arXiv:2101.03712v2 [cs.DB] 2 Feb 2021

Enumeration Algorithms for Conjunctive Queries with Projection

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

Xiao Hu
Department of Computer Sciences, Duke University, Durham, North Carolina, USA

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

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, for star queries, we design combinatorial algorithms that provide instance-specific delay guarantees in 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 new 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 also be interpreted as sparse, output-sensitive matrix multiplication with delay guarantees.

2012 ACM Subject Classification Theory of computation → Database theory

Keywords and phrases Query result enumeration, joins

Digital Object Identifier 10.4230/LIPIcs.ICDT.2021.3

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

Acknowledgements We would like to thank the anonymous reviewers for their careful reading and valuable comments that contributed greatly in improving the manuscript.

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 [32, 20, 19]. 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.

© S. Deep, X. Hu and P. Koutris;
licensed under Creative Commons License CC-BY 4.0
24th International Conference on Database Theory (ICDT 2021).
Editors: Ke Yi and Zhewei Wei; Article No. 3; pp. 3:1–3:24
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 achieve linear delay after linear preprocessing time [3]. Prior work that uses structural decomposition methods [14] generalized these results to arbitrary CQs with free variables and showed that the projected solutions can be enumerated with $O(|D|^{\texttt{fhw}})$ delay. Moreover, a dichotomy about the classes of conjunctive queries with fixed arities where such answers can be computed with polynomial delay (WPD) is also shown. 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* [13] of the query. We should note here that we can always compute and materialize the result of the query during preprocessing to achieve constant delay enumeration but at the cost of using exponential amount of space in general.

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: $Q_{\texttt{two-path}} = \pi_{x,z}(R(x,y) \bowtie S(y,z))$, 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 boolean matrix multiplication exponent is $\omega = 2$. 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, seminal work by Kara et al. [17] 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 $Q_{\texttt{two-path}}$, the main result of of [17] 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 [9] for an overview). The most common star query seen in practice is $Q_{\texttt{two-path}}$. 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 left-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 $Q_{\texttt{two-path}}$, 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.

[1] Hierarchical CQs are a strict subset of acyclic CQs.

Queries	Preprocessing	Delay	Source
Arbitrary acyclic CQ	$O(\|D\|)$	$O(\|D\|)$	[3]
Free-connex CQ (projections)	$O(\|D\|)$	$O(1)$	[3]
Full CQ	$O(\|D\|^{\texttt{fhw}})$	$O(1)$	[21]
Full CQ	$O(\|D\|^{\texttt{subw}} \log \|D\|)$	$O(1)$	[1]
Hierarchical CQ (with projections)	$O(\|D\|^{1+(\texttt{w}-1)\epsilon})$	$O(\|D\|^{1-\epsilon})$ $\epsilon \in [0,1]$	[17]
Star query with k relations (with projections)	$O(\|D\|)$	$O(\frac{\|D\|^{k/(k-1)}}{\|\texttt{OUT}_{\bowtie}\|^{1/(k-1)}})$	this paper
Path query with k relations (with projections)	$O(\|D\|^{2-\epsilon/(k-1)})$	$O(\|D\|^{\epsilon})$, $\epsilon \in [0,1)$	this paper
Left-deep hierarchical CQ (with projections)	$O(\|D\|)$	$O(\|D\|^k/\|\texttt{OUT}_{\bowtie}\|)$	this paper
Two path query (with projections)	$O(\|D\|^{\omega\cdot\epsilon})$	$O(\|D\|^{1-\epsilon})$, $\epsilon \in [\frac{2}{\omega+1}, 1]$	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. For each class of query, the total running time is $O(\min\{|D|\cdot|\texttt{OUT}_{\pi}|, |D|^{\texttt{subw}} \log |D| + |\texttt{OUT}_{\pi}|\})$ where $|\texttt{OUT}_{\pi}|$ denotes the size of the query result. See the related work section for more discussion on best running times for two path and star queries.

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

At this point, the reader may wonder about the improvement obtained from the above result. [17] implies that with preprocessing time $O(|D|)$, the delay guarantee in the worst-case is $O(|D|)$. This raises the question whether the delay from Theorem 1 is truly an algorithmic improvement rather than an improved analysis of [17]. We answer the question positively. Specifically, we show that there exists a database instance where the delay obtained from Theorem 1 is a polynomial improvement over the actual guarantee [17] and not just the worst-case. When the preprocessing time is linear, the delay implied by our result 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 [17], which would require nearly $O(|D|^2)$ preprocessing time to achieve the same guarantee. 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 [17] in the worst-case; we will show that we can always get at least as good of a tradeoff point as the one in [17]. 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 in Section 4) that achieves output-dependent delay guarantees for star queries after linear preprocessing

[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 using constant delay enumeration for queries without projections [22].

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 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 [17] for the class of hierarchical queries.

2. We introduce the novel idea of *interleaving* join query computation in the context of enumeration algorithms which forms the foundation for our algorithms, and may be of independent interest. 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 (Lemma 4 and Lemma 5) in Section 3.
3. We show how fast matrix multiplication can be used to obtain a tradeoff between preprocessing time and delay that further improves upon the tradeoff in [17]. We also present an algorithm for left-deep hierarchical queries with linear preprocessing time and output-dependent delay guarantees (Section 5).
4. Finally, we present new results on preprocessing time-delay tradeoffs for a non-hierarchical query with projections, for the class of path queries (Section 6). 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 $O(|D|^\epsilon)$ with preprocessing time $O(|D|^{2-\epsilon/(k-1)})$ for any $\epsilon \in [0,1)$.

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 the sets of atoms in which they occur are disjoint or one is contained in the other [28]. 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 [15], 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. 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 $O(UVW\beta^{\omega-3})$.*

Observe that in Lemma 2, matrix multiplication cost dominates the time required to construct the input matrices (if they have not been constructed already) for all $\omega \geq 2$. Fixing $\omega = 2$, rectangular matrix multiplication can be done in time $O(UVW/\beta)$. A long line of research on fast square matrix multiplication has dropped the complexity to $O(n^\omega)$, where $2 \leq \omega < 3$. The current best known value is $\omega = 2.3729$ [12], but it is believed that the actual value is 2.

