in the preprocessing phase. Instead, we can simply use the BFS and sort algorithm for all chosen nodes which need to be materialized. This is a faster approach in practice as we avoid use of priority queues but priority queues cannot be avoided for enumeration phase. We observe similar trend for $\text{DBLP}_{3\text{star}}$, $\text{IMDB}_{3\text{star}}$ on both datasets as well.

6.2.2 Cyclic Queries. We also compare the performance of our algorithm to other systems for cyclic queries. We choose four cyclic queries found commonly in practice inspired by [62]: four cycle, six cycle, eight cycle and bowtie query (two four cycles joined at a common attribute). Figure 10 shows the performance of our algorithm on the DBLP dataset for the sum function. As the table shows, our algorithm is able to process all queries within 200 seconds, with the bowtie query being the most computationally intensive. In contrast, for $k = 10$ the fastest performing engine Neo4J required 240s (450s) for four cycle (six cycle). It did not finish execution for eight cycle and bowtie query due to an out of memory error. For the IMDB dataset, our algorithm was able to process all queries, while Neo4J was not able to process any query (except four cycle) due to its large memory requirement. We defer those experiments to [23].

6.3 Large Scale Experiments and Scalability

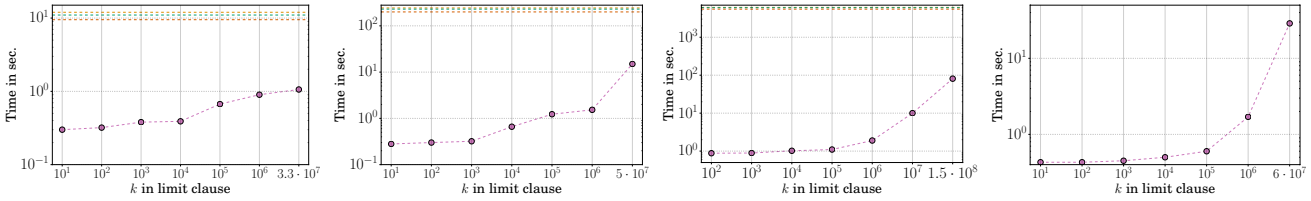
In this section, we investigate the performance of our techniques on the large scale datasets. Figure 8a and 8b shows the time to find the top-k answers for the Memetracker dataset on two neighborhood and three neighborhood queries. Compared to the small scale datasets, the execution time increases rapidly even for low values of k. This is attributed to the high duplication of answers, which leads to a rapidly increasing priority queue size. *None of MariaDB, Postgres and Neo4J were able to finish, or even to find the top-*10 *answers, within* 5 *hours in our experiments.* The same trend is also observed for the Friendster dataset as shown in Figure 8d and 8c. Similar to the small-scale datasets, lexicographic functions were faster than the sum function for our algorithm but DBMS engines were unable to finish query execution. We also conduct scalability experiments on LDBC benchmark queries that contain the `ORDER BY` clause. Figure 9 shows the scalability of our algorithm for finding answers of queries **Q3, Q10, Q11**. As the scale factor increases, the execution time also increases linearly. For each of these queries, all engines require more than 3 hours to compute the result even for SF = 10 and $k = 10$. This is because of the serial execution plan generated by the engines, forcing the materialization of the unsorted result before sorting and filtering for top-k.



(a) $DBLP_{2hop}$ (b) $DBLP_{3hop}$ (c) $DBLP_{4hop}$ (d) $DBLP_{3star}$

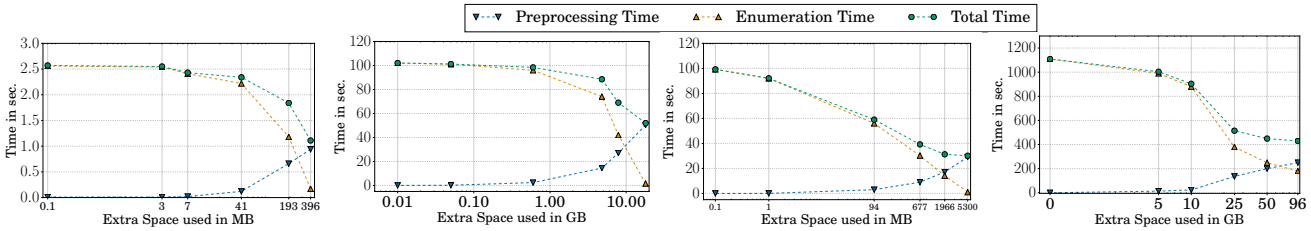
(e) $IMDB_{2hop}$ (f) $IMDB_{3hop}$ (g) $IMDB_{4hop}$ (h) $IMDB_{3star}$

