TASK-AWARE MATERIALIZATION FOR FAST DATA ANALYTICS

by

Shaleen Deep

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2021

Date of final oral examination: 22 November, 2021

The dissertation is approved by the following members of the Final Oral Committee:

Paraschos Koutris, Assistant Professor, Computer Sciences

Jignesh Patel, Professor, Computer Sciences

Theodoros Rekatsinas, Assistant Professor, Computer Sciences

Dimitris Papailiopoulos, Assistant Professor, Electrical and Computer Engineering

ACKNOWLEDGMENTS

Undertaking this Ph.D. was a life-changing experience for me and it would not have been possible without the support of so many people that I have relied upon. I would like to take this opportunity to thank everyone who helped me in this journey.

First and foremost, I would like to express my gratitude to my advisor Paris Koutris. Without his belief in me, guidance, support, and advice over the years, I would not have made it. I cannot thank him enough for being so generous with his time, countless hours of brainstorming on the whiteboard, his brilliant and creative ideas, endless patience when I made mistakes, and long walks discussing research that was a great source of happiness for me. I am constantly in awe of how effortlessly he can generate such sharp insights to do research of the highest quality. I will forever be grateful to him for teaching me everything that I now know and shaping me as a researcher. It was an honor to be advised by Paris and as I start my career after graduation, I will do everything I can to make him a proud advisor.

I also thank the other members of my committee, Jignesh Patel, Theodoros Rekatsinas, and Dimitris Papailiopoulos for their feedback and thought provoking questions. Jignesh's advice and suggestions on my work have been instrumental in shaping my thinking on how to approach research. I thank him for making the time to meet me, despite his busy schedule, whenever I was in need of help. I am also very grateful to Xiao Hu for the opportunity to collaborate, her constant encouragement, and for her help whenever I was stuck.

I am grateful to Computer Sciences staff Angela Thorp, Hilar Heffley, and Patricia Rentner for their help in administrative matters over the years. I am extremely fortunate to have my undergraduate (in India) and graduate education fully paid for in the form of scholarships, fellowships, and grant money from NSF. This money primarily comes from the taxes paid by the hardworking men and women in India and America, and subsidies from the

government. I have always kept this fact in mind and worked hard to ensure that every rupee and dollar spent on me is fully utilized and produces some value to society.

Graduate school would not have been so much fun if not for the friends in the Wisconsin DB group. An incomplete list includes Bruhathi Sundarmurthy, Pradap Konda, Paul Suganthan, Adel Ardalan, Kevin Gaffney, Yannis Chronis, Zhihan Guo, Aarati Kakaraparthy, Han Li, Navneet Potti, Rogers Jeffrey Leo John, Zifan Liu, Ankur Goswami, and Adalbert Gerald Soosai Raj. In particular, I thank Yash Govind and Harshad Deshmukh for being such amazing friends and their brotherly advice whenever I needed it. I also thank Xiating Ouyang and Zhiwei Fan for being such wonderful officemates over the years.

I am also very fortunate to have met some great friends over the years in graduate school. An incomplete list includes Ali Hussain Hitwala, Mushahid Alam, Mohit Verma, Ashish Shenoy, Srinivas Tunuguntla, Shruthi Racha, Shreya Kamath, Ekta Sardana, Anshul Purohit, Nivetha Singara Vadivelu, Atasi Panda, Aayushi Jain (batchmates from 2015); Kausik Subramanian, Arjun Singhvi, Ayon Sen, Arjun Balasubramium (badminton group); Swapnil Haria, Sukriti Singh[1], Kshiteej Mahajan, Neha Godwal, Akhil Guliani, and other Winter Soldiers; Ramanathan Alagappan (for his advice and the countless hours of tennis); Rex Fernando, Yifeng Teng, Mark Mansi, Aishwarya Ganesan, Supriya Hirukar. Vinayak Sood, my close friend from undergraduate, helped a lot by listening to me vent. I thank Junaid Khalid and Shoban Chandrabose for being such great friends and helping me when things got tough. Lastly, I thank Amrita Roy Chowdhury for being a close confidant and for her friendship over the years.

I would be remiss if I failed to express my gratitude towards Anja Gruenheid and Stratis Viglas, with whom I had the privilege of working during my internships at Google. I thoroughly enjoyed working with them and learned a lot. I cannot be thankful enough for their help in my job search and for being such great mentors. I would also like to thank Jeff Naughton for the opportunity to work with Google Madison. Jeff was very kind, generous with his time, plentiful in his wisdom, and provided a gentle guiding hand.

Before coming to graduate school, I had the opportunity to work at Goldman Sachs, Bangalore. My learning in the two years I worked there was beneficial in many ways. In

[1]and their beautiful cat Pixie

particular, I thank Ira Patra, Shaurya Vardhan, and Sriraman Seshadri who invested a lot of their time and energy to teach me valuable technical and non-technical skills.

Lastly, I cannot thank enough my parents - Pradeep and Manju Chaudhary; and my sister Anamika Avni. My parents ensured that both I and my sister received the absolute best possible education. Being educators themselves, they provided me with a top-notch academic environment at home that fostered my inquisitiveness and love for learning. They always believed in my decisions and supported me in moments of self-doubt. All of my achievements are a testament to their strong foundational values. I hope I continue to make them proud.

TABLE OF CONTENTS

Page

ABSTRACT viii

1 Introduction 1

1.1 Motivation 4
1.2 Contributions 6
1.3 Organization 10

2 Background 11

2.1 Data Model and Queries 11
2.2 Computational Model 13
2.3 Fast Matrix Multiplication 14
2.4 General Framework 15

3 Compressed Representations of Conjunctive Query Results 16

3.1 Related Work 18
3.2 Problem Statement 20
3.2.1 Some Basic Results 21
3.3 First Main Result 22
3.3.1 The Basic Structure 31
3.3.2 Answering a Query 36
3.4 Second Main Result 39
3.4.1 Constant Delay Enumeration 42
3.4.2 Beyond Constant Delay 43
3.4.2.1 Comparing width notions 48
3.5 The Complexity of Minimizing Delay 48

Page

4 Space-Time Tradeoffs for Answering Boolean Conjunctive Queries . . . 51

4.1 Problem Statement . . . 53
4.2 General Space-Time Tradeoffs . . . 53
4.3 Space-Time Tradeoffs via Tree Decompositions . . . 55
4.4 Extension to CQs with Negation . . . 57
4.5 Path Queries . . . 58
4.5.1 Length-4 Path . . . 58
4.5.2 General Path Queries . . . 59
4.6 Lower Bounds . . . 60

5 Unranked Enumeration of Conjunctive Queries with Projections . . . 63

5.1 Related Work . . . 67
5.2 Main Result . . . 68
5.2.1 Helper Lemmas . . . 69
5.2.2 Star Queries . . . 72
5.2.3 Comparison with Prior Work . . . 73
5.2.4 Warm-up: Two-Path Query . . . 75
5.2.5 Proof of Main Theorem . . . 78
5.2.6 Interleaving with Join Computation . . . 80
5.3 Left-Deep Hierarchical Queries . . . 83
5.4 Path Queries . . . 84

6 Join-Project Query Evaluation using Fast Matrix Multiplication . . . 87

6.1 Computing Join-Project . . . 89
6.1.1 The 2-Path Query . . . 90
6.1.2 The Star Query . . . 94
6.1.3 Boolean Set Intersection . . . 96
6.2 Speeding Up SSJ and SCJ . . . 97

Page

6.3 Cost-Based Optimization . . . 100
6.4 System Implementation . . . 103
6.5 Experimental Evaluation . . . 105
6.5.1 Datasets . . . 105
6.5.2 Simple Join Processing . . . 106
6.5.3 Set Similarity . . . 108
6.5.4 Set Containment . . . 109
6.5.5 Boolean Set Intersection . . . 110

7 Ranked Enumeration of Conjunctive Query Results . . . 111

7.1 Related Work . . . 113
7.2 Ranking Functions . . . 115
7.2.1 Problem Parameters . . . 118
7.3 Main Result . . . 119
7.3.1 Applications . . . 119
7.3.2 The Algorithm for the Main Theorem . . . 122
7.4 Extensions . . . 131
7.4.1 Ranked Enumeration of UCQs . . . 131
7.4.2 Improving The Main Result . . . 132
7.5 Lower Bounds . . . 134
7.5.1 The Choice of Ranking Function . . . 134
7.5.2 Beyond Logarithmic Delay . . . 138

8 Ranked Enumeration of Conjunctive Queries with Projections . . . 139

8.1 Related Work . . . 143
8.2 Preliminaries . . . 144
8.2.1 Ranking Functions . . . 145
8.2.2 Problem Parameters . . . 146

Page

8.3 General acyclic queries . . . 147
8.3.1 General Algorithm . . . 148
8.3.2 Improvement for Lexicographic Ranking . . . 155
8.4 Star Queries . . . 157
8.4.1 The Algorithm . . . 157
8.4.2 Tradeoff Optimality . . . 159
8.5 General queries . . . 160
8.6 Experimental Evaluation . . . 162
8.6.1 Experimental Setup . . . 162
8.6.1.1 Small-Scale Datasets . . . 163
8.6.1.2 Large-Scale Datasets . . . 164
8.6.2 Small Scale Experiments . . . 165
8.6.2.1 Enumeration with Preprocessing . . . 168
8.6.2.2 Cyclic Queries . . . 168
8.6.3 Large Scale Experiments and Scalability . . . 169

9 Conclusions . . . 170

9.1 Future Work . . . 171

LIST OF REFERENCES . . . 174

ABSTRACT

The big data era has fundamentally changed the landscape of data management over the last few years. To process the large amounts of data available to users in both industry and science, many modern data science tools create data analysis pipelines that comprises of several independent tasks where the output of one task becomes the input of another task. In such cases, it is often useful to materialize a subset of the output if some downstream process in the pipeline plans to repeatedly access it. However, current state-of-the-art solutions only perform materialization to optimize each task in isolation, without taking advantage of the higher-level logical structure of these chained components in the pipeline. In this dissertation, we fundamentally rethink how materialization should be done for data management tasks by taking advantage of the information about the access pattern of how the materialized data is used by the downstream tasks, enabling us to make fine-grained decisions.

In the first part of this dissertation, we study the construction of space-efficient compressed representations of the output of conjunctive query (select-project-join queries) results, with the goal of supporting the efficient access of the intermediate compressed result for a given access pattern. In particular, we study an important trade-off: minimizing the *space* necessary to store the compressed result, versus minimizing the *delay* (time difference between any two consecutive answers) for an access request over the result, and consequently the *total answer time*. We construct a novel parameterized data structure, which can be tuned to trade-off space for answer time. The trade-off allows us to control the space requirement of the data structure precisely and depends both on the structure of the query and the access pattern. Among our results, we also refute two popular conjectures made in the existing literature by constructing unexpectedly better algorithms for popular data structure problems.

In most real-world applications, join queries almost always come with projections in the query. This makes query execution challenging because multiple join answers in the absence of a projection operator may map to the same output tuple after the projection operator is applied under set semantics. In the second part of this dissertation, we undertake the challenge of building efficient data structures with low materialization overhead and novel enumeration algorithms for two important classes of join-project queries, star and path queries. Further, we also show how fast matrix multiplication can be used to improve the join processing time for these queries, both in theory and practice.

Join query tasks may also have an ordering imposed on the output. For interactive use-cases, users may also be interested in only the first few results. In the last part of this dissertation, we show that for join queries (possibly with projections) and a large class of ranking functions frequently used in practice, there exist efficient algorithms that require only linear time preprocessing and minimal materialization before they start enumerating the result with a small delay. We also complement our algorithms with nearly matching lower bounds. This thoroughly resolves an open problem stated in Dagstuhl Seminar 19211 on ranked enumeration. Finally, a comprehensive experimental evaluation demonstrates that the idea of using a delay-based formulation for join processing can lead to orders of magnitude performance improvement, both in terms of execution time and space requirement, over popular relational and graph processing engines.

Chapter 1

Introduction

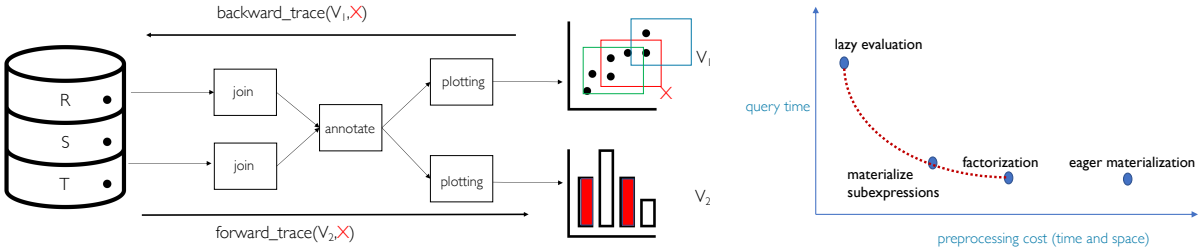
Our society today is becoming increasingly data-driven. Decades of progress and breakthroughs in the database community have led to data processing, storage, and analysis at an astonishing scale. This has fundamentally transformed scientific research across a variety of fields (ranging from astronomy and economics to biology and oceanography), business analytics, social networks, journalism, and electronic commerce. In order to extract value from the large amounts of data available to users in both industry and science, the collected data is typically moved through a pipeline of several independent data processing tasks. For example, consider a practical visualization pipeline [PW18] as shown in Figure 1.1a that takes in relational tables, performs a join task whose output is fed into an annotator to perform labeling of tuples for recording provenance, and finally, a plotting library generates two visualizations. As the user interacts via brushing with the chart V_1 (on top), the highlighted points in box X are then highlighted in chart V_2. Relational data management systems and more recently, data science workflow tools, have become the de-facto workhorse for performing these data processing tasks at scale. Thanks to the declarative power of DBMS, the user only needs to specify the task in a high-level language such as SQL without worrying about how the task will be accomplished. The actual execution is a resource allocation problem for the engine which will decide how to use resources available to it or provision more resources for itself as is common in cloud environments.

However, the growth in data and its processing needs is at a point where it is no longer enough to throw more resources – it is critical to design optimizations that make data processing faster and resource-efficient. This goal was also identified by database practitioners where systems such as Quickstep [PDZ+18, ZDP19, DSP20, SDKN20] and HyPer [KNF+12] aim to extract the maximum performance out of each core. One such

fundamental optimization that is used in database management systems is that of *materialization*[1]. Materialization (also referred to as intermediate results) is a natural embodiment of the ideas of precomputation and caching in databases. The ability of materialized views to speed up queries benefits nearly all database applications [CY+11]. Because of their wide applicability, materialized views are a well-studied topic with both a rich research literature and mature commercial implementations. Indeed, selective materialization of data for analytics applications has been intensely studied by the database community [HSK98, CYZ+08, PKZT01, LDH+08, SHK00, LPH02, ZLXH11, LHG04]. However, most of the materialization techniques known so far (even in modern data management tools like Weld [PTS+17] and Spark [ZCF+10]) operate at a low level and fail to exploit the higher-level logical structure of the tasks. In the OLAP setting, materialization of data cubes over large datasets may require TBs of main memory, and even materializing small sub-cubes can exhaust the main memory very quickly. If the working set is too large, then expensive I/O operations are involved, which in turn negatively impacts the runtime performance. Different applications in the real world have different requirements for space and runtime. For space-bound applications such as graph analytics and pattern retrieval, the objective is to minimize space, while for interactive applications (e.g., visualization) it is critical to optimize for the answer time of each request. As an example, when the user highlights points in chart V_1 in Figure 1.1a, it is necessary to perform a fast backward trace that identifies the contributing tuples (using the provenance) to all points in X and then send this information to chart V_2 for highlighting. Hence, materialization algorithms should allow for a smooth trade-off space and preprocessing time for answer time to achieve optimal performance.

The motivating question of this dissertation is the following: *Can we develop context-aware optimal materialization algorithms that can achieve optimal performance for data analytics applications even under space budget constraints?* We hypothesize that just as how indexes built on base relations are useful for optimized access to service data access patterns (such as equality, range, or band predicates) in join query processing, lifting the same principle to materialization and building indexes optimized for its specific access pattern should also be beneficial. In this dissertation, we confirm this hypothesis by designing, implementing, and evaluating new materialization techniques using three novel ideas: (*i*) task-aware materialization, i.e., taking into account the context of how the materialized output will be used, (*ii*) fine-grained decisions on what and how to materialize, and (*iii*) multiple design points that trade-off space for time for optimal performance of data analytics pipelines. We design novel algorithms and techniques for query processing by making use of limited space, with a particular focus on the task of join processing, both ranked and unranked. Existing

[1]While materialization is also used in optimizing the performance of queries by exploiting intra-query parallelism, this dissertation solely focuses on the benefit of materialization across multiple join queries (i.e. the inter-query aspect).



(a) Example pipeline that processes a relational dataset and generates a visualization (adapted from [PW18]).

(b) Comparison of existing techniques (solid dots) and our proposal (dashed line)

Figure 1.1: Data analysis pipeline example (left); comparing our proposal with prior work (right)

solutions [XD17a, XKD15, XSD17] follow a more coarse-grained approach, where the whole output (or a set of subqueries in the query plan) is fully materialized. We set ourselves apart from prior works by constructing materializations that are controlled in a fine-grained manner – we decide what part of the output we materialize and how we materialize by examining the downstream access pattern. For example, for the problem of linked brushing in Figure 1.1a, the downstream access pattern is fast identification of contributing tuples in charts. In this case, it would be beneficial to create an index on top of the join result that allows us to perform fast provenance computation. Building upon the foundation of how to do fine-grained materialization, we make significant progress in furthering our understanding of efficient enumeration algorithms (both ranked and unranked) for important classes of queries that are commonly seen in practice: *star* and *path* queries which have widespread applications in problems such as similarity search [YSN+12], citation graph analysis [RT05], and network analysis [EL05, Bir08]. For certain join queries, it is the case that improved running time guarantees are only achievable by building a more expensive data structure ahead of time. Our ideas from fine-grained materialization are directly applicable here and are instrumental to the discovery of improved algorithms.

In the following sections of this chapter, we will delve deeper into the need for rethinking materialization in join query processing, followed by the contributions made in this dissertation.

1.1 Motivation

Central to this era of big data is the data-to-knowledge pipeline, which consumes data in raw format, and then cleans, transforms, integrates, stores, processes, and analyzes the data to extract knowledge in a usable and easily digestible format. Data analytics pipelines are complex, and their workflows typically consist of several simpler tasks – such as relational queries, graph algorithms, learning and inference tasks, visualizations, user inputs – chained together. To speed up such a pipeline, in addition to optimizing each component separately, it is critical to apply optimizations that span multiple tasks.

A ubiquitous optimization technique is to materialize the intermediate result of a task (typically a relational query), so that downstream tasks in the pipeline can access the intermediate data efficiently. This is critical when the downstream task needs to access the result multiple times. This pattern occurs in a wide spectrum of applications spanning graph analytics [XD17a], visualization [PW18], and statistical inference [NZRS]. For example, a feature generation query will materialize all feature vectors for a learning or inference task; in data mining applications on social networks, the full graph is materialized before any graph algorithm is executed. Let us now look at a simple running example that will be useful for illustration.

Example 1. *Consider a data scientist Alice who wants to perform analysis on the co-authors in the DBLP dataset. The dataset contains information about which authors write which papers through a table $R(author, paper)$ of size N, where a tuple (a, b) denotes that author* a *has written paper* b*. To analyze the relationships between co-authors, the pipeline first extracts the co-author graph, which can be expressed as the query $Q_1(x, y, p) = R(x, p), R(y, p)$. Then, we can run any graph algorithm on the extracted graph; such algorithms typically access the graph through an API. Once the graph is extracted, multiple tasks can now be accomplished. For instance, Alice may ask to find all papers that David DeWitt and Mike Stonebreaker have co-authored together using the query $Q_1($"David Dewitt", "Mike Stonebreaker", $p)$. Alice can now change the values of x and y to other authors and create a workload of queries that need to be answered.*

Eagerly materializing and indexing the full result of a query has several drawbacks for data analytics applications. First, the output result can be extremely large, and thus materialization can be prohibitively expensive in terms of storage. Even when the input data is in the order of GBs, the memory overhead of full materialization can be in the order of TBs. Second, materialization can be inefficient in terms of runtime, which would be prohibitive in applications that require interactive responses from the system (e.g., in interactive visualization). Third, part of the materialized data may never be used by the downstream task, resulting in unnecessary computation and space.

Alternative to eager materialization, we can serve each request of the downstream task by being lazy, and executing the pipeline only for the particular part of the data that we may be interested in. This solution is at the other end of the spectrum, since it materializes nothing (so it is space-efficient), but can lead to runtime inefficiencies when the same computation has to be repeated multiple times. A third approach uses factorization techniques [OZ15a, OS16] to compress the full result, but in many cases offers no benefit to naïve materialization (or even if it does, the space requirement is still prohibitively large). These existing solutions are highlighted in Figure 1.1b.

Example 2. *Continuing Example 1, it is easy to see that eager materialization of the view* $Q_1(x, y, p)$ *may take as much as* $\Omega(N^2)$ *space since a large subset of authors could be on a single paper. On the other hand, lazy materialization does nothing and in this case,* $Q_1($*"David Dewitt"*, *"Mike Stonebreaker"*$, p)$ *can take as much as* $\Omega(N)$ *time to find out whether the answer is true or false. For this query, factorization will use the variable order* x, y, p *and materialize the view* $V(x, y, p) = R(x, y), R(y, p)$ *which is as expensive as eager materialization. Observe that the number of possible queries that Alice could ask here is* N^2*. If the space is limited, OLAP approaches perform selective materialization of the output of some of these queries using knapsack style algorithms by taking into account the most popular queries in the workload. However, even in this approach, the worst-case running time* $\Omega(N)$ *for some queries remains a bottleneck as there is not enough space to store the answer for all possible queries.*

The right materialization strategy also depends on the task at hand. For example, a join query containing `ORDER BY` requires a different set of indexes to be built for optimal evaluation compared to the same join query without any ordering condition imposed. Similarly, the structure of the query, as well as the data, plays a crucial role in characterizing which tasks can admit efficient evaluation.

Since we analyze the problem from a theoretical angle as well, let us look at the metric for what constitutes an efficient algorithm. Enumeration complexity is used as a yardstick to identify whether the answers produced by evaluating a query can be done efficiently or not. When it comes to query answering, it is common to use data complexity. We treat every query as fixed, and we identify it with the following enumeration problem: given a database and query as input, create a data structure in a *preprocessing phase*, and find all answers to the query over the given database in the *enumeration phase* using the built data structure. The best time guarantee we can hope for is to output all answers with a constant delay between consecutive answers after a linear preprocessing phase. This is the time it takes to read the database and then write the answers one by one.

The most commonly studied class of queries is the Conjunctive Queries (CQs) which consist of join queries followed by projection. Introducing projection increases the difficulty

of answering queries in constant delay. Two different join answers may become identical after projection. Since we do not allow to output duplicates, this reduces the total number of answers, and so we allow the algorithm less time in total to perform the join. In practice, it is often required that the output of a CQ is also ranked in a particular order rather than being enumerated in some arbitrary ordering. For interactive applications, a user may also specify that they are only interested in, say, top 100 tuples. As we saw before, SQL allows expressing these constraints using ORDER BY and LIMIT clause. Once again, the challenge in this setting is to identify how to construct data structures with low materialization overhead in the preprocessing phase and achieve a small delay when enumeration begins. As we will see in the chapters that follow, achieving small delay is not only theoretically interesting but also practically relevant. Let us look at a concrete example in this setting.

Example 3. *Consider the same setting as Example 1 with DBLP relation $R(A, P)$, suppose that given an author a, the function h-index(a) returns the h-index of a. Alice wishes to find all co-authors pairs who have authored at least one paper together. This task can be expressed in SQL by the following query.*

$$Q_2 = \texttt{SELECT DISTINCT } R_1.A, R_2.A \texttt{ FROM } R \texttt{ AS } R_1, \ R \texttt{ AS } R_2 \texttt{ WHERE } R_1.P = R_2.P;$$

Suppose that Alice also wants the pairs of authors returned in decreasing order of the sum of their h-indexes, since she is only interested in the top-100 most influential answers. The following SQL query captures this task.

$$Q_3 = \texttt{SELECT DISTINCT } R_1.A, R_2.A \texttt{ FROM } R \texttt{ AS } R_1, \ R \texttt{ AS } R_2 \texttt{ WHERE } R_1.P = R_2.P \texttt{ ORDER BY h\_index}(R_1.A) + \texttt{h\_index}(R_2.A) \texttt{ LIMIT 100;}$$

The state-of-the-art algorithm can evaluate Q_2 with linear delay after linear preprocessing [KNOZ20a] and Q_3 with delay $O(\log N)$ after $O(N^2)$ preprocessing.

1.2 Contributions

In this section, we give an overview of the different settings considered in this dissertation and discuss our contribution in each setting.

Compressed Representations and Space-time trade-offs. A join query Q over a database D can be accessed by assigning attribute values to certain attributes in the query. Such attributes are called *bound* attributes. The remaining attributes in the query are *free* and the goal is to evaluate the query after specifying the bindings. This concept of bound and free attributes can be used to model access patterns over any query.

Example 4. *The task of finding all co-authors of a given author from Example 1 can be formalized through an* adorned query $Q_1^{\mathsf{bbf}}(x, y, p) = R(x, p), R(y, p)$. *The above formalism says that the query Q_1 will be accessed as follows: given values for the* bound (b) *attributes x, y, we have to return the values for the* free (f) *attribute p such that the tuple is in the query Q. The sequence* bbf *is called the* access pattern *for the adorned query.*

Given an adorned query Q^η, database D, we show that it is possible to develop a data structure that can trade-off the delay guarantees and total running time of a query with the space usage of the data structure. The space usage is parameterized by a quantity τ that can be used as a knob and choose any point in the full continuum of space usage of lazy materialization $O(|D|)$ to eager materialization $O(|D|^{\texttt{fhw}})$ where `fhw` is the fractional hypertree width of the query. The delay guarantee obtained by the algorithm is $O(\tau \log |D|)$. In conjunction with tree decompositions [BDG07a], the space usage of the data structure can be shown to be conditionally optimal for several queries of practical interest. Figure 1.1b shows this proposal as a dashed line that recovers the existing approaches as special cases.

Example 5. *Using our results, it can be shown that for any $1 \leq \tau \leq N$, there exists a data structure of size $S = O(N^2 \log N/\tau^2)$ that can answer $Q_1^{\mathsf{bbf}}(x, y, p)$ in time $O(\tau \log |D|)$, a tunable trade-off in sharp contrast to all strategies outlined in Example 2.*

Next, using the framework introduced, we show that it is possible to go one step further. An important class of queries that are commonly seen in practice is that of Boolean adorned CQs. For example, consider the 2-Set Disjointness problem: given a universe of elements U and a collection of m sets $C_1, \ldots, C_m \subseteq U$, we want to create a data structure such that for any pair of integers $1 \leq i, j \leq m$, we can efficiently decide whether $C_i \cap C_j$ is empty or not. Previous work [CP10b, GKLP17] has shown that the space-time trade-off for 2-Set Disjointness is captured by the equation $S \cdot T^2 = N^2$, where N is the total size of all sets. The data structure obtained is conjectured to be optimal [GKLP17], and its optimality was used to develop conditional lower bounds for other problems, such as approximate distance oracles [AGHP11, Aga14]. Similar trade-offs have been independently established for other data structure problems as well (k-Reachability [GKLP17, CP10a] and edge triangle detection problem [GKLP17]). We show that a unified framework can capture several widely-studied data structure problems in the algorithmic community and recover prior results using a single algorithm. Remarkably, we also explicitly improve the best-known space-time trade-off for the k-Reachability problem for $k \geq 4$. To the best of our knowledge, this is the first non-trivial improvement for the k-Reachability problem. We also refute a lower bound conjecture for the edge triangles detection problem established by [GKLP17].

Enumeration of Join-Project Queries. There has been a long line of work that has studied the problem of obtaining optimal delay guarantees using minimal preprocessing.

While the story for enumerating full acyclic CQ results is relatively complete, the same is not true for general CQs, even for acyclic CQs with projections. For instance, consider the non self-join version of query Q_2 from Example 3: $Q_{\texttt{two-path}} = \pi_{x,z}(R(x,y) \bowtie S(y,z))$, which joins two binary relations and then projects out the join attribute. For this query, [BDG07a] 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.

We show that for the important class of *star queries* and *path queries*, there exist output-sensitive enumeration algorithms that require only linear time preprocessing. Seminal work by Kara et al. [KNOZ20a] showed that for any hierarchical CQ (hierarchical CQs are a strict subset of acyclic CQs), possibly with projections, there always exists a smooth trade-off between preprocessing time and delay. This is the first improvement over the results of Bagan et al. [BDG07a] in over a decade for queries involving projections. Applied to the query $Q_{\texttt{two-path}}$, the main result of of [KNOZ20a] shows that for any $\epsilon \in [0,1]$, we can obtain $O(|D|^{1-\epsilon})$ delay with $O(|D|^{1+\epsilon})$ preprocessing time. Our results show that this trade-off is redundant for star queries – either we have enough preprocessing time to materialize the output of the query and achieve constant delay, or we can achieve the desirable delay with only linear preprocessing time. We show that there exists an algorithm that can always match the linear delay guarantee of linear preprocessing. However, depending on the database instance, it is possible to obtain a sub-linear delay guarantee after only linear preprocessing but the result of [KNOZ20a] can only provide $\Omega(|D|)$ delay. We also identify another subset of hierarchical queries that we call *left-deep* where we can get improved delay guarantees in linear time, but only for certain values of delay. Finally, using fast matrix multiplication, we show how we can improve the trade-off between preprocessing time and delay when compared to [KNOZ20a].

Traditionally, fast matrix multiplication has been thought of as a technique only of theoretical interest due to the large constant involved in the algorithm. However, recent advancement in processor architecture and the advent of GPUs has enabled the development of fast math-kernel libraries that have dramatically improved the performance of matrix multiplication. One could view this improvement as a potentially smaller ω, the matrix multiplication exponent. We investigate how matrix multiplication can be used to improve the big Oh running time complexity of star queries. We do this by fixing an error and generalizing the analysis from prior work [AP09], and show how fast matrix multiplication can be applied to important problems such as set similarity and set containment. Finally, we also undertake an experimental investigation to understand the practical benefits. Our experiments indicate that using the math-kernel libraries, orders of magnitude performance improvement

is possible when the input dataset contains a dense component, a condition that is easy to identify by examining the dataset.

