Enumeration Algorithms for Conjunctive Queries with Projection

Shaleen Deep
Department of Computer Sciences, University of Wisconsin-Madison, Madison, Wisconsin, USA
shaleen@cs.wisc.edu

Xiao Hu
Department of Computer Sciences, Duke University, Durham, North Carolina, USA
xh102@cs.duke.edu

Paraschos Koutris
Department of Computer Sciences, University of Wisconsin-Madison, Madison, Wisconsin, USA
paris@cs.wisc.edu

Abstract

We investigate the enumeration of query results for an important subset of CQs with projections, namely star and path queries. The task is to design data structures and algorithms that allow for efficient enumeration with delay guarantees after a preprocessing phase. Our main contribution is a series of results based on the idea of interleaving precomputed output with further join processing to maintain delay guarantees, which maybe of independent interest. In particular, we design combinatorial algorithms that provide instance-specific delay guarantees in nearly linear preprocessing time. These algorithms improve upon the currently best known results. Further, we show how existing results can be improved upon by using fast matrix multiplication. We also present the first known results involving tradeoff between preprocessing time and delay guarantees for enumeration of path queries that contain projections. CQs with projection where the join attribute is projected away is equivalent to boolean matrix multiplication. Our results can therefore be also interpreted as sparse, output-sensitive matrix multiplication with delay guarantees.

1 Introduction

The efficient evaluation of join queries over static databases is a fundamental problem in data management. There has been a long line of research on the design and analysis of algorithms that minimize the total runtime of query execution in terms of the input and output size [22, 14, 13]. However, in many data processing scenarios it is beneficial to split query execution into two phases: the *preprocessing phase*, which computes a space-efficient intermediate data structure, and the *enumeration phase*, which uses the data structure to enumerate the query results as fast as possible, with the goal of minimizing the *delay* between outputting two consecutive tuples in the result. This distinction is beneficial for several reasons. For instance, in many scenarios, the user wants to see one (or a few) results of the query as fast as possible: in this case, we want to minimize the time of the preprocessing phase, such that we can output the first results quickly. On the other hand, a data processing pipeline may require that the result of a query is accessed multiple times by a downstream task: in this case, it is better to spend more time during the preprocessing phase, to guarantee a faster enumeration with smaller delay.

Previous work in the database literature has focused on finding the class of queries that can be computed with $O(|D|)$ preprocessing time (where D is the input database instance) and constant delay during the enumeration phase. The main result in this line of work shows that full (i.e., without projections) acyclic Conjunctive Queries (CQs) admit linear preprocessing time and constant delay [3]. If the CQ is not full but its free variables satisfy the *free-connex* property, the same preprocessing time and delay guarantees can still be achieved. It is also known that for any (possibly non-full) acyclic CQ, it is possible to

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

achieve linear delay after linear preprocessing time [3]. When the CQ is full but not acyclic, factorized databases uses $O(|D|^{\texttt{fhw}})$ preprocessing time to achieve constant delay, where $\texttt{fhw}$ is the *fractional hypertree width* of the query. We should note here that it is always possible to spend enough preprocessing time to achieve constant delay, since we can always compute and materialize the result of the query during preprocessing.

The aforementioned prior work investigates specific points in the preprocessing time-delay tradeoff space. While the story for full acyclic CQs is relatively complete, the same is not true for general CQs, even for acyclic CQs with projections. For instance, consider the simplest such query: $Q_{\text{path}} = \pi_{x,z}(R(x,y) \bowtie S(y,z))$, which joins two binary relations and then projects out the join attribute. For this query, [3] ruled out a constant delay algorithm with linear time preprocessing unless the matrix multiplication exponent $\omega = 2$. However, we can obtain $O(|D|)$ delay with $O(|D|)$ preprocessing time. We can also obtain $O(1)$ delay with $O(|D|^2)$ preprocessing by computing and storing the full result. It is worth asking whether there are other interesting points in this tradeoff between preprocessing time and delay. Towards this end, Kara et. al [12] showed that for any hierarchical CQ [1] (possibly with projections), there always exists a smooth tradeoff between preprocessing time and delay. This is the first improvement over the results of Bagan et. al [3] in over a decade for queries involving projections. Applied to the query Q_{path}, the main result of of [12] shows that for any $\epsilon \in [0,1]$, we can obtain $O(|D|^{1-\epsilon})$ delay with $O(|D|^{1+\epsilon})$ preprocessing time.

In this paper, we continue the investigation of the tradeoff between preprocessing time and delay for CQs with projections. We focus on two classes of CQs: *star queries*, which are a popular subset of hierarchical queries, and a useful subset of non-hierarchical queries known as *path queries*. We focus narrowly on these two classes for two reasons. First, star queries are of immense practical interest given their connections to set intersection, set similarity joins and applications to entity matching (we refer the reader to [7] for an overview). The most common star query seen in practice is the query Q_{path}. The same holds true for path queries, which are fundamental in graph processing. Second, as we will see in this paper, even for the simple class of star queries, the tradeoff landscape is complex and requires the development of novel techniques. We also present a result on another subset of hierarchical CQs that we call lef-deep.

Our key insight is to design enumeration algorithms that depend not only on the input size $|D|$, but are also aware of other data-specific parameters such as the output size. To give a flavor of our results, consider the query Q_{path}, and denote by $\texttt{OUT}_{\bowtie}$ the output of the corresponding query without projections, $R(x,y) \bowtie S(y,z)$. We can show the following result.

▶ **Theorem 1.** *Given a database instance D, we can enumerate the output of $Q_{path} = \pi_{x,z}(R(x,y) \bowtie S(y,z))$ with preprocessing time $O(|D| \log |D|)$ and delay $O(|D|^2/|\texttt{OUT}_{\bowtie}|)$.*

Note here that the preprocessing time is (almost) linear, while the delay is dependent on the size of the full join. In the worst case where $|\texttt{OUT}_{\bowtie}| = \Theta(|D|^2)$, we actually obtain the best delay, which will be constant. Compare this to the result of [12], which would require $O(|D|^2)$ preprocessing time to achieve the same guarantee, giving us a polynomial improvement. On the other hand, if $|\texttt{OUT}_{\bowtie}| = \Theta(|D|)$, we obtain only a linear delay guarantee of $O(|D|)$. [2] The reader may wonder how our result compares in general with the tradeoff in [12]; we will in fact show that the tradeoff in [12] is suboptimal and that Theorem 1 implies that we

[1] Hierarchical CQs are a strict subset of acyclic CQs.
[2] We do not need to consider the case where $|\texttt{OUT}_{\bowtie}| \leq |D|$), since then we can simply materialize the full result during the preprocessing time.

Queries	Preprocessing	Delay	Source
Full acyclic CQ	$O(\|D\|)$	$O(\|D\|)$	[3]
Free-connex CQ (projections)	$O(\|D\|)$	$O(1)$	[3]
Full CQ	$O(\|D\|^{\texttt{fhw}})$	$O(1)$	[15]
Full CQ	$O(\|D\|^{\texttt{subw}} \log \|D\|)$	$O(1)$	[1]
Hierarchical CQ (projections)	$O(\|D\|^{1+(\texttt{w}-1)\epsilon})$	$O(\|D\|^{1-\epsilon}), \epsilon \in [0,1]$	[12]
Star query with k relations (projections)	$O(\|D\| \log \|D\|)$	$O(\|D\|^{k/(k-1)}/\|\texttt{OUT}_{\bowtie}\|^{1/k-1})$	this paper
Path query with k relations (projections)	$O(\|D\|^{2-\epsilon/(k-1)})$	$O(\|D\|^{\epsilon} \log \|D\|), \epsilon \in [0,1)$	this paper
Left-deep hierarchical CQ (projections)	$O(\|D\| \cdot \log \|D\|)$	$O(\|D\|^{k}/\|\texttt{OUT}_{\bowtie}\|)$	this paper
Two path query (projections)	$O(\|D^{2\epsilon})$	$O(\|D\|^{1-\epsilon}), \epsilon \in [2/3,1]$	this paper

Figure 1 Preprocessing time and delay guarantees for different queries. $|\texttt{OUT}_{\bowtie}|$ denotes the size of join query under consideration but without any projections. $\texttt{subw}$ denotes the submodular width of the query.

can always get a strictly better tradeoff point. Figure 1 summarizes the prior work and the results present in this paper.