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 [26], 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 duration between outputting 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. As Figure 1 shows, when Q is full, with $T_p = O(|D|^{\mathtt{fhw}})$, we can enumerate the results with constant delay $O(1)$ [21]. In the particular case where Q is acyclic i.e. $\mathtt{fhw} = 1$, 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, [17] 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. This is the first non-trivial result that improves upon the linear delay guarantees given by [3] for queries with projections.

▶ **Theorem 3** (due to [17]). *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*

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

The factorization width w of a query, originally introduced as $s^{\uparrow}$ [22], is a generalization of the fractional hypertree width from boolean to arbitrary CQs. For $\pi_{\mathbf{x}_1,\ldots,\mathbf{x}_k}(Q_k^*)$, the factorization width is $\mathsf{w} = k$. Observe that preprocessing time T_p is always smaller than the time required to evaluate the full join result. This is because if $T_p = \Theta(|\mathtt{OUT}_{\bowtie}|)$, we can evaluate the full join and deduplicate the projection output, allowing us to obtain constant delay in the enumeration phase. This implies that ϵ can only take values between 0 and $(\log_{|D|} |\mathtt{OUT}_{\bowtie}| - 1)/(\mathsf{w} - 1)$.

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 note that idea of interleaving computation has been explored in the past to develop dynamic algorithms with good worst-case bounds using static data structures [23].

We 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. Since we are using the uniform-cost RAM model, each operation takes one unit of time.

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

1. *$\mathcal{A}$ enumerates query results in total time at most T with no delay guarantees.*
2. *$\mathcal{A}'$ enumerates query results with delay δ and runs in total time at least T'.*
3. *The outputs of $\mathcal{A}$ and $\mathcal{A}'$ are disjoint.*
4. *T and T' are provided as input to the algorithm.*

Then, the union of the outputs of $\mathcal{A}$ and $\mathcal{A}'$ can be enumerated with delay $c \cdot \delta \cdot \max\{1, T/T'\}$ for some constant c.

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 $\mathcal{A}'$. Note that we need to know the exact values of T and T' (by calculating the number of operations in the algorithms $\mathcal{A}$ and $\mathcal{A}'$ to bound the running time). The next lemma introduces our second key idea of interleaving stored output result with *on-the-fly* query computation (the full algorithm and proof can be found in Appendix A).

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

The final helping lemma allows us to enumerate the union of (possibly overlapping) results of m different algorithms where each algorithm outputs its result according to a total order $\preceq$, such that the union is also enumerated in sorted order according to $\preceq$. This lemma is based on the idea presented as Fact 3.1.4 in [18].

▶ **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 delay $O(\delta)$ according to the total order $\preceq$. Then, the union of their output can be enumerated (without duplicates) with $O(m \cdot \delta)$ delay and in sorted order according to $\preceq$.*

Directly implied by Lemma 6 is the fact that 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 according to $\preceq$, then the union of all lists $\bigcup_{i=1}^{m} L_i$ can be enumerated in sorted order given by $\preceq$ 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 $\textsc{ListMerge}(L_1, L_2, \cdots, L_m)$.

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 detailed discussion on how our result is an improvement over prior work in Subsection 4.1. Then, we present a warm-up proof for $\pi_r(Q_k^*)$ in Subsection 4.2, followed by the proof for the general result in Subsection 4.3.

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

[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.

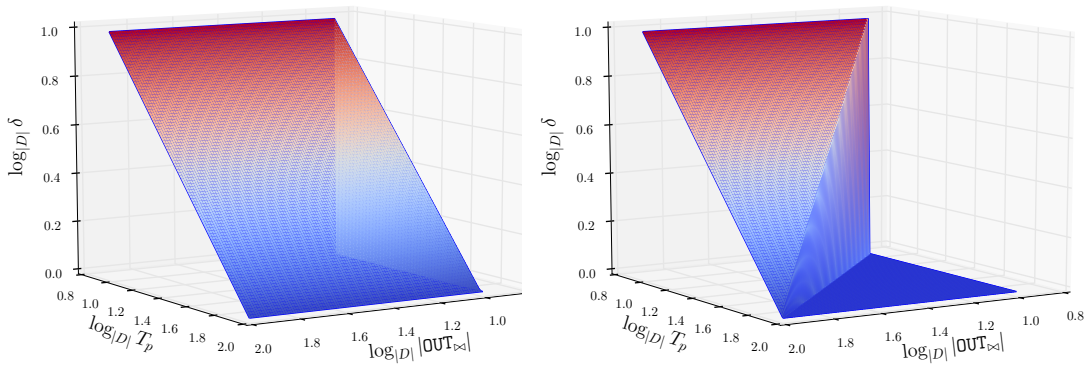


Figure 2 Worst-case tradeoffs given by Theorem 3 without (left) and with (right) taking $|\mathtt{OUT}_{\bowtie}|$ into consideration.

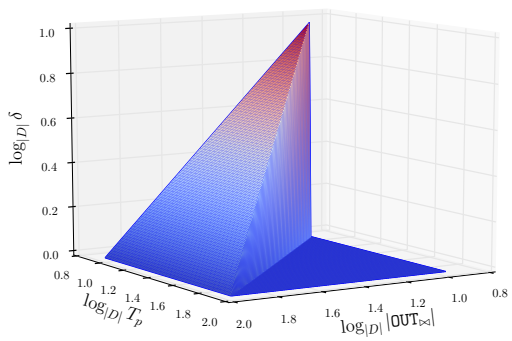


Figure 3 Theorem 7 for $k = 2$.

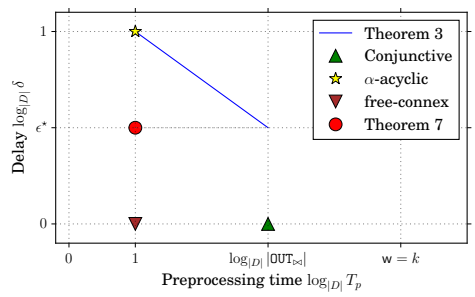


Figure 4 Trade-off in the worst-case for star query.

case when $|\mathtt{OUT}_{\bowtie}| = \Theta(|D|^k)$, it achieves constant delay with linear time preprocessing. In the other extreme, when $|\mathtt{OUT}_{\bowtie}| = \Theta(|D|)$, it achieves linear delay.