Ranked Enumeration of Full and Join-Project Queries. For many data processing applications, enumerating query results according to an order given by a ranking function is a fundamental task. For example, [YAG+18, CLZ+15] consider a setting where users want to extract the top patterns from an edge-weighted graph, where the rank of each pattern is the sum of the weights of the edges in the pattern. Ranked enumeration also occurs in `SQL` queries with an `ORDER BY` clause [QCS07, ISA+04]. In the above scenarios, the user often wants to see the first k results in the query as quickly as possible, but the value of k may not be predetermined. Hence, it is critical to construct algorithms that can output the first tuple of the result as fast as possible, and then output the next tuple in the order with a very small delay. Our main contribution is a novel algorithm that uses query decomposition techniques in conjunction with the structure of the ranking function. The preprocessing phase sets up priority queues that maintain partial tuples at each node of the decomposition. During the enumeration phase, the algorithm materializes the output of the subquery formed by the subtree rooted at each node of the decomposition *on-the-fly*, in sorted order according to the ranking function. To define the rank of the partial tuples, we require that the ranking function can be *decomposed* with respect to the particular decomposition at hand. We show that with $O(|D|^{\texttt{fhw}})$ preprocessing time, we can enumerate with delay $O(\log |D|)$. We then discuss how to apply our main result to commonly used classes of ranking functions. Our work thoroughly resolves an open problem stated at the Dagstuhl Seminar 19211 [BKPS19] on ranked enumeration (see Question 4.6). We also show how to extend our algorithm can be applied to enumerating full UCQs and complement our upper bounds with nearly matching lower bounds.

Extending our results to join-project queries is a challenging task. As we discussed before, adding projections can lead to two different join answers obtained without using projections map to the same result, which makes obtaining delay guarantees difficult. In fact, as prior work [TGR21a, TGR20] remarked, only projections that form a *free-connex* structure [BDG07a] can be handled; if any other projection is involved, the algorithm degenerates to the trivial strategy of materializing the full result and then sorting. However, this conversion requires an expensive materialization step. On the practical side, all RDBMS and graph processing engines evaluate join-project queries in the presence of ranking functions by performing three operations in serial order: (*i*) materializing the result of the full join query, (*ii*) de-duplicating the query result (since the query has `DISTINCT` clause), and (*iii*) sorting the de-duplicated result according to the ranking function. The first step in this process is a show-stopper due to the prohibitive cost of materialization. Further, even if the user is interested in the top-ranked tuple (`LIMIT 1`), the engines would still perform

the entire materialization. Our first main result shows that for any *acyclic* query (the most common fragment of queries in practice [BMT20]) with arbitrary projection attributes, it is possible to develop efficient enumeration algorithms. Specifically, it is possible to obtain $O(|D| \log |D|)$ delay after only $O(|D|)$ preprocessing. This implies that query Q_3 from Example 3 can be enumerated with near-linear delay after linear preprocessing. For the class of star queries, we show that there exists a smooth trade-off between preprocessing time and delay guarantees. Finally, we compare the practical performance of our algorithm against state-of-the-art relational and graph processing engines. We observe that our algorithms have orders of magnitude smaller memory footprint and execution time. Even for queries containing unions and cycles, we still maintain a performance improvement over the existing engines. Our results demonstrate that delay-based algorithms are not only optimal in theory but also in practice, an observation that we hope leads to delay being a first-class citizen in all data processing engines.

1.3 Organization

We begin this dissertation by providing some background and terminology, along with the exposition of some technical tools, in Chapter 2. In Chapter 3, we present our results on how to construct a space-efficient representation of the output of CQ results. Building on those results, we show in Chapter 4 how multiple data structure problems can be cast as answering CQs over relational databases. We also show new results and prove lower bounds for some of those problems. Chapter 5 is dedicated to the study of unranked enumeration of star and path queries containing projections. Using fast matrix multiplication, we show in Chapter 6 how the improved execution time for join query processing can also benefit important problems such as set similarity and set containment. Chapter 7 and Chapter 8 study the problem of ranked enumeration for full and non-full CQs. We finally conclude in Chapter 9.

Chapter 2

Background

In this chapter, we present the basic notions and terminology that are necessary for the reader to follow this dissertation. We present in detail the class of *conjunctive queries*, which are the queries that this dissertation focuses on. We then lay some notation and definitions that are common across all the chapters to follow.

2.1 Data Model and Queries

Data Model. A schema $\mathbf{x} = (x_1, \ldots, x_n)$ is a non-empty ordered set of distinct variables. Each variable x_i has a discrete domain $\mathbf{dom}(x_i)$. A tuple t over schema $\mathbf{x}$ is an element from $\mathbf{dom}(\mathbf{x}) = \mathbf{dom}(x_1) \times \cdots \times \mathbf{dom}(x_n)$. A relation R over schema $\mathbf{x}$ (denoted $R(\mathbf{x})$) is a function $R : \mathbf{dom}(\mathbf{x}) \to \mathbb{Z}$ such that the multiplicity $R(t)$ is non-zero for finitely many t. A tuple t exists in R, denoted by $t \in R$, if $R(t) > 0$. The size of relation R, denoted as $|R|$, is the size of set $\{t \mid t \in R\}$. A database D is a set of relations and the size of the database $|D|$ is the sum of sizes of all its relations. Given a tuple t over schema $\mathbf{x}$ and a set of variables $\mathbf{s} \subseteq \mathbf{x}$, $t[\mathbf{s}]$ denotes the restriction of t to $\mathbf{s}$ and the values of $t[\mathbf{s}]$ follows the same variable ordering as $\mathbf{s}$. We also define the *selection* operator $\sigma_{\mathbf{s}=t}(R) = \{u \in R \mid u[\mathbf{s}] = t\}$ and *projection* operator $\pi_{\mathbf{s}}(R) = \{u[\mathbf{s}] \mid u \in R\}$.

Queries. In this dissertation, we focus on the class of *conjunctive queries* (CQs), which are expressed as

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

Here, the symbols $\mathbf{y}, \mathbf{x}_1, \ldots, \mathbf{x}_n$ are vectors that contain *variables* or *constants*, the atom $Q(\mathbf{y})$ is the *head* of the query, and the atoms $R_1(\mathbf{x}_1), R_2(\mathbf{x}_2), \ldots, R_n(\mathbf{x}_n)$ form the *body*. The variables in the head are a subset of the variables that appear in the body. A CQ is *full* if every variable in the body appears also in the head, and it is *boolean* if the head contains no variables, *i.e.* it is of the form $Q()$. We typically use the symbols $x, y, z, \ldots$ to denote variables, and $a, b, c, \ldots$ to denote constants. We use $\mathsf{vars}(Q)$ to denote the set of all variables in Q, i.e., $\mathbf{y} \cup \mathbf{x}_1 \cup \cdots \cup \mathbf{x}_n$. If D is an input database, we denote by $Q(D)$ the result of

running Q over D. A CQ with negation, denoted as $CQ^\neg$, is a CQ where some of the atoms can be negative, i.e., $\neg R_i(\mathbf{x}_i)$ is allowed. For $\varphi \in CQ^\neg$, we denote by φ^+ the conjunction of the positive atoms in φ. A $CQ^\neg$ is said to be *safe* if every variable appears in at least some positive atom. In this dissertation, we restrict our scope to the class of safe $CQ^\neg$, a standard assumption [WL03, NL04] ensuring that query results are well-defined and do not depend on domains.

We are 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.

A *Union of Conjunctive Queries* $\varphi = \bigcup_{i\in\{1,\ldots,\ell\}} \varphi_i$ is a set of CQs where $\text{head}(\varphi_{i_1}) = \text{head}(\varphi_{i_2})$ for all $1 \le i_1, i_2 \le \ell$. Semantically, $\varphi(D) = \bigcup_{i\in\{1,\ldots,\ell\}} \varphi_i(D)$. A UCQ is said to be full if each φ_i is full.

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 [SORK11]. For example, Q_k^* is hierarchical for any k, while P_k is hierarchical only when $k \le 2$.

Natural Joins. If a CQ is full, has no constants and no repeated variables in the same atom, then we say it is a *natural join query*. For instance, the triangle query $\Delta(x, y, z) = R(x, y), S(y, z), T(z, x)$ is a natural join query. A natural join can be represented equivalently as a *hypergraph* $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of variables, and for each hyperedge $F \in \mathcal{E}$ there exists a relation R_F with variables F. We write the join as $\bowtie_{F\in\mathcal{E}} R_F$. The size of relation R_F is denoted by $|R_F|$. Given a set of variables $I \subseteq \mathcal{V}$, we define $\mathcal{E}_I = \{F \in \mathcal{E} \mid F \cap I \neq \emptyset\}$.

Valuations. A *valuation* v over a subset V of the variables is a total function that maps each variable $x \in V$ to a value $v(x) \in \mathbf{dom}(x_i)$. Given a valuation v of the variables $(x_{i_1}, \ldots, x_{i_\ell})$, we denote $R_F(v) = R_F \ltimes \{v(x_{i_1}, \ldots, x_{i_\ell})\}$.

Query Evaluation. An *answer* is a tuple $t(\mathsf{vars}(Q))$ which is a mapping from $\mathsf{vars}(Q)$ to $\mathbf{dom}$ such that $t[\mathbf{x}_i] \in R_i$. $Q(D)$ is defined as the set of all answers. Evaluating a query Q means computing the set of answers $Q(D)$ given a database instance D.

CQ Hypergraph. We associate a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ to a CQ Q where the vertices are the variables of Q, and every hyperedge is a set of variables occurring in a single atom of Q. In other words, $\mathcal{E} = \{\{v_1, \ldots, v_n\} \mid R_i(v_1, \ldots, v_n) \in \mathsf{atoms}(Q)\}$.

Join Size Bounds. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph, and $S \subseteq \mathcal{V}$. A weight assignment $\mathbf{u} = (u_F)_{F \in \mathcal{E}}$ is called a *fractional edge cover* of S if (i) for every $F \in \mathcal{E}, u_F \geq 0$ and (ii) for every $x \in S, \sum_{F: x \in F} u_F \geq 1$. The *fractional edge cover number* of S, denoted by $\rho^*_{\mathcal{H}}(S)$ is the minimum of $\sum_{F \in \mathcal{E}} u_F$ over all fractional edge covers of S. We write $\rho^*(\mathcal{H}) = \rho^*_{\mathcal{H}}(\mathcal{V})$.

In a celebrated result, Atserias, Grohe,To and Marx [AGM13] proved that for every fractional edge cover $\mathbf{u}$ of $\mathcal{V}$, the size of a natural join is bounded using the following inequality, known as the *AGM inequality*:

$$| \bowtie_{F \in \mathcal{E}} R_F | \leq \prod_{F \in \mathcal{E}} |R_F|^{u_F} \tag{2.1}$$

The above bound is constructive [NRR13, NPRR12]: there exist worst-case algorithms that compute the join $\bowtie_{F \in \mathcal{E}} R_F$ in time $O(\prod_{F \in \mathcal{E}} |R_F|^{u_F})$ for every fractional edge cover $\mathbf{u}$ of $\mathcal{V}$.

Tree Decompositions. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph of a natural join query Q. A *tree decomposition* of $\mathcal{H}$ is a tuple $(\mathcal{T}, (\mathcal{B}_t)_{t \in V(\mathcal{T})})$ where $\mathcal{T}$ is a tree, and every $\mathcal{B}_t$ is a subset of $\mathcal{V}$, called the *bag* of t, such that

1. each edge in $\mathcal{E}$ is contained in some bag $\mathcal{B}_t$; and
2. for each $x \in \mathcal{V}$, the set of nodes $\{t \mid x \in \mathcal{B}_t\}$ is connected in $\mathcal{T}$.

The *fractional hypertree width* of a tree decomposition is defined as $\max_{t \in V(\mathcal{T})} \rho^*(\mathcal{B}_t)$, where $\rho^*(\mathcal{B}_t)$ is the minimum fractional edge cover of the vertices in $\mathcal{B}_t$. The fractional hypertree width of a query Q, denoted $\texttt{fhw}(Q)$, is the minimum fractional hypertree width among all tree decompositions of its hypergraph. A query Q is called acyclic if and only if it has $\texttt{fhw}(Q) = 1$.

Submodular Width. Submodular width, $\texttt{subw}(Q)$, of a query allows us to find data-dependent tree decompositions. The notion of submodular width was introduced in [Mar13]. To present its definition, we need the following terminology. A function $g : 2^{\mathsf{vars}(Q)} \to \mathbb{R}_{\geq 0}$ is (i) monotone, if $g(U) \leq g(V)$ for all $U \subseteq V \subseteq \mathsf{vars}(Q)$, (ii) edge-dominated, if $g(\mathsf{vars}(\alpha)) \leq 1$ for every $\alpha \in \mathsf{atoms}(Q)$, (iii) submodular, if $g(U) + g(V) \geq g(U \cap V) + g(U \cup V)$ for every $U, V \subseteq \mathsf{vars}(Q)$. We denote by $\mathbf{s}(Q)$ the set of all edge-dominated, submodular functions $g : 2^{\mathsf{vars}(Q)} \to \mathbb{R}_{\geq 0}$ that satisfy $g(\emptyset) = 0$, and by $\mathcal{T}$ the set of all tree decompositions of Q. The submodular width of a conjunctive query Q is $\texttt{subw}(Q) = \sup_{g \in \mathbf{s}(Q)} \sup_{(\mathcal{T}, \mathcal{B}_t) \in \mathcal{T}(Q)} \max_{t \in V(\mathcal{T})} g(\mathcal{B}_t)$. It is known that $\texttt{subw}(Q) \leq \texttt{fhw}(Q)$ for every CQ Q.

2.2 Computational Model

Input. Using data complexity for most of our problems, the input is measured only by the size of the database instance D (the query and the schema are treated as fixed). We assume

the input database is given by the reasonable encoding suggested by Flum et al. [FFG02]. When we say linear time, we mean that the number of operations is $O(|D|)$.

Cost Measure. We consider the RAM model of computation. In particular, adding, multiplying, and comparing integers that are polynomial in the cardinality of the input can be done in constant time. RAM model with uniform-cost measure can retrieve the content of any register via its unique address in constant time.

Hash Tables. Each relation (or materialized view) R is implemented by a data structure that stores key-value entries $(\mathbf{x}, R(\mathbf{x}))$ for each tuple $\mathbf{x}, R(\mathbf{x}) \neq 0$ and needs $O(|R|)$ space. This data structure can: (1) look up, insert, and delete entries in constant time, (2) enumerate all stored entries in R with constant delay, and (3) report $|R|$ in constant time.

2.3 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. The following folklore lemma about matrix multiplication is frequently used in the chapters that follow.

Lemma 1. *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 1, 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 $\omega = 2.3729$ [GU18], but it is believed that the actual value is 2.

Adorned Views. In order to model access patterns over a view Q defined over the input database, we use the concept of *adorned views* [Ull85]. In an adorned view, each variable in the head of the view definition is associated with a binding type, which can be either *bound* (b) or *free* (f). A view $Q(x_1, \ldots, x_k)$ is then written as $Q^\eta(x_1, \ldots, x_k)$, where $\eta \in \{\mathsf{b}, \mathsf{f}\}^k$ is called the *access pattern*. We denote by $\mathcal{V}_\mathsf{b}$ (resp. $\mathcal{V}_\mathsf{f}$) the set of bound (resp. free) variables from $\{x_1, \ldots, x_k\}$. We can interpret an adorned view as a function that maps a valuation over the bound variables $\mathcal{V}_\mathsf{b}$ to a relation over the free variables $\mathcal{V}_\mathsf{f}$. In other words, for each valuation v over $\mathcal{V}_\mathsf{b}$, the adorned view returns the answer for the query $Q^\eta[v] = \{\mathcal{V}_\mathsf{f} \mid Q(x_1, \ldots, x_k) \wedge \forall x_i \in \mathcal{V}_\mathsf{b} : x_i = v(x_i)\}$, which we also refer to as an *access request*.

Example 6. $\Delta^{\mathsf{bbf}}(x, y, z) = R(x, y),\ S(y, z), T(z, x)$ *captures the following access pattern: given values* $x = a, y = b$*, list all the* z*-values that form a triangle with the edge* $R(a, b)$*. As another example,* $\Delta^{\mathsf{fff}}(x, y, z) = R(x, y), S(y, z), T(z, x)$ *simply captures the case where we want to perform a full enumeration of all the triangles in the result. Finally,* $\Delta^{\mathsf{b}}(x) = R(x, y), S(y, z), T(z, x)$ *expresses the access pattern where given a node with* $x = a$*, we want to know whether there exists a triangle that contains it or not.*

An adorned view $Q^{\eta}(x_1, \ldots, x_k)$ is *boolean* if every head variable is bound, it is *non-parametric* if every head variable is free, and it is *full* if the CQ if full (*i.e.*, every variable in the body also appears in the head). Of particular interest is the adorned view that is full and non-parametric, which we call the *full enumeration view*, and simply asks to output the whole result.

2.4 General Framework

Throughout this dissertation, we study multiple problems in the enumeration framework similar to that of [Seg15a] where the algorithm can be decomposed into two phases:

1. **Preprocessing Phase**: This phase takes T_p time to compute the data structure $\mathsf{C}_Q(D)$ during the preprocessing phase.
2. **Enumeration Phase**: This phase outputs $Q(D)$ (possibly in some specific order) with no repetitions. The enumeration phase has full access to data structure $C_Q(D)$ constructed in the preprocessing phase and can also use additional space, if necessary. The goal is to minimize the answering time of the query. We measure the answer time in two different ways.
   (a) **delay** (δ): the maximum time to output any two consecutive tuples (and also the time to output the first tuple, and the time to notify that the enumeration has completed).
   (b) **total answer time** (T_A): the total time to output the result.

We now discuss the rationale for why such a framework is useful. Computing the first answer requires at least linear time (to read the input and decide whether an answer exists), but sometimes we can achieve a smaller delay between the subsequent answers. For this reason, we separate the requirement regarding the time before the first answer from that of the following answers. An enumeration algorithm $\mathcal{A}$ is given an input D and Q, and it may build data structures during the preprocessing phase. During the enumeration phase, $\mathcal{A}$ can access the data structures built during preprocessing, and it emits the answers $Q(D)$, one by one, without repetitions. Note that we do not impose a restriction on the memory used. In particular, such an algorithm may use additional constant memory for writing between two consecutive answers.

Chapter 3

Compressed Representations of Conjunctive Query Results

In this chapter, we study the problem of constructing space-efficient *compressed representations* of the output of conjunctive query results, with the goal of efficiently supporting a given access pattern directly over the compressed result, instead of the original input database. In many data management tasks, the data processing pipeline repeatedly accesses the result of a conjunctive query (CQ) using a particular access pattern. In the simplest case, this access pattern can be to enumerate the full result (*e.g.*, in a multi-query optimization context). Generally, the access pattern can specify, or *bound*, the values of some variables, and ask to enumerate the values of the remaining variables that satisfy the query.

Currently, there are two extremal solutions for this problem. In one extreme, we can materialize the full result of the CQ and index the result according to the access pattern. However, since the output result can often be extremely large, storing this index can be prohibitively expensive. In the other extreme, we can service each access request by executing the CQ directly over the input database every time. This solution does not need extra storage but can lead to inefficiencies since computation has to be done from scratch and maybe redundant. In this work, we explore the design space between these two extremes. In other words, we want to compress the query output such that it can be stored in a space-efficient way, while we can support a given access pattern over the output as fast as possible.

Example 7. *Suppose we want to perform an analysis about mutual friends of users in a social network. The friend relation is represented by a symmetric binary relation R of size N, where a tuple $R(a, b)$ denotes that user* a *is a friend of user* b*. The data analysis involves accessing the database through the following pattern:* given any two users x and z who are friends, return all mutual friends y. *We formalize this task through an* adorned view $V^{\mathsf{bfb}}(x, y, z) = R(x, y), R(y, z),\ R(z, x)$. *The above formalism says that the view V of the database will be accessed as follows: given values for the* bound (b) *variables x, z, we have to return the values for the* free (f) *variable y such that the tuple is in the view V. The sequence* bfb *is called the* access pattern *for the adorned view.*

One option to solve this problem is to satisfy each access by evaluating a query on the input database. This approach is space-efficient since we work directly on the input and need space $O(N)$. However, we may potentially have to wait $\Omega(N)$ time to even learn whether there is any returned value for y. A second option is to materialize the view $V(x, y, z)$ and build a hash index with key (x, z): in this case, we can satisfy any access optimally with constant delay $\tilde{O}(1)$.[1] *On the other hand, the space needed for storing the view can be $\Omega(N^{3/2})$.*

In this scenario, we would like to construct representations that trade-off between space and delay (or answer time). As we will show later, for this particular example we can construct a data structure for any parameter τ that needs space $O(N^{3/2}/\tau)$, and can answer any access request with delay $\tilde{O}(\tau)$.

The idea of efficiently compressing query results has recently gained considerable attention, both in the context of factorized databases [OZ15b], as well as constant-delay enumeration [Seg15b, BDG07b]. In these settings, the focus is to construct compressed representations that allow for enumeration of the full result with constant delay: this means that the time between outputting two consecutive tuples is $O(1)$, independent of the size of the data. Using factorization techniques, for any input database D, we can construct a compressed data structure for any CQ without projections, called a d-representation, using space $O(|D|^{\mathsf{fhw}})$, where fhw is the *fractional hypertree width* of the query [OZ15b]. Such a d-representation guarantees constant delay enumeration of the full result. In [Seg13a, BDG07b], the compression of CQs with projections is also studied, but the setting is restricted to $O(|D|)$ time preprocessing –which also restricts the size of the compressed representation to $O(|D|)$.

In this chapter, we show that we can dramatically decrease the space for the compressed representation by both (i) taking advantage of the access pattern, and (ii) tolerating a possibly increased delay. For instance, a d-representation for the query in Example 7 needs $O(N^{3/2})$ space, while no linear-time preprocessing can support constant delay enumeration (under reasonable complexity assumptions [BDG07b]). However, we show that if we are willing to tolerate a delay of $\tilde{O}(N^{1/2})$, we can support the access pattern of Example 7 using only $\tilde{O}(N)$ space, linear in the input size.

Our Contribution. In this chapter, we study the design space for compressed representations of conjunctive queries in the full continuum between optimal space and optimal runtime, when our goal is to optimize for a specific access pattern.

Our main contribution is a novel data structure that (i) *can compress the result for every CQ without projections according to the access pattern given by an adorned view, and* (ii) *can be tuned to trade-off space for the delay and answer time.* At the one extreme, the data structure achieves constant delay $O(1)$; At the other extreme, it uses linear space $O(|D|)$,

[1] the $\tilde{O}$ notation includes a poly-logarithmic dependence on N.

but provides a worst delay guarantee. Our proposed data structure includes as a special case the data structure developed in [CP10a] for the *fast set intersection* problem.

To construct our data structure, we need two technical ingredients. The first ingredient (Theorem 1) is a data structure that trades space with delay with respect to the worst-case size bound of the query result. As an example of the type of trade-offs that can be achieved, for any CQ Q without projections and any access pattern, the data structure needs space $\tilde{O}(|D|^{\rho^*}/\tau)$ to achieve delay $\tilde{O}(\tau)$, where ρ^* is the fractional edge cover number of Q, and $|D|$ the size of the input database. In many cases and for specific access patterns, the data structure can substantially improve upon this trade-off. To prove Theorem 1, we develop novel techniques on how to encode information about expensive sub-instances of the problem in a balanced way.

However, Theorem 1 by its own gives suboptimal trade-offs, since it ignores structural properties of the query (for example, for constant delay it materializes the full result). Our second ingredient (Theorem 2) combines the data structure of Theorem 1 with a type of tree decomposition called *connex tree decomposition* [BDG07b]. This tree decomposition has the property of restricting the tree structure such that the bound variables in the adorned view always forms a connected component at the top of the tree.

Finally, we discuss the complexity of choosing the optimal parameters for our two main theorems, when we want to optimize for delay given a space constraint, or vice versa.

Organization. In Section 3.1, we discuss related work followed by the formal problem statement in Section 3.2. Section 3.3 we provide the detailed algorithm and the proof of the first main result that shows how to trade-off space usage of the materialization and the answering time of the query. A direct application of the main theorem may sometimes lead to suboptimal trade-offs. We remedy that in Section 3.4.

3.1 Related Work

There has been a significant amount of literature on data compression; a common application is to apply compression in column-stores [AMF06]. However, such compression methods typically ignore the logical structure that governs data that is a result of a relational query. The key observation is that we can take advantage of the underlying logical structure to design algorithms that can compress the data effectively. This idea has been explored before in the context of *factorized databases* [OZ15b], which can be viewed as a form of logical compression. Our approach builds upon the idea of using query decompositions as a factorized representation, and we show that for certain access patterns it is possible to go below $|D|^{\mathsf{fhw}}$ space for constant delay enumeration. In addition, our results also allow trading off delay for smaller space requirements of the data structure. A long line of work has also investigated the application of a broader set of queries with projections and aggregations [BOZ12, BKOZ13],

as well as learning linear regression models over factorized databases [SOC16, OS16]. Closely related to our setting is the investigation of join-at-a-time query plans, where at each step, a join over one variable is computed [CO15]. The intermediate results of these plans are partial factorized representations that compress only a part of the query result. Thus, they can be used to trade-off space with delay, albeit in a non-tunable manner.

Our work is also connected to the problem of constant-delay enumeration [Seg13a, Seg15b, BDG07b]: in this case, we want to enumerate a query result with constant delay after a linear time preprocessing step. We can view the linear time preprocessing step as a compression algorithm, which needs space only $O(|D|)$. It has been shown that the class of *connex-free acyclic conjunctive queries* can be enumerated with constant delay after a linear-time preprocessing. Hence, in the case of connex-free acyclic CQs, there exists an optimal compression/decompression algorithm. However, many classes of widely used queries are not factorizable to linear size, and also can not be enumerated with constant delay after linear-time preprocessing. Examples in this case are the triangle query $\Delta^{\mathsf{fff}}(x, y, z) = R(x, y), R(y, z), R(z, x)$, or the 2-path query $P_2^{\mathsf{ff}}(x, y) = R(x, y), R(y, z)$.

Beyond CQs, related work has also focussed on evaluating signed conjunctive queries [BB13, BB12]. CQs that contain both positive and negative atoms allow for tractable enumeration algorithms when they are *free-connex signed-acyclic* [BB13]. Nearby problems include counting the output size $|Q(D)|$ using index structures for enumeration [DM14, DM15], and enumerating more expressive queries over restricted class of databases [KS13].

The problem of finding a class of queries that can be maintained in constant time under updates and admit constant delay enumeration is also of considerable interest. Recent work [BKS17a] considered this particular problem and obtained a dichotomy for self-join free and boolean CQs. Our work is also related to this problem in that the class of such queries has a specific structure that allows constant delay enumeration.

Query compression is also a central problem in graph analytics. Many applications involve extracting insights from relational databases using graph queries. In such situations, most systems load the relational data in-memory and expand it into a graph representation which can become very dense. Analysis of such graphs is infeasible, as the graph size blows up quickly. Recent work [XKD15, XD17b, XSD17] introduced the idea of controlled graph expansion by storing information about high-degree nodes and evaluating acyclic CQs over light sub-instances. However, this work is restricted only to binary views (*i.e.*, graphs), and does not offer any formal guarantees on delay or answer time. It also does not allow the compressed representation to grow more than linear in the size of the input.

Finally, we also present a connection to the problem of set intersection. Set intersection has applications in problems related to document indexing [CP10a, AN16] and proving hardness and bounds for space/approximation trade-off of distance oracles for graphs [PR10,

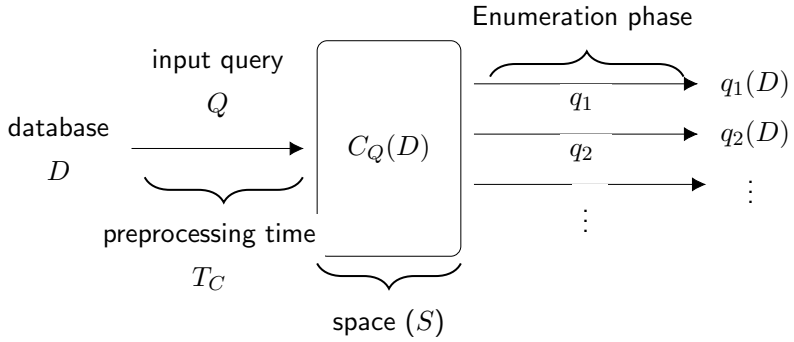


Figure 3.1: Depiction of the enumeration framework along with the parameters.

CP10b]. Previous work [CP10a] has looked at creating a data structure for fast set intersection reporting and the corresponding boolean version. Our main data structure is a strict generalization of the one from [CP10a].

3.2 Problem Statement

Given an adorned view $Q^\eta(x_1, \ldots, x_k)$ and an input database D, our goal is to answer any access request $Q^\eta[v]$ that conforms to the access pattern η. The view Q can be expressed through any type of query, but in this work, we will focus on the case where Q is a conjunctive query.

There are two extremal approaches to handle this problem. The first solution is to answer any such query directly on the input database D, without materializing $Q(D)$. This solution is efficient in terms of space, but it can lead to inefficient query answering. For instance, consider the adorned view $\Delta^{\mathsf{bbf}}(x, y, z) = R(x, y), S(y, z), T(z, x)$. Then, every time we are given new values $x = a, y = b$, we would have to compute all the nodes c that form a triangle with a, b, which can be very expensive.

The second solution is to materialize the view $Q(D)$, and then answer any incoming query over the materialized result. For example, we could choose to materialize all triangles, and then create an appropriate index over the output result. The drawback of this approach is that it requires a lot of space, which may not be available.

We propose to study the solution space between these two extremal solutions, that is, instead of materializing all of $Q(D)$, we would like to store a *compressed representation* $\mathsf{C}_Q(D)$ of $Q(D)$. The compression function C_Q must guarantee that the compression is lossless, *i.e.*, there exists a decompression function D_Q such that for every database D, it holds that $\mathsf{D}_Q(\mathsf{C}_Q(D)) = Q(D)$. We compute the compressed representation $\mathsf{C}_Q(D)$ during a *preprocessing phase*, and then answer any access request in an *online phase*.