Our Contribution. In this paper, we improve the state-of-the-art on the preprocessing time-delay tradeoff for a subset of CQs with projections. We summarize our main technical contributions below (highlighted in Figure 1):

1. Our main contribution consists of a novel algorithm (Theorem 7) that achieves output-dependent delay guarantees for star queries after (almost) linear preprocessing time. Specifically, we show that for the query $\pi_{x_1,\ldots,x_k}(R_1(x_1,y) \bowtie \cdots \bowtie R_k(x_k,y))$ we can achieve delay $O(|D|^{k/(k-1)}/|\texttt{OUT}_{\bowtie}|^{1/k-1})$ with almost linear preprocessing. Our key idea is to identify an appropriate degree threshold to split a relation into partitions of *heavy* and *light*, which allows us to perform efficient enumeration. For star queries, our result implies that there exists no smooth tradeoff between preprocessing time and delay guarantees as stated in [12] for the class of star queries.
2. We introduce the novel idea of *interleaving* join query computation (Section 3) which forms the foundation for our algorithms, and may be of independent interest in the context of enumeration algorithms. Specifically, we show that it is possible to union the output of two algorithms $\mathcal{A}$ and $\mathcal{A}'$ with δ delay guarantee where $\mathcal{A}$ enumerates query results with δ delay guarantees but $\mathcal{A}'$ does not. This technique allows us to compute a subset of a query *on-the-fly* when enumeration with good delay guarantees is impossible.
3. We show (Subsection 4.4) how fast matrix multiplication can be used to obtain a tradeoff between preprocessing time and delay that further improves upon the tradeoff in [12].
4. We present an algorithm for left-deep hierarchical queries with almost linear preprocessing time and output-dependent delay guarantees.
5. Finally, we present the first results on preprocessing time-delay tradeoffs for a non-hierarchical query with projections, for the class of path queries. A path query has the form $\pi_{x_1,x_{k+1}}(R_1(x_1,x_2) \bowtie \cdots \bowtie R_k(x_k,x_{k+1}))$. Our results show that we can achieve delay $O(|D|^{\epsilon} \cdot \log |D|)$ with preprocessing time $O(|D|^{2-\epsilon/(k-1)})$ for any $\epsilon \in [0,1)$.

Organization. We introduce the basic terminology and problem setting in Section 2. In Section 3, we prove several helper lemmas that may be of independent interest for enumeration algorithms. Section 4 presents our main results for star queries, Section 6 for left-deep hierarchical queries, and Section 6 for path queries. We discuss related work in Section 7, and finally conclude in Section 8 with promising future research directions and open problems.

2 Problem Setting

In this section we present the basic notation and terminology.

2.1 Conjunctive Queries

In this paper we will focus on the class of *conjunctive queries* (CQs), which we denote as

$$Q = \pi_{\mathbf{y}}(R_1(\mathbf{x}_1) \bowtie R_2(\mathbf{x}_2) \bowtie \ldots \bowtie R_n(\mathbf{x}_n))$$

Here, the symbols $\mathbf{y}, \mathbf{x}_1, \ldots, \mathbf{x}_n$ are vectors that contain *variables* or *constants*. We say that Q is *full* if there is no projection. We will typically use the symbols $x, y, z, \ldots$ to denote variables, and $a, b, c, \ldots$ to denote constants. We use $Q(D)$ to denote the result of the query Q over input database D.

In this paper, we will focus on CQs that have no constants and no repeated variables in the same atom (both cases can be handled within a linear time preprocessing step, so this assumption is without any loss of generality). Such a query 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 be particularly interested in two families of CQs that are fundamental in query processing, star and path queries. The *star query* with k relations is expressed as:

$$Q_k^* = R_1(\mathbf{x}_1, \mathbf{y}) \bowtie R_2(\mathbf{x}_2, \mathbf{y}) \bowtie \cdots \bowtie R_k(\mathbf{x}_k, \mathbf{y})$$

where $\mathbf{x}_1, \ldots, \mathbf{x}_k$ have disjoint sets of variables. The *path query* with k (binary) relations is expressed as:

$$P_k = R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie \cdots \bowtie R_k(x_k, x_{k+1})$$

In Q_k^*, variables in each relation R_i are partitioned into two sets: variables $\mathbf{x}_i$ that are present only in R_i and a common set of join variables $\mathbf{y}$ present in every relation.

Hierarchical Queries. A CQ Q is *hierarchical* if for any two of its variables, either their sets of atoms are disjoint or one is contained in the other [18]. For example, Q_k^* is hierarchical for any k, while P_k is hierarchical only when $k \leq 2$.

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 $\rho^*(\mathcal{H}) = \rho^*_{\mathcal{H}}(\mathcal{V})$.

Tree Decompositions. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph of a CQ 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 $\{t \mid x \in \mathcal{B}_t\}$ form a connected subtree of $\mathcal{T}$.

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 minimum fractional edge cover of the vertices in $\mathcal{B}_t$. The fractional hypertree width of a query Q, denoted $\texttt{fhw}(Q)$, is the minimum fractional hypertree width among all tree decompositions of its hypergraph. We say that a query is *acyclic* if $\texttt{fhw}(Q) = 1$.

Computational Model. To measure the running time of our algorithms, we will use the uniform-cost RAM model [10], 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.

2.2 Fast Matrix Multiplication

Let A be a $U_1 \times U_3$ matrix and C be a $U_3 \times U_2$ matrix over any field $\mathcal{F}$. $A_{i,j}$ is the shorthand notation for entry of A located in row i and column j. The matrix product is given by $(AC)_{i,j} = \sum_{k=1}^{U_3} A_{i,k}C_{k,j}$. Algorithms for fast matrix multiplication are of extreme theoretical interest given its fundamental importance. Throughout this paper, will assume that the exponent ω in matrix multiplication is $2 + o(1)$ as believed by many researchers. We will frequently use the following folklore lemma about rectangular matrix multiplication.

▶ **Lemma 2.** *Let ω be the smallest constant such that an algorithm to multiply two $n \times n$ matrices that runs in time $O(n^\omega)$ is known. Let $\beta = \min\{U, V, W\}$. Then fast matrix multiplication of matrices of size $U \times V$ and $V \times W$ can be done in time $O(UVW\beta^{\omega-3})$.*

Fixing $\omega = 2 + o(1)$, rectangular matrix multiplication can be done in time $O(UVW/\beta)$. A long line of research on fast matrix multiplication has dropped the complexity to $O(n^\omega)$, where $2 \leq \omega < 3$. The current best known value is $\omega = 2.373$ [9], but it is believed that the actual value is $2 + o(1)$.

2.3 Problem Statement

Given a Conjunctive Query Q and an input database D, we want to enumerate the tuples in $Q(D)$ in any order. We will study this problem in the enumeration framework similar to that of [16], where an algorithm can be decomposed into two phases:

- **Preprocessing phase:** it computes a data structure that takes space S_p in *preprocessing time* T_p.
- **Enumeration phase:** it outputs $Q(D)$ with no repetitions. This phase has 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 pair of consecutive tuples (and also the time to output the first tuple, and the time to notify that the enumeration phase has completed).

In this work, our goal is to study the relationship between the preprocessing time T_p and delay δ for a given CQ Q. Ideally, we would like to achieve the best possible delay in linear preprocessing time but this may not be always possible. As Figure 1 shows, when Q is full, with $T_p = O(|D|^{\texttt{fhw}})$, we can enumerate the results with constant delay $O(1)$ [15]. In the particular case where Q is acyclic, $\texttt{fhw} = 1$, and hence we can achieve constant delay with only linear preprocessing time. On the other hand, [3] shows that for every acyclic CQ, we can achieve linear delay $O(|D|)$ with linear preprocessing time $O(|D|)$

Recently, [12] showed that it is possible to get a tradeoff between the two extremes, for the class of hierarchical queries. Note that hierarchical queries are acyclic but not necessarily free-connex.