When $|\mathtt{OUT}_{\bowtie}|$ has linear size, we can compute and materialize the result of the query in linear preprocessing time and achieve constant delay enumeration. Generalizing this observation, when T_p is sufficient to evaluate the full join result, we can always achieve constant delay.

4.1 Comparison with Prior Work

It is instructive now to compare the worst-case delay guarantee obtained by Theorem 3 for $Q_k^*(D)$ with Theorem 7. Suppose that we want to achieve delay $\delta = O(|D|^{1-\epsilon})$ for some $\epsilon \in [0, (\log_{|D|}|\mathtt{OUT}_{\bowtie}| - 1)/(k-1)]$. Theorem 3 tells us that this requires $O(|D|^{1+\epsilon(k-1)})$ preprocessing time. Then, it holds that:

$$|D|^{1-\epsilon} \geq |D|^{1-\frac{(\log_{|D|}|\mathtt{OUT}_{\bowtie}|-1)}{k-1}} = |D|^{\frac{k-\log_{|D|}|\mathtt{OUT}_{\bowtie}|}{k-1}} = |D|^{k/k-1}/|\mathtt{OUT}_{\bowtie}|^{1/k-1}$$

(a) Database D_0 with full join size N^2.

(b) Database D_1 with full join size $N^{3/2}$.

Figure 5 $D_0 \cup D_1$ forms a database where Theorem 7 improves the delay of Theorem 3.

In other words, either we have enough preprocessing time to materialize the output and achieve constant delay, or we can achieve the desirable delay with linear preprocessing time.

Figure 2, Figure 3 and Figure 4 show the existing and new tradeoff results. Figure 2 shows the tradeoff curve obtained from Theorem 3 by adding $|\texttt{OUT}_\bowtie|$ as a third dimension, and adding the optimization for constant delay when $T_p \geq O(|\texttt{OUT}_\bowtie|)$. Figure 3 shows the tradeoff obtained from our result, while Figure 4 shows other existing results for a fixed value of $|\texttt{OUT}_\bowtie|$. For a fixed value of $|\texttt{OUT}_\bowtie|$, the delay guarantee does not change in Figure 3 as we increase T_p from $|D|$ to $|\texttt{OUT}_\bowtie|$. It remains an open question to further decrease the delay if we allow more preprocessing time. Such an algorithm would correspond to a curve connecting the red point(●) and the green triangle(▲) in Figure 4.

Our results thus imply that, depending on $|\texttt{OUT}_\bowtie|$, one must choose a different algorithm to achieve the optimal tradeoff between preprocessing time and delay. Since $|\texttt{OUT}_\bowtie|$ can be computed in linear time (using a simple adaptation of Yannakakis algorithm [32, 24]), this can be done without affecting the preprocessing bounds.

Next, we show how our result provides an algorithmic improvement over Theorem 3. Consider the instances D_0, D_1 depicted in Figure 5a and Figure 5b respectively, and assume we want to use linear preprocessing time. For D_1, the algorithm of Theorem 3 materializes nothing, since no y valuation has a degree of $O(|D|^0)$, and the delay will be $\Theta(\sqrt{N})$. No materialization also occurs for D_0, but here the delay will be $O(1)$. It is easy to check that our algorithm matches the delay on both instances. Now, consider the instance $D = D_0 \cup D_1$. The input size for D is $\Theta(N)$, while the full join size is $N^{3/2} + N^2 = \Theta(N^2)$. The algorithm of Theorem 3 will again achieve only a $\Theta(\sqrt{N})$ delay, since after the linear time preprocessing no y valuations can be materialized. In contrast, our algorithm still guarantees a constant delay. This algorithmic improvement is a result of the careful overlapping of the constant-delay computation for instance D_0 with the computation for D_1.

The above construction can be generalized as follows. Let $\alpha \in (0, 1)$ be some constant. D_0 remains the same. For D_1, we construct R to be the cross product of N^α x-values and $N^{1-\alpha}$ y-values, and S to be the cross product of N^α z-values and $N^{1-\alpha}$ y-values. As before, let $D = D_0 \cup D_1$. The input size for D is $\Theta(N)$, while the full join size is $N^{2-\alpha} + N^2 = \Theta(N^2)$. Hence, our algorithm achieves constant delay with linear preprocessing time. In contrast, the algorithm of Theorem 3 achieves $\Theta(N^{1-\alpha})$ delay with linear preprocessing time. In fact, the $\Theta(N^{1-\alpha})$ delay occurs even if we allow $O(N^{1+\epsilon})$ preprocessing time for any $\epsilon < \alpha$. We can now use the same idea to show that there also exists an instance where achieving constant delay using Theorem 3 requires near quadratic preprocessing time (Example 14 in the appendix).

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.

4.2 Warm-up: Two-Path Query

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

At a high level, we will decompose the join into two 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. Using any comparison based sorting technique requires $\Omega(|D| \log |D|)$ time in general. Thus, if we wish to remove the $\log |D|$ factor, we must use non-comparison based sorting algorithms. In this paper, we will use count sort [8] which has complexity $O(|D| + r)$ where r is the range of the non-negative key values. However, we need to ensure that all relations in the database D satisfy the bounded range requirement. This can be easily accomplished by introducing a bijective function $f : \mathbf{dom}(D) \to \{1, 2, \ldots, |D|\}$ that maps all values in the active domain of the database to some integer between 1 and $|D|$ (both inclusive). Both f and its inverse f^{-1} can be stored as hash tables as follows: suppose there is a counter $c \leftarrow 1$. We perform a linear pass over the database and check if some value $v \in \mathbf{dom}(D)$ has been mapped or not (by checking if there exists an entry $f(v)$). If not, we set $f(v) = c, f^{-1}(c) = v$ and increment c. Once the hash tables f and f^{-1} have been created, we modify the input relation R (and S similarly) by replacing every tuple $t \in R$ with tuple $t' = f(t)$. Since the mapping is a relabeling scheme, such a transformation preserves the degree of all the values. The codomain of f is also equipped with a total order $\preceq$ (we will use $\leq$). Note that f is not an order-preserving transformation in general but this property is not required in any of our algorithms.