Parameters. Our goal is to construct a compression that is as space-efficient as possible, while it guarantees that we can efficiently answer any access query. In particular, we are interested in measuring the trade-off between the following parameters, which are also depicted in Figure 3.1:

Compression Time (T_C): the time to compute $\mathsf{C}_Q(D)$ during the preprocessing phase.

Space (S): the size of $\mathsf{C}_Q(D)$.

Answer Time: this parameter measures the time to enumerate a query result, where the query is of the form $Q^\eta[v]$. The enumeration algorithm must (i) enumerate the query result without any repetitions of tuples, and (ii) use only $O(\log |D|)$ extra memory[2]. We will measure answer time in two different ways.

1. **delay** (δ): the maximum time to output any two consecutive tuples (and also the time to output the first tuple, and the time to notify that the enumeration has completed).
2. **total answer time** (T_A): the total time to output the result.

In the case of a boolean adorned view, the delay and the total answer time coincide. In an ideal situation, both the compression time and the space are linear to the input size and any query can be answered with constant delay $O(1)$. As we will see later, this is achievable in certain cases, but in most cases, we have to trade-off space and preprocessing time for delay and total answer time.

3.2.1 Some Basic Results

We present here some basic results that set up a baseline for our framework. We will study the case where the given view definition Q is a conjunctive query.

Our first observation is that if we allow the compression time to be at least $\Omega(|D|)$, we can assume without loss of generality that the adorned view Q^η has no constants or repeated variables in a single atom. Indeed, we can first do a linear time computation to rewrite the adorned view Q^η to a new view where constants and repeated variables are removed, and then compute the compressed representation for this new view (with the same adornment).

Example 8. *Consider* $Q^{\mathsf{fb}}(x, z) = R(x, y, a), S(y, y, z)$. *We can first compute in linear time* $R'(x, y) = R(x, y, a)$ *and* $S'(y, z) = S(y, y, z)$, *and then rewrite the adorned view as* $Q^{\mathsf{fb}}(x, z) = R'(x, y), S'(y, z)$.

[2]Memory requirement also depends on the memory required for executing the join algorithm. Note that worst-case optimal join algorithms such as NPRR [NPRR12] can be executed using $\log |D|$ memory assuming query size is constant and all relations are sorted and indexed.

Hence, whenever the adorned view is a full CQ, we can w.l.o.g. assume that it is a natural join query. We now state a simple result for the case where the adorned view is full and every variable is bound.

Proposition 1. *Suppose that the adorned view is a natural join query with head* $Q^{\mathsf{b}\cdots\mathsf{b}}(x_1, \ldots, x_k)$. *Then, in time* $T_C = O(|D|)$, *we can construct a data structure with space* $S = O(|D|)$, *such that we can answer any access request over* D *with constant delay* $\delta = O(1)$.

Next, consider the full enumeration view $Q^{\mathsf{f}\cdots\mathsf{f}}(x_1, \ldots, x_k)$. A first observation is that if we store the materialized view, we can enumerate the result in constant delay. From the AGM bound, to achieve this we need space $|D|^{\rho^*(\mathcal{H})}$, where $\mathcal{H}$ is the hypergraph of Q. However, it is possible to improve upon this naive solution using the concept of a factorized representation [OZ15b]. Let $\texttt{fhw}(Q)$ denote the *fractional hypertree width* of Q. Then, the result from [OZ15b] can be translated in our terminology as follows.

Proposition 2 ([OZ15b])**.** *Suppose that the adorned view is a natural join query with head* $Q^{\mathsf{f}\cdots\mathsf{f}}(x_1, \ldots, x_k)$. *Then, in compression time* $T_C = \tilde{O}(|D|^{\texttt{fhw}(Q)})$, *we can construct a data structure with space* $S = O(|D|^{\texttt{fhw}(Q)})$, *such that we can answer any access request over* D *with constant delay* $\delta = O(1)$.

Since every acyclic query has $\texttt{fhw}(Q) = 1$, for acyclic CQs without projections both the compression time and space become linear, $O(|D|)$. In the next section, we will see how we can generalize the above result to an arbitrary adorned view that is full.

3.3 First Main Result

Consider a full adorned view $Q^{\eta}(x_1, \ldots, x_k)$, where Q is a natural join query expressed by the hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. Recall that $\mathcal{V}_{\mathsf{b}}, \mathcal{V}_{\mathsf{f}}$ are the bound and free variables respectively. Since the query is a natural join and there are no projections, we have $\mathcal{V}_{\mathsf{b}} \cup \mathcal{V}_{\mathsf{f}} = \mathcal{V}$. We will denote by $\mu = |\mathcal{V}_{\mathsf{f}}|$ the number of free variables. We also impose a lexicographic order on the enumeration order of the output tuples. Specifically, we equip the domain **dom** with a total order $\leq$, and then extend this to a total order for output tuples in $\mathbf{dom}^{\mu}$ using some order $x^1_{\mathsf{f}}, x^2_{\mathsf{f}}, \ldots, x^{\mu}_{\mathsf{f}}$ of the free variables.[3]

Example 9. *As a running example, consider*

$$Q^{\mathsf{fffbbb}}(x, y, z, w_1, w_2, w_3) = R_1(w_1, x, y), R_2(w_2, y, z), R_3(w_3, x, z).$$

We have $\mathcal{V}_{\mathsf{f}} = \{x, y, z\}$ *and* $\mathcal{V}_{\mathsf{b}} = \{w_1, w_2, w_3\}$. *To keep the exposition simple, assume that* $|R_1| = |R_2| = |R_3| = N$.

[3] There is no restriction imposed on the lexicographic ordering of the free variables.

If we materialize the result and create an index with composite key (w_1, w_2, w_3), then in the worst-case, we need space $S = O(N^3)$, but we will be able to enumerate the output for every access request with constant delay. On the other hand, if we create three indexes, one for each R_i with key w_i, we can compute each access request with worst-case running time and delay of $O(N^{3/2})$. Indeed, once we fix the bound variables to constants c_1, c_2, c_3, we need to compute the join $R_1(c_1, x, y) \bowtie R_2(c_2, y, z) \bowtie R_3(c_3, x, z)$, which needs time $O(N^{3/2})$ using any worst-case optimal join algorithm.

For any fractional edge cover $\mathbf{u}$ of $\mathcal{V}$, and $S \subseteq \mathcal{V}$, we define the *slack* of $\mathbf{u}$ for S as:

$$\alpha(S) = \min_{x \in S} \left(\sum_{F : x \in F} u_F \right) \tag{3.1}$$

Intuitively, the slack is the maximum positive quantity such that $(u_F/\alpha(S))_{F \in \mathcal{E}}$ is still a fractional edge cover of S . By construction, the slack is always at least one, $\alpha(S) \geq 1$. For our running example, suppose that we pick a fractional edge cover for $\mathcal{V}$ with $u_{R_1} = u_{R_3} = u_{R_3} = 1$. Then, the slack of $\mathbf{u}$ for $\mathcal{V}_\mathsf{f}$ is $\alpha(\mathcal{V}_\mathsf{f}) = 2$.

Theorem 1. *Let Q^η be an adorned view over a natural join query with hypergraph $(\mathcal{V}, \mathcal{E})$. Let $\mathbf{u}$ be any fractional edge cover of $\mathcal{V}$. Then, for any input database D and parameter $\tau > 0$ we can construct a data structure with*

$$\textit{compression time } T_C = \tilde{O}(|D| + \prod_{F \in \mathcal{E}} |R_F|^{u_F})$$
$$\textit{space } S = \tilde{O}(|D| + \prod_{F \in \mathcal{E}} |R_F|^{u_F} / \tau^{\alpha(\mathcal{V}_\mathsf{f})})$$

such that for any access request $q = Q^\eta[v]$, we can enumerate its result $q(D)$ in lexicographic order with

$$\textit{delay } \delta = \tilde{O}(\tau)$$
$$\textit{answer time } T_A = \tilde{O}(|q(D)| + \tau \cdot |q(D)|^{1/\alpha(\mathcal{V}_\mathsf{f})})$$

Example 10. *Let us apply Theorem 1 to our running example for $\mathbf{u} = (1, 1, 1)$ and $\tau = N^{1/2}$. The slack for the free variables is $\alpha(\mathcal{V}_\mathsf{f}) = 2$. The theorem tells us that we can construct in time $\tilde{O}(N^3)$ a data structure with space $\tilde{O}(N^2)$, such that every access request q can be answered with delay $\tilde{O}(N^{1/2})$ and answer time $\tilde{O}(|q(D)| + \sqrt{N \cdot |q(D)|})$.*

Next, we show an example application of the theorem.

Example 11. *Consider the adorned view over the* star join

$$S_n^{\mathsf{b}\cdots\mathsf{b}\mathsf{f}}(x_1, \ldots, x_n, z) = R_1(x_1, z), R_2(x_2, z), \ldots, R_n(x_n, z)$$

The star join is acyclic, which means that the d-representation of the full result takes only linear space. This d-representation [OZ15a] can be used for any adornment of S_n where z is a bound variable; hence, in all these cases we can guarantee $O(1)$ delay using linear compression space. However, we cannot get any guarantees when z is free, as is in the adornment used above. Note that for the fractional edge cover where $u_1 = \cdots = u_n = 1$, the slack is $\alpha(\mathcal{V}_\mathsf{f}) = n$. Hence, Theorem 1 tells us that with space $\tilde{O}(|D|^n/\tau^n)$ we get delay $\tilde{O}(\tau)$ and answer time $\tilde{O}(|Q(D)| + \tau \cdot |Q(D)|^{1/n})$.

We should note here that our data structure strictly generalizes the data structure proposed in [CP10a] for the problem of *fast set intersection*. Given a family of sets $S_1, \ldots, S_n$, the goal in this problem is to construct a space-efficient data structure, such that given any two sets S_i, S_j we can compute their intersection $S_i \cap S_j$ as fast as possible. It is easy to see that this problem is captured by the adorned view $S_2^{\mathsf{bbf}}(x_1, x_2, z) = R(x_1, z), R(x_2, z)$, where R is a relation that describes set membership ($R(S_i, a)$ means that $a \in S_i$).

We will now describe the detailed construction of the data structure for Theorem 1. Before we present the compression procedure, we first introduce two important concepts in our construction, f-intervals and f-boxes, both of which describe subspaces of the space of all possible tuples in the output.

Intervals. The *active domain* $\mathbf{D}[x]$ of each variable x is equipped with a total order $\leq$ induced from the order of **dom**. We will use $\bot, \top$ to denote the smallest and largest element of the active domain respectively (these will always exist, since we assume finite databases). An *interval* for variable x is any subset of $\mathbf{D}[x]$ of the form $\{u \in \mathbf{D}[x] \mid a \leq u \leq b\}$, where $a, b \in \mathbf{D}[x]$, denoted by $[a, b]$. We adopt the standard notation for closed and open intervals and write $[a, b) = \{u \in \mathbf{D}[x] \mid a \leq u < b\}$, and $(a, b] = \{u \in \mathbf{D}[x] \mid a < u \leq b\}$. The interval $[a, a]$ is called the *unit interval* and represents a single value. We will often write a for the interval $[a, a]$, and the symbol $\square$ for the interval $\mathbf{D}[x]$.

By lifting the order from a single domain to the lexicographic order of tuples in $\mathbf{D}_\mathsf{f} = \mathbf{D}[x_\mathsf{f}^1] \times \cdots \times \mathbf{D}[x_\mathsf{f}^\mu]$, we can also define intervals over $\mathbf{D}_\mathsf{f}$, which we call f-*intervals*. For instance, if $\mathbf{a} = \langle a_1, \ldots, a_\mu \rangle$ and $\mathbf{b} = \langle b_1, \ldots, b_\mu \rangle$, the f-interval $\mathbf{I} = [\mathbf{a}, \mathbf{b})$ represents all valuations v_f over $\mathcal{V}_\mathsf{f}$ that are lexicographically at least $\mathbf{a}$, but strictly smaller than $\mathbf{b}$.

Boxes. It will be useful to consider another type of subset of $\mathbf{D}_\mathsf{f}$, which we call f-boxes.

Definition 1 (f-box)**.** *An* f-box *is defined as a tuple of intervals $\mathbf{B} = \langle I_1, \ldots, I_\mu \rangle$, where I_i is an interval of $\mathbf{D}[x_\mathsf{f}^i]$. The* f-*box represents all valuations v_f over $\mathcal{V}_\mathsf{f}$, such that $v_\mathsf{f}(x_\mathsf{f}^i) \in I_i$ for every $i = 1, \ldots, \mu$.*

We say that a f-box is *canonical* if whenever $I_i \neq \square$, then every I_j with $j < i$ is a unit interval. A canonical f-box is always of the form $\langle a_1, \ldots, a_{i-1}, I_i, \square, \ldots \rangle$. For ease of

notation, we will omit the $\square$ intervals in the end of a canonical f-box, and simply write $\langle a_1, \ldots, a_{i-1}, I_i \rangle$.

A f-box satisfies the following important property:

Proposition 3. *For every* f*-box* $\mathbf{B}$, $(\bowtie_{F \in \mathcal{E}} R_F) \ltimes \mathbf{B} = \bowtie_{F \in \mathcal{E}} (R_F \ltimes \mathbf{B})$.

Proof. Suppose that the f-box is $\mathbf{B} = \langle I_1, \ldots, I_\mu \rangle$. Consider some valuation v over $\mathcal{V}$ that belongs in $(\bowtie_{F \in \mathcal{E}} R_F) \ltimes \mathbf{B}$. Then, for every $F \in \mathcal{E}$ we have $v(F) \in R_F$, and also for every variable x^i_{f} we have $v(x^i_{\mathsf{f}}) \in I_i$. Since for every variable in $F \cap \mathcal{V}_{\mathsf{f}}$ we have $v(x^i_{\mathsf{f}}) \in I_i$ as well, we conclude that $v(F) \in (R_F \ltimes \mathbf{B})$. Thus, v belongs in $\bowtie_{F \in \mathcal{E}} (R_F \ltimes \mathbf{B})$ as well.

For the opposite direction, consider some valuation v over $\mathcal{V}$ that belongs in $\bowtie_{F \in \mathcal{E}} (R_F \ltimes \mathbf{B})$. Since $(R_F \ltimes \mathbf{B}) \subseteq R_F$, we have that for every $F \in \mathcal{E}$, $v(F) \in R_F$. Thus, in order to show the desired result, it suffices to show that for every x^i_{f} we have $v(x^i_{\mathsf{f}}) \in I_i$. Indeed, take any hyperdge F such that $x^i_{\mathsf{f}} \in F$: then, $v(F) \in (R_F \ltimes \mathbf{B})$ implies that $v(x^i_{\mathsf{f}}) \in I_i$. $\square$

In other words, if we want to compute the restriction of output to tuples in $\mathbf{B}$, it suffices to first restrict each relation to $\mathbf{B}$ and then perform the join. We denote this restriction of the relation as $R_F(\mathbf{B}) = R_F \ltimes \mathbf{B}$. Unfortunately, Proposition 3 does not extend to f-intervals. As we show in the example below, it is generally not possible to first restrict each relation to $R_F \ltimes \mathbf{I}$ and then perform the join.

Example 12. *Consider the adorned view* $V^{\mathsf{fbff}}(x, y, z, w) = R_1(x, y), R_2(y, z), R_3(z, w), R_4(w, x)$. *Assume that the active domain is* $\mathbf{D}[x] = \mathbf{D}[y] = \mathbf{D}[z] = \mathbf{D}[w] = \{1, 2\}$. *Since* $\mathcal{V}_{\mathsf{f}} = \{x, z, w\}$, *consider the* f*-interval* $\mathbf{I} = [\mathbf{a}, \mathbf{b}]$ *where* $\mathbf{a} = \langle 1, 2, 1 \rangle$ *and* $\mathbf{b} = \langle 2, 1, 2 \rangle$. *In other words, interval* $\mathbf{I}$ *contains the following valuations for* $\mathcal{V}_{\mathsf{f}}$*:* $(1, 2, 1), (1, 2, 2), (2, 1, 1), (2, 1, 2)$. *It is easy to verify that* $R_i \ltimes \mathbf{I} = R_i$ *for every* $i = 1, 2, 3, 4$ *and that* $(1, 1, 1, 1)$ *is an output tuple. However,* $(\bowtie_{F \in \mathcal{E}} R_F) \ltimes \mathbf{I}$ *filters out* $(1, 1, 1, 1)$ *as* $(1, 1, 1)$ *does not lie in the interval* $\mathbf{I}$.

As we will see next, we can partition each f-interval to a set of f-boxes of constant size.

Box Decomposition. It will be useful to represent a f-interval $\mathbf{I} = (\mathbf{a}, \mathbf{b})$ as a union of canonical f-boxes. Let j be the first position such that $a_j \neq b_j$. Then, we define the *box decomposition* of $\mathbf{I}$, denoted $\mathcal{B}(\mathbf{I})$, as the following set of canonical f-boxes:

$$
\begin{aligned}
\mathbf{B}^{\ell}_{\mu} &= \langle a_1, \ldots, a_{\mu-1}, (a_\mu, \top] \rangle \\
&\ldots \\
\mathbf{B}^{\ell}_{j+1} &= \langle a_1, \ldots, a_j, (a_{j+1}, \top] \rangle \\
\mathbf{B}_{j} &= \langle a_1, \ldots, a_{j-1}, (a_j, b_j) \rangle \\
\mathbf{B}^{r}_{j+1} &= \langle b_1, \ldots, b_j, [\bot, b_{j+1}) \rangle \\
&\ldots
\end{aligned}
$$

$$\mathbf{B}^r_\mu = \langle b_1, \ldots, b_{\mu-1}, [\bot, b_\mu)\rangle$$

Intuitively, a box decomposition divides an interval into a set of disjoint, lexicographically ordered intervals. We give next an example of an f-interval and its decomposition into canonical f-boxes.

Example 13. *For our running example (Example 9), let the active domain be* $\mathbf{D}[w_i] = \{1, 2, \ldots, 1000\}$ *for* $i = 1, 2, 3$. *Consider an open* f*-interval* $\mathbf{I} = (\langle 10, 50, 100\rangle, \langle 20, 10, 50\rangle)$. *The box decomposition of* $\mathbf{I}$ *consists of the following 5 canonical* f*-boxes:*

$$\mathbf{B}^\ell_3 = \langle 10, 50, (100, \top]\rangle, \quad \mathbf{B}^\ell_2 = \langle 10, (50, \top]\rangle$$
$$\mathbf{B}_1 = \langle (10, 20)\rangle,$$
$$\mathbf{B}^r_2 = \langle 20, [\bot, 10)\rangle \quad \mathbf{B}^r_3 = \langle 20, 10, [\bot, 50)\rangle$$

For another f*-interval* $\mathbf{I}' = [\langle 10, 50, 100\rangle, \langle 10, 50, 200\rangle)$, *where the first two positions coincide, the box decomposition consists of one* f*-box:* $\mathbf{B}_3 = \langle 10, 50, [100, 200)\rangle$.

The following lemma summarizes several important properties of the box decomposition:

Lemma 2. *Let* $\mathbf{I}$ *be an* f*-interval and* $\mathcal{B}(\mathbf{I})$ *be its box decomposition. Then:*

1. *The* f*-boxes in* $\mathcal{B}(\mathbf{I})$ *form an order,* $\mathbf{B}^\ell_\mu \leq \cdots \leq \mathbf{B}^\ell_{j+1} \leq \mathbf{B}_j \leq \mathbf{B}^r_{j+1} \leq \cdots \leq \mathbf{B}^r_\mu$, *such that two tuples from different* f*-boxes are ordered according to the order of their* f*-boxes.*
2. *The non-empty* f*-boxes of* $\mathcal{B}(\mathbf{I})$ *form a partition of* $\mathbf{I}$.
3. $|\mathcal{B}(\mathbf{I})| \leq 2\mu - 1$, *where* $\mu = |\mathcal{V}_\mathsf{f}|$.

Proof. To show item (1), we begin by considering two consecutive f-boxes of the form $\mathbf{B}^\ell_i$. Consider the largest element $\mathbf{a}^> \in \mathbf{B}^\ell_i$ and the smallest element $\mathbf{a}^< \in \mathbf{B}^\ell_{i-1}$, for any $i = j+2, \ldots, \mu$. Note that $\mathbf{a}^>$ has value a_{i-1} in the $(i-1)$-th position and $\mathbf{a}^>$ has a value from $(a_{i-1}, \top]$ in its $(i-1)$-th position. Since a_{i-1} appears before any element in the set $(a_{i-1}, \top]$ and both boxes agree on the first $i-2$ positions, it follows that $\mathbf{a}^> < \mathbf{a}^<$ (notice that the inequality here is strict). A similar argument applies to all other consecutive f-boxes in the decomposition.

We next show item (2). We have already shown that the f-boxes in the decomposition are all disjoint. It is also easy to observe that every f-box in $\mathcal{B}(\mathbf{I})$ is a subset of $\mathbf{I}$. Thus, in order to show that the non-empty f-boxes form a partition of $\mathbf{I}$, it suffices to show that every $\mathbf{c} \in \mathbf{I}$ belongs in some f-box of $\mathcal{B}(\mathbf{I})$. Let $\mathbf{I} = (\mathbf{a}, \mathbf{b})$ and $\mathbf{c} = \langle c_1, \ldots, c_\mu\rangle$ such that $\mathbf{c} \in \mathbf{I}$.

We start by looking at the value of c_j where j is the first position such that $a_j \neq b_j$. We distinguish three cases. If $a_j < c_j < b_j$, then we have that $\mathbf{c} \in \mathbf{B}_j$ and we are done. Suppose

now that $c_j = a_j$, and consider the first position k such that $c_k \neq a_k$. (Note that such a k always exists, otherwise $\mathbf{c} = \mathbf{a} \notin \mathbf{I}$.) Then it is easy to see that $\mathbf{c} \in \mathbf{B}_k^{\ell}$. If $c_j = b_j$, then we symmetrically consider the first position k such that $c_k \neq b_k$; then one can see that $\mathbf{c} \in \mathbf{B}_k^{r}$.

To prove item (3), observe that $|\mathcal{B}(\mathbf{I})| = (\mu - j) + 1 + (\mu - j) = 2\mu + (1 - 2j) \leq 2\mu - 1$, where the last inequality follows because $j \geq 1$. □

The above lemma implies the following corollary:

Corollary 1. *Let* $\mathbf{I}$ *be an* f*-interval and* $\mathcal{B}(\mathbf{I})$ *be its box decomposition.Then:*

$$\bigcup_{\mathbf{B} \in \mathcal{B}(\mathbf{I})} \bowtie_{F \in \mathcal{E}} (R_F \ltimes \mathbf{B}) = (\bowtie_{F \in \mathcal{E}} R_F) \ltimes \mathbf{I}$$

We describe here the intuition behind the compression representation. Our data structure is parametrized by an integer $\tau \geq 0$, which can be viewed as a threshold parameter that works as a knob. We seek to compute the result $(\bowtie_{F \in \mathcal{E}} R_F(v_{\mathsf{b}})) \ltimes \mathbf{I}$, where $\mathbf{I}$ is initially the f-interval that represents all possible valuations. We can upper bound the running time for this instance using the AGM bound. If the bound is less than τ, we can compute the answer in time and delay at most τ.

Otherwise, we do two things: (i) we store a bit (1 if the answer is nonempty, and 0 if it is empty), and (ii) we split the f-interval into two smaller f-intervals. Then, we recursively apply the same idea for each of the two f-intervals. Since we need to store one bit for every valuation that exceeds the given threshold for a given f-interval, we need to bound the number of such valuations: this bound will be our first ingredient. Second, we split each f-interval in the same way for every valuation; we do it such that we can balance the information we need to store for each smaller f-interval. The method to split the f-intervals in a balanced way is our second key ingredient.

Bounding the Heavy Valuations. Given a valuation v_{b} for the bound variables, suppose we are asked to compute the result restricted in some f-interval $\mathbf{I}$, in other words $(\bowtie_{F \in \mathcal{E}} R_F(v_{\mathsf{b}})) \ltimes \mathbf{I}$. Let $R_F(v, \mathbf{B}) = R_F(v) \ltimes \mathbf{B} = (R_F \ltimes v) \ltimes \mathbf{B}$. For an f-box $\mathbf{B}$ and valuation v over any variables, we define:

$$T(\mathbf{B}) = \prod_{F \in \mathcal{E}} |R_F(\mathbf{B})|^{\hat{u}_F}, \quad T(v, \mathbf{B}) = \prod_{F \in \mathcal{E}} |R_F(v, \mathbf{B})|^{\hat{u}_F}$$

We overload T to apply to an f-interval $\mathbf{I}$ and valuation v over any variables as follows:

$$T(\mathbf{I}) = \sum_{B \in \mathcal{B}(\mathbf{I})} T(\mathbf{B}), \quad T(v, \mathbf{I}) = \sum_{B \in \mathcal{B}(\mathbf{I})} T(v, \mathbf{B})$$

Proposition 4. *The output* $(\bowtie_{F \in \mathcal{E}} R_F(v_{\mathsf{b}})) \ltimes \mathbf{I}$ *can be computed in time* $O(T(v_{\mathsf{b}}, \mathbf{I}))$.

Proof. Consider the box decomposition $\mathcal{B}(\mathbf{I})$. First, observe that for any $\mathbf{B} \in \mathcal{B}(\mathbf{I})$ the join $(\bowtie_{F\in\mathcal{E}} R_F(v_\mathsf{b}, \mathbf{B}))$ is over the variables $\mathcal{V}_\mathsf{f}$. Since every variable in $\mathcal{V}_\mathsf{f}$ is covered by $\hat{\mathbf{u}}$, we can use any worst-case optimal algorithm to compute the join in time at most $T(v_\mathsf{b}, \mathbf{B})$. By Corollary 1, we can now compute the join over every $\mathbf{B}$ and union the (disjoint) results to obtain the desired result. The time needed for this is at most $T(v_\mathsf{b}, \mathbf{I}) = \sum_{\mathbf{B}\in\mathcal{B}(\mathbf{I})} T(v_\mathsf{b}, \mathbf{B})$. □

We will use the above bound on the running time as a threshold of when it means that a particular interval is expensive to compute.

Definition 2. *A pair* $(v_\mathsf{b}, \mathbf{I})$ *is* τ-heavy *for a fractional edge cover* $\mathbf{u}$ *if* $T(v_\mathsf{b}, \mathbf{I}) > \tau$.

Observe that if a pair is not τ-heavy, this means that we can compute the corresponding sub instance over $\mathbf{I}$ in time at most $O(\tau)$. The following proposition provides an upper bound for the number of such τ-heavy pairs.

Proposition 5. *Given a* f*-interval* $\mathbf{I}$ *and integer* τ*, let* $\mathcal{H}(\mathbf{I}, \tau)$ *be the valuations* v_b *such that the pair* $(v_\mathsf{b}, \mathbf{I})$ *is* τ*-heavy for* $\mathbf{u}$*. Then,*

$$|\mathcal{H}(\mathbf{I}, \tau)| \leq \left(\frac{T(\mathbf{I})}{\tau}\right)^\alpha$$

Proof. For the sake of simplicity, we will write $\mathcal{H}$ instead of $\mathcal{H}(\mathbf{I}, \tau)$. We can now write:

$$\begin{aligned}
\tau|\mathcal{H}| &\leq \sum_{v_\mathsf{b}\in\mathcal{H}} \sum_{\mathbf{B}\in\mathcal{B}(\mathbf{I})} \prod_{F\in\mathcal{E}} |R_F(v_\mathsf{b}, \mathbf{B})|^{\hat{u}_F} \\
&= \sum_{\mathbf{B}\in\mathcal{B}(\mathbf{I})} \sum_{v_\mathsf{b}\in\mathcal{H}} 1^{1-1/\alpha} \cdot \left(\prod_{F\in\mathcal{E}} |R_F(v_\mathsf{b}, \mathbf{B})|^{u_F}\right)^{1/\alpha} \\
&\leq \sum_{\mathbf{B}\in\mathcal{B}(\mathbf{I})} \left(\sum_{v_\mathsf{b}\in\mathcal{H}} 1\right)^{1-1/\alpha} \left(\sum_{v_\mathsf{b}\in\mathcal{H}} \prod_{F\in\mathcal{E}} |R_F(v_\mathsf{b}, \mathbf{B})|^{u_F}\right)^{1/\alpha} \\
&= |\mathcal{H}|^{1-1/\alpha} \cdot \sum_{\mathbf{B}\in\mathcal{B}(\mathbf{I})} \left(\sum_{v_\mathsf{b}\in\mathcal{H}} \prod_{F\in\mathcal{E}} |R_F(\mathbf{B}) \ltimes v_\mathsf{b}|^{u_F}\right)^{1/\alpha} \\
&\leq |\mathcal{H}|^{1-1/\alpha} \cdot \sum_{\mathbf{B}\in\mathcal{B}(\mathbf{I})} \left(\prod_{F\in\mathcal{E}} |R_F(\mathbf{B})|^{u_F}\right)^{1/\alpha} \\
&= |\mathcal{H}|^{1-1/\alpha} \sum_{\mathbf{B}\in\mathcal{B}(\mathbf{I})} T(\mathbf{B})
\end{aligned}$$

The first inequality comes directly from the definition of a τ-heavy pair. The second inequality is an application of Hölder's inequality. The third inequality is an application of the Query Decomposition Lemma from [NRR13]. □

Example 14. *Consider the following instance for our running example.*

$\mathbf{w_1}$	x	y
1	1	1
1	1	2
1	2	1
2	1	1
3	1	1

R_1

$\mathbf{w_2}$	y	z
1	1	2
1	2	1
1	2	2
2	1	1
2	1	2

R_2

$\mathbf{w_3}$	x	z
1	1	1
1	1	2
1	2	1
2	1	1
2	1	2

R_3

We will use $\mathbf{u} = (1, 1, 1)$ *as the fractional edge cover for* $\mathcal{V}$*. Recall that the slack is* $\alpha = 2$*, and thus* $\hat{\mathbf{u}} = (1/2, 1/2, 1/2)$*. Observe that* $\mathbf{D}[x] = \mathbf{D}[y] = \mathbf{D}[z] = \{1, 2\}$*,* $\mathbf{D}[w_1] = \{1, 2, 3\}$*,* $\mathbf{D}[w_2] = \{1, 2\}$*,* $\mathbf{D}[w_3] = \{1, 2, 3\}$*. Consider the root interval* $\mathbf{I}(r) = [\langle 1, 1, 1\rangle, \langle 2, 2, 2\rangle]$*. The box decomposition* $\mathcal{B}(\mathbf{I}(r))$ *is:*

$$\mathbf{B}_3^\ell = \langle 1, 1, [1, 2]\rangle, \quad \mathbf{B}_2^\ell = \langle 1, (1, 2]\rangle$$
$$\mathbf{B}_2^r = \langle 2, [1, 2)\rangle \quad \mathbf{B}_3^r = \langle 2, 2, [1, 2]\rangle$$

We can then compute $T(\mathbf{I}(r)) = \sqrt{|3||3||4|} + \sqrt{|1||2||4|} + \sqrt{|1||3||1|} + 0 \approx 10.56$*. Consider* $v_{\mathsf{b}}(w_1, w_2, w_3) = (1, 1, 1)$*. One can compute* $T(v_{\mathsf{b}}, \mathbf{I}(r)) = \sqrt{2} + 2 + 1 = 4.414$*. If we pick* $\tau = 4$*, then* $(v_{\mathsf{b}}, \mathbf{I}(r))$ *is* τ*-heavy.*

Splitting an Interval. We next discuss how we perform a balanced splitting of an f-interval $\mathbf{I}$.