▶ **Theorem 3** (due to [12]). *Consider a hierarchical CQ Q with factorization width w, and an input instance D. Then, for any $\epsilon \in [0, 1]$ there exists an algorithm that can preprocess D in time $T_p = O(|D|^{1+(\mathsf{w}-1)\epsilon})$ and space $S_p = O(|D|^{1+(\mathsf{w}-1)\epsilon})$ such that we can enumerate the query output with*

$$\text{delay } \delta = O(|D|^{1-\epsilon}) \qquad \text{space } S_e = O(1).$$

The factorization width w of a hierarchical query is a notion defined by [12]. Intuitively, it refers to the AGM exponent of the query after removing all relations that do not contain any projection variables. For $\pi_{\mathbf{x}_1,\ldots,\mathbf{x}_k}(Q_k^*)$, the factorization width is $\mathsf{w} = k$. [12] is the first non-trivial result that improves upon the linear delay guarantees given by [3].

3 Helper Lemmas

Before we present the proof of our main results, we discuss three useful lemmas which will be used frequently, and may be of independent interest for enumeration algorithms. The first two lemmas are based on the key idea of *interleaving query results* which we describe next. We will say that an algorithm $\mathcal{A}$ provides *no delay guarantees* to mean that its delay guarantee is its total execution time. In other words, if an algorithm requires time T to complete, its delay guarantee is upper bounded by T.

▶ **Lemma 4.** *Consider two algorithms $\mathcal{A}$ and $\mathcal{A}'$ such that*

1. *$\mathcal{A}$ enumerates query results in total time T with no delay guarantees.*
2. *$\mathcal{A}'$ enumerates query results with delay δ and takes T' total time.*
3. *The outputs of $\mathcal{A}$ and $\mathcal{A}'$ are disjoint.*

Then, the union of the outputs of $\mathcal{A}$ and $\mathcal{A}'$ can be enumerated with delay $O(\delta \cdot \max\{1, T/T'\})$.

Proof. Let Δ denote a parameter to be fixed upon later. Note that in every δ time, we can emit one output result from $\mathcal{A}'$. But since we also want to compute the output from $\mathcal{A}$ that takes overall time T, we need to slow down the enumeration of $\mathcal{A}'$ sufficiently so that we do not run out of output from $\mathcal{A}'$. This can be done by interleaving the two algorithms in the following way: we allow $\mathcal{A}'$ to run for δ time and emit (at least) one output tuple, then switch to $\mathcal{A}$ and run it for $O(\Delta)$ time and then go back to $\mathcal{A}'$. We keep alternating between the two algorithms. This alternation takes $O(1)$ time in RAM model where the state of registers and program counter can be stored and retrieved enabling pause and resume of any algorithm. Our goal is to find a value of Δ such that $\mathcal{A}'$ does not run out of enumeration output until $\mathcal{A}$ has finished. This condition is satisfied for the smallest value of Δ such that $T \leq T'/\delta \cdot \Delta$ which yields $\Delta \geq T/T' \cdot \delta$. The overall delay guarantee is $O(\delta + \Delta) = O(\delta \cdot \max\{1, T/T'\})$. ◀

Lemma 4 tells us that as long as $T = O(T')$, the output of $\mathcal{A}$ and $\mathcal{A}'$ can be combined without giving up on delay guarantees by pacing the output of $\mathcal{A}'$. The next lemma introduces our second key idea of interleaving stored output result with *on-the-fly* query computation (the full algorithm is located in Appendix A).

▶ **Lemma 5.** *Consider an algorithm $\mathcal{A}$ that enumerates query results in total time T with no delay guarantees. Suppose that J output tuples have been stored apriori with no duplicates tuples, where $J \leq T$. Then, there exists an algorithm that enumerates the output with delay guarantee $\delta = O(T/J)$.*

Proof. Let $\delta = c \cdot T/J$, where c is a constant. We first store the J output results in a hash map. Using a similar interleaving strategy as above, we emit one result from J and allow algorithm $\mathcal{A}$ to run for δ time. Whenever $\mathcal{A}$ wants to emit an output tuple, it probes the hash map first and emits t only if t does not appear in the hash map. Each probe takes $O(1)$ time, so the total running time of $\mathcal{A}$ is still $O(T)$. Note that $\mathcal{A}$ terminates before the materialized output runs out. It can be easily checked that no duplicated result is emitted and $O(\delta)$ delay is guaranteed between every pair of consecutive results. Again, observe that we need the algorithm $\mathcal{A}$ to be pausable, which means that we should be able to resume the execution from where we left off. This can be achieved by storing the contents of all registers in the memory and loading it when required to resume execution. ◀

The final helping lemma allows us to enumerate the union of (possibly overlapping) results of m different algorithms, with the additional condition that all algorithms must output the results using exactly the same order.

▶ **Lemma 6.** *Consider m algorithms $\mathcal{A}_1, \mathcal{A}_2, \cdots, \mathcal{A}_m$ such that each $\mathcal{A}_i$ enumerates its output L_i with $O(\delta)$ delay using the same order. Then, the union of their output can be enumerated (without duplicates) with $O(m \cdot \delta)$ delay in the same order.*

Algorithm 1: MERGE($\mathcal{A}_1, \mathcal{A}_2, \cdots, \mathcal{A}_m$)

```
1  S ← {1, 2, ··· , m};
2  foreach i ∈ S do
3  |   e_i ← A_i.first() ;
4  while S ≠ ∅ do
5  |   w ← min_{i∈S} e_i ;          /* finds the smallest output over all algorithms */
6  |   output w ;
7  |   foreach i ∈ S do
8  |   |   if e_i = w then
9  |   |   |   e_i ← A_i.next();
10 |   |   if e_i = null then
11 |   |   |   S ← S − {i}          /* the algorithm completes its output */
```

Proof. We describe Algorithm 1. For simplicity of exposition, we assume that $\mathcal{A}_i$ outputs a null value when it finishes enumeration. Note that results enumerated by one algorithm are in order, thus it always outputs the locally minimum result over the remaining result to be enumerated. Algorithm 1 goes over all locally minimum results over all algorithms and outputs the smallest one as globally minimum result. Once a result is enumerated, all algorithms needs check whether it matches its locally minimum result. If yes, then $\mathcal{A}_i$ needs to update its locally minimum result by finding the next one. Then, Algorithm 1 just repeats this loop until all algorithms finish enumeration.

Observe that one distinct result is enumerated in each iteration of the while loop. It takes $O(m)$ time to find the globally minimum result (line 5) and $O(m \cdot \delta)$ to update all local minimum results (line 7-9). Thus, Algorithm 1 has a delay guarantee of $O(m \cdot \delta)$. ◀

Directly implied by Lemma 6, the *list merge* problem can be enumerated with delay guarantees: Given m lists $L_1, L_2, \cdots, L_m$ whose elements are drawn from a common domain, if elements in L_i are distinct (i.e no duplicates) and ordered, then the union of all lists $\bigcup_{i=1}^{m} L_i$ can be enumerated in order with delay $O(m)$. Note that the enumeration algorithm $\mathcal{A}_i$ degenerates to going over elements one by one in list L_i, which has $O(1)$ delay guarantee as long as indexes/pointers within L_i are well-built. Throughout the paper, we use this primitive as LISTMERGE($L_1, L_2, \cdots, L_m$).

4 Star Queries

In this section, we study enumeration algorithms for the star query $\pi_r(Q_k^*)$ where $r \subseteq \bigcup_{i \in \{1,2,\cdots,k\}} \mathbf{x}_i$. Our main result is Theorem 7 that we present below. We first present a warm-up proof for $\pi_r(Q_k^*)$ in Subsection 4.1, and then in Subsection 4.2 give the proof for the general result. In the next two subsections, we show how we can obtain different tradeoffs that can give even better guarantees for certain input instances.

▶ **Theorem 7.** *Consider the star query with projection $\pi_r(Q_k^*)$ where $r \subseteq \bigcup_{i\in\{1,2,\cdots,k\}} \mathbf{x}_i$ and an instance D [3]. There exists an algorithm with preprocessing time $T_p = O(|D| \log |D|)$ and preprocessing space $S_p = O(|D|)$, such that we can enumerate $Q_k^*(D)$ with*

$$\textit{delay } \delta = O\left(\frac{|D|^{k/k-1}}{|\mathtt{OUT}_{\bowtie}|^{1/k-1}}\right) \textit{ and space } S_e = O(|D|).$$

In the above theorem, the delay depends on the full join result size $|\mathtt{OUT}_{\bowtie}| = |Q_k^*(D)|$. As the join size increases, the algorithm can obtain better delay guarantees. In the extreme case when $|\mathtt{OUT}_{\bowtie}| = \Theta(|D|^k)$, it achieves constant delay with (almost) linear time preprocessing. In the other extreme, when $|\mathtt{OUT}_{\bowtie}| = \Theta(|D|)$, it achieves linear delay.