Figure 5: Comparing our algorithm with state-of-the-art engines for sum function



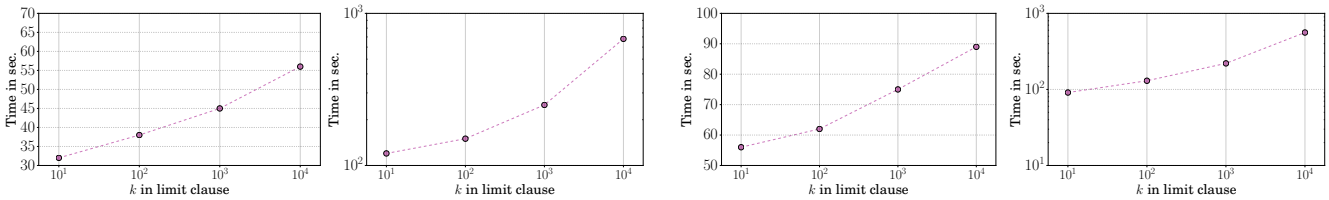
(a) $DBLP_{2hop}$ (b) $DBLP_{3hop}$ (c) $DBLP_{4hop}$ (d) $DBLP_{3star}$

Figure 6: Comparing our linear delay algorithm with state-of-the-art engines for lexicographic function.



(a) $DBLP_{2hop}$ (b) $IMDB_{2hop}$ (c) $DBLP_{3star}$ (d) $IMDB_{3star}$

Figure 7: Comparing the preprocessing and enumeration tradeoff for sum function when enumerating the entire result



(a) Memetracker 2-neighborhood (b) Memetracker 3-neighborhood (c) Friendster 2-neighborhood (d) Friendster 3-neighborhood

Figure 8: LinDelay performance on large-scale datasets

	SF = 10	SF = 20	SF = 30	SF = 40	SF = 40
Q3	5.91s	9.63s	13.47s	18.23s	22.18s
Q10	2.82s	3.47s	4.65s	5.23s	6.46s
Q11	0.78s	1.07s	1.56s	1.82s	2.36s

Figure 9: Scalability for different scale factors (SF) in LDBC

	k = 10	k = 10^2	k = 10^3	k = 10^4
four cycle	0.85s	0.95s	1.2s	1.8s
six cycle	13.1s	17.7s	25.6s	38.2s
eight cycle	33.4s	48.7s	63.9s	77.8s
bowtie	112s	125s	156s	192s

Figure 10: Cyclic query performance on the DBLP dataset for different values of k in the `LIMIT` clause.

7 RELATED WORK

Top-k. Top-k ranked enumeration of full join queries has been studied extensively by the database community for both certain [5, 12, 35, 36, 45, 46, 55, 61] and uncertain databases [57, 68]. 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 [35] for a comprehensive survey. We note that none of these works consider non-trivial join-project queries (see Appendix in [23] for more discussion). Ours is the first work to consider the ranked enumeration of arbitrary join-project queries.

Rank aggregation algorithms. Top-k processing over ranked lists of objects has a rich history. The problem was first studied by Fagin et al. [28, 29] 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. A key limitation of these works is that it expects the input to be materialized, i.e., $Q(D)$ must already be computed and stored for the algorithm to perform random access.

Unranked enumeration of query results. Recent work by Kara et al. [37] showed that for a small but important fragment of CQs known as hierarchical queries, it is possible to obtain a tradeoff between preprocessing and delay guarantees. Importantly, this result is applicable even in the presence of arbitrary projection. However, the authors did not investigate how to add ranking because adding priority queues at different location in the join tree leads to different complexities. In fact, follow up work [22] showed that the same unranked enumeration could be performed with better delay guarantees under certain settings. Our work considers the class of CQs with arbitrary projections and we are also able to extend the main result to UCQs, an even broader class of queries. Naturally, our algorithm automatically recovers the existing results for full CQs as well [24, 62], in addition to the first extensive empirical evidence on how ranked enumeration can be performed for CQs containing projections beyond free-connex queries.

Factorization and Aggregation. Factorized databases [11, 19, 53] exploit the distributivity of product over union to represent query results compactly and generalize the results on bounded `fhwto` the non-Boolean case [53]. [3] captures a wide range of aggregation problems over semirings. Factorized representations can also enumerate the query results with constant delay according to lexicographic orders of the variables [10]. For that to work, the lexicographic order must "agree" with the factorization order. However, it was shown in [24] that the algorithm for lexicographic ordering is not optimal. Further, since all prior work in this space using the concept of variable ordering, adding projections to the query forces the building of a GHD that can materialize the entire join query result, which is expensive and an unavoidable drawback.