Lemma 3. *Let* $\mathbf{B} = \langle I_1, \ldots, I_i, \ldots\rangle$ *be an* f*-box, and* $J_1, \ldots, J_p$ *a partition of the interval* I_i*. Denote* $\mathbf{B}_k = \langle I_1, \ldots, J_k, \ldots\rangle$*. Then,* $\sum_{k=1}^p T(\mathbf{B}_k) \le T(\mathbf{B})$*.*

Proof. Let $\mathcal{F}$ be the hyperedges that include the variable x_{f}^i. Notice that if $F \notin \mathcal{F}$, then $R_F(\mathbf{B}_k) = R_F(\mathbf{B})$ for every $k = 1, \ldots, p$. Moreover, observe that for every $F \in \mathcal{F}$, we have $\sum_{k=1}^p |R_F(\mathbf{B}_k)| = |R_F(\mathbf{B})|$. Thus, to prove the lemma it suffices to show that

$$\sum_{k=1}^{p} \prod_{F\in\mathcal{F}} |R_F(\mathbf{B}_k)|^{\hat{u}_F} \le \prod_{F\in\mathcal{F}} \left(\sum_{k=1}^{p} |R_F(\mathbf{B}_k)|\right)^{\hat{u}_F}$$

The above inequality is an application of Friedgut's inequality [Fri04] called the generalized Hölder inequality, which we can apply because $\sum_{F\in\mathcal{F}} \hat{u}_F \ge 1$. □

Lemma 4. *Consider the canonical* f*-box*

$$\mathbf{B} = \langle a_1, \ldots, a_{i-1}, [\beta_L, \beta_U]\rangle.$$

Then, for any $t \ge 0$*, there exists* $\beta \in \mathbf{D}[x_{\mathsf{f}}^i]$ *such that*

1. $T(\langle a_1, \ldots, a_{i-1}, [\beta_L, \beta)\rangle) \leq t$
2. $T(\langle a_1, \ldots, a_{i-1}, (\beta, \beta_U]\rangle) \leq \max\{0, T(\mathbf{B}) - t\}$.

Moreover, we can compute β in time $\tilde{O}(1)$.

Proof. Let $\beta_L = b_1, \ldots, b_n = \beta_U$ be the elements of the interval $[\beta_L, \beta_U]$ in sorted order. Define $v_i = T(\langle a_1, \ldots, a_{i-1}, [\beta_L, b_i]\rangle)$ for $i = 1, \ldots, n$. Observe that we have $v_1 \leq v_2 \leq \ldots v_n = T(\mathbf{B})$. Hence, we can view the elements b_i as being sorted in increasing order w.r.t. to the value v_i. We now perform binary search to find $\beta = \min_i\{v_i \geq \min(T(\mathbf{B}), t)\}$; such an element always exists since v_i is increasing and $v_n = T(\mathbf{B})$. We can create an index that returns the count $|R_F(\mathbf{B})|$ in logarithmic time, hence the running time to find β is $\tilde{O}(1)$. By construction, we have $T(\langle a_1, \ldots, a_{i-1}, [\beta_L, \beta)\rangle) \leq \min(T(\mathbf{B}), t) \leq t$. Finally, since the intervals $[\beta_L, \beta]$, $[\beta, \beta]$ and and $(\beta, \beta_U]$ form a partition of $[\beta_L, \beta_U]$, we can apply Lemma 3 to obtain that $T(\langle a_1, \ldots, a_{i-1}, (\beta, \beta_U]\rangle) \leq T(\mathbf{B}) - \min(T(\mathbf{B}), t) = \max(0, T(\mathbf{B}) - t)$. □

We now present Algorithm 1, an algorithm that allows for balanced splitting of an f-interval $\mathbf{I}$.

Algorithm 1: Splitting an f-interval $\mathbf{I}$

1 $\mathcal{B}(\mathbf{I}) = \{\mathbf{B}_1, \ldots, \mathbf{B}_k\}$ in lexicographic order
2 $T \leftarrow \sum_{i=1}^{k} T(\mathbf{B}_i)$
3 $s \leftarrow \arg\min_j\{\sum_{i=1}^{j} T(\mathbf{B}_i) > T/2\}$
`/* let` $\mathbf{B}_s = \langle c_1, \ldots, c_{k-1}, I_k, \ldots, I_\mu\rangle$ `*/`
4 $\gamma_{k-1} \leftarrow \sum_{i=1}^{s-1} T(\mathbf{B}_i), \quad \Delta_{k-1} \leftarrow T(\mathbf{B}_s)$
5 **for** j=k **to** μ **do**
6 find min c_j s.t. $T(\langle c_1, \ldots, c_{j-1}, I_j \cap [\bot, c_j]\rangle) \geq \min\{\Delta_{j-1}, T/2 - \gamma_{j-1}\}$
7 $\Delta_j \leftarrow T(\langle c_1, \ldots, c_j\rangle)$
8 $\gamma_j \leftarrow \gamma_{j-1} + T(\langle c_1, \ldots, c_{j-1}, I_j \cap [\bot, c_j)\rangle$
9 **return** $(c_1, \ldots, c_\mu)$

Proposition 6. *Let $\mathbf{I} = [\mathbf{a}, \mathbf{b}]$ be an f-interval. Then, Algorithm 1 returns $\mathbf{c} \in \mathbf{D}_\mathsf{f}$ that splits $\mathbf{I}$ into $\mathbf{I}^{\prec} = [\mathbf{a}, \mathbf{c})$ and $\mathbf{I}^{\succ} = (\mathbf{c}, \mathbf{b}]$ such that $T(\mathbf{I}^{\prec}) \leq T(\mathbf{I})/2$ and $T(\mathbf{I}^{\succ}) \leq T(\mathbf{I})/2$. Moreover, it terminates in time $\tilde{O}(1)$.*

Proof. Notice first that line (6) of the algorithm always finds a c_j, following Lemma 4. Hence, the algorithm always returns a split point $\mathbf{c} = (c_1, \ldots, c_\mu)$.

Define $\mathbf{B}_j^{\prec} = \langle c_1, \ldots, c_{j-1}, I_j \cap [\bot, c_j)\rangle$ and $\mathbf{B}_j^{\succ} = \langle c_1, \ldots, c_{j-1}, I_j \cap (c_j, \top]\rangle$ for $j = k, \ldots, \mu$. Similarly to γ_j, define $\bar{\gamma}_{k-1} = \sum_{i=s+1}^{\mu} T(\mathbf{B}_i)$, and for $j = k, \ldots, \mu$, $\bar{\gamma}_j = \bar{\gamma}_{j-1} + T(\mathbf{B}_j^{\succ})$.

Now, consider the following sets of canonical f-boxes:

$$\mathcal{B}^{\prec} = \mathbf{B}_1, \ldots, \mathbf{B}_{s-1}, \mathbf{B}_k^{\prec}, \ldots, \mathbf{B}_\mu^{\prec}$$
$$\mathcal{B}^{\succ} = \mathbf{B}_1, \ldots, \mathbf{B}_{s-1}, \mathbf{B}_k^{\succ}, \ldots, \mathbf{B}_\mu^{\succ}$$

The key observation is that $\mathcal{B}^{\prec} = \mathcal{B}(\mathbf{I}^{\prec})$ and $\mathcal{B}^{\succ} = \mathcal{B}(\mathbf{I}^{\succ})$. Moreover, by construction $\gamma_\mu = \sum_{\mathbf{B}\in\mathcal{B}^{\prec}} T(\mathbf{B})$ and also $\bar{\gamma}_\mu = \sum_{\mathbf{B}\in\mathcal{B}^{\succ}} T(\mathbf{B})$. Thus, to prove the statement, it suffices to show that $\gamma_\mu, \bar{\gamma}_\mu \leq T/2$.

We will first show that for any $j = k-1, \ldots, \mu : \gamma_j \leq T/2$. For γ_{k-1} this follows by our choice of s. For some $j \geq k$, we have $\gamma_j = \gamma_{j-1} + T(\mathbf{B}_j^{\prec}) \leq \gamma_{j-1} + \min\{\Delta_{j-1}, T/2 - \gamma_{j-1}\} \leq T/2$, where the first inequality follows from the choice of c_j.

Second, we will show by induction that for $j = k-1, \ldots, \mu : \bar{\gamma}_j \leq T/2$. For $\bar{\gamma}_{k-1}$, we have $\bar{\gamma}_{k-1} = T - \sum_{i=1}^{s} T(\mathbf{B}_i) \leq T - T/2 = T/2$. Now, let $j \geq k$. We can write:

$$\begin{aligned}
\bar{\gamma}_j &= \bar{\gamma}_{j-1} + T(\mathbf{B}_j^{\succ}) \\
&\leq \bar{\gamma}_{j-1} + \max\{0, \Delta_{j-1} - (T/2 - \gamma_{j-1})\} \\
&= \max\{\bar{\gamma}_{j-1}, (\Delta_{j-1} + \bar{\gamma}_{j-1} + \gamma_{j-1}) - T/2\}
\end{aligned}$$

The first inequality follows from item (2) of Lemma 4. By the inductive hypothesis we have $\bar{\gamma}_{j-1} \leq T/2$. We next show that $\bar{\gamma}_j + \gamma_j \leq T - \Delta_j$. From Lemma 3, it holds for every $j = k, \ldots, \mu$:

$$T(\mathbf{B}_j^{\prec}) + T(\mathbf{B}_j^{\succ}) \leq \Delta_{j-1} - \Delta_j$$

Using the above inequality, we can write:

$$\begin{aligned}
\bar{\gamma}_j + \gamma_j &= \sum_{i\neq s} T(\mathbf{B}_i) + \sum_{i=k}^{j} (T(\mathbf{B}_i^{\prec}) + T(\mathbf{B}_i^{\succ})) \\
&\leq \sum_{i\neq s} T(\mathbf{B}_i) + \sum_{i=k}^{j} (\Delta_{j-1} - \Delta_j) \\
&= \sum_{i\neq s} T(\mathbf{B}_i) + \Delta_{k-1} - \Delta_j \\
&= T - \Delta_j
\end{aligned}$$

The runtime bound of $\tilde{O}(1)$ follows from Lemma 4, which tells us that we can compute each c_j (line (6)) in time $\tilde{O}(1)$. □

3.3.1 The Basic Structure

We now have all the necessary pieces to describe how we construct the compressed representation. Recall that our data structure is parametrized by a threshold parameter τ, and by

a weight assignment $\mathbf{u} = (u_F)_{F\in\mathcal{E}}$ that covers the variables in $\mathcal{V}$. The construction consists of two steps.

1) The Delay-Balanced Tree. In the first step, we construct an annotated binary tree $\mathcal{T}$. Each node $w \in V(\mathcal{T})$ is annotated with an f-interval $\mathbf{I}(w)$ and a value $\beta(w) \in \mathbf{D}_{\mathsf{f}}$, which is chosen according to Algorithm 1. The tree is constructed recursively.

Initially, we create a *root* r with interval $\mathbf{I}(r) = \mathbf{D}_{\mathsf{f}}$. Let w be a node at level ℓ with interval $\mathbf{I}(w) = [\mathbf{a}, \mathbf{c}]$, and define the threshold at level ℓ to be $\tau_\ell = \tau/2^{\ell(1-1/\alpha)}$. In the case where $T(\mathbf{I}(w)) < \tau_\ell$, w is a leaf of the tree. Otherwise, using $\beta(w)$ as a splitting point, we construct two sub-intervals of $\mathbf{I}$:

$$\mathbf{I}^{\prec} = [\mathbf{a}, \beta(w)) \text{ and } \mathbf{I}^{\succ} = (\beta(w), \mathbf{c}].$$

If $\mathbf{I}^{\prec} \neq \emptyset$, we create a new node w_l as the left child of w, with interval $\mathbf{I}(w_l) = \mathbf{I}^{\prec}$. Similarly, if $\mathbf{I}^{\succ} \neq \emptyset$, we create a new node w_r as the right child of w, with interval $\mathbf{I}(w_r) = \mathbf{I}^{\succ}$. We call the resulting tree $\mathcal{T}$ a *delay-balanced tree.*

Lemma 5. *Let $\mathcal{T}$ be a delay-balanced tree. Then:*

1. *For every node $w \in V(\mathcal{T})$ at level ℓ, we have $T(\mathbf{I}(w)) \leq T(\mathbf{I}(r))/2^\ell$.*
2. *The depth of $\mathcal{T}$ is at most $O(\log T)$ and its size at most $O(T)$, where $T = \prod_{F\in\mathcal{E}} |R_F|^{u_F}/\tau^\alpha$.*

Proof. If w_1 is a child of w_2, then we have that $T(\mathbf{I}(w_1)) \leq T(\mathbf{I}(w_2))/2$ by Proposition 6. Item (1) follows by a simple induction on the depth of the tree.

Suppose that w is a node at level ℓ. From the condition that we use to stop expanding a node, we have:

$$\begin{aligned}\tau_\ell &\leq T(\mathbf{I}(w)) \leq T(\mathbf{I}(r))/2^\ell \\ &\leq (2\mu - 1) \cdot \prod_{F\in\mathcal{E}} |R_F|^{\hat{u}_F}/2^\ell\end{aligned}$$

The bound on the size follows from the fact that the tree is binary. □

Example 15. *Continuing our running example, we will construct the delay-balanced tree. Since $\ell = 0$ for root node, $\tau_\ell = \tau$. We begin by finding the split point $\beta(r)$ for root node. We start with unit interval $\mathbf{I}(r)^{\prec} = [\langle 1,1,1\rangle, \langle 1,1,1\rangle]$ and keep increasing the interval range until the join evaluation cost $T(\mathbf{I}(r)^{\prec}) > T(\mathbf{I}(r))/2$. For interval $\mathbf{I}(r)^{\prec} = [\langle 1,1,1\rangle, \langle 1,1,1\rangle]$, the box decomposition is $\mathcal{B}(\mathbf{I}(r)^{\prec}) = \mathbf{B}_3^\ell = \langle 1,1,1\rangle$.*

The reader can verify that $T(\mathbf{I}(r)^{\prec}) = \sqrt{|3||1||2|} \approx 2.44$ units and changing the interval to $\mathbf{I}'(r)^{\prec} = [\langle 1,1,1\rangle,\ \langle 1,1,2\rangle]$ gives $T(\mathbf{I}'(r)^{\prec}) = \sqrt{|3||3||4|} > T(\mathbf{I}(r))/2$. Thus, $\beta(r) = (1,1,2)$ and $\mathbf{I}(r)^{\succ} = [\langle 1,2,1\rangle, \langle 2,2,2\rangle]$ with $T(\mathbf{I}(r)^{\succ}) = \sqrt{|1||2||4|} + \sqrt{|1||3||1|} \approx 4.56$.

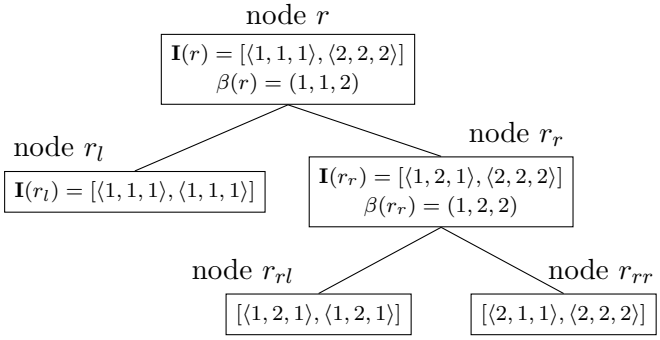


Figure 3.2: Delay balanced tree for running example

For the next level $\ell = 1$, the threshold $\tau_\ell = \tau/\sqrt{2} \approx 2.82$. Since $T(\mathbf{I}(r)^{\prec}) \leq 2.82$, it is a leaf node. We recursively split $\mathbf{I}(r_r) = \mathbf{I}^{\succ}(r) = [\langle 1,2,1\rangle, \langle 2,2,2\rangle]$ into $\mathbf{I}^{\prec}(r_r)$ and $\mathbf{I}^{\succ}(r_r)$. Fixing $\beta(r_r) = (1,2,2)$, we get $T(\mathbf{I}^{\prec}(r_r)) = \sqrt{|1||2||1|} \approx 1.414$ and $\mathbf{I}^{\succ}(r_r) = [\langle 2,1,1\rangle, \langle 2,2,2\rangle]$, $T(\mathbf{I}^{\succ}(r_r)) = \sqrt{3}$. Since both worst case running times are smaller than $\tau_2 = \tau/2 = 2$, our tree construction is complete. We demonstrate the final delay-balanced tree $\mathcal{T}$ in Figure 3.2.

2) Storing Auxiliary Information. The second step is to store auxiliary information for the heavy valuations at each node of the tree $\mathcal{T}$. Recall that the threshold for a heavy valuation at a node in level ℓ is $\tau_\ell = \tau/2^{\ell(1-1/\alpha)}$. We will construct a *dictionary* $\mathcal{D}$ that takes as arguments a node $w \in V(\mathcal{T})$ at level ℓ and a valuation v_b such that $(v_\mathsf{b}, \mathbf{I}(w))$ is τ_ℓ-heavy and returns in constant time:

$$\mathcal{D}(w, v_\mathsf{b}) = \begin{cases} 0, & \text{if } (\bowtie_{F\in\mathcal{E}} R_F(v_\mathsf{b})) \ltimes \mathbf{I}(w) = \emptyset, \\ 1, & \textit{otherwise.} \end{cases}$$

If $(v_\mathsf{b}, \mathbf{I}(w))$ is not τ_ℓ-heavy, then there is no entry for this pair in the dictionary and it simply returns $\perp$. In other words, $\mathcal{D}$ remembers for the pairs that are heavy whether the answer is empty or not for the restriction of the result to the f-interval $\mathbf{I}(w)$.

We next provide an upper bound on the size of $\mathcal{D}$.

Lemma 6. $|\mathcal{D}| = \tilde{O}(\prod_{F\in\mathcal{E}} |R_F|^{u_F}/\tau^\alpha)$.

Proof. We first bound the number of (w, v_b) pairs that are stored in the dictionary for a node w at level ℓ. Notice that for node w we will store an entry for at most the τ_ℓ-heavy valuations. By Proposition 5, these are at most

$$\begin{aligned} |\mathcal{H}(\mathbf{I}(w), \tau_\ell)| &\leq \left(\frac{T(\mathbf{I}(w))}{\tau_\ell}\right)^\alpha \leq \left(\frac{T(\mathbf{I}(r))}{2^\ell \tau_\ell}\right)^\alpha \\ &\leq (2\mu - 1)^\alpha 2^{-\ell\alpha} \tau_\ell^{-\alpha} \prod_{F\in\mathcal{E}} |R_F|^{u_F} \end{aligned}$$

$$= c \cdot 2^{-\ell} \tau^{-\alpha} \prod_{F \in \mathcal{E}} |R_F|^{u_F}$$

where $c = (2\mu - 1)^{\alpha}$ is a constant. At level ℓ we have at most 2^{ℓ} nodes. Hence, the total number of nodes if the tree has L levels is at most:

$$\sum_{\ell=0}^{L} 2^{\ell} \left(\tau^{-\alpha} 2^{-\ell} \prod_{F \in \mathcal{E}} |R_F|^{u_F} \right) \leq \log |D| \cdot \tau^{-\alpha} \prod_{F \in \mathcal{E}} |R_F|^{u_F}$$

This concludes the proof. □

The final compressed representation consists of the pair $(\mathcal{T}, \mathcal{D})$, along with the necessary indexes on the base relations (that need only linear space).

Example 16. *The last step for our running example is to construct the dictionary for all τ_ℓ-heavy valuations. Consider the valuation $v_{\mathsf{b}}(w_1, w_2, w_3) = (1, 1, 1)$, which we have shown to be τ-heavy. Next, we store a bit in the dictionary at each node for v_{b} denoting if the join output is non-empty for the restriction of result to interval $\mathbf{I}$. The reader can verify that $(v_{\mathsf{b}}, \mathbf{I}(r))$ and $(v_{\mathsf{b}}, \mathbf{I}(r_r))$ are τ_0- and τ_1-heavy respectively. Thus, the dictionary will store two entries for v_{b}: $\mathcal{D}(\mathbf{I}(r), v_{\mathsf{b}}) = 1, \mathcal{D}(\mathbf{I}(r_r), v_{\mathsf{b}}) = 1$.*

Lastly, we now show a detailed construction that allows us to build the dictionary $\mathcal{D}$ in time $\tilde{O}(\prod_{F \in \mathcal{E}} |R_F|^{u_F})$, using at most $\tilde{O}(\prod_{F \in \mathcal{E}} |R_F|^{u_F} / \tau^{\alpha})$ space, *i.e.* no more space than the size of the dictionary.

Lemma 7. *The dictionary $\mathcal{D}$ can be constructed in time $\tilde{O}(\prod_{F \in \mathcal{E}} |R_F|^{u_F})$, using at most $\tilde{O}(\prod_{F \in \mathcal{E}} |R_F|^{u_F} / \tau^{\alpha})$ space.*

We construct the dictionary $\mathcal{D}$ as follows:

a) Find Heavy Valuations. The first step of the algorithm is to compute the list of heavy valuations v_{b} for any interval $\mathbf{I}$.

Proposition 7. *Let $\mathcal{L}_{\mathbf{I}}$ denote the sorted list of all valuations of $\mathcal{V}_{\mathsf{b}}$ such that $(v_{\mathsf{b}}, \mathbf{I})$ is τ-heavy. Then, $\mathcal{L}_{\mathbf{I}}$ can be constructed in time $\tilde{O}(\sum_{\mathbf{B} \in \mathcal{B}(\mathbf{I})} \prod_{F \in \mathcal{E}_{\mathcal{V}_{\mathsf{b}}}} |R_F \ltimes \mathbf{B}|^{u_F})$ and using space at most $O((T(\mathbf{I})/\tau)^{\alpha})$.*

Proof. The first observation is that for all heavy $(v_{\mathsf{b}}, \mathbf{I})$ valuations, since $T(v_{\mathsf{b}}, \mathbf{I}) > \tau$, there exists a $\mathbf{B} \in \mathcal{B}(\mathbf{I})$ such that $R_F(v_{\mathsf{b}}) \ltimes \mathbf{B}$ is non-empty for each $F \in \mathcal{E}_{\mathcal{V}_{\mathsf{b}}}$. This implies that $\pi_{F \cap \mathcal{V}_{\mathsf{b}}}(v_{\mathsf{b}}) \in \pi_{F \cap \mathcal{V}_{\mathsf{b}}}(R_F \ltimes \mathbf{B})$ (otherwise the relation will be empty and $T(v_{\mathsf{b}}, \mathbf{I}) = 0$). Thus, it is sufficient to compute $\pi_{\mathcal{V}_{\mathsf{b}}}((\bowtie_{F \in \mathcal{E}_{\mathcal{V}_{\mathsf{b}}}} R_F) \ltimes \mathbf{I})$ to find all heavy valuations.

We can construct the list $\mathcal{L}_{\mathbf{I}}$ by running a worst case join algorithm in time $O(\sum_{\mathbf{B} \in \mathcal{B}(\mathbf{I})} \prod_{F \in \mathcal{E}_{\mathcal{V}_{\mathsf{b}}}} |R_F \ltimes \mathbf{B}|^{u_F})$. Additionally, as soon as the worst case join algorithm generates an output v_{b}, we check if v_{b} is τ-heavy in $\tilde{O}(1)$ time. This can be done by using linear

sized indexes on base relations to count the number of tuples in each relation $R_{F\in\mathcal{E}}(v_\mathsf{b}, \mathbf{I})$ and using $\mathbf{u}$ as cover to check whether the execution time is greater than the threshold τ. Since we need only heavy valuations, we only retain those in memory. Proposition 5 bounds the space requirement of $\mathcal{L}_\mathbf{I}$ to at most $O((T(\mathbf{I})/\tau)^\alpha)$. Sorting $\mathcal{L}_\mathbf{I}$ introduces at most an additional $O(\log|D|)$ factor. □

b) Using join output to create $\mathcal{D}$. Consider the delay balanced tree $\mathcal{T}$ as constructed in the first step. Without loss of generality, assume that the tree is full. We bound the time taken to create $\mathcal{D}(w, v_\mathsf{b})$ for all nodes w_L at some level L (applying the same steps to other levels introduces at most $\log|D|$ factor). The detailed algorithm is presented below.

Algorithm 2: Create Dictionary $\mathcal{D}(w, v_\mathsf{b})$ for level L nodes in $\mathcal{T}$

input : tree $\mathcal{T}$
output: $\mathcal{D}(w, v_\mathsf{b})$ for all leaf nodes

forall w *in* w_L **do**
  /* Run NPRR on $(\bowtie_{F\in\mathcal{E}_{\mathcal{V}_\mathsf{b}}} R_F) \ltimes \mathbf{I}(w)$ to compute $\mathcal{L}_{\mathbf{I}(w)}$ */
  forall $v_\mathsf{b} \in \mathcal{L}_{\mathbf{I}(w)}$ **do**
    $\mathcal{D}(w, v_\mathsf{b}) = 0$ /* initializing $\mathcal{D}$ with all heavy pairs */
  /* Run NPRR on $(\bowtie_{F\in\mathcal{E}} R_F) \ltimes \mathbf{I}(w)$ */
  forall $j \leftarrow$ *output tuple from NPRR* /* requires $\log|D|$ main memory */
  do
    $v_\mathsf{b} \leftarrow \Pi_{\mathcal{V}_\mathsf{b}}(j)$
    if $v_\mathsf{b} \in \mathcal{L}_{\mathbf{I}(w)}$ **then** /* binary search over $\mathcal{L}_{\mathbf{I}(w)}$ */
      $\mathcal{D}(w, v_\mathsf{b}) = 1$

Algorithm Analysis. We first bound the running time of the algorithm.

Proposition 8. *Algorithm 2 runs in time* $\tilde{O}(\prod_{F\in\mathcal{E}} |R_F|^{u_F})$.

Proof. We will first compute the time needed to construct the list $\mathcal{L}_{\mathbf{I}(w)}$ for all nodes w at level L. Proposition 7 tells us that to find the heavy valuations for an interval $\mathbf{I}(w)$ we need time $\tilde{O}(\sum_{\mathbf{B}\in\mathcal{B}(\mathbf{I}(w))} \prod_{F\in\mathcal{E}_{\mathcal{V}_\mathsf{b}}} |R_F \ltimes \mathbf{B}|^{u_F})$. We will apply Lemma 3 to show that $\sum_{w\in w_L} \sum_{\mathbf{B}\in\mathcal{B}(\mathbf{I}(w))} \prod_{F\in\mathcal{E}} |R_F \ltimes \mathbf{B}|^{u_F} = O(\prod_{F\in\mathcal{E}} |R_F|^{u_F})$. Consider all the f-boxes in the box decomposition of $\mathbf{I}(w), \forall w \in w_L$. All f-boxes that have the first $\mu-1$ variables fixed are of the form $\mathbf{B}^k_\mu = \langle a_1, \ldots, a_{\mu-1}, (a^k_\mu, b^k_\mu)\rangle$. We apply Lemma 3 with $i = \mu$ to all such boxes. Thus, $\sum_k T(\mathbf{B}^k_\mu) \le T(\langle a_1, \ldots, a_{\mu-1}, \square\rangle)$.

After this step, all f-boxes have unit interval prefix of length at most $\mu-1$ and have the domain of x^f_μ as $\square$. Now, we repeatedly apply lemma 3 to all boxes with $i = \mu-1, \mu-2, \ldots, 1$

sequentially. Each application merges the boxes and fixes the domain of $x^i_{\mathsf{f}} = \square$. The last step merges all f-boxes of the form $\langle a \rangle$ to $\mathbf{I}(r) = \langle \square, \ldots, \square \rangle$. This gives us,

$$\sum_{w \in w_L} \sum_{\mathbf{B} \in \mathcal{B}(\mathbf{I}(w))} \prod_{F \in \mathcal{E}} |R_F \ltimes \mathbf{B}|^{u_F} \leq \prod_{F \in \mathcal{E}} |R_F \ltimes \mathbf{I}(r)|^{u_F}$$
$$= O(\prod_{F \in \mathcal{E}} |R_F|^{u_F})$$

The second step is to bound the running time of the worst case join optimal algorithm to compute $(\bowtie_{F \in \mathcal{E}} R_F) \ltimes \mathbf{I}(w)$ in line 4. Observe that this join can also be computed in worst case time $\sum_{w \in w_L} \sum_{\mathbf{B} \in \mathcal{B}(\mathbf{I}(w))} \prod_{F \in \mathcal{E}} |R_F \ltimes \mathbf{B}|^{u_F} = O(\prod_{F \in \mathcal{E}} |R_F|^{u_F})$.

Finally, note that all steps in line 6-line 8 are $\tilde{O}(1)$ operations. □

Next, we analyze the space requirement of Algorithm 2.

Proposition 9. *Algorithm 2 requires space* $O(\prod_{F \in \mathcal{E}} |R_F|^{u_F} / \tau^{\alpha})$

Proof. Lines 2-3 take $|\mathcal{D}|$ amount of space. The NPRR algorithm requires $\log |D|$ amount of memory to keep track of pointers [4]. Since we are only streaming through the join output, there is no additional memory overhead in this step. Thus, the bound on memory required follows from Proposition 7 (bounding the size of $\mathcal{L}_{\mathbf{I}(r)}$) and Lemma 6. □

3.3.2 Answering a Query

We now explain how we can use the data structure to answer an access request $q = Q^{\eta}[v]$ given by a valuation v. The detailed algorithm is depicted in Algorithm 3.