Observe that when $|\mathtt{OUT}_{\bowtie}| \leq |D|$, we can compute and materialize the result of the query in linear preprocessing time and achieve constant delay enumeration. Generalizing this observation, with $T_p = O(|\mathtt{OUT}_{\bowtie}|)$ preprocessing time, we can always achieve constant delay.

It is instructive now to compare Theorem 3 with Theorem 7. Suppose that we want to achieve delay $\delta = O(|D|^{1-\epsilon})$ for some $\epsilon \in [0, 1]$. Theorem 3 tells us that this requires $O(|D|^{1+\epsilon(k-1)})$ preprocessing time. We distinguish two cases:

- $|\mathtt{OUT}_{\bowtie}| \leq |D|^{1+\epsilon(k-1)}$: we can simply compute the full join $Q_k^*(D)$ and materialize it during the preprocessing phase, hence achieving $O(1)$ delay.
- $|\mathtt{OUT}_{\bowtie}| > |D|^{1+\epsilon(k-1)}$: we invoke the algorithm in Theorem 7 to achieve a better delay than Theorem 3 using only (nearly) linear preprocessing time, since the inequality $|D|^{k/k-1}/|\mathtt{OUT}_{\bowtie}|^{1/k-1} < |D|^{1-\epsilon}$ strictly holds in this range of $|\mathtt{OUT}_{\bowtie}|$.

In other words, *there exists no tradeoff between delay and preprocessing time as Theorem 3 suggests*: either we have enough preprocessing time to achieve constant delay, or we can achieve the desirable delay with (nearly) linear preprocessing time. Our results thus imply that, depending on $|\mathtt{OUT}_{\bowtie}|$, one must choose a different algorithm to achieve the optimal tradeoff between preprocessing time and delay. In the rest of the paper, for simplicity of exposition, we assume that all variable vectors $\mathbf{x}_i, \mathbf{y}$ in Q_k^* are singletons (i.e, all the relations are binary) and $r = \{x_1, x_2, \ldots, x_k\}$. The proof for the general query is a straightforward extension of the binary case.

4.1 Warm-up: 2-Path Query

As a warm-up step, we will present an algorithm for the query $Q_{\text{path}} = \pi_{x,z}(R(x, y) \bowtie S(y, z))$ that achieves $O(|D|^2/|\mathtt{OUT}_{\bowtie}|)$ delay with (nearly) linear preprocessing time.

At a high level, we will decompose the join into subqueries with disjoint outputs. The subqueries will be generated based on whether a valuation for x is *light* or not based on its degree in relation R. For all light valuations of x (degree at most δ), we will show that their enumeration is achievable with delay δ. For the heavy x valuations, we will show that they also can be computed *on-the-fly* while maintaining the delay guarantees.

Preprocessing Phase. We first process the input relations such that we remove any dangling tuples. During the preprocessing phase, we will store the input relations as a hash map and sort the valuations for x in increasing order of their degree. More specifically, for every tuple $t \in R(x, y)$, we create a hash map with key $\pi_x(t)$ and value $\pi_y(t)$; and for every

[3] We assume that r contains at least one variable from each $\mathbf{x}_i$. Otherwise, we can remove relations with no projection variables after the preprocessing phase.

tuple $t \in S(y,z)$, we create a hash map with key $\pi_y(t)$ and value $\pi_z(t)$. Finally, we sort all values in $\pi_x(R)$ in increasing order of their degrees in R (i.e $|\sigma_{x=v_i}R(x,y)|$ is the sort key). Let $\mathcal{L} = \{v_1, \ldots, v_n\}$ denote these values sorted by their degree and let $d_1, \ldots, d_n$ be their respective degrees. This list is also generated in the preprocessing phase. It is easy to see that this step takes time $O(|D| \log |D|)$.

Algorithm 2: Enumeration Phase

```
1  left ← 1
2  right ← n;
3  while left < right do
4  |  Assume π_y σ_{x=v_left} R_1 = {u_1, u_2, ··· , u_ℓ} and π_y σ_{x=v_right} R_1 = {u'_1, u'_2, ··· , u'_h};
5  |  ListMerge(π_z σ_{y=u_1} R_2, π_z σ_{y=u_2} R_2, ··· , π_z σ_{y=u_ℓ} R_2) for O(1) time;
6  |  left ← left + 1 if enumeration of left completed;
7  |  if left = right then
8  |  |  break
9  |  ListMerge(π_z σ_{y=u'_1} R_2, π_z σ_{y=u'_2} R_2, ··· , π_z σ_{y=u'_h} R_2) for O(1) time;
10 |  right ← right − 1 if enumeration of right completed;
11 ListMerge(π_z σ_{y=u'_1} R_2, π_z σ_{y=u'_2} R_2, ··· , π_z σ_{y=u'_h} R_2);
```

Enumeration Phase. The enumeration algorithm is presented in Algorithm 2. The algorithm alternates between low-degree and high-degree values in $\mathcal{L}$. The first key observation is that, for a given $v_i \in \mathcal{L}$, we can enumerate the result of the subquery $\sigma_{x=v_i}(Q_{\text{path}})$ with delay $O(d_i)$. This can be accomplished by observing that the subquery is equivalent to list merging and using Algorithm 1. Note that the enumeration output of Algorithm 1 is in lexicographic order. Our second observation is that we can alternate between low-degree and high-degree values in $\mathcal{L}$ in the following way: we spend $O(1)$ time applying Algorithm 1 on v_1(left pointer) and then spend $O(1)$ time applying Algorithm 1 on v_n(right pointer). We keep alternating until either v_1 or v_n finishes and then move on to the next (or previous respectively) value in $\mathcal{L}$.

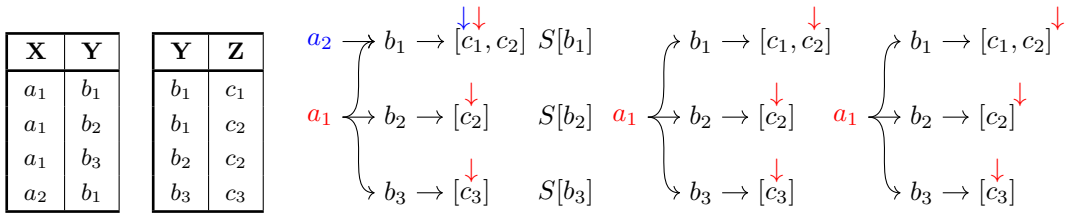
▶ **Lemma 8.** *For the query Q_{path} and an instance D, Algorithm 2 enumerates $Q_{path}(D)$ with delay $\delta = O(|D|^2/|\texttt{OUT}_\bowtie|)$ and $S_e = O(|D|)$.*

Proof. Let δ denote the degree d_i when left = right. First, we claim that the delay will be $O(\delta)$. Indeed, the left side of the algorithm will emit a result every $O(\delta)$ time. Because of the interleaving, the enumeration algorithm will also have the same guarantee.

By construction, when the left and right pointers meet, the algorithm has spent equal amount of time processing light ($d_i \leq \delta$) and heavy ($d_i > \delta$) values of $\mathcal{L}$. Let $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ denote the set of values that are processed by the left and right pointers respectively. When the two pointers meet, we must have that full join result of the valuations enumerated by right (i.e $\sum_{v \in \mathcal{L}_{\text{right}}} |R(v,y) \bowtie S(y,z)|$) is $J_h = \Omega(|\texttt{OUT}_\bowtie|/2)$. On the other hand, $J_h \leq |D|^2/\delta$ since there are most $|D|/\delta$ heavy values in $\mathcal{L}$, and each heavy value can produce at most $|D|$ tuples for the full join. Combining the two inequalities gives us the claimed delay guarantee. ◀

▶ **Example 9.** Consider relations R and S as shown in Figure 2a and Figure 2b.