Next, for every tuple $t \in R(x,y)$, we create a hash map with key $\pi_x(t)$ and the value is a list $\pi_y(t)$; and for every tuple $t \in S(y,z)$, we create a hash map with key $\pi_y(t)$ and the value is a list $\pi_z(t)$. For the second hash map, we sort the value list using sort order $\preceq$ for each key, once each tuple $t \in S(y,z)$ has been processed. Finally, we sort all values in $\pi_x(R)$ in increasing order of their degree in R (i.e $|\sigma_{x=v_i} R(x,y)|$ is the sort key). Let $\mathcal{L} = \{v_1, \ldots, v_n\}$ denote the ordered set of these values sorted by their degree and let $d_1, \ldots, d_n$ be their respective degrees. Creating the sorted list $\mathcal{L}$ takes $O(|D|)$ time since the degrees d_i satisfy the bounded range requirement (i.e $1 \leq d_i \leq |D|$). Next, we identify the smallest index i^* such that

$$\sum_{v:\{v_1, v_2, \ldots, v_{i^*}\}} |R(v,y) \bowtie S(y,z)| \geq \sum_{v:\{v_{i^*+1}, \ldots, v_n\}} |R(v,y) \bowtie S(y,z)| \tag{1}$$

This can be computed by doing a linear pass on $\mathcal{L}$ using a simple adaptation of Yannakakis algorithm [32, 24]. This entire phase takes time $O(|D|)$.

Enumeration Phase. The enumeration algorithm interleaves the following two loops using the construction in Lemma 4. Specifically, it will spend an equal amount of time (a constant) before switching to the computation of the other loop.

Algorithm 1 EnumTwoPath

for $i = 1, \ldots, i^*$ **do**
  Let $\pi_y(\sigma_{x=v_i}(R)) = \{u_1, u_2, \cdots, u_\ell\}$;
  output $(v_i, f^{-1}(\textsc{ListMerge}(\pi_z\sigma_{y=u_1}S, \pi_z\sigma_{y=u_2}S, \cdots, \pi_z\sigma_{y=u_\ell}S)))$[4]
  run for $O(1)$ time then switch
for $i = i^* + 1, \ldots, n$ **do**
  Let $\pi_y(\sigma_{x=v_i}(R)) = \{u_1, u_2, \cdots, u_\ell\}$;
  output $(v_i, f^{-1}(\textsc{ListMerge}(\pi_z\sigma_{y=u_1}S, \pi_z\sigma_{y=u_2}S, \cdots, \pi_z\sigma_{y=u_\ell}S)))$
  run for $O(1)$ time then switch

The algorithm alternates between low-degree and high-degree values in $\mathcal{L}$. The main idea is that, for a given $v_i \in \mathcal{L}$, we can enumerate the result of the subquery $\sigma_{x=v_i}(Q_{\texttt{two-path}})$ with delay $O(d_i)$. This can be accomplished by observing that the subquery is equivalent to list merging and so we can use Algorithm 3.

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

The reader should note that the delay of $\delta = O(|D|^2/|\texttt{OUT}_\bowtie|)$ is only an upper bound. Depending on the skew present in the database instance, it is possible that Algorithm 1 achieves much better delay guarantees in practice (as shown in Example 16 in the Appendix).

4.3 Proof of Main Theorem

We now generalize Algorithm 1 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 the remaining light subqueries.

Preprocessing Phase. Assume all relations are reduced without dangling tuples, which can be achieved in linear time [32]. The full join size $|\texttt{OUT}_\bowtie|$ can also be computed in linear time. Similar to the preprocessing phase in the previous section, we construct the hash tables f, f^{-1} to perform the domain compression and modify all the input relations by replacing tuple t with $f(t)$. Set $\Delta = (2 \cdot |D|^k/|\texttt{OUT}_\bowtie|)^{\frac{1}{k-1}}$. For each relation R_i, a value v for attribute x_i is *heavy* if 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 in time $O(|D|)$. 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 in a way so that there is no overlap between the output of any subquery. In order to do so, we create k subqueries of the form

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

[4] Abusing the notation, $f^{-1}(\mathcal{B})$ for some ordered list (or tuple) $\mathcal{B}$ returns an ordered list (tuple) $\mathcal{B}'$ where $\mathcal{B}'(i) = f^{-1}(\mathcal{B}(i))$.

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 each output tuple t is generated by exactly one of the Q_i and thus the output of all subqueries is disjoint. This implies that each $f^{-1}(t)$ is also generated by exactly one subquery. Similar to the preprocessing phase of two path query, we store all R_i^ℓ and R_i^h in hashmaps where the values in the maps are lists sorted in lexicographic order.

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 the user with delay guarantees and also make progress on computing the full join of Q_H. Our goal is to reason about the running time of enumerating Q_L (denoted by T_L) and the running time of Q_H (denoted by T_H) and make sure that while we compute Q_H, we do not run out of the output from Q_L.

Next, we introduce the algorithm that enumerates output for any specific valuation v of attribute x_i, which is described in Lemma 9. This algorithm can be viewed as another instantiation of Algorithm 3.

▶ **Lemma 9.** *Consider an arbitrary value $v \in \mathbf{dom}(x_i)$ with degree d in relation $R_i(x_i, y)$. Then, its query result $\pi_{x_1,x_2,\cdots,x_k}\sigma_{x_i=v}R_1^h(x_1, y) \bowtie R_2^h(x_2, y) \bowtie \cdots \bowtie R_i(x_i, y) \bowtie \cdots \bowtie R_k^\star(x_k, y)$ can be enumerated with $O(d)$ delay guarantee.*

Let $c^\star$ be an upper bound on the number of operations in each iteration of ListMerge. This can be calculated by counting the number of operations in the exact implementation of the algorithm. Directly implied by Lemma 9, the result of any subquery in Q_L can be enumerated with delay $O(\Delta)$. Let Q_H^* denote the corresponding full query of Q_H, i.e, the head of Q_H^* also includes the variable y (Q_L^* is defined similarly). Then, Q_H^* can be evaluated in time $T_H \leq c^\star \cdot |Q_H^*| \leq c^\star \cdot |\mathtt{OUT}_\bowtie|/2$ by using ListMerge on subquery Q_H. This follows from the bound $|Q_H^*| \leq |D| \cdot (|D|/\Delta)^{k-1}$ and our choice of $\Delta = (2 \cdot |D|^k/|\mathtt{OUT}_\bowtie|)^{\frac{1}{k-1}}$. Since $|Q_H^*| + |Q_L^*| = |\mathtt{OUT}_\bowtie|$, it holds that $|Q_L^*| \geq |\mathtt{OUT}_\bowtie|/2$ given our choice of Δ. Also, the running time T_L is lower bounded by $|Q_L^*|$ (since we need at least one operation for every result). Thus, $T_L \geq |\mathtt{OUT}_\bowtie|/2$.