Ranked enumeration. Both Chang et al. [16] and Yang et al. [65] provide any-k algorithms for *graph queries* instead of the more general CQs; Kimelfeld and Sagiv [39] give an any-k algorithm for acyclic queries with polynomial delay. Recent work on ranked enumeration of MSO logic over words is also of particular interest [15]. None of these existing works give any non-trivial guarantees for CQs with projections. Ours is the first work in this space that provides non-trivial guarantees.

8 CONCLUSION

In this paper, we study the problem of ranked enumeration for CQs with projections. We present a general algorithm that can enumerate query results according to two commonly-used ranking functions (`SUM`, `LEXICOGRAPHIC`) with near-linear delay after near-linear preprocessing time. We also show how to extend our results to a broader class of queries known as UCQs. For star queries, an important and practical fragment of CQs, we further show how to achieve a smooth tradeoff between the delay, preprocessing time and space used for data structure. Extensive experiments demonstrate that our methods are up to three orders of magnitude better when compared to popular open-source RDBMS and specialized graph engines. This work opens up several exciting future work challenges. The first important problem is to extend our results from main memory setting to the distributed setting. Since the cost of I/O must also be taken into account, it becomes important to identify the optimal priority queue storage layout to ensure that access cost is low. It would also be interesting to develop output balanced algorithms. The second exciting challenge is to incorporate approximation into the ranking. For some applications, it may be sufficient to get an approximately ordered output which could lead to improved running time guarantees. Finally, it would be useful to re-rank the query results when the ranking function is changed by the user and extend our ideas to non-monotone ranking functions.

ACKNOWLEDGMENTS

This research was supported in part by National Science Foundation grants CRII-1850348 and III-1910014. We would like to thank the anonymous reviewers for their careful reading and valuable comments. We also thank Wim Martens for pointing us to the reference [14] that motivates the study of star queries.

REFERENCES

[1] Amir Abboud and Virginia Vassilevska Williams. 2014. Popular conjectures imply strong lower bounds for dynamic problems. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*. IEEE, 434–443.

[2] Amir Abboud, Virginia Vassilevska Williams, and Huacheng Yu. 2018. Matching triangles and basing hardness on an extremely popular conjecture. *SIAM J. Comput.* 47, 3 (2018), 1098–1122.

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

[4] Mahmoud Abo Khamis, Hung Q Ngo, and Dan Suciu. 2017. 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*. 429–444.

[5] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. 2011. Best position algorithms for efficient top-k query processing. *Information Systems* 36, 6 (2011), 973–989.

[6] Noga Alon, Raphael Yuster, and Uri Zwick. 1994. Finding and counting given length cycles. In *European Symposium on Algorithms*. Springer, 354–364.

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

[8] Renzo Angles and Claudio Gutierrez. 2011. Subqueries in SPARQL. *AMW* 749 (2011), 12.

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

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

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

[12] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Christian Theobalt, and Gerhard Weikum. 2006. Io-top-k: Index-access optimized top-k query processing. (2006).

[13] Maria Biryukov. 2008. Co-author network analysis in DBLP: Classifying personal names. In *International Conference on Modelling, Computation and Optimization in Information Systems and Management Sciences*. Springer, 399–408.

[14] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *The VLDB Journal* 29, 2 (2020), 655–679.

[15] Pierre Bourhis, Alejandro Grez, Louis Jachiet, and Cristian Riveros. 2021. Ranked enumeration of MSO logic on words. *ICDT* (2021).

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

[17] Jinpeng Chen, Yu Liu, Guang Yang, and Ming Zou. 2018. Inferring tag co-occurrence relationship across heterogeneous social networks. *Applied Soft Computing* 66 (2018), 512–524.

[18] Philipp Christmann, Rishiraj Saha Roy, and Gerhard Weikum. 2021. Efficient Contextualization using Top-k Operators for Question Answering over Knowledge Graphs. *arXiv preprint arXiv:2108.08597* (2021).

[19] Radu Ciucanu and Dan Olteanu. 2015. *Worst-case optimal join at a time*. Technical Report. Technical report, Oxford.

[20] Olivier Corby and Catherine Faron-Zucker. 2007. Implementation of SPARQL query language based on graph homomorphism. In *International Conference on Conceptual Structures*. Springer, 472–475.

[21] Nilesh Dalvi and Dan Suciu. 2007. Efficient query evaluation on probabilistic databases. *The VLDB Journal* 16, 4 (2007), 523–544.