Figure 2c shows the sorted valuations a_2 and a_1 by their degree and the valuations for Z as sorted lists $S[b_1], S[b_2]$ and $S[b_3]$. For both a_1 and a_2, the pointers point to the head of the lists. We will now show how ListMerge$(S[b_1], S[b_2], S[b_3])$ is executed for a_1. Since there



(a) Table R **(b)** Table S **(c)** output (a_1, c_1) **(d)** output (a_1, c_2) **(e)** output(a_1, c_3)

Figure 2 Example for two path query enumeration

are three sorted lists that need to be merged, the algorithm finds the smallest valuation across the three lists. c_1 is the smallest valuation and the algorithm outputs (a_1, c_1). Then, we need to increment pointers of all lists which are pointing to c_1 ($S[b_1]$ is the only list containing c_1). Figure 2d shows the state of pointers after this step. The pointer for $S[b_1]$ points to c_2 and all other pointers are still pointing to the head of the lists. Next, we continue the list merging by again finding the smallest valuation from each list. Both $S[b_1]$ and $S[b_2]$ pointers are pointing to c_2 and the algorithm outputs (a_1, c_2). The pointers for both $S[b_1]$ and $S[b_2]$ are incremented and the enumeration for both the lists is complete as shown in Figure 2e. In the last step, only $S[b_3]$ list remains and we output (a_1, c_3) and increment the pointer for $S[b_3]$. All pointers are now past the end of the lists and the enumeration is now complete.

The reader should note that the output of all subqueries is computed by going over their entire join results (and deduplicating smartly). However, this is not the only method to compute the join-project result for heavy valuations. Indeed, as we will see later, one can compute the join using fast matrix multiplication (such as [2]) to further improve the delay guarantees based on the database instance.

Observe also that the delay of $\delta = O(|D|^2/|\mathtt{OUT}_{\bowtie}|)$ is only an upper bound. Depending on the skew present in the database instance, it is possible that Algorithm 2 achieves much better delay guarantees in practice. We conclude this subsection with a concrete example to illustrate this.

▶ **Example 10.** Consider a relation $R(x, y)$ of size $O(N)$ that contains values $v_1, \ldots, v_N$ for attribute x. Suppose that each of $v_1, \ldots, v_{N-1}$ have degree exactly 1, and each one is connected to a unique value of y. Also, v_N has degree $N-1$ and is connected to all $N-1$ values of y. Suppose we want to compute Q_{path}. It is easy to see that $\mathtt{OUT}_{\bowtie} = \Theta(N)$. Thus, applying the bound of $\delta = O(N^2/|\mathtt{OUT}_{\bowtie}|)$ gives us $O(N)$ delay. However, Algorithm 2 will achieve a delay guarantee of $O(1)$. This is because all of $v_1, \ldots, v_{N-1}$ are processed by the left pointer in $O(1)$ delay as they produce exactly one output result, while the right pointer processes v_N on-the-fly in $O(N)$ time.

4.2 Proof of Main Theorem

We now generalize Algorithm 2 for any star query. At a high level, we will decompose the join query $\pi_{x_1,\ldots,x_k}(Q_k^*)$ into a union of $k+1$ subqueries whose output is a partition of the result of original query. These subqueries will be generated based on whether a value for some x_i is *light* or not. We will show if any of the values for x_i is light, the enumeration delay is small. The $k+1$-th subquery will contain heavy values for all attributes. Our key idea again is to *interleave* the join computation of the "heavy" subquery with generating the

output from any of the remaining subqueries by performing constant amount of work and alternate between the heavy subquery and the remaining light subqueries.

Preprocessing Phase. Assume all relations are reduced without dangling tuples, which can be achieved in linear time. The full join size $|\texttt{OUT}_\bowtie|$ can also computed in linear time.

Set $\Delta = (\frac{2|D|^k}{|\texttt{OUT}_\bowtie|})^{\frac{1}{k-1}}$. For each relation R_i, a value v for attribute x_i is *heavy* is its degree (i.e $|\pi_y \sigma_{x_i=v} R(x_i, y)|$) is greater than Δ, and *light* otherwise. Moreover, a tuple $t \in R_i$ is identified as heavy or light depending on whether $\pi_{x_i}(t)$ is heavy or light. In this way, each relation R is divided into two relations R^h and R^ℓ, containing heavy and light tuples respectively.

The original query can be decomposed into subqueries of the following form: $\pi_{x_1,x_2,\cdots,x_k}(R_1^? \bowtie R_2^? \bowtie \cdots \bowtie R_k^?)$ where ? can be either h, ℓ or $\star$. Here, $R_i^\star$ simply denotes the original relation R_i. However, care must be taken to generate the subqueries so that there is no overlap between the output of any subquery. In order to do so, we create k subqueries of the form

$$Q_i = \pi_{x_1,\ldots,x_k}(R_1^h \bowtie \cdots \bowtie R_{i-1}^h \bowtie R_i^\ell \bowtie R_{i+1}^\star \bowtie \cdots \bowtie R_k^\star)$$

In subquery Q_i, relation R_i has superscript ℓ, all relations $R_1, \ldots, R_{i-1}$ have superscript h and relations $R_{i+1}, \ldots, R_k$ have superscript $\star$. The $k+1$-th query with all ? as h is denoted by Q_H. Note that the results of all these subqueries are disjoint.

Enumeration Phase. We next describe how enumeration is performed. The key idea is the following: We will show that for $Q_L = Q_1 \cup \cdots \cup Q_k$, we can enumerate the result in delay $O(\Delta)$. Since Q_H contains all heavy valuations from all relations, we compute its join *on-the-fly* by alternating between some subquery in Q_L and Q_H. This will ensure that we can give some output to user with delay guarantees and also make progress on compute the full join of Q_H. The key observation is to make sure that while we compute Q_H, we do not run out of the output from Q_L, i.e, $|Q_H| \leq |Q_L|$, similar to the algorithm for two-path query. Moreover, it can be easily shown that each output $t = (x_1, x_2, \cdots, x_k)$ is emitted by exactly one query.

Next, we introduce the algorithm that enumerates output for any specific valuation v of attribute x_i, which is described in Lemma 11. This algorithm can be viewed as another instantiation of Algorithm 1.

▶ **Lemma 11.** *Consider an arbitrary value $v \in \mathbf{dom}(x_i)$ with degree d in relation $R_i(x_i, y)$. Then, its query result $\times_{x_1,x_2,\cdots,x_k} \sigma_{x_i=v} R_1(x_1, y) \bowtie R_2(x_2, y) \bowtie \cdots \bowtie R_k(x_k, y)$ can be enumerated in order with $O(d)$ delay guarantee.*

Proof. Let u be an arbitrary neighbor of v_i. Each u is associated with a list of valuations over attributes $(x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_k)$, which is a cartesian product of $k-1$ sub-lists $\sigma_{y=u} R_j(x_j, y)$. Note that such a list is not materialized as that for 2-path query, but this is not a problem.

We next define the enumeration algorithm $\mathcal{A}_u$ for each neighbor $u \in \pi_y \sigma_{x_i=v} R_i(x_i, y)$, with lexicographical ordering of attributes $(x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_k)$. Note that elements in each list $\pi_{x_j} \sigma_{y=u} R_j(x_j, y)$ can be enumerated with $O(1)$ delay. Then, $\mathcal{A}_u$ enumerates all results in $\times_{j \neq i : j \in \{1,2\cdots,k\}} \sigma_{y=u} R_j(x_j, y)$ by $k-1$ level of nested loops in ordering of $(x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_k)$, which has $O(k-1) = O(1)$ delay. After applying Algorithm 1, we can obtain enumeration algorithm that enumerates the union of query results over all neighbors with $O(d)$ delay guarantee. ◀

Directly implied by Lemma 11, the result of any subquery in Q_L can be enumerated with $\delta = O(\Delta)$. Observe that subquery Q_H can be computed in time $|D| \cdot (\frac{|D|}{\Delta})^{k-1} = \frac{1}{2}|\texttt{OUT}_\bowtie|$ in

the worst case. Thus, Q_H can be computed on the fly without running out of output from Q_L. This is because $|Q_L| = |\texttt{OUT}_\bowtie| - |Q_H| \geq \frac{1}{2}|\texttt{OUT}_\bowtie|$. We can now apply Lemma 4 where $\mathcal{A}$ is the full join computation of Q_H, $\mathcal{A}'$ is the enumeration algorithm applied to Q_L. Our choice of Δ ensures that the join output is $T = T' = \frac{1}{2}|\texttt{OUT}_\bowtie|$ in the worst-case (and $T \leq T'$ in general). Overall, we have completed the proof for Theorem 7.