We start traversing the tree starting from the root r. For a node w, if $\mathcal{D}(w, v_{\mathsf{b}}) = \perp$, we compute the corresponding sub instance using a worst-case optimal algorithm for every box in the box decomposition. If $\mathcal{D}(w, v_{\mathsf{b}}) = \mathsf{0}$, we do nothing. If $\mathcal{D}(w, v_{\mathsf{b}}) = \mathsf{1}$, we recursively traverse the left child (if it exists), compute the instance for the unit interval $[\beta(w), \beta(w)]$, then recursively traverse the right child (if it exists). This traversal order guarantees that the tuples are output in lexicographic order.

Algorithm Analysis. We now analyze the performance of Algorithm 3. Let $\mathcal{T}_v$ be the subtree of $\mathcal{T}$ that contains the nodes visited by Algorithm 3. The algorithm stops traversing down the tree only when it finds a node $w \in V(\mathcal{T})$ such that $\mathcal{D}(w, v_{\mathsf{b}}) \neq \mathsf{1}$. (The leaf nodes of $\mathcal{T}$ have all $\perp$ entries since by construction they contain no heavy pairs.) Thus, the leaves of $\mathcal{T}_v$ have $\mathcal{D}(w, v_{\mathsf{b}}) \in \{\mathsf{0}, \perp\}$ and the internal nodes have $\mathcal{D}(w, v_{\mathsf{b}}) = \mathsf{1}$. Figure 3.3 depicts an instance of such an incomplete binary tree.

[4] If we have at least $|D|$ memory, i.e, all relations can fit in memory, then we require no subsequent I/O's

Algorithm 3: Answering a query $q = Q^{\eta}[v_{\mathsf{b}}]$

input : tree $\mathcal{T}$, dictionary $\mathcal{D}$, valuation v

output: query answer $q(D)$

```
1  eval(r, v_b)                                          /* start from the root */
2  return
3  procedure eval(w, v_b)
4      if D(w, v_b) = ⊥ then
5          forall B ∈ B(I(w)) do
6              output ⋈_{F∈E} R_F(v_b, B)
7      else if D(w, v_b) = 1 then
8          if w has left child w_ℓ then
9              eval(w_ℓ, v_b)
10         output ⋈_{F∈E} R_F(v_b, [β(w), β(w)])
11         if w has right child w_r then
12             eval(w_r, v_b)
13     return
```

Lemma 8. *Let w be a node in $\mathcal{T}_v$. Algorithm 3 spends $O(1)$ time at w if $\mathcal{D}(w, v_{\mathsf{b}}) \neq \perp$; otherwise it spends time $O(\tau_\ell)$, where ℓ is the level of node w.*

Proof. It takes constant time to retrieve the value $\mathcal{D}(w, v_{\mathsf{b}})$ from the dictionary. If the result is $\mathsf{0}$, we do nothing more on node w. If the result is $\mathsf{1}$, we also need to evaluate the subinstance $\bowtie_{F\in\mathcal{E}} R_F(v_{\mathsf{b}}, [\beta(w), \beta(w)])$. But this can be done in constant time, since $[\beta(w), \beta(w)]$ is a unit interval, and thus the evaluation can be done by checking a constant number of hash tables.

If $\mathcal{D}(w, v_{\mathsf{b}}) = \perp$ and w is at level ℓ, the algorithm will evaluate the subinstance with interval $\mathbf{I}(w)$, which by definition takes time $O(\tau_\ell)$. □

Proposition 10. *Algorithm 3 enumerates $q(D)$ in lexicographic order with delay $\delta = \tilde{O}(\tau)$.*

Proof. Suppose that Algorithm 3 outputs $t \in q(D)$, and the lexicographically next tuple exists and is t'. We will show that the time to output t' after t is $\tilde{O}(\tau)$.

If t, t' are output while Algorithm 3 is at the same node w, it must be that $\mathcal{D}(w, v_{\mathsf{b}}) = \perp$, in which case the delay will be trivially bounded by $O(\tau)$. Otherwise let w be the node where t is output, and w' the node where t' is output. Notice that t will be the last tuple from w that is output, and t' the first tuple from w'. Now, let P be the unique path in T_v that connects w with w', with nodes $w = w_1, w_2, \ldots, w_k = w'$. An example of P is depicted in Figure 3.3. All the nodes in the path, except possibly the endpoints w_1, w_k are internal

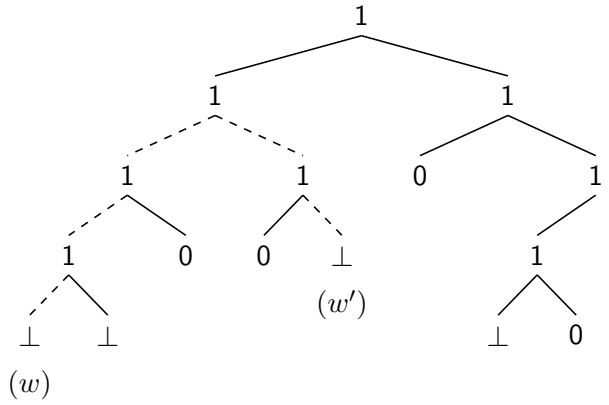


Figure 3.3: An example subtree $\mathcal{T}_v$ traversed by Algorithm 3 to answer $q = Q^\eta[v]$. Each node w is annotated by the dictionary entry $\mathcal{D}(w, v_\mathsf{b})$. The dashed edges show the path from node w that outputs tuple t, to node w' that outputs the lexicographically next tuple t'.

nodes and thus we have $\mathcal{D}(w_i, v_\mathsf{b}) = 1$ for $i = 2, \ldots, k-1$. Moreover, there must exist some $q = 1, \ldots, k$ such that: (i) if $j \leq q$, then w_{j-1} is a child of w_j, and (ii) if $j > q$, then w_j is a child of w_{j-1}.

Let us consider the first segment of the path, where $j \leq q$. If w_{j-1} is the right child of w_j, then the algorithm will exit w_j and visit the next node in the path. If it is the left child, then the algorithm will visit the subtree rooted at its right child first. However, the subtree can only have a single node w'' with $\mathcal{D}(w''_i, v_\mathsf{b}) \neq 1$, since otherwise t' would not have been the next tuple to be output. Thus, after at most $O(\tau)$ time, the algorithm will visit the next node in the path. By a symmetric argument, the algorithm will take at most $O(\tau)$ time to visit the next node in the path for the second segment, where $j \geq q$. Since the length of P is at most 2 times the depth of the tree, which is $O(\log |D|)$, the algorithm will visit w' (and thus output t') in time $O(\tau \log |D|)$.

In the case where there is no next tuple after t, it is easy to see that there exists again a path P that ends at the root node r. A similar argument can be done to bound the time to output the first tuple. □

We now proceed to bound the time to answer the query. The next lemma relates the output size $|q(D)|$ to the size of the tree $\mathcal{T}_v$.

Lemma 9. *The number of nodes in $\mathcal{T}_v$ is $\tilde{O}(|q(D)|)$.*

Proof. Let F be the set of internal nodes of $\mathcal{T}_v$, such that there is no child with entry 1. The key observation is that $|q(D)| \geq |F|$, since the intervals of the nodes in F do not overlap, and each interval will produce at least one output tuple. We can easily also see that $|V(\mathcal{T}_v)| \leq |F| \cdot \log |D|$. Hence, $|V(\mathcal{T}_v)| = O(|q(D)| \cdot \log |D|)$. □

Proposition 11. *Algorithm 3 enumerates $q(D)$ in lexicographic order in $T_A = \tilde{O}(|q(D)| + \tau \cdot |q(D)|^{1/\alpha})$ time.*

Proof. We first bound the time needed to visit the nodes w in $\mathcal{T}_v$ with entry $\neq \perp$. Since every such node requires constant time to visit, and by Lemma 9 the total number of nodes in tree is $\tilde{O}(|q(D)|)$, we need $\tilde{O}(|q(D)|)$ time. Second, we bound the time to visit the nodes with entry $= \perp$. Let V be the set of such nodes. Every node in V is a leaf in $\mathcal{T}_v$. For a node w, let ℓ_w be its level. The answer time can be bounded by:

$$\begin{aligned}\sum_{w \in V} \tau_{\ell_w} &= \sum_{w \in V} \tau \cdot 2^{-\ell_w(1-1/\alpha)} \\ &= \tau \cdot \sum_{w \in V} 1^{1/\alpha} (2^{-\ell_w})^{1-1/\alpha} \\ &\leq \tau |V|^{1/\alpha} \left(\sum_{w \in V} 2^{-\ell_w} \right)^{1-1/\alpha} \\ &\leq \tilde{O}(\tau \cdot |q(D)|^{1/\alpha})\end{aligned}$$

The first inequality is an application of Hölders inequality. The second inequality is an application of Kraft's inequality [McM56], which states that for a binary tree we have $\sum_{w \text{ leaf}} 2^{-depth(w)} \leq 1$. □

3.4 Second Main Result

The direct application of Theorem 1 can lead to suboptimal trade-offs between space and time/delay, since it ignores the structural properties of the query. In this section, we show how to overcome this problem by combining Theorem 1 with tree decompositions. We first need to introduce a variant of a tree decomposition of a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, defined with respect to a given subset $C \subseteq \mathcal{V}$.

Definition 3 (Connex Tree Decomposition [BDG07b])**.** *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph, and $C \subseteq \mathcal{V}$. A* C-connex tree decomposition *of $\mathcal{H}$ is a tuple $(\mathcal{T}, A)$, where:*

1. *$\mathcal{T} = (\mathcal{T}, (\mathcal{B}_t)_{t \in V(\mathcal{T})})$ is a tree decomposition of $\mathcal{H}$; and*
2. *A is a connected subset of $V(\mathcal{T})$ such that $\bigcup_{t \in A} \mathcal{B}_t = C$.*

In a C-connex tree decomposition, the existence of the set A forces the set of nodes that contain some variable from C to be connected in the tree.

Example 17. *Consider the hypergraph $\mathcal{H}$ in Figure 3.4. The decomposition depicted on the left is a C-connex tree decomposition for $C = \emptyset$. The C-connex tree decomposition on the right is for $C = \{v_1, v_5, v_6\}$. In both cases, A consists of a single bag (colored grey) which contains exactly the variables in C.*

In [BDG07b], C-connex decompositions were used to obtain compressed representations of CQs with projections (where C is the set of the head variables). In our setting, we will choose C to be the set of bound variables in the adorned view, *i.e.*, $C = \mathcal{V}_\mathsf{b}$. Additionally, we will use a novel notion of width, which we introduce next. Given a $\mathcal{V}_\mathsf{b}$-connex tree decomposition $(\mathcal{T}, A)$, we orient the tree $\mathcal{T}$ from some node in A. For any node $t \in V(\mathcal{T}) \setminus A$, we denote by $\mathsf{anc}(t)$ the union of all the bags for the nodes that are the ancestors of t. Define $\mathcal{V}_\mathsf{b}^t = \mathcal{B}_t \cap \mathsf{anc}(t)$ and $\mathcal{V}_\mathsf{f}^t = \mathcal{B}_t \setminus \mathcal{V}_\mathsf{b}^t$. Intuitively, $\mathcal{V}_\mathsf{b}^t$ (resp. $\mathcal{V}_\mathsf{f}^t$) are the bound (resp. free) variables for the bag t as we traverse the tree top-down. Figure 3.4 depicts each bag $\mathcal{B}_t$ as $\mathcal{V}_\mathsf{f}^t \mid \mathcal{V}_\mathsf{b}^t$.

Given a $\mathcal{V}_\mathsf{b}$-connex tree decomposition, a *delay assignment* is a function $\delta : V(\mathcal{T}) \to [0, \infty)$ that maps each bag to a non-negative number, such that $\delta(t) = 0$ for $t \in A$. Intuitively, this assignment means that we want to achieve a delay of $|D|^{\delta(t)}$ for traversing this particular bag. For a bag t, define

$$\rho_t^+ = \min_{\mathbf{u}} \left(\sum_F u_F - \delta(t) \cdot \alpha(\mathcal{V}_\mathsf{f}^t) \right) \tag{3.2}$$

where $\mathbf{u}$ is a fractional edge cover of the bag $\mathcal{B}_t$. The *$\mathcal{V}_\mathsf{b}$-connex fractional hypertree δ-width* of $(\mathcal{T}, A)$ is defined as $\max_{t \in V(\mathcal{T}) \setminus A} \rho_t^+$. It is critical that we ignore the bags in the set A in the max computation. We also define $u_t^+ = \sum_F u'_F$ where $\mathbf{u}'$ is the fractional edge cover of bag $\mathcal{B}_t$ that minimizes ρ_t^+.

When $\delta(t) = 0$ for every bag t, the δ-width of any $\mathcal{V}_\mathsf{b}$-connex tree decomposition becomes $\max_{t \in V(\mathcal{T}) \setminus A} \rho^*(\mathcal{B}_t)$, where $\rho^*(\mathcal{B}_t)$ is the fractional edge cover number of $\mathcal{B}_t$. Define $\mathsf{fhw}(\mathcal{H} \mid \mathcal{V}_\mathsf{b})$ as the smallest such quantity among all $\mathcal{V}_\mathsf{b}$-connex tree decompositions of $\mathcal{H}$. When $\mathcal{V}_\mathsf{b} = \emptyset$, then $\mathsf{fhw}(\mathcal{H} \mid \mathcal{V}_\mathsf{b}) = \mathsf{fhw}(\mathcal{H})$, thus recovering the notion of fractional hypertree width. Appendex 3.4.2.1 shows the relationship between $\mathsf{fhw}(\mathcal{H} \mid \mathcal{V}_\mathsf{b})$ and other hypergraph related parameters.

Finally, we define the δ-height of a $\mathcal{V}_\mathsf{b}$-connex tree decomposition to be the maximum weight root-to-leaf path, where the weight of a path P is defined as $\sum_{t \in P} \delta(t)$.

Example 18. *Consider the decomposition on the right in Figure 3.4, and a delay assignment δ that assigns $1/3$ to node t_1 with $\mathcal{B}_{t_1} = \{v_2, v_4, v_1, v_5\}$, $1/6$ to the bag t_2 with $\mathcal{B}_{t_2} = \{v_2, v_3, v_4\}$, and 0 to the node t_3 with $\mathcal{B}_{t_3} = t_3 = \{v_6, v_7\}$. The δ-height of the tree is $h = \max\{1/3 + 1/6, 0\} = 1/2$. To compute the fractional hypertree δ-width, observe that we can cover the bag $\{v_2, v_4, v_1, v_5\}$ by assigning weight of 1 to the edges $\{v_1, v_2\}, \{v_4, v_5\}$, in which case $\rho_{t_1}^+ = (1 + 1) - 1/3 \cdot 1 = 5/3$. We also have $\rho_{t_2}^+ = (1 + 1) - 1/6 \cdot 2 = 5/3$, and $\rho_{t_3}^+ = 1$. Hence, the fractional hypertree δ-width is $5/3$. Also, observe that $u_{t_1}^+ = u_{t_2}^+ = 2$ and $u_{t_3}^+ = 1$.*

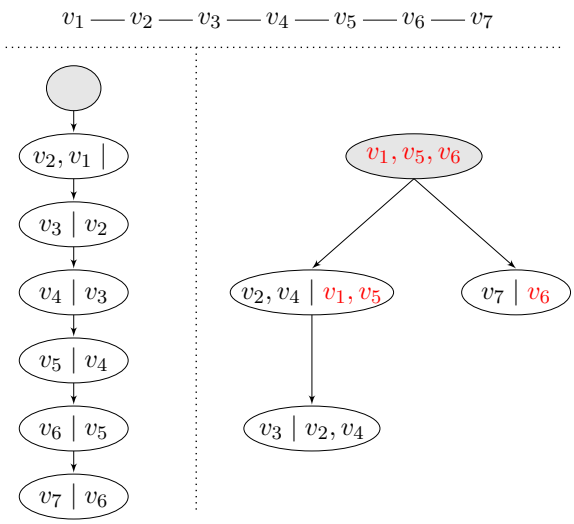


Figure 3.4: The hypergraph $\mathcal{H}$ for a path query of length 6, along with two C-connex tree decompositions. The decomposition on the left has $C = \emptyset$, and the decomposition on the right $C = \{v_1, v_5, v_6\}$. The variables in C are colored red, and the grey nodes are the ones in the set A.

Theorem 2. *Let $q = Q^\eta$ be an adorned view over a natural join query with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. Suppose that $\mathcal{H}$ admits a $\mathcal{V}_\mathsf{b}$-connex tree decomposition. Fix any delay assignment δ, and let f be the $\mathcal{V}_\mathsf{b}$-connex fractional hypertree δ-width, h the δ-height of the decomposition, and $u^* = \max_{t \in V(\mathcal{T}) \setminus A} u_t^+$.*

Then, for any input database D, we can construct a data structure in compression time $T_C = \tilde{O}(|D| + |D|^{u^* + \max_t \delta(t)})$ with space $S = \tilde{O}(|D| + |D|^f)$, such that we can answer any access request with delay $\tilde{O}(|D|^h)$.

If we write the delay in the above result as $\tilde{O}(\prod_{t \in P} |D|^{\delta(t)})$, where P is the maximum-weight path, Theorem 2 tells us that the delay is essentially multiplicative in the same branch of the tree, but additive across branches. Unlike Theorem 1, the lexicographic ordering of the result $q(D)$ for Theorem 2 now depends on the tree decomposition.

For our running example, Theorem 2 implies a data structure with space $\tilde{O}(|D| + |D|^{5/3})$ and delay $\tilde{O}(|D|^{1/2})$. This data structure can be computed in time $\tilde{O}(|D| + |D|^{7/3})$. Notice that this is much smaller than the $O(|D|^4)$ time required to compute the worst case output.

Consider an adorned view over a natural join query (with hypergraph $\mathcal{H}$), with bound variables $\mathcal{V}_\mathsf{b}$. Fix a $\mathcal{V}_\mathsf{b}$-connex tree decomposition $(\mathcal{T}, A)$. Observe that, since the bags in A do not play any role in the width or height, we can assume w.l.o.g. that A consists of a single bag t_b. We consider as the running example in this section the one in Figure 3.4.

3.4.1 Constant Delay Enumeration

Suppose now that our goal is to achieve constant delay. From Theorem 2, to do this we have to choose the delay assignment to be 0 everywhere. In this case, we have the following result (which slightly strengthens Theorem 2 in this special case by dropping the polylogarithmic dependence).

Proposition 12. *Let Q^η be a full adorned view over a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. Then, for any input database D, we can construct a data structure in compression time and space $S = O(|D|^{\mathsf{fhw}(\mathcal{H}|\mathcal{V}_\mathsf{b})})$, such that we can answer any access request with delay $O(1)$.*

Observe that when all variables are free, then $\mathcal{V}_\mathsf{b} = \emptyset$, in which case $\mathsf{fhw}(\mathcal{H} \mid \mathcal{V}_\mathsf{b}) = \mathsf{fhw}(\mathcal{H})$, thus recovering the compression result of a d-representation. Moreover, since the delay assignment is 0 for all bags, the compression time $T_C = \tilde{O}(|D| + |D|^{\mathsf{fhw}(\mathcal{H}|\mathcal{V}_\mathsf{b})})$. Further, we can take any optimal tree decomposition with fractional hypertree width $\mathsf{fhw}(\mathcal{H})$ (so a $\mathcal{V}_\mathsf{b}$-connex decomposition), materialize the tuples in each bag by running the query restricted on the vertices of the bag, and finally apply a sequence of semi-joins in a bottom-up order to remove any tuples from the bags that do not participate in the final result. Additionally, for each node $t \in V(\mathcal{T})$, we construct a hash index over the materialized result with key $\mathcal{V}_\mathsf{b}^t = \mathcal{B}_t \cap \mathsf{anc}(t)$, and output values for the variables in $\mathcal{V}_\mathsf{f}^t = \mathcal{B}_t \setminus \mathcal{V}_\mathsf{b}^t$. For example, the node with bag $\{v_2, v_3\}$ constructs an index with key v_2 that returns all the matching values of v_3.

Given such indexes, we can perform a full enumeration in constant delay starting from the root (which will be the empty bag), and visiting the nodes of the tree $\mathcal{T}$ in a pre-order fashion by following the indexes at each bag. This construction uses the same idea as d-representations [OZ15b], and requires space $O(|D|^{\mathsf{fhw}(\mathcal{H})})$.

When $\mathcal{V}_\mathsf{b} \neq \emptyset$, the standard tree decomposition may not be useful to achieve constant delay enumeration. For instance, for the example hypergraph of Figure 3.4, if the adorned view has $\mathcal{V}_\mathsf{b} = \{v_1, v_5, v_6\}$, then the pre-order traversal of the decomposition on the left will fail to achieve constant delay. However, we can use a $\mathcal{V}_\mathsf{b}$-connex decomposition to successfully answer any access request (*e.g.*, the right decomposition in Figure 3.4). We first materialize all the bags, except for the bag of t_b. Then, we run a sequence of semi-joins in a bottom-up manner, where we stop right before the node t_b (since it is not materialized). For each node $t \in V(\mathcal{T}) \setminus \{t_\mathsf{b}\}$, we construct a hash index over the materialized result with key $\mathcal{V}_\mathsf{b}^t$. Finally, for the root node t_b, we construct a hash index that tests membership for every hyperedge of $\mathcal{H}$ that is contained in $\mathcal{V}_\mathsf{b}$. For the example in Figure 3.4, we construct one such index for the hyperedge $\{v_5, v_6\}$.

Given a valuation v_b over $\mathcal{V}_\mathsf{b}$, we answer the access request as follows. We start by checking in constant time whether $v_\mathsf{b}(v_5, v_6)$ is in the input. Then, we use the hash index of the node $\{v_2, v_4, v_1, v_5\}$ to find the matching values for v_2, v_4 (since v_1, v_5 are bound by

v_{b}), and subsequently use the hash index of $\{v_3, v_2, v_4\}$ to find the matching values for v_3; similarly, we also traverse the right subtree starting of the root node to find the matching values of v_7. We keep doing this traversal until all tuples are enumerated. We describe next how this algorithm generalizes for any adorned view and beyond constant delay.

3.4.2 Beyond Constant Delay

We will now sketch the construction and query answering for the general case. Along with the tree decomposition, let us fix a delay assignment δ.

Construction Sketch. The first step is to apply for each node t in $\mathcal{T}$ (except t_{b}) the construction of the data structure from Theorem 1, with the following parameters: (i) hypergraph $\mathcal{H}' = (\mathcal{V}', \mathcal{E}')$ where $\mathcal{V}' = \mathcal{B}_t$ and $\mathcal{E}' = \mathcal{E}_{\mathcal{B}_t}$, (ii) bound variables $\mathcal{V}'_{\mathsf{b}} = \mathcal{V}^t_{\mathsf{b}}$, (iii) $\tau = |D|^{\delta(t)}$, and (iv) $\mathbf{u}$ the fractional edge cover that minimizes ρ^+_t. For the root node t_{b}, we simply construct a hash index that tests membership for every hyperedge of $\mathcal{H}$ that is contained in $\mathcal{V}_{\mathsf{b}}$. This construction uses for each bag space $\tilde{O}(|D| + |D|^{\rho^+_t - \delta(t)\cdot\alpha(\mathcal{V}^t_{\mathsf{f}})})$, which means that the compressed representation has size $\tilde{O}(|D| + |D|^f)$.

The second step is to set run a sequence of semi-joins in a bottom-up fashion. However, since the bags are not fully materialized anymore, this operation is not straightforward. Instead, we set any entry of the dictionary $\mathcal{D}_t(w, v_{\mathsf{b}})$ of the data structure at node t that is $\mathsf{1}$ to $\mathsf{0}$ if no valuation in the interval $\mathbf{I}(w)$ joins with its child bag. This step is necessary to guarantee that if we visit an interval in the delay-balanced tree with entry $\mathsf{1}$, we are certain to produce an output for the full query (and not only the particular bag). To perform this check, we do not materialize the bag of the child, but we simply use its dictionary (hence costing an extra factor of $\max_t \delta(t)$ during preprocessing time).

Detailed Construction. Let us now look at the detailed construction of data structure. Without loss of generality, we assume that all bound variables are present in a single bag t_{b} in the $\mathcal{V}_{\mathsf{b}}$-connex tree decomposition of the hypergraph $\mathcal{H}$. This can be achieved by simply merging all the bags t with $\mathcal{B}_t \subseteq \mathcal{V}_{\mathsf{b}}$ into t_{b} which is also designated as the root. Note that the delay assignment for root node is $\delta_{t_{\mathsf{b}}} = 0$. Let $\lambda(\mathcal{T})$ denote the set of all root to leaf paths in $\mathcal{T}$, h be the δ-height of the decomposition and f be the $\mathcal{V}_{\mathsf{b}}$-connex fractional hypertree δ-width of the decomposition. We also define the quantity $\mathbf{u}^* = \max_{t \in V(\mathcal{T}) \setminus t_{\mathsf{b}}}(\sum_F u_F)$ where u is the fractional edge cover for bag $\mathcal{B}_t$. We will show that in time $T_C = \tilde{O}(|D| + |D|^{\mathbf{u}^* + \max_t \delta_t})$, we can construct required data structure using space $S = \tilde{O}(|D| + |D|^f)$ for a given $\mathcal{V}_{\mathsf{b}}$-connex tree decomposition $\mathcal{T}$ of δ-width f. The construction will proceed in two steps:

(i) Apply Theorem 1 to decomposition. We apply theorem 1 to each bag (except t_{b}) in the decomposition with the following parameters: $(i)\mathcal{H}^t = (\mathcal{V}^t, \mathcal{E}^t)$ where $\mathcal{V}^t = \mathcal{B}_t$ and $\mathcal{E}^t = \mathcal{E}_{\mathcal{B}_t}$, $(ii)\mathcal{V}^t_{\mathsf{b}} = \mathsf{anc}(t) \cap \mathcal{B}_t$ and $\mathcal{V}^t_{\mathsf{f}} = \mathcal{B}_t \setminus \mathsf{anc}(t)$, and (iii) the edge cover u is the cover of the node t in the decomposition corresponding to ρ^+_t. Thus, in time $\tilde{O}(|D| + |D|^{\mathbf{u}^*})$ we can

the construct delay-balanced tree $\mathcal{T}_t$ and the corresponding dictionary $\mathcal{D}_t$ for each bag other than the root. The space requirement for each bag is no more than $\tilde{O}(|D| + |D|^f)$.

However, the dictionary $\mathcal{D}_t$ needs to be modified for each bag since there can be *dangling tuples* in a bag that may participate only in the join output of the bag but not in the join output of the branch containing t. In the following description, we will use the notation v_{b}^t to denote a valuation over variables $\mathcal{V}_{\mathsf{b}}^t$. Note that v_{b} is the valuation over $\mathcal{V}_{\mathsf{b}}$.

Algorithm 4: Modifying $(\mathcal{D})_{t \in V(\mathcal{T})}$

input : $\mathcal{V}_{\mathsf{b}}$-bound decomposition $\mathcal{T}$, $(\mathcal{T}, \mathcal{D})_{t \in V(\mathcal{T})}$

forall $t \in \mathcal{T} \setminus \{t_{\mathsf{b}} \cup$ *children of* $t_{\mathsf{b}}\}$ *in post-order fashion* **do**
  $\mathsf{parent} \leftarrow$ *parent of* t
  forall $w \in w_L$ *of* $\mathcal{T}_{\mathsf{parent}}$ /* w_L represents all nodes at level L in $\mathcal{T}_{\mathsf{parent}}$ */
  do
    forall *heavy* $v_{\mathsf{b}}^{\mathsf{parent}} \in w$ *and* $\mathcal{D}_{\mathsf{parent}}(w, v_{\mathsf{b}}^{\mathsf{parent}}) = 1$ **do**
      forall $k \leftarrow Q_{\mathsf{parent}}(v_{\mathsf{b}}^{\mathsf{parent}}, D) \ltimes \mathbf{I}(w)$ /* computing $(\bowtie_{F \in \mathcal{E}_{\mathcal{V}^t}} R_F)$ via box decomposition */
      do
        if *Algorithm 3 on* t *with* $v_{\mathsf{b}}^t = \pi_{\mathcal{B}_{\mathsf{parent}} \cap \mathcal{B}_t}(k)$ *is empty for all* k **then**
          $\mathcal{D}_t(w, \pi_{\mathcal{V}_{\mathsf{b}}^{\mathsf{parent}}}(k)) = 0$

(ii) Modify $\mathcal{D}_t$ using semijoins. Algorithm 4 shows the construction of the modified dictionary $\mathcal{D}_t$ to incorporate the semijoin result. The goal of this step is to ensure that if $\mathcal{D}_t(w, v_{\mathsf{b}}^t) = 1$, then there exists a set of valid valuations for all variables in the subtree rooted at t. We will apply a sequence of semijoin operations in a bottom-up fashion which we describe next.

Bottom Up Semijoin. In this phase, the bags are processed according to *post-order* traversal of the tree in bottom-up fashion. The key idea is to stream over all heavy valuations of a node in $\mathcal{T}_t$ and ensure that they join with some tuple in the child bags. Let $Q_t(v_{\mathsf{b}}^t, D)$ denote the NPRR join instance on the relations covering variables $\mathcal{V}^t$ where bound variables are fixed to v_{b}^t. When processing a non-root (or non-child of root) node t_j, a semijoin is performed with its parent t_i to flip all dictionary entries of t_i from 1 to 0 if the entry does not join with any tuple in t_j on their common attributes $\mathcal{B}_{t_i} \cap \mathcal{B}_{t_j}$. To perform this operation, we stream over all tuples $k \leftarrow Q_{t_i}(v_{\mathsf{b}}^{t_i}, D)$ and check if $\pi_{\mathcal{B}_{t_i} \cap \mathcal{B}_{t_j}}(k)$ is present in the join output of relations covering t_j. This check in bag t_j can be performed by invoking Algorithm 3 with bound valuation $\pi_{\mathcal{B}_{t_i} \cap \mathcal{B}_{t_j}}(k)$ in time $\tilde{O}(|D|^{\delta_{t_j}})$.

Algorithm Analysis. We will show that Algorithm 4 can be executed in time $\tilde{O}(|D|^{\mathbf{u}^* + \max_t \delta_t})$.

Proposition 13. *Algorithm 4 executes in time* $\tilde{O}(|D|^{\mathbf{u}^* + \max_t \delta_t})$.