We are now ready to apply Lemma 4 with the following parameters:

1. $\mathcal{A}$ is the full join computation of Q_H and $T = c^\star \cdot |\mathtt{OUT}_\bowtie|/2$.
2. $\mathcal{A}'$ is the enumeration algorithm applied to Q_L with delay guarantee $\delta = O(\Delta)$ and $T' = |\mathtt{OUT}_\bowtie|/2$.
3. T and T' are fixed once $|\mathtt{OUT}_\bowtie|, \Delta$ and the constant $c^\star$ are known.

By construction, the outputs of Q_H and Q_L are also disjoint. Thus, the conditions of Lemma 4 apply and we obtain a delay of $O(\Delta)$.

4.4 Interleaving with Join Computation

Theorem 7 obtains poor delay guarantees when the full join size $|\mathtt{OUT}_\bowtie|$ is close to input size $|D|$. 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 computing 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 10.** *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 $\delta = O\left(|\mathtt{OUT}_\bowtie|/|\mathtt{OUT}_\pi|^{1/k}\right)$ and space $S_e = O(|D|)$*

In the above theorem, we obtain delay guarantees that depend on both the full join result $\mathtt{OUT}_\bowtie$ and the projection output size $\mathtt{OUT}_\pi$.

However, one does not need to know $\mathtt{OUT}_\bowtie$ or $\mathtt{OUT}_\pi$ to apply the result. We first compare the result with Theorem 7. First, observe that both Theorem 10 and Theorem 7 require $O(|D|)$ preprocessing time. Second, the delay guarantee provided by Theorem 10 can be better than Theorem 7. This happens when $|\mathtt{OUT}_\bowtie| \leq |D| \cdot J^{1-1/k}$, a condition that can be easily checked in linear time.

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 before, 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 can now invoke Lemma 5 to achieve the claimed delay. The final observation is to express J in terms of $|\mathtt{OUT}_\pi|$. Note that $|\mathtt{OUT}_\pi| \leq \Pi_{i\in[k]} |\mathbf{dom}(x_i)|$ which implies that $\max_{i\in[k]} |\mathbf{dom}(x_i)| \geq |\mathtt{OUT}_\pi|^{1/k}$. Thus, it holds that $J \geq |\mathtt{OUT}_\pi|^{1/k}$ which gives us the desired bound on the delay guarantee.

4.5 Fast Matrix Multiplication

Both Theorem 7 and Theorem 3 are combinatorial algorithms. 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.

▶ **Theorem 11.** *Consider the star query $\pi_{x_1,\ldots,x_k}(Q_k^*)$ and an input database instance D. Then, there exists an algorithm that requires preprocessing $T_p = O((|D|/\delta)^{\omega+k-2})$ and can enumerate the query result with delay $O(\delta)$ for $1 \leq \delta \leq |D|^{(\omega+k-3)/(\omega+2\cdot k-3)}$.*

For the two-path query and the current best value of $\omega = 2.373$, we get the tradeoff as $T_p = O((|D|/\delta)^{2.373})$ and a delay guarantee of $O(\delta)$ for $|D|^{0.15} < \delta \leq |D|^{0.40}$. 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 safely assume that $|\mathtt{OUT}_\bowtie| > |D|^{1.422}$, otherwise one can simply compute the full join in time $c^\star \cdot |D|^{1.422}$ using ListMerge, deduplicate and get constant delay enumeration. Applying Theorem 7 with $|\mathtt{OUT}_\bowtie| > |D|^{1.422}$ tells us that we can obtain delay as $O(|D|^2/|\mathtt{OUT}_\bowtie|) = O(|D|^{0.58})$. Thus, we can offer the user both choices and the user can decide which enumeration algorithm to use.

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:

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

It is easy to see that $Q^k_{\texttt{leftdeep}}$ 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 $\pi_{w_1,\ldots,w_k}(Q^k_{\texttt{leftdeep}})$, where we project out all the join variables. We show that the following result holds:

▶ **Theorem 12.** *Consider the query $\pi_{w_1,\ldots,w_k}(Q^k_{\texttt{leftdeep}})$ and any input database D. Then, there exists an algorithm that enumerates the query after preprocessing time $T_p = O(|D|)$ with delay $O(|D|^k/|\texttt{OUT}_\bowtie|)$.*

In the above theorem, $\texttt{OUT}_\bowtie$ is the full join result of the query $Q^k_{\texttt{leftdeep}}$ without projections. The AGM exponent for $Q^k_{\texttt{leftdeep}}$ is $\rho^* = k$. Observe that Theorem 12 is of interest when $|\texttt{OUT}_\bowtie| > |D|^{k-1}$ to ensure that the delay is smaller than $O(|D|)$. When the condition $|\texttt{OUT}_\bowtie| > |D|^{k-1}$ holds, the delay obtained by Theorem 12 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 linear preprocessing time, compared to Theorem 3 that would require $\Theta(|D|^k)$ preprocessing time to achieve the same delay. The decision of when to apply Theorem 12 or Theorem 3 can be made in linear time by checking whether $|D|^k/|\texttt{OUT}_\bowtie|$ is smaller or larger than the actual delay guarantee obtained by the algorithm of Theorem 3 after linear time preprocessing.

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 [17] does not apply. A subset of path queries, namely 3-path and 4-path counting queries were considered in [16]. The algorithm used for counting the answers of 3-path and 4-path queries under updates constructed a set of views that can be used for the task of enumerating the query results under the static setting. Our result extends the same idea to apply to arbitrary length path queries, which we state next.