4.3 Interleaving with Join Computation

Theorem 7 obtains poor delay guarantees when the full join size $|\texttt{OUT}_\bowtie|$ approximates the input size $O(|D|)$. In this section, we present an alternate algorithm that provides good delay guarantees in this case.

The algorithm is an instantiation of Lemma 5 on the star query, which degenerates to compute as many distinct output results as possible in limited preprocessing time. An observation is that for each valuation u of attribute y, the cartesian product $\times_{i\in\{1,2,\cdots,k\}}\pi_{x_i}\sigma_{y=u}R_i(x_i,y)$ is a subset of output results without duplication. Thus, this subset of output result is readily available since no deduplication needs to be performed. Similarly, after all relations are reduced, it is also guaranteed that each valuation of attribute x_i of relation R_i generates at least one output result. Thus, $\max_{i=1}^k |\mathbf{dom}(x_i)|$ results are also readily available that do not require deduplication. We define J as the larger of the two quantities, i.e, $J = \max\left\{\max_{i=1}^k |\mathbf{dom}(x_i)|, \max_{u\in\mathbf{dom}(y)} \prod_{i=1}^k |\sigma_{y=u}R_i(x_i,y)|\}\right\}$. Together with these observations, we can achieve the following theorem.

▶ **Theorem 12.** *Consider star query $\pi_{x_1,\ldots,x_k}(Q_k^*)$ and an input database instance D. There exists an algorithm with preprocessing time $O(|D|)$ and space $O(|D|)$, such that $\pi_{x_1,\ldots,x_k}(Q_k^*)$ can be enumerated with delay $\delta = O\left(\frac{|\texttt{OUT}_\bowtie|}{|\texttt{OUT}_\pi|^{1/k}}\right)$ and space $S_e = O(|D|)$*

In the above theorem, we obtain delay guarantees that depend on both the full join result $\texttt{OUT}_\bowtie$ and the projection output size $\texttt{OUT}_\pi$. However, one does not need to know $\texttt{OUT}_\bowtie$ or $\texttt{OUT}_\pi$ to apply the result. We first compare the result with Theorem 7. First, observe that Theorem 12 requires $O(|D|)$ preprocessing instead of $O(|D|\log|D|)$ required by Theorem 7. Second, the delay guarantee provided by Theorem 12 can be better than Theorem 7. This happens when $O(\frac{|\texttt{OUT}_\bowtie|}{|\texttt{OUT}_\pi|^{1/k}}) \leq O(\frac{|D|^{k/k-1}}{|\texttt{OUT}_\bowtie|^{1/k-1}})$, which can be further simplified to the condition $|\texttt{OUT}_\bowtie| \leq |D| \cdot O(|\texttt{OUT}_\pi|^{(k-1)/k^2})$. Theorem 3 obtains a linear delay guarantee when $\epsilon = 0$. This result is recovered by Theorem 12 due to the inequality $|\texttt{OUT}_\bowtie|/|\texttt{OUT}_\pi|^{1/k} \leq |D|$ for $\pi_{x_1,\ldots,x_k}(Q_k^*)$ (see [2]).

We now proceed to describe the algorithm. First, we compute all the statistics for computing J in linear time. If $J = |\mathbf{dom}(x_j)|$ for some integer $j \in \{1,2,\cdots,k\}$, we just materialize one result for each valuation of x_j. Otherwise, $J = \prod_{i=1}^k |\sigma_{y=u}R_i(x_i,y)|$ for some valuation u in attribute y. Note that we do need to explicitly materialize the cartesian product but only need to store the tuples in $\bigcup_{i\in\{1,2,\cdots,k\}}\sigma_{y=u}R_i(x_i,y)$. As mentioned, each output in $\times_{i=1}^k(\pi_y\sigma_{y=u}R_i(x_i,y))$ can be enumerated with $O(1)$ delay. This preprocessing phase takes $O(|D|)$ time and $O(|D|)$ space. We invoke Lemma 5 at last. The final observation is to relate the size of J in terms of $|\texttt{OUT}_\pi|$. Note that $|\texttt{OUT}_\pi| \leq \Pi_{i\in[k]}|\mathbf{dom}(x_i)|$ which implies that $\max_{i\in[k]}|\mathbf{dom}(x_i)| \geq |\texttt{OUT}_\pi|^{1/k}$. Thus, it holds that $|J| \geq |\texttt{OUT}_\pi|^{1/k}$ which gives us the desired delay guarantee.

4.4 Fast Matrix Multiplication

Theorem 7 tells us that there exists no tradeoff between preprocessing time and delay guarantees as claimed by Theorem 3 for $\pi_{x_1,\ldots,x_k}(Q_k^*)$. In this section, we will show how fast matrix multiplication can be used to obtain a tradeoff between preprocessing time and delay that is better than Theorem 3 for some values of delay δ.

▶ **Theorem 13.** *Consider $Q_{path} = \pi_{x,z}(R(x,y) \bowtie S(y,z))$. Assume that the $\omega = 2$ for matrix multiplication. Then, we can enumerate the result of Q_{path} with $T_p = O(|D|^2/\delta^2)$ and delay $O(\delta)$, for $1 \leq \delta \leq |D|^{1/3}$.*

Proof. Our key insight is to use fast matrix multiplication for preprocessing. Let δ be the degree threshold for deciding whether a valuation is heavy or light. We can partition the original query into the following subqueries: $\pi_{x,z}(R_1(x^?, y^?) \bowtie R_2(y^?, z^?))$ where ? can be either h, ℓ or $\star$. The input tuples can also be partitioned into four different cases (which can be done in linear time since δ is fixed). We handle each subquery separately.

- x has $? = \ell$ and z has $? = \star$. In this case, we can just invoke ListMerge(v_i) for each valuation v_i in attribute x and enumerate the output.
- x has $? = h$ and z has $? = \ell$. In this case, we can invoke ListMerge(v_i) for each valuation v_i in attribute z and enumerate the output. Note that there is no overlap of output between this case and the previous case.
- both x, z have $? = h$. We compute the output of $\pi_{x,z} R(x^h, y^?) \bowtie S(y^?, z^h)$ in preprocessing phase and obtain $O(1)$-delay enumeration.
  - y has $? = \ell$. We compute the full join $R(x^h, y^\ell) \bowtie S(y^\ell, z^h)$ and materialize all distinct output results, which takes $O(|D| \cdot \delta)$ time.
  - y has $? = h$. There are at most $|D|/\delta$ valuations in all attributes. We now have a square matrix multiplication instance for all dimensions are of size $O(|D|/\delta)$. Using Lemma 2 with $\omega = 2$, we can evaluate the join in time $O((|D|/\delta)^2)$.

Overall, the preprocessing time is $T_p = O((|D|/\delta)^2 + |D| \cdot \delta)$. The matrix multiplication term dominates whenever $\delta \leq O(|D|^{1/3})$ which gives us the desired time-delay tradeoff. ◀

For the current best value of $\omega = 2.373$, we get the tradeoff as $T_p = O((|D|/\delta)^{2.37})$ and a delay guarantee of $O(\delta), |D|^{0.15} < \delta \leq |D|^{0.40}$. If we choose $\delta = |D|^{0.40}$, the preprocessing time is $T_p = O(|D|^{1.422})$. In contrast, Theorem 3 requires a preprocessing time of $T_p = O(|D|^{1.6})$, which is suboptimal compared to the above theorem. On the other hand, since $T_p = O(|D|^{1.422})$, we can assume that $|\texttt{OUT}_\bowtie| = \Omega(|D|^{1.422})$, otherwise one can simply compute the full join, deduplicate and get constant delay enumeration. Applying Theorem 7 with $|\texttt{OUT}_\bowtie| = \Omega(|D|^{1.422})$ tells us that we can obtain delay as $O(|D|^2/|\texttt{OUT}_\bowtie|) \leq O(|D|^{0.58})$. Thus, we can now offer the user both the choices and the user can make the decision on which enumeration algorithm to use.