Algorithm 5: Query Answering using $\mathcal{V}_{\mathsf{b}}$-connex decomposition

input : tree $\mathcal{T}$, $(\mathcal{T}, \mathcal{D})_{t \in V(\mathcal{T})}, v_b$

output: query answer $Q(D)$

1 Initialize $t_{visited} \leftarrow 0$ for all nodes, $v \leftarrow v_b, t \leftarrow$ left child of t_{b}, $parent(t) \leftarrow t$

2 Check if $R_F(v_{\mathsf{b}}) \neq \emptyset, F \in \mathcal{E}, F \subseteq C$

3 **forall** *nodes in pre-order traversal starting from* t **do**

4 $\quad v \leftarrow \pi_{\mathcal{V}^t_{\mathsf{pred}}}(v)$

5 $\quad v^t_{\mathsf{f}} \leftarrow \mathsf{next}_t(\pi_{\mathcal{V}^t_{\mathsf{b}}}(v))$

6 $\quad$ **if** v^t_{f} *is empty and* $t_{visited} = 0$ **then**

7 $\quad\quad t \leftarrow parent(t)$

8 $\quad\quad$ *continue*

9 $\quad$ **if** v^t_{f} *is empty and* $t_{visited} = 1$ **then**

10 $\quad\quad t_{visited} \leftarrow 0$

11 $\quad\quad t \leftarrow predecessor(t)$

12 $\quad\quad$ *continue*

13 $\quad t_{visited} \leftarrow 1$

14 $\quad v \leftarrow (v, v^t_{\mathsf{f}})$

15 $\quad$ **if** t *is last node in the tree* **then**

16 $\quad\quad$ **if** $R_F(v) \neq \emptyset, F \in \mathcal{E}$ **then**

17 $\quad\quad\quad$ emit v

18 $\quad\quad$ **go to** line 4 /* If t is last node in tree, find next valuation for $\mathcal{V}^t_{\mathsf{f}}$ */

Proof. The main observation is that the join $Q_{\mathsf{parent}}(v^t_{\mathsf{b}}, D) \ltimes \mathbf{I}(w)$ for all nodes at level L in $\mathcal{T}_{\mathsf{parent}}$ can be computed in time at most $O(|D|^{\mathbf{u}^*})$ as shown in Proposition 8. Since the operation in Line 8 can be performed in time $\tilde{O}(|D|^{\delta_t})$ for each k and $\mathcal{T}_{\mathsf{parent}}$ has at most logarithmic number of levels, the total overhead of the procedure is dominated by the semijoin operation where delay for bag t is largest. This gives us the running time of $\tilde{O}(|D|^{\mathbf{u}^* + \max_t \delta_t})$ for the procedure. □

Proposition 14. *If* $\mathcal{D}_t(w, v^t_{\mathsf{b}}) = 1$*, then there exists a set of valuations for all variables in the subtree rooted at* t *for* v^t_{b}*.*

Proof. Consider a valuation such that $\mathcal{D}_t(w, v^t_{\mathsf{b}}) = 1$. If v^t_{b} does not join with the relations of any child bag c, then Line 8 would be true, and Algorithm 4 would have flipped the dictionary entry to 0. Thus, there exists a valuation for $\mathcal{V}^c_{\mathsf{f}}$. Applying the same reasoning inductively to each child bag c till we reach the leaf nodes gives us the desired result. □

The modified dictionary, along with the enumeration algorithm, will guarantee that the valuation of free variables that is output by Algorithm 3 for a particular bag will also produce

an output for the entire query. Note that the main memory requirement of Algorithm 4 is only $O(1)$ pointers and the data structures for each bag which takes $\tilde{O}(|D| + |D|^f)$ space.

Query Answering Sketch. To answer an access query with valuation v_{b}, we start from the root node t_b of the decomposition and check using the indexes of the node whether v_{b} belongs to all relations R_F such that $F \subseteq \mathcal{V}_{\mathsf{b}}$. Then, we invoke Algorithm 3 on the leftmost child t_0 of t_b, which outputs a new valuation in time at most $\tilde{O}(|D|^{\delta(t_0)})$, or returns nothing. As soon as we obtain a new output, we recursively proceed to the next bag in pre-order traversal of $\mathcal{T}$, and find valuations for the (still free) variables in the bag. If there are no such valuations returned by Algorithm 3 for the node under consideration, this means that the last valuation outputted by the parent node does not lead to any output. In this case, we resume the enumeration for the parent node. Finally, when Algorithm 3 finishes the enumeration procedure, then we resume the enumeration for the pre-order predecessor of the current node (and not the parent). Intuitively, we go to the predecessor to fix our next valuation, to enumerate the cartesian product of all free variables in the subtree rooted at the least common ancestor of the current node and predecessor node.

The delay guarantee of $\tilde{O}(|D|^h)$ comes from the fact that, at every node t in the tree, we will output in time $\tilde{O}(|D|^{\delta(t)})$ at most $\tilde{O}(|D|^{\delta(t)})$ valuations, one of which will produce a final output tuple. Moreover, when a node has multiple children, then for a fixed valuation of the node, the traversal of each child is independent of the other children: if one subtree produces no result, then we can safely exit all subtrees and continue the enumeration of the node.

Detailed Algorithm. We present the detailed query answering algorithm for the given $\mathcal{V}_{\mathsf{b}}$-connex decomposition. We will first add some metadata to each bag in the decomposition and then invoke algorithm 3 for each bag in a pre-order fashion.

Adding pointers for each bag. Consider the decomposition $\mathcal{T}$ along with $(\mathcal{T}, \mathcal{D})_{t \in V(\mathcal{T}) \setminus t_{\mathsf{b}}}$ for each bag. We will modify $\mathcal{T}$ as follows: for each node of the tree, we fix a pointer $predecessor(t)$, that will point to the *pre-order predecessor* of the node. Intuitively, a pre-order predecessor of a node is the last node where valuation for a free variable will be fixed in the pre-order traversal of the tree just before visiting the current node. Figure 3.5 shows an example of a modified decomposition. This transformation can be done in $O(1)$ time.

For ease of description of the algorithm, we assume that the Algorithm 3 answering $Q^{\eta}[v_{\mathsf{b}}^t]$ for any bag t is accessible using the procedure $\mathsf{next}_t(v_{\mathsf{b}}^t)$. Let $\mathcal{V}_{\mathsf{pred}}^t$ represent all bound variables encountered in the pre-order traversal of the tree from t_{b} to t (including bound variables of t).

Algorithm Description. The algorithm begins from the root node and fixes the valuation for all free variables in the root bag. Then, it proceeds to the next bag recursively considering all ancestor variables as bound variables and finds a valuation for $\mathcal{B}_t \setminus \mathsf{anc}(t)$. At the first

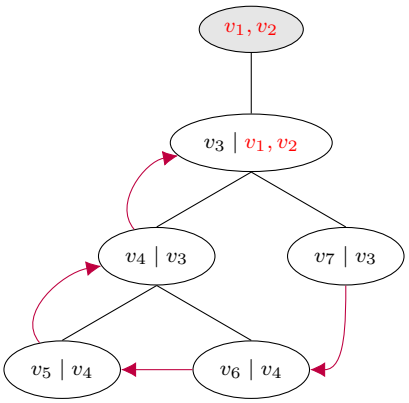


Figure 3.5: Example of the modified tree decomposition: the arrows in color are the predecessor pointers

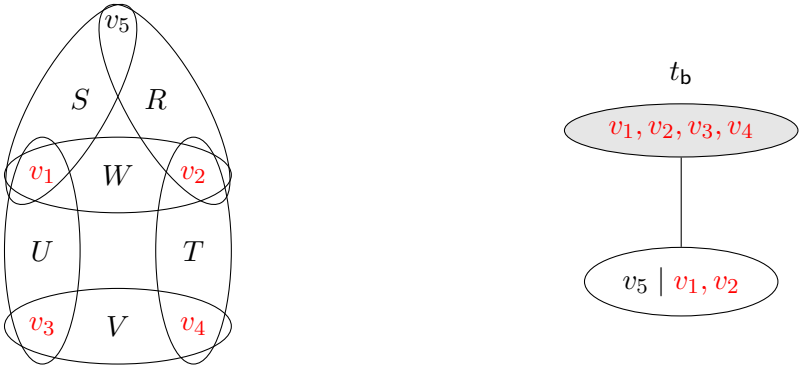


Figure 3.6: Query hypergraph and corresponding C-bound tree decomposition with $C = \{v_1, v_2, v_3, v_4\}$

visit to any bag, if the bound variables v_b^t do not produce an output in delay $\tilde{O}(|D|^{\delta_t})$, then we proceed to the next valuation in the parent bag. However, if the enumeration for some v_b^t did produce output tuples but the procedure $\mathsf{next}_t(v_\mathsf{b}^t)$ has finished, we proceed to the predecessor of the bag to fix the next valuation for variables in predecessor bag. In other words, the ancestor variables remain fixed and we enumerate the cartesian product of the remaining variables.

Lemma 10. *Algorithm 5 enumerates the answers with delay at most $\tilde{O}(|D|^h)$ where h is the δ-height of the decomposition tree. Moreover, it requires at most $O(\log |D|)$ memory*

Proof. Since the size of the decomposition is a constant, we require at most $O(1)$ pointers for predecessors and $O(1)$ pointers for storing the valuations of each free variable. Let n_ℓ be the set of nodes at depth ℓ. We will express the delay of the algorithm in terms of the delay of the subtrees of every node. The delay at the root t_b after checking whether valuation v_b is in the base relations is $d_{t_\mathsf{b}} = O(\sum_{t \in n_1} d_t)$. This is because the enumeration of each subtree rooted

at depth $\ell = 1$ depends only on its ancestor variables and is thus independent of the other subtrees at that depth. Since each node in the tree can produce at most $|D|^{\delta_t}$ valuations in $\tilde{O}(|D|^{\delta_t})$ time, the recursive expansion of δ_{t_b} gives $\delta_{t_\mathsf{b}} = O(\sum_{p \in \lambda(\mathcal{T})} \tilde{O}(|D|^{\sum_{t \in p} \delta_t}))$. The largest term over all root to leaf paths is $\tilde{O}(|D|^h)$ which gives us the desired delay guarantee. □

3.4.2.1 Comparing width notions

We briefly discuss the connection of $\mathsf{fhw}(\mathcal{H} \mid \mathcal{V}_\mathsf{b})$ for a $\mathcal{V}_\mathsf{b}$-connex decomposition with other related hypergraph notions. The first observation is that the minimum edge cover number ρ^* is always an upper bound on $\mathsf{fhw}(\mathcal{H} \mid \mathcal{V}_\mathsf{b})$. On the other hand, the $\mathsf{fhw}(\mathcal{H} \mid \mathcal{V}_\mathsf{b})$ is incomparable with $\mathtt{fhw}(\mathcal{H})$. Indeed, Example 20 shows that $\mathsf{fhw}(\mathcal{H} \mid \mathcal{V}_\mathsf{b}) < \mathtt{fhw}(\mathcal{H})$, and the example below shows that the inverse situation can happen as well.

Example 19. *The query $R(x, y), S(y, z)$ is acyclic and has $\mathtt{fhw}(\mathcal{H}) = 1$. Let $\mathcal{V}_\mathsf{b} = \{x, z\}$. The only valid $\mathcal{V}_\mathsf{b}$-bound decomposition is the one with two bags, $\{x, z\}, \{x, y, z\}$, and hence $\mathsf{fhw}(\mathcal{H} \mid \mathcal{V}_\mathsf{b}) = 2$. In this scenario, $\mathsf{fhw}(\mathcal{H} \mid \mathcal{V}_\mathsf{b}) > \mathtt{fhw}(\mathcal{H})$.*

Example 20. *Figure 3.6 shows an example hypergraph and a $\mathcal{V}_\mathsf{b}$-bound tree decomposition (the variables in $\mathcal{V}_\mathsf{b}$ are colored red). For this example, $\mathsf{fhw}(\mathcal{H}) = 2$, but $\mathsf{fhw}(\mathcal{H} \mid \mathcal{V}_\mathsf{b}) = 3/2$. Indeed, observe that we can cover the lower bag of the tree decomposition with a fractional edge cover of value only $3/2$.*

3.5 The Complexity of Minimizing Delay

In this section, we study the computational complexity of choosing the optimal parameters for Theorem 1 and Theorem 2. We identify two objectives that guide the parameter choice: (i) given a space constraint, minimize the delay, and (ii) given a delay constraint, minimize the necessary space.

We start with the following computational task, which we call MinDelayCover. We are given as input a full adorned view Q^η over a CQ, the sizes $|R_F|$ of each relation F, and a positive integer Σ as a space constraint. The size of Q^η, denoted $|Q^\eta|$, is defined as the length of Q^η when viewed as a word over alphabet that consists of a variable set $\mathcal{V}$, **dom** and atoms in the body of the query. The goal is to output a fractional edge cover $\mathbf{u}$ that minimizes the delay in Theorem 1, subject to the space constraint $S \leq \Sigma$.

We observe that we can express MinDelayCover as a linear fractional program with a bounded and non-empty feasible region. Such a program can always be transformed to an equivalent linear program [CC62], which means that the problem can be solved in polynomial time.

minimize	τ
subject to	$\rho \log \lvert D\rvert \leq \log \lvert\Sigma\rvert + \alpha \log \tau$
	$\rho = \sum_{F \in \mathcal{E}} u_F$
	$\forall x \in \mathcal{V}_\mathsf{f} : \sum_{F:x \in F} u_F \geq \alpha$
	$\forall x \in \mathcal{V} : \sum_{F:x \in F} u_F \geq 1$
	$\forall F \in \mathcal{E} : 0 \leq u_F \leq 1$
	$\alpha \geq 1$

(a) Linear program with bilinear constraint

minimize	$\hat{\tau}/\alpha$
subject to	$\rho \log \lvert D\rvert \leq \log \lvert\Sigma\rvert + \hat{\tau}$
	$\rho = \sum_{F \in \mathcal{E}} u_F$
	$\forall x \in \mathcal{V}_\mathsf{f} : \sum_{F:x \in F} u_F \geq \alpha$
	$\forall x \in \mathcal{V} : \sum_{F:x \in F} u_F \geq 1$
	$\forall F \in \mathcal{E} : 0 \leq u_F \leq 1$
	$\alpha, \hat{\tau} \geq 1$

(b) Transformed linear fractional program

Figure 3.7: Left to right: Bilinear program to minimize delay; Equivalent linear fractional program

Proposition 15. MinDelayCover *can be solved in polynomial time in the size of the adorned view, the relation sizes, and the space constraint.*

Proof. Consider the bilinear program in Figure 3.7a. Without loss of generality, assume that all relations are of the same size. The first constraint ensures that $|D|^{\sum u_F}/\tau^\alpha \leq \Sigma$, while the fourth constraint encodes the fractional edge covers. However, the program is not an LP as $\alpha \log \tau$ is a bilinear constraint. We can easily transform it into a linear fractional program as shown in Figure 3.7b where $\hat{\tau} = \alpha \log \tau$. Notice that we can replace the objective in program 3.7a from τ to $\log \tau$ without changing the optimal solution. The key idea is that we can convert the linear fractional program to a linear program using the Charnes-Cooper transformation [CC62] provided that the feasible region is bounded and non-empty. Our claim follows from the observation that the region is indeed bounded since $u_F \leq 1, \alpha \leq |Q|, \hat{\tau} \leq |Q|^2 \log |D|$ and non-empty as $u_F = 1, \alpha = 1, \tau = |D|^{|Q|}$ is a valid solution. □

We also consider the inverse task, called MinSpaceCover: given as input a full adorned view Q^η over a CQ, the sizes $|R_F|$ of each relation F, and a positive integer Δ as a delay constraint, we want to output a fractional edge cover $\mathbf{u}$ that minimizes the space S in Theorem 1, subject to the delay constraint $\tau \leq \Delta$.

To solve MinSpaceCover, observe that we can simply perform a binary search over the space parameter S, from $|D|$ to $|D|^k$, where k is the number of atoms in Q. For each space, we then run MinDelayCover and check whether the minimum delay returned satisfies the delay constraint.

Proposition 16. MinSpaceCover *can be solved in polynomial time in the size of the adorned view, the relation sizes, and the delay constraint.*

We next turn our attention to how to optimize for the parameters in Theorem 2.

Suppose we are given a full adorned view Q^η over a CQ, the database size $|D|$, and a space constraint Σ, and we want to minimize the delay. If we are given a fixed $\mathcal{V}_\mathtt{b}$-connex tree decomposition, then we can compute the optimal delay assignment δ and optimal fractional edge cover for each bag as follows: we iterate over every bag in the tree decomposition, and then solve MinDelayCover for each bag using the space constraint. It is easy to see that the delay that we obtain in each bag must be the delay of an optimal delay assignment. For the inverse task where we are given a $\mathcal{V}_\mathtt{b}$-connex tree decomposition, a delay constraint and our goal is to minimize the space, we can apply the same binary search technique as in the case of MinSpaceCover (observe the that δ-height is also easily computable in polynomial time). In other words, we can compute the optimal parameters for our given objective in polynomial time, as long as we are provided with a tree decomposition.

In the case where the tree decomposition is not given, then the problem of finding the optimal data structure according to Theorem 2 becomes intractable. Indeed, we have already seen that if we want to achieve constant delay $\tau = 1$, then the tree decomposition that minimizes the space S is the one that achieves the $\mathcal{V}_\mathtt{b}$-connex fractional hypertree width, $\mathtt{fhw}(\mathcal{H} \mid \mathcal{V}_\mathtt{b})$. Since for $\mathcal{V}_\mathtt{b} = \emptyset$ we have $\mathtt{fhw}(\mathcal{H} \mid \mathcal{V}_\mathtt{b}) = \mathtt{fhw}(\mathcal{H})$, and finding the optimal fractional hypertree width is NP-hard [GGS14], finding the optimal tree decomposition for our setting is also NP-hard.

Chapter 4

Space-Time Tradeoffs for Answering Boolean Conjunctive Queries

Recent work has made remarkable progress in developing data structures and algorithms for answering set intersection problems [GKLP17], reachability oracles and directed reachability [AGHP11, Aga14, CP10b], histogram indexing [CL15, KRR13], and problems related to document retrieval [AN16, LMNT15]. A fundamental algorithmic question related to these problems is the trade-off between the space S necessary for data structures and the answering time T for requests.

For example, consider the 2-Set Disjointness problem: given a universe of elements U and a collection of m sets $C_1, \ldots, C_m \subseteq U$, we want to create a data structure such that for any pair of integers $1 \leq i, j \leq m$, we can efficiently decide whether $C_i \cap C_j$ is empty or not. Previous work [CP10b, GKLP17] has shown that the space-time trade-off for 2-Set Disjointness is captured by the equation $S \cdot T^2 = N^2$, where N is the total size of all sets. The data structure obtained is conjectured to be optimal [GKLP17], and its optimality was used to develop conditional lower bounds for other problems, such as approximate distance oracles [AGHP11, Aga14]. Similar trade-offs have been independently established for other data structure problems as well. In the k-Reachability problem [GKLP17, CP10a] we are given as an input a directed graph $G = (V, E)$, an arbitrary pair of vertices u, v, and the goal is to decide whether there exists a path of length k between u and v. In the edge triangle detection problem [GKLP17], we are given an input undirected graph $G = (V, E)$, the goal is to develop a data structure that takes space S and can answer in time T whether a given edge $e \in E$ participates in a triangle or not. Each of these problems has been studied in isolation and, as a result, the algorithmic solutions are not generalizable due to a lack of a comprehensive framework.

In this chapter, we cast many of the above problems into answering CQs over a relational database. CQs are a powerful class of relational queries with widespread applications in data analytics and graph exploration [XKD15, XD17b, DK18]. For example, by using the relation $R(x, y)$ to encode that element x belongs to set y, 2-Set Disjointness can be captured by the

following CQ: $Q(y_1, y_2) = R(x, y_1), R(x, y_2)$. As we will see later, k-Reachability can also be naturally captured by a CQ.

The insight of casting data structure problems into CQs over a database allows for a unified treatment for developing algorithms within the same framework, which in turn allows for improved algorithms and data structures. In particular, we can leverage the techniques developed by the data management community through a long line of research on efficient join evaluation [Yan81, NRR13, NPRR12], including worst-case optimal join algorithms [NPRR12] and tree decompositions [GGS14, RS86]. The use of these techniques has been a subject of previous work [AKKNS20, GS13, DK18, OS16, KNOZ20a, KNN$^+$19] for enumerating query results under static and dynamic settings. In this chapter, we build upon the aforementioned techniques to develop a framework that allows us to obtain general space-time tradeoffs for any Boolean CQ (a Boolean CQ is one that outputs only true or false). As a consequence, we recover state-of-the-art tradeoffs for several existing problems (e.g., 2-Set Disjointness as well as its generalization k-Set Disjointness and k-Reachability) as special cases of the general trade-off. We can even obtain improved tradeoffs for some specific problems, such as edge triangles detection, thus falsifying existing conjectures.

Our Contribution. We summarize our main technical contributions below.

1. ***A Comprehensive Framework.*** We propose a unified framework that captures several widely-studied data structure problems. More specifically, we resort to the formalism of CQs and the notion of *Boolean adorned queries*, where the values of some variables in the query are fixed by the user (denoted as an *access pattern*) and aim to evaluate the Boolean query. We then show how this framework captures the 2-Set Disjointness and k-Reachability problems. Our first main result (Theorem 3) is an algorithm that builds a data structure to answer any Boolean CQ under a specific access pattern. Importantly, the data structure can be tuned to trade off space for answering time, thus capturing the full continuum between optimal space and answering time. At one extreme, the data structure achieves constant answering time by explicitly storing all possible answers. At the other extreme, the data structure stores nothing, but we execute each request from scratch. We show how to recover existing and new tradeoffs using this general framework. The first main result may sometimes lead to suboptimal tradeoffs since it does not take into account the structural properties of the query. Our second main result (Theorem 4) combines tree decompositions of the query structure with access patterns to improve space efficiency. We then show how this algorithm can handle Boolean CQs with negation.

2. ***Improved Algorithms.*** In addition to the main result above, we explicitly improve the best-known space-time trade-off for the k-Reachability problem for $k \geq 4$. For any $k \geq 2$, the trade-off of $S \cdot T^{2/(k-1)} = O(|E|^2)$ was conjectured to be optimal

by [GKLP17], where $|E|$ is the number of edges in the graph and was used to conditionally prove other lower bounds on space-time tradeoffs. We show that for a regime of answer time T, it can be improved to $S \cdot T^{2/(k-2)} = O(|E|^2)$, thus breaking the conjecture. To the best of our knowledge, this is the first non-trivial improvement for the k-Reachability problem. We also refute a lower bound conjecture for the edge triangles detection problem established by [GKLP17].

3. ***Conditional Lower Bounds.*** Finally, we show a reduction between lower bounds for the problem of k-Set Disjointness for different values of k, which generalizes the 2-Set Disjointness to computing the intersection between k given sets, for $k \geq 2$.

Organization. In the next section, we will formally describe the problem statement. Section 4.2 describes the general algorithm applicable for any boolean CQ. This trade-off can be further improved using tree decompositions in Section 4.3. Using the ideas introduced in these two sections the children's, we show how existing conjectures can be disproved in Section 4.5. Lastly, we present some new lower bounds in Section 4.6.

4.1 Problem Statement

We say that an adorned query is *Boolean* if every head variable is bound. In this case, the answer for every access request is also Boolean, i.e., true or false. We will use the concept of adorned queries introduced in the previous chapter.

Given a Boolean adorned query Q^η and an input database D, our goal is to construct a data structure, such that we can answer any access request that conforms to the access pattern η as fast as possible. Our goal is to study the relationship between the space of the data structure S and the answering time T for a given adorned query Q^η.

4.2 General Space-Time Tradeoffs

We can now state our first main theorem.

Theorem 3. *Let Q^η be a Boolean adorned query with hypergraph $(\mathcal{V}, \mathcal{E})$. Let $\mathbf{u}$ be any fractional edge cover of $\mathcal{V}$. Then, for any input database D, we can construct a data structure that answers any access request in time $O(T)$ and takes space*

$$S = O\left(|D| + \prod_{F \in \mathcal{E}} |R_F|^{u_F} / T^\alpha\right)$$

Proof. Let $\mathcal{V}_\mathsf{b} = \{x_1, \ldots, x_k\}$. Recall that an access request $\mathbf{a} = (a_1, \ldots, a_k)$ corresponds to the query $Q[a_1/x_1, \ldots, a_k/x_k]$; in other words, we substitute each occurrence of a bound variable x_i with the constant a_i. Define the hypergraph $\mathcal{H}_\mathsf{b} = (\mathcal{V}_\mathsf{b}, \mathcal{E}_\mathsf{b})$, where $\mathcal{E}_\mathsf{b} = \{F \cap \mathcal{V}_\mathsf{b} \mid$

$F \in \mathcal{E}\}$. We say that an access request $\mathbf{a}$ is *valid* if it is an answer for the query Q^{b} corresponding to $\mathcal{H}_{\mathsf{b}}$, i.e. $\mathbf{a} \in Q^{\mathsf{b}}(D)$. We can construct hash indexes of linear size $O(|D|)$ during the preprocessing phase so that we can check whether any access request is valid in constant time $O(1)$.

For every relation R_F in the query, let $R_F(\mathbf{a}) = \sigma_{x_i=a_i|x_i\in\mathcal{V}_{\mathsf{b}}\cap F}(R_F)$. In other words, $R_F(\mathbf{a})$ is the subrelation that we obtain once we filter out the tuples that satisfy the selection condition implied by the access request.

If α is the slack for the fractional edge cover $\mathbf{u}$, define $\hat{u}_F = u_F/\alpha$ for every $F \in \mathcal{E}$. As we have discussed earlier, $\hat{\mathbf{u}} = \{\hat{u}_F\}_{F\in\mathcal{E}}$ is a fractional edge cover for the query $Q[a_1/x_1, \ldots, a_k/x_k]$: indeed, it is necessary to cover only the non-bound variables, since all bound variables are replaced by constants in the query. Hence, using a worst-case optimal join algorithm, we can compute the access request $Q[a_1/x_1, \ldots, a_k/x_k]$ with running time

$$T(\mathbf{a}) = \prod_{F\in\mathcal{E}} |R_F(\mathbf{a})|^{u_F/\alpha}.$$

We can now describe the preprocessing phase and the data structure we build. The data structure simply creates a hash index. Let J be the set of valid access requests such that $T(\mathbf{a}) > T$. For every $\mathbf{a} \in J$, we add to the hash index a key-value entry, where the key is $\mathbf{a}$ and the value the (boolean) answer to the access request $Q[a_1/x_1, \ldots, a_k/x_k]$.

We claim that the answer time using the above data structure is at most $O(T)$. Indeed, we first check whether $\mathbf{a}$ is valid, which we can do in constant time. If it is not valid, we simply output no. If it is valid, we probe the hash index. If $\mathbf{a}$ exists in the hash index, we obtain the answer in time $O(1)$ by reading the value of the corresponding entry. Otherwise, we know that $T(\mathbf{a}) < T$ and hence we can compute the answer to the access request in time $O(T)$.

It remains to bound the size of the data structure we constructed during the preprocessing phase. Since the size is $O(|J|)$, we will bound the size of J. Indeed, we have:

$$\begin{aligned}
T \cdot |J| \leq \sum_{\mathbf{a}\in J} T(\mathbf{a}) &= \sum_{\mathbf{a}\in J} \prod_{F\in\mathcal{E}} |R_F(\mathbf{a})|^{u_F/\alpha} \\
&= \sum_{\mathbf{a}\in J} 1^{1-1/\alpha} \cdot \left(\prod_{F\in\mathcal{E}} |R_F(\mathbf{a})|^{u_F}\right)^{1/\alpha} \\
&\leq \left(\sum_{\mathbf{a}\in J} 1\right)^{1-1/\alpha} \cdot \left(\sum_{\mathbf{a}\in J} \prod_{F\in\mathcal{E}} |R_F(\mathbf{a})|^{u_F}\right)^{1/\alpha} \\
&\leq |J|^{1-1/\alpha} \cdot \prod_{F\in\mathcal{E}} |R_F|^{u_F/\alpha}
\end{aligned}$$

Here, the first inequality follows directly from the definition of the set J. The second inequality is Hölders inequality. The third inequality is an application of the query decomposition lemma from [NRR13]. By rearranging the terms, we can now obtain the desired bound. □

We should note that Theorem 3 applies when the relation sizes are different; this gives us sharper upper bounds compared to the case where each relation is bounded by the total size of the input. Indeed, if using $|D|$ as an upper bound on each relation, we obtain a space requirement of $O(|D|^{\rho^*}/T^{\alpha})$ for achieving answering time $O(T)$, where ρ^* is the fractional edge cover number. Since $\alpha \geq 1$, this gives us at worst a linear trade-off between space and time, i.e., $S \cdot T = O(|D|^{\rho^*})$. For cases where $\alpha \geq 1$, we can obtain much better trade-off.

Example 21. *Consider the query $Q^{\mathsf{b}\dots\mathsf{b}}(y_1, \dots, y_k) = R_1(x, y_1), R_2(x, y_2), \dots, R_k(x, y_k)$. We obtain an improved trade-off: $S \cdot T^k = O(|D|^k)$. Note that this result matches the best-known space-time trade-off for the k-Set Disjointness problem [GKLP17]. (Note that all atoms use the same relation symbol R, so $|R_i| = |D|$ for every $i = 1, \dots, k$.)*

4.3 Space-Time Tradeoffs via Tree Decompositions

Theorem 3 does not always give us the optimal trade-off. For the k-reachability problem with the adorned query $Q^{\mathsf{bb}}(x_1, x_{k+1}) = R_1(x_1, x_2), \dots, R_k(x_k, x_{k+1})$, Theorem 3 gives a trade-off $S \cdot T = |D|^{\lceil (k+1)/2 \rceil}$, by taking the optimal fractional edge covering number $\rho^* = \lceil (k+1)/2 \rceil$ and slack $\alpha = 1$, which is far from efficient. In this section, we will show how to leverage tree decompositions to further improve the space-time trade-off in Theorem 3.