[22] Shaleen Deep, Xiao Hu, and Paraschos Koutris. 2021. Enumeration Algorithms for Conjunctive Queries with Projection. In *24th International Conference on Database Theory (ICDT 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[23] Shaleen Deep, Xiao Hu, and Paraschos Koutris. 2022. Ranked Enumeration of Join Queries with Projections. *arXiv preprint arXiv:2201.05566* (2022).

[24] Shaleen Deep and Paraschos Koutris. 2021. Ranked Enumeration of Conjunctive Query Results. In *24th International Conference on Database Theory*.

[25] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The design and operation of CloudLab. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 1–14.

[26] Ergin Elmacioglu and Dongwon Lee. 2005. On six degrees of separation in DBLP-DB and more. *ACM SIGMOD Record* 34, 2 (2005), 33–40.

[27] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

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

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

[30] Chenyuan Feng, Zuozhu Liu, Shaowei Lin, and Tony QS Quek. 2019. Attention-based graph convolutional network for recommendation system. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 7560–7564.

[31] Michael L Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)* 34, 3 (1987), 596–615.

[32] Todd J Green, Shan Shan Huang, Boon Thau Loo, Wenchao Zhou, et al. 2013. *Datalog and recursive query processing*. Now Publishers.

[33] Stephen Harris and Nigel Shadbolt. 2005. SPARQL query processing with conventional relational database systems. In *International Conference on Web Information Systems Engineering*. Springer, 235–244.

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

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

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

[37] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2020. 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*. 375–392.

[38] Mehdi Kargar, Lukasz Golab, Divesh Srivastava, Jaroslaw Szlichta, and Morteza Zihayat. 2020. Effective Keyword Search over Weighted Graphs. *IEEE Transactions on Knowledge and Data Engineering* (2020).

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

[40] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. 2016. Higher lower bounds from the 3SUM conjecture. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 1272–1287.

[41] Onur Küçüktunç, Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. 2012. Recommendation on academic networks using direction aware citation analysis. *arXiv preprint arXiv:1205.1143* (2012).

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

[43] Jyoti Leeka, Srikanta Bedathur, Debajyoti Bera, and Medha Atre. 2016. Quark-X: An efficient top-k processing framework for RDF quad stores. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 831–840.

[44] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

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

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

[47] Xiaoming Li, Hui Fang, and Jie Zhang. 2019. Supervised user ranking in signed social networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 184–191.

[48] Stefan Manegold, Martin L Kersten, and Peter Boncz. 2009. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1648–1653.

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

[50] Dániel Marx. 2021. Modern Lower Bound Techniques in Database Theory and Constraint Satisfaction. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 19–29.

[51] Seth A Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. 2014. Information network or social network? The structure of the Twitter follow graph. In *Proceedings of the 23rd International Conference on World Wide Web*. 493–498.

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

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

[54] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)* 34, 3 (2009),

1–45.

[55] Yan Qi, K Selçuk Candan, and Maria Luisa Sapino. 2007. 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*. VLDB Endowment, 507–518.

[56] Erhard Rahm and Andreas Thor. 2005. Citation analysis of database publications. *ACM Sigmod Record* 34, 4 (2005), 48–53.

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

[58] Luc Segoufin. 2015. Constant Delay Enumeration for Conjunctive Queries. *SIGMOD Record* 44, 1 (2015), 10–17. https://doi.org/10.1145/2783888.2783894

[59] Juan F Sequeda and Daniel P Miranker. 2013. Ultrawrap: SPARQL execution on relational data. *Journal of Web Semantics* 22 (2013), 19–39.

[60] Yizhou Sun and Jiawei Han. 2013. Mining heterogeneous information networks: a structural analysis approach. *Acm Sigkdd Explorations Newsletter* 14, 2 (2013), 20–28.

[61] Panayiotis Tsaparas, Themistoklis Palpanas, Yannis Kotidis, Nick Koudas, and Divesh Srivastava. 2003. Ranked join indices. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. IEEE, 277–288.

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

[63] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2021. Beyond Equi-joins: Ranking, Enumeration and Factorization. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, Vol. 14. 2599–2612.

[64] Konstantinos Xirogiannopoulos and Amol Deshpande. 2017. Extracting and Analyzing Hidden Graphs from Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 897–912.

[65] Xiaofeng Yang, Deepak Ajwani, Wolfgang Gatterbauer, Patrick K Nicholson, Mirek Riedewald, and Alessandra Sala. 2018. Any-k: Anytime Top-k Tree Pattern Retrieval in Labeled Graphs. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 489–498.

[66] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *VLDB*, Vol. 81. 82–94.

[67] Xiao Yu, Yizhou Sun, Brandon Norick, Tiancheng Mao, and Jiawei Han. 2012. User guided entity similarity search using meta-path selection in heterogeneous information networks. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. 2025–2029.

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