We can also extend the algorithm to $\pi_{x_1,\ldots,x_k}(Q_k^*)$ as well. When any of the x_i valuation is light, we can partition the relations into k subqueries as described for star queries and apply Lemma 6 allowing for $O(\delta)$ enumeration. The $k+1th$ query is $Q_H = \pi_{x_1,\ldots x_k}(R_1^h(x_1,y) \bowtie R_2^h(x_2,y) \bowtie \ldots \bowtie R_k^h(x_k,y))$ which can be evaluated in the preprocessing step in time $O(|D| \cdot \delta^{k-1} + (|D|/\delta)^k)$ [7] (assuming $\omega = 2 + o(1)$). Thus, we get the tradeoff as $T_p = O((|D|/\delta)^k)$ and delay $O(\delta)$ for $1 \leq \delta \leq |D|^{1/(k+1)}$ for projection over star queries.

5 Left-Deep Hierarchical Queries

In this section, we will apply our techniques to another subset of hierarchical queries, which we call *left-deep*. A left-deep hierarchical query is of the following form:

$$Q^k_{\text{leftdeep}} = R_1(w_1, x_1) \bowtie R_2(w_2, x_1, x_2) \bowtie \ldots \bowtie R_{k-1}(w_{k-1}, x_1, \ldots, x_{k-1}) \bowtie R_k(w_k, x_1, \ldots, x_{k-1})$$

It is easy to see that Q^k_{leftdeep} is a hierarchical query for any $k \geq 1$. Note that for $k = 2$, we get the two-path query. For $k = 3$, we get $R(w_1, x_1) \bowtie S(w_2, x_1, x_2) \bowtie T(w_3, x_1, x_2)$. We will be interested in computing the query $\pi_{w_1,\ldots,w_k}(Q^k_{\text{leftdeep}})$, where we project out all the join variables. We show that the following result holds:

▶ **Theorem 14.** *Consider the query $\pi_{w_1,\ldots,w_k}(Q^k_{leftdeep})$ and any input database D. Then, there exists an algorithm that enumerates the query after preprocessing time $T_p = O(|D| \log |D|)$ with delay $O(|D|^k / |\texttt{OUT}_\bowtie|)$.*

In the above theorem, $\texttt{OUT}_\bowtie$ is the full join result of the query without projections, Q^k_{leftdeep}. The AGM exponent for Q^k_{leftdeep} is $\rho^* = k$. Observe that Theorem 14 is of interest when $|\texttt{OUT}_\bowtie| \geq |D|^{k-1}$ to ensure that the delay is smaller than $O(|D|)$. When the condition $|\texttt{OUT}_\bowtie| \geq |D|^{k-1}$ holds, the delay obtained by Theorem 14 is also better than the one given by the tradeoff in Theorem 3. In the worst case when $|\texttt{OUT}_\bowtie| = \Theta(|D|^k)$, we can achieve constant delay enumeration after (almost) linear preprocessing time, compared to Theorem 3 that would require $\Theta(|D|^k)$ preprocessing time to achieve the same delay.

6 Path Queries

In this section, we will study path queries. In particular, we will present an algorithm that enumerates the result of the query $\pi_{x_1,x_{k+1}}(P_k)$, i.e., the CQ that projects the two endpoints of a path query of length k. Recall that for $k \geq 3$, P_k is not a hierarchical query, and hence the tradeoff from [12] does not apply. Instead, we show that the following tradeoff holds.

▶ **Theorem 15.** *Consider the query $\pi_{x_1,x_{k+1}}(P_k)$ with $k \geq 2$. For any input instance D and parameter $\epsilon \in [0, 1)$ there exists an algorithm that enumerates the query with preprocessing time (and space) $T_p = O(|D|^{2-\epsilon/(k-1)})$ and delay $O(|D|^\epsilon \cdot \log |D|)$.*

We should note here that for $\epsilon = 1$, we can obtain a delay $O(|D|)$ using only linear preprocessing time $O(|D|)$, while for $\epsilon \to 1$ the above theorem would give preprocessing time $O(|D|^{2-1/(k-1)})$. Hence, for $k \geq 3$, we observe a discontinuity in the time-delay tradeoff.

A second observation following from Theorem 15 is that as $k \to \infty$, the tradeoff collapses to only two extremal points: one where we get constant delay with $T_p = O(|D|^2)$, and the other where we get linear delay with $T_p = O(|D|)$.

7 Related Work

We overview prior work on static query evaluation for acyclic join-project queries. The result of any acyclic conjunctive query can be enumerated with constant delay after linear-time preprocessing if and only if it is free-connex [3]. This is based on the conjecture that Boolean multiplication of $n \times n$ matrices cannot be done in $O(n^2)$ time. Acyclicity itself is necessary for having constant delay enumeration: A conjunctive query admits constant delay enumeration after linear-time preprocessing if and only if it is free-connex acyclic [6]. This is based on a stronger hypothesis that the existence of a triangle in a hypergraph of

n vertices cannot be tested in time $O(n^2)$ and that for any k, testing the presence of a k-dimensional tetrahedron cannot be tested in linear time. We refer the reader to an overview of pre-2015 for problems and progress related to constant delay enumeration [17]. Prior work also exhibits a dependency between the space and enumeration delay for conjunctive queries with access patterns [8]. It constructs a succinct representation of the query result that allows for enumeration of tuples over some variables under value bindings for all other variables. As noted by [12], it does not support enumeration for queries with free variables, which is also its main contribution. Our work demonstrates that for a subset of hierarchical queries, the tradeoff shown in [12] is not optimal. Our work introduces fundamentally new ideas that may be useful in improving the tradeoff for arbitrary hierarchical queries and enumeration of UCQs. There has also been some experimental work by the database community on problems related to enumerating join-project query results efficiently but without any formal delay guarantees. Seminal work [20, 19, 21] has studied how compressed representations can be create apriori that allow for faster enumeration of query results. Also related is the problem of dynamic evaluation of hierarchical queries. Recent work [11, 12, 4, 5] has studied the tradeoff between amortized update time and delay guarantees.

8 Conclusion and Open Problems

In this paper, we studied the problem of enumerating query results for an important subset of CQs with projections, namely star and path queries. We presented data-dependent algorithms that improve upon existing results by achieving non-trivial delay guarantees in (almost) linear preprocessing time. Our results are based on the idea of interleaving join query computation to achieve meaningful delay guarantees. Further, we showed how non-combinatorial algorithms (fast matrix multiplication) can be used for faster preprocessing to improve the tradeoff between preprocessing time and delay. We also presented the first results on time-delay tradeoffs for a subset of non-hierarchical queries for the class of path queries. The results in this paper take the first step towards introducing techniques that may be of independent interest for enumerating UCQs and achieving better instance specific delay guarantees for CQs. Our results also open several new tantalizing questions that open up possible directions for future work.

Extending to hierarchical queries. The first main question is to understand whether our techniques can be extended to arbitrary hierarchical queries. Theorem 14 suggests that this may be possible, depending on the output size of the query.

More preprocessing time for star queries. The second major open question is to show whether Theorem 7 can benefit from more preprocessing time to achieve lower delay guarantees. For instance, if we can afford the algorithm preprocessing time $T_p = O(|\mathtt{OUT}_{\bowtie}|^{1-\epsilon} + |D|)$ time, can we expect to get delay $\delta = O(|D|^{\epsilon})$ for all $\epsilon \in (0, 1)$?

Sublinear delay guarantees for two-path query. It is not known whether we can achieve sublinear delay guarantee in linear preprocessing time for Q_{path} query. This question is equivalent to the following problem: for what values of $|\mathtt{OUT}_{\pi}|$ can Q_{path} be evaluated in linear time. If $|\mathtt{OUT}_{\pi}| = |D|^{\epsilon}$, then the best known algorithms can evaluate Q_{path} in time $O(|D|^{1+\epsilon/3})$ (using fast matrix multiplication) [7] but this is still superlinear.

Space-delay bounds. The last question is to study the tradeoff between space vs delay for arbitrary hierarchical queries and path queries. Using some of our techniques, it may be possible to smartly materialize a certain subset of joins that could be used to achieve delay guarantees by interleaving with join computation.