Theorem 4. *Let Q^{η} be a Boolean adorned query with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. Consider any $\mathcal{V}_{\mathsf{b}}$-connex tree decomposition of $\mathcal{H}$. For some parametrization δ of the decomposition, let f be its δ-width, and h be its δ-height. Then, for any input database D, we can construct a data structure that answers any access request in time $T = O(|D|^h)$ in space $S = O(|D| + |D|^f)$.*

Proof. We recall some of the notation from Section 3.4. Let $\mathcal{T} = (\mathcal{T}, A)$ denote the $\mathcal{V}_{\mathsf{b}}$-connex tree decomposition with f as its δ-width, and h as its δ-height. For each node $t \in \mathcal{T} \setminus A$, we denote by $\mathsf{anc}(t)$ the union of all the bags for the nodes that are the ancestors of t, $\mathcal{V}_{\mathsf{b}}^t = \mathcal{B}_t \cap \mathsf{anc}(t)$ and $\mathcal{V}_{\mathsf{f}}^t = \mathcal{B}_t \setminus \mathcal{V}_{\mathsf{b}}^t$. Intuitively, $\mathcal{V}_{\mathsf{b}}^t(\mathcal{V}_{\mathsf{f}}^t)$ are the bound (resp. free) variables for the bag t as we traverse the tree top-down.

Data Structure Construction We apply Theorem 3 to each bag (except the root bag) in $\mathcal{T}$ with the following parameters: $(i)\mathcal{H}^t = (\mathcal{V}^t, \mathcal{E}^t)$ where $\mathcal{V}^t = \mathcal{B}_t$ and $\mathcal{E}^t = \mathcal{E}_{\mathcal{B}_t}$; (ii) bound variables are $\mathcal{V}_{\mathsf{b}}^t = \mathsf{anc}(t) \cap \mathcal{B}_t$; and (iii) fractional edge cover $\mathbf{u}$ corresponding to bag $\mathcal{B}_t$. The space requirement for the data structure corresponding to bag $\mathcal{B}_t$ is $S = O(|D| + |D|^{\rho_t(\delta)}) \leq$

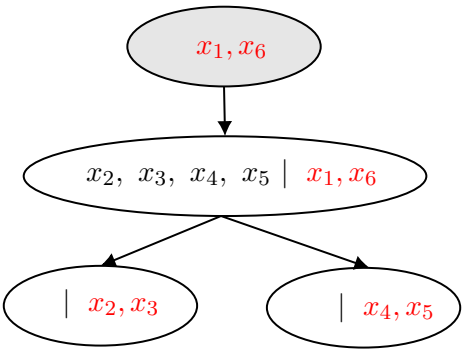


Figure 4.1: C-connex decomposition for Example 22.

$O(|D| + |D|^f)$. This follows directly from the definition of $\delta(t)$ width of bag t which is assumed to be at most f. Recall that the data structure stores a list of all access requests $\mathbf{a}$ defined over the schema $\mathcal{V}_\mathsf{b}^t$ such that answering time $T > O(|D|^f)$. Let us call this stored list as $\mathcal{L}(t)$.

Query Answering We now describe the query answering algorithm. Let $C = \{x_1, \ldots, x_k\}$ and access request $\mathbf{a} = (a_1, \ldots, a_k)$. We first need to check whether $\mathbf{a}$ is valid. If the request is not valid, we can simply output no. This can be done in constant time after creating hash indexes of size $O(|D|)$ during the preprocessing phase. If the access request is valid, the second step is to check whether $Q(\mathbf{a})$ is true or false. Let $\mathcal{P}$ denote the set of bags that are children of root bag. Then, for each bag $\mathcal{B}_t \in \mathcal{P}$, we check whether $\pi_{\mathcal{V}_\mathsf{b}^t}(\mathbf{a}) \in \mathcal{L}(t)$. If it is stored, it means that that running time of $\pi_{\mathcal{V}_\mathsf{b}^t}(\mathbf{a})$ is greater than $O(|D|^{\delta(t)})$. If the entry for $\pi_{\mathcal{V}_\mathsf{b}^t}(\mathbf{a})$ is false in the data structure, we can output false immediately since we know that no output tuple can be formed by the subtree rooted at bag $\mathcal{B}_t$.

If there is no entry for $\pi_{\mathcal{V}_\mathsf{b}^t}(\mathbf{a})$ in $\mathcal{L}(t)$, this means that answering time of evaluating the join at node t is $T \leq O(|D|^{\delta(t)})$. Thus, we can evaluate the join for the bag by fixing $\mathcal{V}_\mathsf{b}^t$ as $\pi_{\mathcal{V}_\mathsf{b}^t}(\mathbf{a})$ using any worst-case optimal join algorithm, which guarantees that the total running time is at most $O(|D|^{\delta(t)})$. If no output is generated, the algorithm outputs false since no output tuple can be formed by subtree rooted at $\mathcal{B}_t$. If there is output generated, then there can be at most $O(|D|^{\delta(t)})$ tuples. For each of these tuples, we recursively proceed to the children of bag $\mathcal{B}_t$ and repeat the algorithm. Each fixing of variables at bag t acts as the bound variables for the children's bag. In the worst case, all bags in $\mathcal{T}$ may require join processing. Since the query size is a constant, it implies that the number of root to leaf paths is also constant. Thus, the answering time is dominated by the longest root to leaf path, i.e the δ-height of the decomposition. Thus, $T = O(|D|^{\sum_{t \in P} \delta(t)}) = O(|D|^h)$. □

4.4 Extension to CQs with Negation

In this section, we present a simple but powerful extension of our result to adorned Boolean CQs with negation. Given a query $Q \in CQ^{\neg}$, we build the data structure from Theorem 4 for Q^+ but impose two constraints on the decomposition: (i) no leaf node(s) contains any free variable, (ii) for every negated relation R^-, all variables of R^- must appear together as bound variables in some leaf node(s). In other words, there exists a leaf node such that $\mathsf{vars}(R^-)$ are present in it. It is easy to see that such a decomposition always exists. Indeed, we can fix the root bag to be $C = \mathcal{V}_\mathsf{b}$, its child bag with free variables as $\mathsf{vars}(Q^+) \setminus C$ and bound variables as C, and the leaf bag, which is connected to the child of the root, with bound variables as $\mathsf{vars}(Q^-)$ without free variables. Observe that the bag containing $\mathsf{vars}(Q^+)$ free variables can be covered by only using the positive atoms since Q is safe. The intuition is the following: during the query answering phase, we wish to find the join result overall variables $\mathcal{V}_\mathsf{f}$ before reaching the leaf nodes; and then, we can check whether there the tuples satisfy the negated atoms or not, in $O(1)$ time. The next example shows the application of the algorithm to adorned path queries containing negation.

Example 22. *Consider the query $Q^{\mathsf{bb}}(x_1, x_6) = R(x_1, x_2), \neg S(x_2, x_3), T(x_3, x_4), \neg U(x_4, x_5), V(x_5, x_6)$. Using the decomposition in Figure 4.1, we can now apply Theorem 4 to obtain the trade-off $S = O(|D|^3/\tau)$ and $T = O(\tau)$. Both leaf nodes only require linear space since a single atom covers the variables. Given an access request $x_1 \leftarrow a, x_6 \leftarrow b$, we check whether the answer for this request has been materialized or not. If not, we proceed to the query answering phase and find at most $O(\tau)$ answers after evaluating the join in the middle bag. For each of these answers, we can now check in constant time whether the tuples formed by values for x_2, x_3 and x_4, x_5 are not present in relations S and U respectively.*

For adorned queries where $\mathcal{V}_\mathsf{b} \subseteq \mathsf{vars}(Q^-)$, we can further simplify the algorithm. In this case, we no longer need to create a constrained decomposition since the check to see if the negated relations are satisfied or not can be done in constant time at the root bag itself. Thus, we can directly build the data structure from Theorem 4 using the query Q^+.

Example 23 (Open Triangle Detection)**.** *Consider the query $Q^{\mathsf{bb}}(x_2, x_3) = R_1(x_1, x_2), \neg R_2(x_2, x_3), R_3(x_1, x_3)$, where Q^- is $\neg R_2(x_2, x_3)$ and Q^+ is $R_1(x_1, x_2), R_3(x_1, x_3)$ with the adorned view as $Q^{+\mathsf{bb}}(x_2, x_3) = R_1(x_1, x_2), R_3(x_1, x_3)$. Observe that $\{x_2, x_3\} \subseteq \mathsf{vars}(Q^-)$. We apply Theorem 4 to obtain the trade-off $S = O(|E|^2/\tau^2)$ and $T = O(\tau)$ with root bag $C = \{x_2, x_3\}$, its child bag with $\mathcal{V}_\mathsf{b} = C$ and $\mathcal{V}_\mathsf{f} = \{x_1\}$, and the leaf bag to be $\mathcal{V}_\mathsf{b} = C$ and $\mathcal{V}_\mathsf{f} = \emptyset$. Given an access request $x_2 \leftarrow a, x_3 \leftarrow b$, we check whether the answer for this request has been materialized or not. If not, we traverse the decomposition and evaluating the join to find if there exists a connecting value for x_1. For the last bag, we simply check whether (a, b) exists in R_2 or not in $O(1)$ time.*

A note on optimality. It is easy to see that the algorithm obtained for Boolean CQs with negation is conditionally optimal assuming the optimality of Theorem 4. Indeed, if all negated relations are empty, the join query is equivalent to Q^+, and the algorithm now simply applies Theorem 4 to Q^+. In example Example 23, assuming relation R_2 is empty, the query is equivalent to set intersection whose tradeoffs are conjectured to be optimal.

4.5 Path Queries

In this section, we present an algorithm for the adorned query $P_k^{\mathsf{bb}}(x_1, x_{k+1}) = R_1(x_1, x_2), \ldots, R_k(x_k, x_{k+1})$ that improves upon the conjectured optimal solution. Before diving into the details, we first state the upper bound on the trade-off between space and query time.

Theorem 5 (due to [GKLP17]). *There exists a data structure for solving $P_k^{\mathsf{bb}}(x_1, x_{k+1})$ with space S and answering time T such that $S \cdot T^{2/(k-1)} = O(|D|^2)$.*

Note that for $k = 2$, the problem is equivalent to SetDisjointness with the space/time trade-off as $S \cdot T^2 = O(N^2)$. [GKLP17] also conjectured that the trade-off is essentially optimal.

Conjecture 1 (due to [GKLP17]). *Any data structure for $P_k^{\mathsf{bb}}(x_1, x_{k+1})$ with answering time T must use space $S = \tilde{\Omega}(|D|^2/T^{2/(k-1)})$.*

If k is not a constant, Conjecture 1 implies that $\Theta(|D|^2)$ space is needed for achieving $O(1)$ answering time. Building upon Conjecture 1, [GKLP17] also showed a result on the optimality of approximate distance oracles. Our results implies that Theorem 5 can be improved further, thus refuting Conjecture 1. The first observation is that the trade-off in Theorem 5 is only useful when $T \leq |D|$. Indeed, we can always answer any boolean path query in linear time using breadth-first search. Surprisingly, it is also possible to improve Theorem 5 for the regime of small answering time as well. In what follows, we will show the improvement for paths of length 4; we will generalize the algorithm for any length in the next section.

4.5.1 Length-4 Path

Lemma 11. *There exists a parameterized data structure for solving $P_4^{\mathsf{bb}}(x_1, x_5)$ that uses space S and answering time $T \leq \sqrt{|D|}$ that satisfies the trade-off $S \cdot T = O(|D|^2)$.*

For $k = 4$, Theorem 5 gives us the trade-off $S \cdot T^{2/3} = O(|D|^2)$ which is always worse than the trade-off in Lemma 11. We next present our algorithm in detail.

Preprocessing Phase. Consider $P_4^{\mathsf{bb}}(x_1, x_5) = R(x_1, x_2), S(x_2, x_3), T(x_3, x_4), U(x_4, x_5)$. Let Δ be a degree threshold. We say that a constant a is *heavy* if its frequency on attribute

x_3 is greater than Δ in both relations S and T; otherwise, it is *light*. In other words, a is heavy if $|\sigma_{x_3=a}(S)| > \Delta$ and $|\sigma_{x_3=a}(T)| > \Delta$. We distinguish two cases based on whether a constant for x_3 is heavy or light. Let $\mathcal{L}_{\mathsf{heavy}}(x_3)$ denote the unary relation that contains all heavy values, and $\mathcal{L}_{\mathsf{light}}(x_3)$ the one that contains all light values. Observe that we can compute both of these relations in time $O(|D|)$ by simply iterating over the active domain of variable x_3 and checking the degree in relations S and T. We compute two views:

$$V_1(x_1, x_3) = R(x_1, x_2) \wedge S(x_2, x_3) \wedge \mathcal{L}_{\mathsf{heavy}}(x_3)$$
$$V_2(x_3, x_5) = \mathcal{L}_{\mathsf{heavy}}(x_3) \wedge T(x_3, x_4) \wedge U(x_4, x_5)$$

We store the views as a hash index that, given a value of x_1 (or x_5), returns all matching values of x_3. Both views take space $O(|D|^2/\Delta)$. Indeed, $|\mathcal{L}_{\mathsf{heavy}}| \leq |D|/\Delta$. Since we can construct a fractional edge cover for V_1 by assigning a weight of 1 to R and $\mathcal{L}_{\mathsf{heavy}}$, this gives us an upper bound of $|D| \cdot (|D|/\Delta)$ for the query output. The same argument holds for V_2. We also compute the following view for light values: $V_3(x_2, x_4) = S(x_2, x_3), \mathcal{L}_{\mathsf{light}}(x_3), T(x_3, x_4)$. This view requires space $O(|D| \cdot \Delta)$, since the degree of the light constants is at most Δ. We can now rewrite the original query as $P_4^{\mathsf{bb}}(x_1, x_5) = R(x_1, x_2), V_3(x_2, x_4), U(x_4, x_5)$.

The rewritten query is a three path query. Hence, we can apply Theorem 4 to create a data structure with answering time $T = O(|D|/\Delta)$ and space $S = O(|D|^2/(|D|/\Delta)) = O(|D| \cdot \Delta)$.

Query Answering. Given an access request, we first check whether there exists a 4-path that goes through some heavy value in $\mathcal{L}_{\mathsf{heavy}}(x_3)$. This can be done in time $O(|D|/\Delta)$ using the views V_1 and V_2. Indeed, we obtain at most $O(|D|/\Delta)$ values for x_3 using the index for V_1, and $O(|D|/\Delta)$ values for x_3 using the index for V_3. We then intersect the results in time $O(|D|/\Delta)$ by iterating over the $O(|D|/\Delta)$ values for x_3 and checking if the bound values for x_1 and x_5 from a tuple in V_1 and V_2 respectively. If we find no such 4-path, we check for a 4-path that uses a light value for x_3. From the data structure we have constructed in the preprocessing phase, we can do this in time $O(|D|/\Delta)$.

Tradeoff Analysis. From the above, we can compute the answer in time $T = O(|D|/\Delta)$. From the analysis in the preprocessing phase, the space needed is $S = O(|D|^2/\Delta + |D| \cdot \Delta)$. Thus, whenever $\Delta \geq \sqrt{|D|}$, the space becomes $S = O(|D| \cdot \Delta)$, completing our analysis.

4.5.2 General Path Queries

We can now use the algorithm for the 4-path query to improve the space-time trade-off for general path queries of length greater than four.

Theorem 6. *Let D be an input instance. For $k \geq 4$, there is a data structure for $P_k^{\mathsf{bb}}(x_1, x_{k+1})$ with space $S = O(|D| \cdot \Delta)$ and answer time $T = O\left((\frac{|D|}{\Delta})^{\frac{k-2}{2}}\right)$ for $\Delta \geq \sqrt{|D|}$.*

Proof. Fix some $\Delta \geq \sqrt{|D|}$. We construct the data structure for a path of length k recursively. The base case is when $k = 4$, with answer time $T_4 = |D|/\Delta$ and space $S_4 = |D| \cdot \Delta$.

In the recursive step, similar to the previous section, we set $\sqrt{\frac{|D|}{\Delta}}$ as the degree threshold for any constant that variables x_1 and x_{k+1} can take. Let $\mathcal{L}^1_{\mathsf{heavy}}, \mathcal{L}^{k+1}_{\mathsf{heavy}}$ be unary relations that store the heavy values for x_1, x_{k+1} respectively. We compute and store the result of

$$V(x_1, x_{k+1}) = \mathcal{L}^1_{\mathsf{heavy}}(x_1), R_1(x_1, x_2), \ldots, R_k(x_k, x_{k+1}), \mathcal{L}^{k+1}_{\mathsf{heavy}}(x_{k+1}).$$

This view has size bounded by $\left(|D| \cdot \sqrt{\frac{\Delta}{|D|}}\right)^2 = |D| \cdot \Delta$. We consider the following queries:

$$V_1^{\mathsf{bb}}(x_2, x_{k+1}) = R_2(x_2, x_3), \ldots, R_k(x_k, x_{k+1}).$$

$$V_2^{\mathsf{bb}}(x_1, x_k) = R_1(x_1, x_2), \ldots, R_{k-1}(x_{k-1}, x_k).$$

both of which correspond to the $(k-1)$-path, so we can recursively apply the data structure here. Let S_k, T_k be the space and time for k-path. For space, we have following observation:

$$S_k = |D| \cdot \Delta + S_{k-1}$$

As $S_4 = |D| \cdot \Delta$, we obtain $S_k = O(|D| \cdot \Delta)$.

Given an access request, we answer it by distinguishing two cases. If x_1, x_{k+1} is heavy, we probe the stored view $V(x_1, x_{k+1})$ in time $O(1)$. If one of them is light (say w.l.o.g. x_1), we call recursively the data structure V_1 for every one of the $\leq \sqrt{|D|/\Delta}$ values connected with x_1. This gives us the following recurrence formula for answer time:

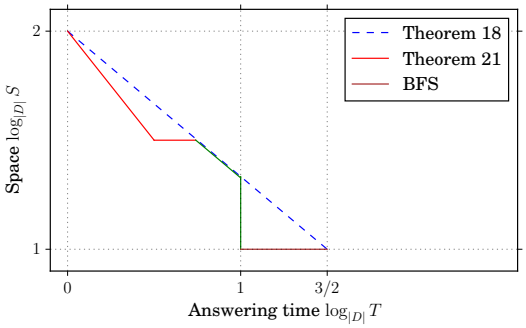
$$T_k = (|D|/\Delta)^{1/2} \cdot T_{k-1}$$

Solving the recursive formula gives us $T_k = (|D|/\Delta)^{(k-2)/2}$. □

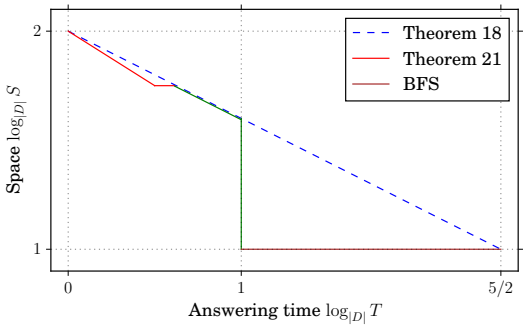
The space-time trade-off obtained from Theorem 5 is $S \cdot T^{2/(k-2)} = O(|D|^2)$, but only for $T \leq |D|^{(k-2)/4}$. To compare it with the trade-off of $S \cdot T^{2/(k-1)} = O(|D|^2)$ obtained from Theorem 5, it is instructive to look at Figures 4.2a and 4.2b, which plot the space-time tradeoffs for $k = 4$ and $k = 6$ respectively. For $k = 4$, we can see that the new trade-off is better for $T \leq |D|^{1/2}$. Once T exceeds $|D|^{1/2}$, it is still better to use the data structure from Theorem 6 until Theorem 5 takes over. For $k = 6$, the switch point also happens at $T = |D|^{1/2}$ but requires more space. In general, as k grows, the new trade-off line becomes flatter and approaches Theorem 5.

4.6 Lower Bounds

In this section, we study the lower bounds for adorned star and path queries. We first present conditional lower bounds for the k-Set Disjointness problem using the conditional optimality of ℓ-Set Disjointness where $\ell < k$. First, we review the known results from [GKLP17] starting with the conjecture for k-Set Disjointness .



(a) Tradeoff for $P_4^{\mathsf{bb}}(x_1, x_5)$



(b) Tradeoff for $P_6^{\mathsf{bb}}(x_1, x_7)$

Figure 4.2: Space/time tradeoffs for path query of length $k = 4, 6$. The line in blue (dashed) shows the trade-off obtained from Theorem 5. The highlighted portion in brown shows the improved trade-off using BFS. The red curve is the new trade-off obtained using Theorem 6. The green portion of the original curve is still the best possible when Theorem 6 is not applicable.

Conjecture 2 (due to [GKLP17])**.** *Any data structure for k-Set Disjointness problem that answers queries in time T must use space $S = \Omega(|D|^k/T^k)$.*

Conjecture 2 was shown to be conditionally optimal based on conjectured lower bound for the $(k+1)$-Sum Indexing problem, however, it was subsequently shown to be false [KP19], which implies that Conjecture 2 is still an open problem. Conjecture 2 can be further generalized to the case when input relations are of unequal sizes as follows.

Conjecture 3. *Any data structure for $Q_*^{\mathsf{b}\ldots\mathsf{b}}(y_1, \ldots, y_k) = R_1(x, y_1), \ldots, R_k(x, y_k)$ that answers queries in time T must use space $S = \Omega(\Pi_{i=1}^k |R_i|/T^k)$.*

We now state the main result for star queries.

Theorem 7. *Suppose that any data structure for $Q_*^{\mathsf{b}\ldots\mathsf{b}}(y_1, \ldots, y_k)$ with answering time T must use space $S = \Omega(\Pi_{i=1}^k |R_i|/T^k)$. Then, any data structure for $Q_*^{\mathsf{b}\ldots\mathsf{b}}(y_1, \ldots, y_\ell)$ with answering time T must use space $S = \Omega(\Pi_{i=1}^\ell |R_i|/T^\ell)$, for $2 \leq \ell < k$.*

Proof. Let $\Delta = T$ be the degree threshold for the k bound variables $y_1, \ldots y_k$. If any of the k variables is light (i.e $|\sigma_{y_i = a[y_i]} R_i(y_i, x)| \leq \Delta$), then we can check whether the intersection between k sets is empty or not in time $O(T)$ by indexing all relations in a linear time preprocessing phase. The remaining case is when all k variables are heavy. We now create ℓ views $V_1, \ldots V_\ell$ by arbitrarily partitioning the k relations into the ℓ views followed by materializing the join of all relations in each view. Let view V_i contain the join of k_i relations. Then, $|V_i| = O((\Pi_{R \in J_i} |R|/T^{k_i - 1}))$ where J_i is the set of all relations assigned to view V_i.

We have now reduced the k-star query where all k variables are heavy into an instance of ℓ-star query where the input relations are $V_1, \ldots, V_\ell$. Suppose that there exists a data

structure that can answer queries in time T using space $S = o(\Pi_{i=1}^{\ell}|V_i|/T^{\ell})$. Then, we can use such a data structure for answering the original query where all variables are heavy. The space used by this oracle is

$$\begin{aligned} S &= o(\Pi_{i=1}^{\ell}|V_i|/T^{\ell}) = o((\Pi_{i=1}^{\ell}\Pi_{R\in J_i}|R|/T^{k_i-1}) \cdot (1/T^{\ell})) \\ &= o((\Pi_{i=1}^{k}|R_i|/T^{k-\ell}) \cdot (1/T^{\ell})) = o(\Pi_{i=1}^{k}|R_i|/T^{k}) \end{aligned}$$

which contradicts the space lower bound for k-star. □

Theorem 7 creates a hierarchy for k-Set Disjointness , where the optimality of smaller set disjointness instances depends on larger set disjointness instances. Next, we show conditional lower bounds on the space requirement of path queries. We begin by proving a simple result for optimality of P_2^{bb} (equivalent to 2-Set Disjointness) assuming the optimality of P_3^{bb} query.

Theorem 8. *Suppose that any data structure for P_3^{bb} that answers queries in time T, uses space S such that $S \cdot T = \Omega(|D|^2)$. Then, for P_2^{bb} , it must be the case that any data structure that uses space $S = O(|D|^2/T^2)$, the answering time is $\Omega(T)$.*

Proof. Let $\Delta = T$ be the degree threshold for all vertices. If both bound variables in P_3^{bb} are heavy, then we can answer the query in constant time using space $\Theta(|D|^2/T^2)$ by materializing the answers to all heavy-heavy queries. In the remaining cases, at least one of the bound valuations is light. Without loss of generality, suppose x_1 is light. Then, we can make Δ calls to the oracle for query $P_2^{\mathsf{bb}}(x_2, x_4) = R_2(x_2, x_3), R_3(x_3, x_4)$.

Suppose that there exists a data structure with space $O(|D|^2/T^2)$ for $P_2^{\mathsf{bb}}(x_2, x_4)$ and answering time $o(T)$. Then, we can answer P_3^{bb} with light x_1 in time $o(T^2)$. This improves the trade-off for P_3^{bb} since the product of space usage and answering time is $o(|D|^2)$ for any non-constant T, coming to a contradiction. □

Using a similar argument, it can be shown that the conditional optimality of Theorem 6 for $k = 4$ implies that $S \cdot T = \Omega(|D|^2)$ trade-off for P_3^{bb} is also optimal (but only for the range $T \leq \sqrt{|D|}$ when the result is applicable).

Chapter 5

Unranked Enumeration of Conjunctive Queries with Projections

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 [Yan81, NRR13, NPRR12]. 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 [BDG07a]. 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 [BDG07a]. Prior work that uses structural decomposition methods [GS13] 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* [GGS14] 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 trade-off 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, [BDG07a] 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 trade-off between preprocessing time and delay. Towards this end, seminal work by Kara et al. [KNOZ20a] showed that for any hierarchical CQ[1] (possibly with projections), there always exists a smooth trade-off between preprocessing time and delay. This is the first improvement over the results of Bagan et al. [BDG07a] in over a decade for queries involving projections. Applied to the query $Q_{\texttt{two-path}}$, the main result of of [KNOZ20a] 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 chapter, we continue the investigation of the trade-off 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 [DHK20] for an overview). The most common star query seen in practice is $Q_{\texttt{two-path}}$ [BMT20]. The same holds true for path queries, which are fundamental in graph processing. Second, as we will see in this chapter, even for the simple class of star queries, the trade-off 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.

Theorem 9. *Given a database instance D, we can enumerate the output of $Q_{\texttt{two-path}} = \pi_{x,z}(R(x,y) \bowtie S(y,z))$ 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. [KNOZ20a] 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 9 is truly an algorithmic improvement rather than an improved analysis of [KNOZ20a]. We answer the question positively. Specifically, we show that there exists a database instance where the delay obtained from Theorem 9 is a polynomial improvement over the actual guarantee [KNOZ20a] 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. For the worst-case output size where $|\texttt{OUT}_\bowtie| = \Theta(|D|^2)$, we actually obtain the best possible delay, which will be constant. Compare this to the result of [KNOZ20a], 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 trade-off in [KNOZ20a] in the worst-case; we will show that we can always get at least as good of a trade-off point as the one in [KNOZ20a]. Figure 5.1 summarizes the prior work and the results present in this chapter.

Our Contribution. In this chapter, we improve the state-of-the-art on the preprocessing time-delay trade-off for a subset of CQs with projections. We summarize our main technical contributions below (highlighted in Figure 5.1):

1. Our main contribution consists of a novel algorithm (Theorem 11) that achieves output-dependent delay guarantees for star queries after 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 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 trade-off between preprocessing time and delay guarantees as stated in [KNOZ20a] for the general 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.

[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 [OZ15a].

Queries	**Preprocessing**	**Delay**	**Source**
Arbitrary acyclic CQ	$O(\|D\|)$	$O(\|D\|)$	[BDG07a]
Free-connex CQ (projections)	$O(\|D\|)$	$O(1)$	[BDG07a]
Full CQ	$O(\|D\|^{\texttt{fhw}})$	$O(1)$	[OS16]
Full CQ	$O(\|D\|^{\texttt{subw}} \log \|D\|)$	$O(1)$	[AKNS17]
Hierarchical CQ (with projections)	$O(\|D\|^{1+(\texttt{w}-1)\epsilon})$	$O(\|D\|^{1-\epsilon})$, $\epsilon \in [0,1]$	[KNOZ20a]
Star query with k relations (with projections)	$O(\|D\|)$	$O(\frac{\|D\|^{k/(k-1)}}{\|\texttt{OUT}_{\bowtie}\|^{1/(k-1)}})$	this thesis
Path query with k relations (with projections)	$O(\|D\|^{2-\epsilon/(k-1)})$	$O(\|D\|^{\epsilon})$, $\epsilon \in [0,1)$	this thesis
Left-deep hierarchical CQ (with projections)	$O(\|D\|)$	$O(\|D\|^k/\|\texttt{OUT}_{\bowtie}\|)$	this thesis
Two path query (with projections)	$O(\|D\|^{\omega\cdot\epsilon})$	$O(\|D\|^{1-\epsilon})$, $\epsilon \in [\frac{2}{\omega+1}, 1]$	this thesis

Figure 5.1: 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.

3. We show how fast matrix multiplication can be used to obtain a trade-off between preprocessing time and delay that further improves upon the trade-off in [KNOZ20a]. We also present an algorithm for left-deep hierarchical queries with linear preprocessing time and output-dependent delay guarantees.

4. Finally, we present new results on preprocessing time-delay trade-off 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 $O(|D|^{\epsilon})$ with preprocessing time $O(|D|^{2-\epsilon/(k-1)})$ for any $\epsilon \in [0,1)$.

Organization. Section 5.1 overviews the prior done for join-project queries. Section 5.2 presents we present useful lemmas that are used frequently throughout the chapter and the main result for star queries. Towards the end of the section, we also discuss how matrix multiplication can be useful. Section 5.3 is dedicated to the study of a class of hierarchical

queries that we call left-deep. Finally, Section 5.4 presents a novel trade-off for the important class of path queries.

5.1 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 [BDG07a]. 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 [BB13]. 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 [Seg15a]. Prior work also exhibits a dependency between the space and enumeration delay for conjunctive queries with access patterns [DK18]. 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 [KNOZ20a], 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 trade-off shown in [KNOZ20a] is not optimal. Our work introduces fundamentally new ideas that may be useful in improving the trade-off 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 [XKD15, XD17b, XSD17, DK21] 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 $O(|D| \cdot |\mathtt{OUT}_\pi|^{\frac{(\omega-1)}{(\omega+1)}} + |D|^{\frac{2(\omega-1)}{(\omega+1)}} \cdot |\mathtt{OUT}_\pi|^{\frac{2}{(\omega+1)}})$ [DHK20, AP09]. 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 [KNN+19, KNOZ20a, BKS17b, BKS18] has studied the trade-off 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 [RCWH+20, LCFK21, CCW+21, MCCJ21, ACFR20, CCC+20, RCGvDMJ], directed graphical models [CRJ20], and consistent query answering [KOW21] may also benefit from some of our techniques.