▶ **Theorem 13.** *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 $O(|D|^\epsilon)$.*

We should note here that for $\epsilon = 1$, we can obtain a delay $O(|D|)$ using only linear preprocessing time $O(|D|)$ using the result of [3] since the query is acyclic, 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 13 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 [27]. Prior work also exhibits a dependency between the space and enumeration delay for conjunctive queries with access patterns [10]. 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 [17], 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 [17] 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 [30, 29, 31, 11] has studied how compressed representations can be created apriori that allow for faster enumeration of query results. For the two path query, the fastest evaluation algorithm (with no delay guarantees) evaluates the projection join output in time $O(|D| \cdot |\mathtt{OUT}_\pi|^{\frac{(\omega-1)}{(\omega+1)}} + |D|^{\frac{2(\omega-1)}{(\omega+1)}} \cdot |\mathtt{OUT}_\pi|^{\frac{2}{(\omega+1)}})$ [9, 2]. For star queries, there is no closed form expression but fast matrix multiplication can be used to obtain instance dependent bounds on running time. Also related is the problem of dynamic evaluation of hierarchical queries. Recent work [16, 17, 4, 5] has studied the tradeoff between amortized update time and delay guarantees. Some of our techniques may also lead to new insights and improvements in existing algorithms. Prior work in differential privacy [25] and DGM [7] may also benefit from some of our techniques.

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 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 new results on time-delay tradeoffs for a subset of non-hierarchical queries for the class of path queries. Our results also open several new tantalizing questions that open up possible directions for future work.

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 $T_p = O(|\mathtt{OUT}_\bowtie|/|D|^\epsilon + |D|)$ 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 $Q_{\texttt{two-path}}$ query. This question is equivalent to the following problem: for what values of $|\texttt{OUT}_\pi|$ can $Q_{\texttt{path}}$ be evaluated in linear time. If $|\texttt{OUT}_\pi| = |D|^\epsilon$, then the best known algorithms can evaluate $Q_{\texttt{two-path}}$ in time $O(|D|^{1+\epsilon/3})$ (using fast matrix multiplication) [9] 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. We also believe that the space-delay tradeoff implied by prior work can also be improved for certain ranges of delay by using the ideas introduced in this paper.

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. A. R. Chowdhury, T. Rekatsinas, and S. Jha. Data-dependent differentially private parameter learning for directed graphical models. In *International Conference on Machine Learning*, pages 1939–1951. PMLR, 2020.
8. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Chapter 8.2, Introduction to algorithms*. MIT press, 2009.
9. S. Deep, X. Hu, and P. Koutris. Fast join project query evaluation using matrix multiplication. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1213–1223, 2020.
10. S. Deep and P. Koutris. Compressed representations of conjunctive query results. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 307–322. ACM, 2018.
11. S. Deep and P. Koutris. Ranked enumeration of conjunctive query results. *To appear in Joint 2021 EDBT/ICDT Conferences, ICDT '21 Proceedings*, 2021.
12. 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.
13. G. Gottlob, G. Greco, and F. Scarcello. Treewidth and hypertree width. *Tractability: Practical Approaches to Hard Problems*, 1, 2014.
14. G. Greco and F. Scarcello. Structural tractability of enumerating csp solutions. *Constraints*, 18(1):38–74, 2013.
15. J. E. Hopcroft, J. D. Ullman, and A. Aho. The design and analysis of computer algorithms, 1975.
16. 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.
17. A. Kara, M. Nikolic, D. Olteanu, and H. Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 375–392, 2020.
18. W. Kazana. *Query evaluation with constant delay*. PhD thesis, 2013.
19. 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.
20. 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.
21. D. Olteanu and M. Schleich. Factorized databases. *ACM SIGMOD Record*, 45(2):5–16, 2016.

22 D. Olteanu and J. Závodnỳ. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)*, 40(1):1–44, 2015.

23 M. H. Overmars and J. Van Leeuwen. Dynamization of decomposable searching problems yielding good worst-case bounds. In *Theoretical Computer Science*, pages 224–233. Springer, 1981.

24 A. Pagh and R. Pagh. Scalable computation of acyclic joins. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 225–232, 2006.

25 A. Roy Chowdhury, C. Wang, X. He, A. Machanavajjhala, and S. Jha. Crypt?: Crypto-assisted differential privacy on untrusted servers. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 603–619, 2020.

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

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

28 D. Suciu, D. Olteanu, C. Ré, and C. Koch. Probabilistic databases, synthesis lectures on data management. *Morgan & Claypool*, 2011.

29 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.

30 K. Xirogiannopoulos, U. Khurana, and A. Deshpande. Graphgen: Exploring interesting graphs in relational data. *Proceedings of the VLDB Endowment*, 8(12):2032–2035, 2015.

31 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.

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

A Algorithm for Lemma 5

Algorithm 2 describes the detailed algorithm for Lemma 5.

Algorithm 2 DEDUPLICATE($J, \mathcal{A}$)

Input : Materialized output list J, Algorithm $\mathcal{A}$ with known completion time T
Output : Deduplicated result of $\mathcal{A}$

```
δ ← O(T/J), ptr ← 0, dedup ← 0
H ← ∅                                                    /* empty hash-set */
while ptr < |J| do
    output J[ptr]                  /* output result from J to maintain delay guarantee */
    ptr ← ptr + 1, counter ← 0
    while counter ≤ δ do
        if A has not completed then
            Execute A for c time                         /* c is a constant */
            foreach t ∈ t                /* let t be the output tuples generated (if any) */
             do
                if t ∉ J and t ∉ H then
                    output t
                    insert t in H
        counter ← counter + c;
```

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

Proof. Let δ be a parameter to be fixed later. We first store the J output results in a hash set and create an empty hash set H that will be used for deduplication. 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 set H and J, emits t only if t does not appear in H and J, followed by inserting t in H. Inserting t in H will ensure that $\mathcal{A}$ does not output duplicates[5]. Each probe takes $O(1)$ time, so the total running time of $\mathcal{A}$ is $O(T)$. Our goal is to choose δ such that $\mathcal{A}$ terminates before the materialized output J runs out. This condition is satisfied when $\delta \cdot J \geq O(T)$ which gives us $\delta = O(T/J)$. 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. ◀

B Other Missing Proofs

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

1. *$\mathcal{A}$ enumerates query results in total time at most T with no delay guarantees.*

[5] If $\mathcal{A}$ guarantees that it will generate results with no duplicates, then there is no need to use H.

2. *$\mathcal{A}'$ enumerates query results with delay δ and runs in total time at least T'.*
3. *The outputs of $\mathcal{A}$ and $\mathcal{A}'$ are disjoint.*
4. *T and T' are provided as input to the algorithm.*

Then, the union of the outputs of $\mathcal{A}$ and $\mathcal{A}'$ can be enumerated with delay $c \cdot \delta \cdot \max\{1, T/T'\}$ for some constant c.