References

1. M. Abo Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 429–444, 2017.
2. R. R. Amossen and R. Pagh. Faster join-projects and sparse matrix multiplications. In *Proceedings of the 12th International Conference on Database Theory*, pages 121–126. ACM, 2009.
3. 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.
4. 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.
5. 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.
6. J. Brault-Baron. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. PhD thesis, Université de Caen, 2013.
7. S. Deep, X. Hu, and P. Koutris. Fast join project query evaluation using matrix multiplication. *arXiv preprint arXiv:2002.12459*, 2020.
8. 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.
9. F. L. Gall and F. Urrutia. Improved rectangular matrix multiplication using powers of the coppersmith-winograd tensor. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1029–1046. SIAM, 2018.
10. J. E. Hopcroft, J. D. Ullman, and A. Aho. The design and analysis of computer algorithms, 1975.
11. A. Kara, H. Q. Ngo, M. Nikolic, D. Olteanu, and H. Zhang. Counting triangles under updates in worst-case optimal time. In *22nd International Conference on Database Theory*, 2019.
12. A. Kara, M. Nikolic, D. Olteanu, and H. Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. In *arXiv preprint arXiv:1907.01988*, 2019.
13. 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.
14. 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.
15. D. Olteanu and M. Schleich. Factorized databases. *ACM SIGMOD Record*, 45(2):5–16, 2016.
16. L. Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Record*, 44(1):10–17, 2015.
17. L. Segoufin. Constant delay enumeration for conjunctive queries. *ACM SIGMOD Record*, 44(1):10–17, 2015.
18. D. Suciu, D. Olteanu, C. Ré, and C. Koch. Probabilistic databases, synthesis lectures on data management. *Morgan & Claypool*, 2011.
19. K. Xirogiannopoulos and A. Deshpande. Extracting and analyzing hidden graphs from relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 897–912. ACM, 2017.
20. K. Xirogiannopoulos, U. Khurana, and A. Deshpande. Graphgen: Exploring interesting graphs in relational data. *Proceedings of the VLDB Endowment*, 8(12):2032–2035, 2015.
21. K. Xirogiannopoulos, V. Srinivas, and A. Deshpande. Graphgen: Adaptive graph processing using relational databases. In *Proceedings of the Fifth International Workshop on Graph*

Data-management Experiences & Systems, GRADES'17, pages 9:1–9:7, New York, NY, USA, 2017. ACM.

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

A Algorithm for Lemma 5

Algorithm 3 describes the detailed algorithm for Lemma 5.

Algorithm 3: DEDUPLICATE($J, \mathcal{A}$)

Input : Materialized output J, Algorithm $\mathcal{A}$
Output : Deduplicated result of $\mathcal{A}$ and J

$\delta \leftarrow T/J$, ptr $\leftarrow \&J$, dedup $\leftarrow \&J$
$H \leftarrow \emptyset$ /* empty hashset */
while *ptr* $\neq$ *null* **do**
  output $*$ptr /* output result from J to maintain delay guarantee */
  $*$ptr $\leftarrow *$ptr $+ 1$, counter $\leftarrow 0$
  while counter $\leq O(\delta)$ **do**
    if $\mathcal{A}$ *has not completed* **then**
      Execute/Resume $\mathcal{A}$ for $O(1)$ time and insert any output generated into H;
      counter $\leftarrow$ counter $+ O(1)$;
    else
      if dedup $\neq \&J.end()$ /* remove all J tuples from H */
      then
        Delete $*$dedup from H if it exists;
        dedup $\leftarrow$ dedup $+ 1$, counter $\leftarrow$ counter $+ O(1)$;
foreach $t \in H$ **do**
  output t

B Proofs

▶ **Theorem 14.** *Consider the query* $\pi_{w_1,\ldots,w_k}(Q^k_{leftdeep})$ *and any input database* D. *Then, there exists an algorithm that enumerates the query after preprocessing time* $T_p = O(|D| \log |D|)$ *with delay* $O(|D|^k/|\mathtt{OUT}_{\bowtie}|)$.

Proof. The algorithm is based on LISTMERGE subroutine from Lemma 6. We distinguish two cases based on the degree of valuations of variable w_k. If some valuation of w_k (say v) is light (degree is at most δ), then we can enumerate the join result with delay $O(\delta)$. Since there are at most δ tuples $U = \sigma_{w_k=v}R_k$, each $u \in U$ is associated with a list of valuations over attributes $(w_1, w_2, \ldots, w_{k-1})$, which is a cartesian product of $k-1$ sub-lists $\pi_{w_i}\sigma_{x_1=u[x_1],\ldots,x_i=u[x_i]}R_i$. The elements of each list can be enumerated in $O(1)$ delay in lexicographic order. Thus, we only need to merge the δ sublists which can be accomplished in $O(\delta)$ time using Lemma 6.

We now describe how to process all w_k valuations that are heavy. The key observation here is that the full-join result with no projections for this case can be upper bounded by $|D|^k/\delta$ since there are at most $|D|/\delta$ heavy w_k valuations. The full-join result can be computed and deduplicated in time $O(|D|^k/\delta)$ using any worst-case optimal join algorithm. We can now apply Lemma 5 where $\mathcal{A}'$ is the list-merging algorithm for the light case, $\mathcal{A}$ is the worst-case optimal join algorithm for the heavy case and $\delta = 2 \cdot |D|^k/|\mathtt{OUT}_{\bowtie}|$. Here we should note that $|\mathtt{OUT}_{\bowtie}|$ can be computed during the preprocessing phase that also sorts and indexes the relations, and requires $O(|D| \log |D|)$ time. ◀

▶ **Theorem 15.** *Consider the query $\pi_{x_1,x_{k+1}}(P_k)$ with $k \geq 2$. For any input instance D and parameter $\epsilon \in [0,1)$ there exists an algorithm that enumerates the query with preprocessing time (and space) $T_p = O(|D|^{2-\epsilon/(k-1)})$ and delay $O(|D|^{\epsilon} \cdot \log |D|)$.*

Proof. Let Δ be a parameter that we will fix later. In the preprocessing phase, we first perform a full reducer pass to remove dangling tuples, and then create for each relation $R_i(x_i, x_{i+1})$ a hash map with key x_i, and all its corresponding x_{i+1} values sorted for each key entry. (We also store the degree of each value.) Next, for every $i = 1, \ldots, k$, and every heavy value a of x_i in R_i (with degree $> \Delta$), we compute the query $\pi_{x_{k+1}}(R_i(a, x_{i+1}) \bowtie \cdots \bowtie R_k(x_k, x_{k+1}))$, and store its result sorted in a hash map with key a. Note that each such query can be computed in time $O(|D|)$ through a sequence of semijoins and projections. Since there are at most $|D|/\Delta$ heavy values for each x_i, the total running time (and space necessary) for this step is $O(|D|^2/\Delta)$.

We will present the enumeration algorithm using induction. In particular, we will show that for each $i = k, \ldots, 1$ and for every value a of x_i, the subquery $\pi_{x_{k+1}}(R_i(a, x_{i+1}) \bowtie \cdots \bowtie R_k(x_k, x_{k+1}))$ can be enumerated (using the same order) with delay $O(\Delta^{k-i} \cdot \log |D|)$. This implies that our target path query can be enumerated with delay $O(\Delta^{k-1} \cdot \log |D|)$, by simply iterating through all values of x_1 in R_1. Finally we can obtain the desired result by choosing $\Delta = |D|^{\epsilon/(k-1)}$.

Indeed, for the base case ($i = k$) it is trivial to see that we can enumerate $\pi_{x_{k+1}}(R_k(a, x_{k+1}))$ in constant time $O(1)$ using the stored hash map. For the inductive step, consider some i, and a value a for x_i in R_i. If the value a is heavy, then we can enumerate all the x_{k+1}'s with constant delay by probing the hash map we computed during the preprocessing phase. If the value is light, then there are at most Δ values of x_{i+1}. For each such value b, the inductive step provides an algorithm that enumerates all x_{k+1} with delay $O(\Delta^{k-i-1} \cdot \log |D|)$ (the log factor appears from sorting the lists). Observe that the order across all b's will be the same. Thus, we can apply Lemma 6 to obtain that we can enumerate the union of the results with delay $O(\Delta \cdot \Delta^{k-i-1} \cdot \log |D|) = O(\Delta^{k-i} \cdot \log |D|)$. ◀