Proof. Let η and γ denote two positive values 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 run $\mathcal{A}'$ for γ operations, pause $\mathcal{A}'$, then run $\mathcal{A}$ for η operations, pause $\mathcal{A}$ and resume $\mathcal{A}'$ for γ operations, and so on. The pause and resume takes constant time (say c_{pause} and c_{resume}) 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 η such that $\mathcal{A}'$ does not terminate until $\mathcal{A}$ has finished. This condition is satisfied when the number of iterations of $\mathcal{A}'$ is larger than number of iterations of $\mathcal{A}$. This gives us the condition that,

$$T'/\gamma \leq (\text{Time taken by } \mathcal{A}')/\gamma = (\text{Time taken by } \mathcal{A})/\eta \leq T/\eta$$

Thus, any value of $\eta \leq T \cdot \gamma/T'$ is acceptable. We fix η to be any positive constant and then set γ to be the smallest positive value that satisfies the condition. The delay is bounded by the product of worst-case number of iterations between two answers of $\mathcal{A}'$ and the work done between each iteration which is $(\delta/\gamma) \cdot (\gamma + \eta + c_{\mathsf{pause}} + c_{\mathsf{resume}}) \leq \delta \cdot (1 + T/T' + (c_{\mathsf{pause}} + c_{\mathsf{resume}})/\gamma) = O(\delta \cdot \max\{1, T/T'\})$. ◀

Algorithm 3 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 (using ⪯) 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 */
```

▶ **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 delay $O(\delta)$ according to the total order $\preceq$. Then, the union of their output can be enumerated (without duplicates) with $O(m \cdot \delta)$ delay and in sorted order according to $\preceq$.*

Proof. We describe Algorithm 3. 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 (e_i) over the remaining result to be enumerated. Algorithm 3 goes over all locally minimum results over all algorithms and outputs the smallest one (denoted w) as globally minimum result (line 5). Once a result

is enumerated, each $\mathcal{A}_i$ needs to check whether its e_i matches w. If yes, then $\mathcal{A}_i$ needs to update its locally minimum result by finding the next one. Then, Algorithm 3 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 and $O(m \cdot \delta)$ to update all local minimum results (line 7-line 9). Thus, Algorithm 3 has a delay guarantee of $O(m \cdot \delta)$. ◀

▶ **Example 14.** Continuing our discussion from Subsection 4.1, we now construct an instance where achieving constant delay with Theorem 3 would require close to $\Theta(|D|^2)$ computation. Let us fix N to be a power of 2. We will fix $|\mathbf{dom}(x)| = |\mathbf{dom}(z)| = N \log N$. Let D_i be the database constructed by setting $N^\alpha = 2^i$ for $i \in \{1, 2, \ldots, \log N\}$ where relation R is the cross product of N^α x-values and $N^{1-\alpha}$ y-values, and S is the cross product of N^α z-values and $N^{1-\alpha}$ y-values.. We also construct a database D^* which consists of a single y that is connected to all x and z values. Let $D = D^* \cup D_1 \cup D_2 \cup \cdots \cup D_{\log N}$. It is now easy to see that $|D| = N \cdot \log N, |\mathbf{dom}(y)| = \sum_\alpha N^{1-\alpha} \leq 2N = \Theta(|D|/\log|D|)$ and $|\mathtt{OUT}_\bowtie| = \sum_\alpha N^{1+\alpha} + N^2 \log^2 N = \Theta(|D|^2)$. On this instance, Theorem 3 achieves $\Theta(|D|/\log|D|)$ after linear time preprocessing. Suppose we wish to achieve constant delay enumeration. Let us fix this constant to be c^* (which is also a power of 2, for simplicity). Then, we need enough preprocessing time to materialize the join result of all database instances D_i where $i \in \{1, 2, \ldots, \log(N/c^*)\}$ to ensure that the number of heavy y values that remain is at most c^*. This requires time $T_p > \sum_{i \in \{1,2,\ldots,\log(N/c^*)\}} N \cdot 2^i > N^2/c^* = \Theta(|D|^2/\log|D|)$. This example shows that Theorem 3 requires near quadratic computation to achieve constant delay enumeration.

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

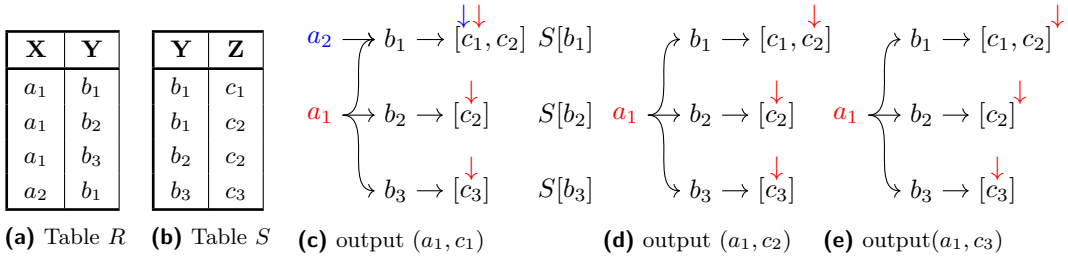
Proof. To prove this result, we will apply Lemma 4, where $\mathcal{A}'$ is the first loop (the one with light-degree values), and $\mathcal{A}$ is the second loop (the one with high-degree values).

Let δ denote the degree of the valuation v_{i^*}. First, we claim that the delay of $\mathcal{A}'$ will be $O(\delta)$. Indeed, ListMerge will output a result every $O(\delta)$ time since the degree of each valuation in the first loop is at most δ. Let $J_h = \sum_{i > i^*} |R(v_i, y) \bowtie S(y, z)|$ and $J_\ell = \sum_{i \leq i^*} |R(v_i, y) \bowtie S(y, z)|$. Then, $\mathcal{A}'$ runs in time at least J_ℓ, and $\mathcal{A}$ in time at most $c^\star \cdot J_h$. Here, $c^\star$ is an upper bound on the number operations in each iteration of the loop in Algorithm 1. Since by construction $J_\ell \geq J_h$, Lemma 4 obtains a total delay of $O(\delta)$.

It now remains to bound δ. First, observe that, since i^* is the smallest index that satisfies Equation 1, it must be that $J_\ell - J_h \leq |D|$ (if not, shifting the smallest index by one decreases the LHS by at most $|D|$ and increases the RHS by at most $|D|$ while still satisfying the condition that $J_\ell \geq J_h$). Combined with the observation that $J_\ell + J_h = |\mathtt{OUT}_\bowtie|$, we get that $J_h \geq |\mathtt{OUT}_\bowtie|/2 - |D|/2 \geq 1/4 \cdot |\mathtt{OUT}_\bowtie|$ assuming $|\mathtt{OUT}_\bowtie| \geq 2 \cdot |D|$. The final observation is that $J_h \leq |D|^2/\delta$ since there are most $|D|/\delta$ heavy values, and each heavy value can join with at most $|D|$ tuples for the full join. Combining the two inequalities gives us the claimed delay guarantee. ◀

▶ **Lemma 9.** *Consider an arbitrary value $v \in \mathbf{dom}(x_i)$ with degree d in relation $R_i(x_i, y)$. Then, its query result $\pi_{x_1, x_2, \cdots, x_k} \sigma_{x_i = v} R_1^h(x_1, y) \bowtie R_2^h(x_2, y) \bowtie \cdots \bowtie R_i(x_i, y) \bowtie \cdots \bowtie R_k^\star(x_k, y)$ can be enumerated with $O(d)$ delay guarantee.*

Proof. Consider some tuple $(v_i, u) \in R_i$. 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



(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 6 Example for two path query enumeration

$\sigma_{y=u} R_j(x_j, y)$. Note that such a list is not materialized as that for two-path query, but present in a factorized form.

We next define the enumeration algorithm $\mathcal{A}_u$ for each $u \in \pi_y \sigma_{x_i=v} R_i(x_i, y)$, 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 lexicographic order $(x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_k)$, which has $O(k-1) = O(1)$ delay. After applying Algorithm 3, we can obtain an enumeration algorithm that enumerates the union of query results over all neighbors with $O(d)$ delay guarantee. For each output tuple t generated by Algorithm 3, we return $f^{-1}(t)$ to the user. ◀

▶ **Example 15.** Consider relations R and S as shown in Figure 6a and Figure 6b. Figure 6c 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 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 6d 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 6e. 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.

▶ **Example 16.** 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 $Q_{\texttt{two-path}}$. It is easy to see that $\texttt{OUT}_{\bowtie} = \Theta(N)$. Thus, applying the bound of $\delta = O(N^2/|\texttt{OUT}_{\bowtie}|)$ gives us $O(N)$ delay. However, Algorithm 1 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.

▶ **Theorem 11.** *Consider the star query $\pi_{x_1,\ldots,x_k}(Q_k^*)$ and an input database instance D. Then, there exists an algorithm that requires preprocessing $T_p = O((|D|/\delta)^{\omega+k-2})$ and can enumerate the query result with delay $O(\delta)$ for $1 \leq \delta \leq |D|^{(\omega+k-3)/(\omega+2\cdot k-3)}$.*

Proof. We sketch the proof for $k = 2$. 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 $? = \ell$, y has $? = \star$ and z has $? = \star$. In this case, we can just invoke ListMerge(v_i) for each valuation v_i of attribute x and enumerate the output.
- x has $? = h$, y has $? = \star$ and z has $? = \ell$. In this case, we can invoke ListMerge(v_i) for each valuation v_i of 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. In the following, we say that y has $? = \ell$ to mean that the join considers all y valuations that have degree at most δ in both R and S.
  - 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 where all dimensions have size $O(|D|/\delta)$. Using Lemma 2, we can evaluate the join in time $O((|D|/\delta)^\omega)$.

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

▶ **Theorem 12.** *Consider the query $\pi_{w_1,\ldots,w_k}(Q^k_{\texttt{leftdeep}})$ and any input database D. Then, there exists an algorithm that enumerates the query after preprocessing time $T_p = O(|D|)$ with delay $O(|D|^k/|\texttt{OUT}_\bowtie|)$.*

Proof. Once again, we will use the same steps of the preprocessing phase as in Lemma 8. We index all the input relations in a hash table where the values are sorted lists after applying the domain compression trick using f and f^{-1}. Thus, count sort now runs in $O(|D|)$ time. We also compute $|\texttt{OUT}_\bowtie|$ using Yannakakis algorithm.

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 $k-1$ sorted 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. Let T_L denote the total time required to enumerate the query result for all light w_k valuations.

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 of the heavy subquery can be done in time $T_H \leq c^\star \cdot |D|^k/\delta$ using ListMerge. Fixing $\delta = 2 \cdot |D|^k/|\texttt{OUT}_\bowtie|$ gives us $T_H \leq c^\star \cdot |\texttt{OUT}_\bowtie|/2$. Since $|Q^*_L| + |Q^*_H| = |\texttt{OUT}_\bowtie|$, our choice of δ ensures that $T_L \geq |Q^*_L| \geq |\texttt{OUT}_\bowtie|/2$.

We can now apply Lemma 4 with (i) $\mathcal{A}'$ is the list-merging algorithm for the light case with $T' = |\texttt{OUT}_\bowtie|/2$; (ii) $\mathcal{A}$ is the worst-case optimal join algorithm for the heavy case with $T = c^\star \cdot |\texttt{OUT}_\bowtie|/2$; (iii) T, T' are fixed once $|\texttt{OUT}_\bowtie|, \delta, c^\star$ have been computed.

Once again, in order to know the exact values of T, T', we need to analyze the exact constant that is used in the join algorithm for ListMerge. By construction, the output of $\mathcal{A}$ and $\mathcal{A}'$ is different. Note that for each output tuple t generated, we return $f^{-1}(t)$ to the the user, a constant time operation. ◀

▶ **Theorem 13.** *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 $O(|D|^{\epsilon})$.*

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, apply the domain transformation technique by creating f and f^{-1} and then create a hash map for each relation $R_i(x_i, x_{i+1})$ 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, and sorting in linear time using count sort. 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 $O(\Delta^{k-i})$. This implies that our target path query can be enumerated with delay $O(\Delta^{k-1})$, 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 $O(\Delta^{k-i-1})$. 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 $O(\Delta \cdot \Delta^{k-i-1}) = O(\Delta^{k-i})$. Finally, For each output tuple t generated, we return $f^{-1}(t)$ to the the user. ◀