In practice, most of the previous work has considered join-project query evaluation by pushing down the projection operator in the query plan [GHQ95a, BGI97, GHQ95b, CG85]. LevelHeaded [ALOR18] and EmptyHeaded [ALT+17] are general linear algebra systems that use highly optimzed set intersections to speed up evaluation of cyclic joins, counting queries and support projections over them. Since Intel MKL is also a linear algebra library, one can also use EmptyHeaded as the underlying framework for performing matrix multiplication. For group-by aggregate queries, [XD19] also used worst-case optimal join algorithms to avoid evaluating binary joins at a time and materializing the intermediate results. However, the running time of their algorithm is not output sensitive with respect to the final projected result.

5.2 Main Result

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 constant delay in linear preprocessing time. As Table 5.1 shows, when Q is full, with $T_p = O(|D|^{\texttt{fhw}})$, we can enumerate the results with constant delay [OS16]. In the particular case where Q is acyclic i.e. $\texttt{fhw} = 1$, we can achieve constant delay with only linear preprocessing time. On the other hand, [BDG07a] shows that for every acyclic CQ, we can achieve linear delay $O(|D|)$ with linear preprocessing time $O(|D|)$.

Recently, [KNOZ20a] showed that it is possible to get a trade-off 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 [BDG07a] for queries with projections.

Theorem 10 ([KNOZ20a])**.** *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 $s^{\uparrow}$ [OZ15a], 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 must always be smaller than the time required to evaluate the full join result. This is because if $T_p = \Theta(|\texttt{OUT}_{\bowtie}|)$, we can evaluate the full join and deduplicate the projection output, allowing us to obtain constant delay in the enumeration phase. Therefore, the trade-off is more meaningful when ϵ can only take values between 0 and $(\log_{|D|} |\texttt{OUT}_{\bowtie}| - 1)/(\mathsf{w} - 1)$. In the worst-case, $|\texttt{OUT}_{\bowtie}| = |D|^{\mathsf{w}}$

and ϵ can take any value between 0 and 1 (both inclusive), which is captured by the general result above.

5.2.1 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 [OVL81].

We say that an algorithm $\mathcal{A}$ provides no delay guarantees to mean that its delay guarantee can be as large as 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 12. *Consider two algorithms $\mathcal{A}, \mathcal{A}'$ and two positive integers T and T' provided as a part of the input 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.*

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 after δ time has passed, 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 and γ 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 equal to the 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 integer constant and then set γ to be the smallest positive integer 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 $(\delta/\gamma) \cdot (\gamma + \eta + c_{\mathsf{pause}} + c_{\mathsf{resume}}) \leq \delta \cdot (1 + T/T' + (c_{\mathsf{pause}} + c_{\mathsf{resume}})/\gamma) = O(\delta \cdot \max\{1, T/T'\})$. □

Lemma 12 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'. This can be accomplished by calculating the number of operations in the algorithms $\mathcal{A}$ and $\mathcal{A}'$ to bound their running time. The next lemma introduces our second key idea of interleaving stored output result with *on-the-fly* query computation. Algorithm 6 describes the detailed algorithm for Lemma 13.

Lemma 13. *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. Suppose the J output results are already stored 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[3]. 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. □

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 [Kaz13].

Lemma 14. *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 $\preceq$.*

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

Algorithm 6: DEDUPLICATE$(J, \mathcal{A})$

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

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

Proof. We describe algorithm 7. 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 7 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 7 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 7 has a delay guarantee of $O(m \cdot \delta)$. □

Directly implied by Lemma 14 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 chapter, we use this primitive as LISTMERGE$(L_1, L_2, \cdots, L_m)$.

Algorithm 7: MERGE($\mathcal{A}_1, \mathcal{A}_2, \cdots, \mathcal{A}_m$)

```
S ← {1, 2, ··· , m};
foreach i ∈ S do
    e_i ← A_i.first() ;
while S ≠ ∅ do
    w ← min_{i∈S} e_i ;        /* finds the smallest output (using ⪯) over all algorithms */
    output w ;
    foreach i ∈ S do
        if e_i = w then
            e_i ← A_i.next() ;
        if e_i = null then
            S ← S − {i}                    /* the algorithm completes its output */
```

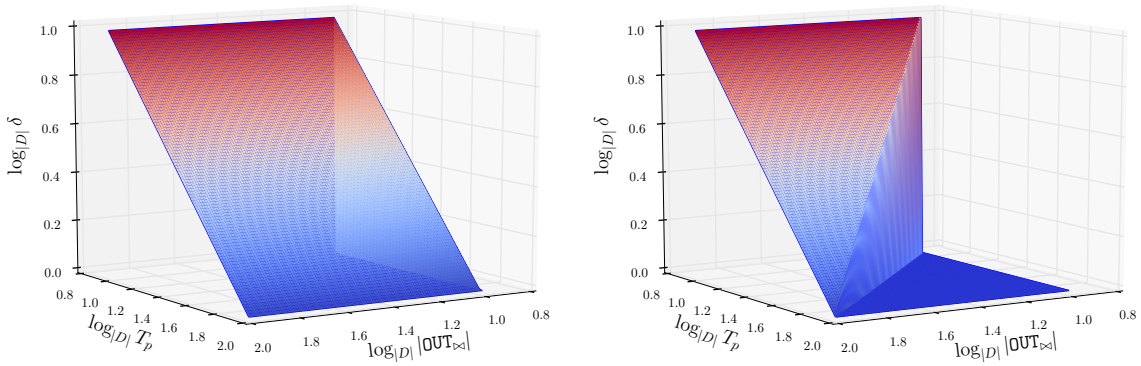


Figure 5.2: Worst-case trade-off given by Theorem 10 without (left) and with (right) taking $|\mathtt{OUT}_{\bowtie}|$ into consideration.

5.2.2 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 11 that we present below. We first present a detailed discussion on how our result is an improvement over prior work in Section 5.1. Then, we present a warm-up proof for $\pi_r(Q_k^*)$ in Section 5.2.4, followed by the proof for the general result in Section 5.2.5.

Theorem 11. *Consider the star query*[4] *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*

[4]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.

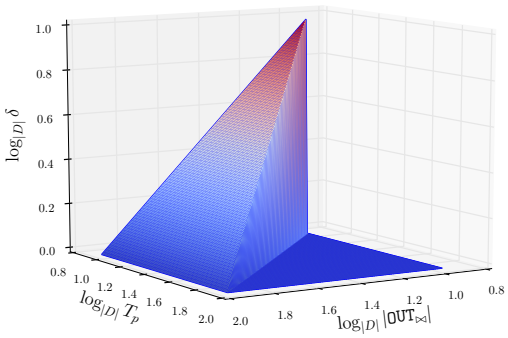


Figure 5.3: Theorem 11 for $k = 2$.

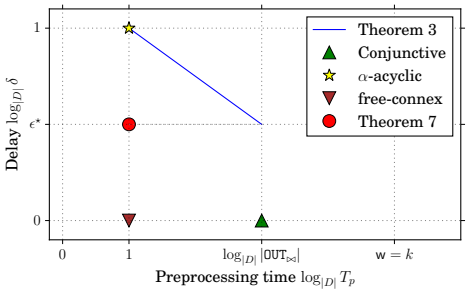


Figure 5.4: Trade-off in the worst-case for star query.

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 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.

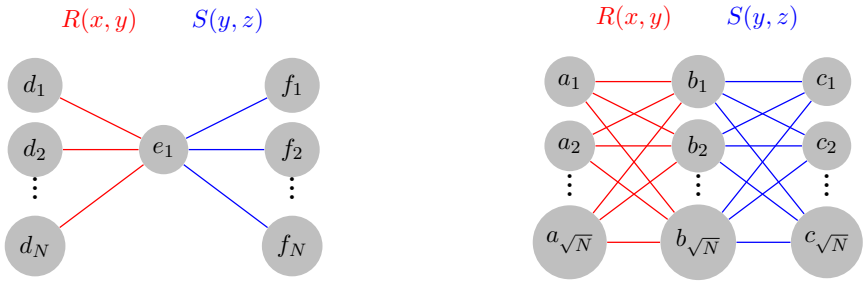
5.2.3 Comparison with Prior Work

It is instructive now to compare the worst-case delay guarantee obtained by Theorem 10 for $Q_k^*(D)$ with Theorem 11. 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 10 tells us that this requires $O(|D|^{1+\epsilon(k-1)})$ preprocessing time. Then, it holds that:

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

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 5.2, 5.3 and 5.4 show the existing and new trade-off results. Figure 5.2 shows the trade-off curve obtained from Theorem 10 by adding $|\mathtt{OUT}_{\bowtie}|$ as a third dimension, and adding the optimization for constant delay when $T_p \geq O(|\mathtt{OUT}_{\bowtie}|)$. Figure 5.3 shows the trade-off obtained from our result, while Figure 5.4 shows other existing results for a fixed



(a) Database D_0 with full join size N^2. (b) Database D_1 with full join size $N^{3/2}$.

Figure 5.5: $D_0 \cup D_1$ forms a database where Theorem 11 improves the delay guarantee of Theorem 10.

value of $|\texttt{OUT}_{\bowtie}|$. For a fixed value of $|\texttt{OUT}_{\bowtie}|$, the delay guarantee does not change in Figure 5.3 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 5.4.

Our results thus imply that, depending on $|\texttt{OUT}_{\bowtie}|$, one must choose a different algorithm to achieve the optimal trade-off between preprocessing time and delay. Since $|\texttt{OUT}_{\bowtie}|$ can be computed in linear time (using a simple adaptation of Yannakakis algorithm [Yan81, PP06]), this can be done without affecting the preprocessing bounds.

Next, we show how our result provides an algorithmic improvement over Theorem 10. Consider the instances D_0, D_1 depicted in Figure 5.5a and Figure 5.5b respectively, and assume we want to use linear preprocessing time.

For D_1, the algorithm of Theorem 10 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 10 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 10 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 10 requires near quadratic preprocessing time as shown Example 24 below.

Example 24. *We construct an instance where achieving constant delay with Theorem 10 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 $|\texttt{OUT}_\bowtie| = \sum_\alpha N^{1+\alpha} + N^2 \log^2 N = \Theta(|D|^2)$. On this instance, Theorem 10 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 10 requires near quadratic computation to achieve constant delay enumeration.*

In the rest of this section, 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.

5.2.4 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 chapter,

we will use count sort [CLRS09a] 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 to which $\pi_y(t)$ is appended; 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 to which $\pi_z(t)$ is appended. 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

$$\sum_{v:\{v_1, v_2, \ldots, v_{i^*}\}} |R(v, y) \bowtie S(y, z)| \geq \sum_{v:\{v_{i^*+1}, \ldots, v_n\}} |R(v, y) \bowtie S(y, z)| \tag{5.1}$$

This can be computed by doing a linear pass on $\mathcal{L}$ using a simple adaptation of Yannakakis algorithm [Yan81, PP06]. This entire phase takes time $O(|D|)$.

Enumeration Phase. The enumeration algorithm interleaves the following two loops using the construction in Lemma 12. Specifically, it will spend an equal amount of time (a constant) before switching to the computation of the other loop.

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 7.

Example 25. *Consider relations R and S as shown in Figure 5.6a and Figure 5.6b. Figure 5.6c 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*

Algorithm 8: EnumTwoPath

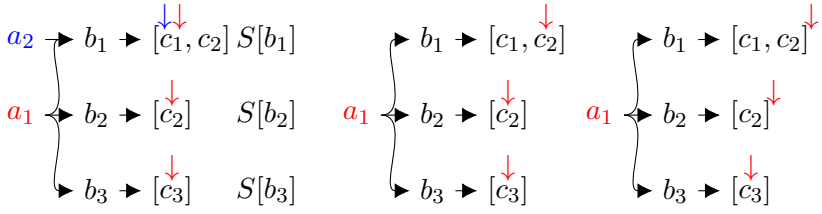
```
for i = 1, ..., i* do                                              ┐
    Let π_y(σ_{x=v_i}(R)) = {u_1, u_2, ··· , u_ℓ};                 │ run for O(1) time
    output (v_i, f^{-1}(ListMerge(π_z σ_{y=u_1} S, π_z σ_{y=u_2} S, ··· , π_z σ_{y=u_ℓ} S)))^5 ┘ then switch
for i = i* + 1, ..., n do                                          ┐
    Let π_y(σ_{x=v_i}(R)) = {u_1, u_2, ··· , u_ℓ};                 │ run for O(1) time
    output (v_i, f^{-1}(ListMerge(π_z σ_{y=u_1} S, π_z σ_{y=u_2} S, ··· , π_z σ_{y=u_ℓ} S))) ┘ then switch
```

X	Y
a_1	b_1
a_1	b_2
a_1	b_3
a_2	b_1

(a) Table R

Y	Z
b_1	c_1
b_1	c_2
b_2	c_2
b_3	c_3

(b) Table S



(c) output (a_1, c_1) (d) output (a_1, c_2) (e) output(a_1, c_3)

Figure 5.6: Example for two path query enumeration

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 5.6d 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 5.6e. 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.*

Theorem 12. *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* $S_e = O(|D|)$*.*

Proof. To prove this result, we will apply Lemma 12, 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 8. Since by construction $J_\ell \geq J_h$, Lemma 12 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 5.1, 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 $J_\ell + J_h = |\texttt{OUT}_\bowtie|$, we get that $J_h \geq |\texttt{OUT}_\bowtie|/2 - |D|/2 \geq 1/4 \cdot |\texttt{OUT}_\bowtie|$ assuming $|\texttt{OUT}_\bowtie| \geq 2 \cdot |D|$. 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. □

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 8 achieves much better delay guarantees in practice as shown in Example 26 below.

Example 26. *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 8 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.*

5.2.5 Proof of Main Theorem

We now generalize Algorithm 8 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 [Yan81]. 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

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

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 with 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 15. This algorithm can be viewed as another instantiation of Algorithm 7.

Lemma 15. *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 $\sigma_{y=u} R_j(x_j, y)$. 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 7, 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 7, we return $f^{-1}(t)$ to the user. □

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 15, 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 $T_H \leq c^\star \cdot |Q_H^*| \leq c^\star \cdot |\texttt{OUT}_\bowtie|/2$ 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 $\Delta = (2 \cdot |D|^k / |\texttt{OUT}_\bowtie|)^{\frac{1}{k-1}}$. Since $|Q_H^*| + |Q_L^*| = |\texttt{OUT}_\bowtie|$, it holds that $|Q_L^*| \geq |\texttt{OUT}_\bowtie|/2$ 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, $T_L \geq |\texttt{OUT}_\bowtie|/2$.

We are now ready to apply Lemma 12 with the following parameters:

1. $\mathcal{A}$ is the full join computation of Q_H and $T = c^\star \cdot |\texttt{OUT}_\bowtie|/2$.
2. $\mathcal{A}'$ is the enumeration algorithm applied to Q_L with delay guarantee $\delta = O(\Delta)$ and $T' = |\texttt{OUT}_\bowtie|/2$.
3. T and T' are fixed once $|\texttt{OUT}_\bowtie|, \Delta$, 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 12 apply and we obtain a delay of $O(\Delta)$.

5.2.6 Interleaving with Join Computation

Theorem 11 obtains poor delay guarantees when the full join size $|\texttt{OUT}_\bowtie|$ 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 13 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 13. *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 size of 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 11. First, observe that both Theorem 13 and Theorem 11 require $O(|D|)$ preprocessing time. Second, the delay guarantee provided by Theorem 13 can be better than Theorem 11. 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 not 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 $\times_{i=1}^{k} (\pi_{x_i} \sigma_{y=u} R_i(x_i, y))$ can be enumerated with $O(1)$ delay. This preprocessing phase takes $O(|D|)$ time and $O(|D|)$ space. We can now invoke Lemma 13 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.

Dynamic Setting. Remarkably, it is also possible to do a simple adaptation of the algorithm for the dynamic setting but only in the case of self-joins, where $R_1 = R_2 = \ldots R_k = R$. In this setting, single tuple updates are allowed to the underlying relation, i.e., a tuple t over variables x, y can be either inserted or deleted from the relation $R(x, y)$ (note that a relation does not allow duplicates). The preprocessing phase in this case simply stores $R(x, y)$ is stored as bidirectional hash map (one map with key as y valuation and the value as the sorted list of x_i valuations connected to it and other map with key as x_i valuation and the value as the sorted list of y valuations). This hash map can be maintained in amortized constant time under single-tuple updates. We also store a hash set called J containing (a, a) for all $a \in \mathbf{dom}(x)$. J can also be maintained under updates in amortized constant time. Indeed, whenever the degree of some x valuation becomes zero and it is removed from the hash maps of R, we can also remove it from J. Similarly, whenever a new domain value is added for the first time to R, we can update insert it into J.

For the enumeration phase, we simply apply Lemma 13 with stored hash set J. Applying the same reasoning as above, we can delay $\delta = O(|\mathtt{OUT}_{\bowtie}|/|\mathtt{OUT}_{\pi}|^{1/k})$. We can now state the result formally.

Theorem 14. *Consider a self-join star query $\pi_{x_1,\ldots,x_k}(Q_k^*)$ over a single relation $R(x,y)$ 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 $S_e = O(|D|)$ with $O(1)$ amortized update time for single-tuple updates.*

To achieve constant amortized update time, the main result that applies to all hierarchical queries (including self-joins) from [KNOZ20a] requires linear preprocessing time and achieves worst-case linear delay. Since $|\mathtt{OUT}_{\bowtie}|/|\mathtt{OUT}_{\pi}|^{1/k} \leq |D|$ [AP09], our strategy provides an alternate algorithm that achieves the same worst-case linear delay but may perform better on certain database instances. Modifying our algorithm for the non self-join case is a non-trivial problem and likely requires development of new technical machinery that we leave as a problem for future work.

Fast Matrix Multiplication. Both Theorem 11 and Theorem 10 are combinatorial algorithms. In this section, we will show how fast matrix multiplication can be used to obtain a trade-off between preprocessing time and delay that is better than Theorem 10 for some values of delay.

Theorem 15. *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 1, 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 trade-off. □

For the two-path query and the current best value of $\omega = 2.373$, we get the trade-off 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 worst-case preprocessing time is $T_p = O(|D|^{1.422})$. In contrast, Theorem 10 requires a worst-case 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 $|\texttt{OUT}_\bowtie| > |D|^{1.422}$, 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 11 with $|\texttt{OUT}_\bowtie| > |D|^{1.422}$ tells us that we can obtain delay as $O(|D|^2/|\texttt{OUT}_\bowtie|) = O(|D|^{0.58})$. Thus, we can offer the user both choices and the user can decide which enumeration algorithm to use.

5.3 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 16. *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 12. 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 14. 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 14. 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 12 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. □

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 16 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 16 is also better than the one given by the trade-off in Theorem 10. In the worst-case when $|\texttt{OUT}_\bowtie| = \Theta(|D|^k)$, we can achieve constant delay enumeration after linear preprocessing time, compared to Theorem 10 that would require $\Theta(|D|^k)$ preprocessing time to achieve the same delay. The decision of when to apply Theorem 16 or Theorem 10 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 10 after linear time preprocessing.

5.4 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 trade-off from [KNOZ20a] does not apply. A subset of path queries, namely 3-path and 4-path counting queries were considered in [KNN+19]. 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 17. *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 14 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. □

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 [BDG07a] 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 trade-off. A second observation following from Theorem 17 is that as $k \to \infty$, the trade-off 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 $T_p = O(|D|)$.

Chapter 6

Join-Project Query Evaluation using Fast Matrix Multiplication

In this chapter, we study the problem of evaluating join queries where the join result does not contain all the variables in the body of the query. In other words, some of the variables have been *projected out* of the join result. The simplest way to evaluate such a query is to first compute the full join, and then make a linear pass over the result, project each tuple and remove the duplicates. While this approach is conceptually simple, it relies on efficient *worst-case optimal join algorithms* for full queries, which have recently been developed in a series of papers [AGM13, NRR13, NPRR12, Vel]. The main result in this line of work is a class of algorithms that run in time $O(|D|^{\rho^*} + |\texttt{OUT}|)$, where D is the database instance and ρ^* is the optimal fractional edge cover of the query [AGM13]. In the worst case, there exists a database D such that $|\texttt{OUT}| = |D|^{\rho^*}$. In practice, most query optimizers create a query plan by pushing down projections in the join tree.

Example 27. *Consider relation $R(x, y)$ of size N that represents a social network graph where an edge between two users x and y denotes that x and y are friends. We wish to enumerate all users pairs who have at least one friend in common [MBO12]. This task is equivalent to the query $\ddot{Q}(x, z) = R(x, y), R(z, y)$, which corresponds to the following SQL query:*

```
SELECT DISTINCT R1.x, R2.x FROM R AS R1, R AS R2 WHERE
    R1.y = R2.y;
```

Suppose that the graph contains a small (constant) number of communities and the users are spread evenly across them. Each community has $O(\sqrt{N})$ users, and there exists an edge between most user pairs within the same community. In this case, the full join result is $\Theta(N^{3/2})$ but $|\ddot{Q}(D)| = \Theta(N)$.

As the above example demonstrates, using worst-case optimal join algorithms can lead to an intermediate output that can be much larger than the final result after projection, especially if there are many duplicate tuples. Thus, we ask whether it is possible to design

faster algorithms that can skip the construction of the full result when this is large and as a result speed up the evaluation. Ideally, we would like to have algorithms that run faster than worst-case optimal join algorithms, are sensitive to the size of the output of the projected result, and do not require large main memory during execution.

In this chapter, we show how to achieve the above goal for a fundamental class of join queries called *star joins*. Star joins are join queries where every relation is joined on the same variable. The motivation to build faster algorithms for star joins with projection is not limited to faster query execution in DBMS systems. We present next a list of three applications that benefit from these faster algorithms.

Set Similarity. Set similarity is a fundamental operation in many applications such as entity matching and recommender systems. Here, the goal is to return all pairs of sets such that have contain at least c common elements. Recent work [DTL18] gave the first output-sensitive algorithm that enumerates all similar sets in time $O(|D|^{2-\frac{1}{c}} \cdot |\texttt{OUT}|^{\frac{1}{2c}})$. As the value of c increases, the running time tends to $O(|D|^2)$. The algorithm also requires $O(|D|^{2-\frac{1}{c}} \cdot |\texttt{OUT}|^{\frac{1}{2c}})$ space. We improve the running time and the space requirement of the algorithm for a large set of values that $|\texttt{OUT}|$ can take, for all c.

Set Containment. Efficient computation of set containment joins over set-valued attributes has been extensively studied in the literature. A long line of research [JP05, YZY+18, KRS+16, LFHDB15] has developed a trie-based join method where the algorithm performs an efficient blocking step that prunes away most of the set verifications. However, the verification step is a simple set merging-based method that checks if set $t \subseteq u$, which can be expensive. We show that for certain datasets, our algorithm can identify set containment relationships much faster than state-of-the-art techniques.

Graph Analytics. In the context of graph analytics, the graph to be analyzed is often defined as a *declarative* query over a relational schema [XD17b, XKD15, XSD17, AHS+15]. For instance, consider the DBLP dataset, which stores which authors write which papers through a table $R(\mathit{author}, \mathit{paper})$. To analyze the relationships between co-authors, we can extract the *co-author graph*, which we can express as the view $V(x, y) = R(x, p), R(y, p)$. Recent work [XD17b] has proposed compression techniques where a preprocessing step generates a succinct representation of $V(x, y)$. However, these techniques require a very expensive preprocessing step, rely on heuristics, and do not provide any formal guarantees on the running time. In the context of querying data through APIs, suppose that we want to support an API where a user checks whether authors a_1 and a_2 have co-authored a paper. This is an example of a *boolean query*. In this scenario, the view $R(x, p), R(y, p)$ is implicit and not materialized. Since such an API may handle thousands of requests per second, it is beneficial to batch B queries together and evaluate them at once. We show that our algorithms can lead to improved performance by minimizing user latency and resource usage.

Our contribution. In this chapter, we show how to evaluate star join queries with projection using output-sensitive algorithms. We summarize our technical contribution below.

1. Our main contribution is an output-sensitive algorithm that evaluates star join queries with projection . We use worst-case optimal joins and matrix multiplication as two fundamental building blocks to split the join into multiple subjoin queries which are evaluated separately. This technique was initially introduced in [AP09], but their runtime analysis is incorrect for certain regimes of the output size. We improve and generalize the results via a more careful application of (fast) matrix multiplication.

2. We show how to exploit the join query algorithms for the problems of set similarity, set containment, join processing, and boolean set intersection. Our algorithms also improve the best known preprocessing time bounds for creating offline data structures for set intersection problems [DK17] and compressing large graphs [XD17b]. In addition, we can show that our approach is much more amenable to parallelization.

3. We develop a series of optimization techniques that address the practical challenges of incorporating matrix multiplication algorithms into join processing.

4. We implement our solution as an in-memory prototype and perform a comprehensive benchmarking to demonstrate the usefulness of our approach. We show that our algorithms can be used to improve the running time for set similarity, set containment, join processing, and boolean query answering over various datasets for both single-threaded and multithreaded settings. Our experiments indicate that matrix multiplication can achieve an order of magnitude speedup on average and up to 50× speedup over the best-known baselines for datasets containing a dense component.

Organization. Section 6.1 contains the main algorithm for two-path and star queries. The algorithm uses the idea from [AP09] combined with an improved analysis. Section 6.2 shows how to use the algorithm for the problems of set similarity and set containment. Section 6.3 and Section 6.4 address the challenges of making the algorithm practical. Lastly, Section 6.5 describes the experimental evaluation.

6.1 Computing Join-Project

In this section, we describe our main technique and its theoretical analysis. Ideally, we would like to compute Q_k^* in time linear to the size of the input and output. However, [BDG07b] showed that $\ddot{Q}$ cannot be evaluated in time $O(|\texttt{OUT}|)$ assuming that exponent ω in matrix multiplication is greater than two. A straightforward way to compute any query that is a join followed by a projection is to compute the join using any worst-case optimal

algorithm, and then deduplicate to find the projection. This gives the following baseline result.

Proposition 17 ([NRR13, NPRR12]). *Any CQ Q with optimal fractional edge cover ρ^* can be computed in time $O(|D|^{\rho^*})$.*

Proposition 17 implies that we can compute the star query Q_k^* in time $O(|D|^k)$, where k is the number of joins. However, the algorithm is oblivious of the actual output $\texttt{OUT}$ and will have the same worst-case running time even if $\texttt{OUT}$ is much smaller than $|D|^k$ – as it happens often in practice. To circumvent this issue, [AP09] showed the following output sensitive bound that uses only combinatorial techniques:

Lemma 16 ([AP09]). *Q_k^* can be computed in time $O(|D| \cdot |\texttt{OUT}|^{1-\frac{1}{k}})$.*

For $k = 2$, the authors also make use of fast matrix multiplication to improve the running time to $\tilde{O}(N^{0.862} \cdot |\texttt{OUT}|^{0.408} + |D|^{2/3} \cdot |\texttt{OUT}|^{2/3})$. Later in the section, we will discuss the flaws in the proof of this result in detail.

6.1.1 The 2-Path Query

Consider the query $\ddot{Q}(x, z) = R(x, y), S(z, y)$. Let N_R and N_S denote the cardinality of relations R and S respectively. Without loss of generality, assume that $N_S \leq N_R$. For now, assume that we know the output size $|\texttt{OUT}|$; we will show how to drop this assumption later.

We will also assume that we have removed any tuples that do not contribute to the query result, which we can do during a linear time preprocessing step.

Algorithm. Our algorithm follows the idea of partitioning the input tuples based on their degree as introduced in [AP09], but it differs on the choice of threshold parameters. It is parameterized by two integer constants $\Delta_1, \Delta_2 \geq 1$. It first partitions each relation into two parts, R^-, R^+ and S^-, S^+:

$$R^- = \{R(a, b) \mid |\sigma_{x=a} R(x, y)| \leq \Delta_2 \text{ or } |\sigma_{y=b} S(z, y)| \leq \Delta_1\}$$
$$S^- = \{S(c, b) \mid |\sigma_{z=c} S(z, y)| \leq \Delta_2 \text{ or } |\sigma_{y=b} S(z, y)| \leq \Delta_1\}$$

In other words, R^-, S^- include the tuples that contain at least one value with a low degree. R^+, S^+ contain the remaining tuples from R, S respectively. Algorithm 9 describes the detailed steps for computing the join. It proceeds by performing a (disjoint) union of the following results:

1. Compute $R^- \bowtie S$ and $R \bowtie S^-$ using any worst-case optimal join algorithm, then project.

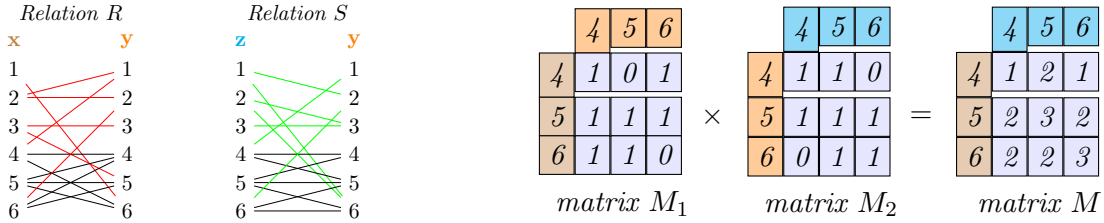
2. Materialize R^+, S^+ as two rectangular matrices and use matrix multiplication to compute their product.

Algorithm 9: Computing $\pi_{xz} R(x,y) \bowtie S(z,y)$

1 $R^- \leftarrow \{R(a,b) \mid |\sigma_{x=a} R(x,y)| \leq \Delta_2 \text{ or } |\sigma_{y=b} S(z,y)| \leq \Delta_1\}$, $R^+ \leftarrow R \setminus R^-$
2 $S^- \leftarrow \{S(c,b) \mid |\sigma_{z=c} S(z,y)| \leq \Delta_2 \text{ or } |\sigma_{y=b} S(z,y)| \leq \Delta_1\}$, $S^+ \leftarrow S \setminus S^+$
3 $T \leftarrow (R^- \bowtie S) \cup (R \bowtie S^-)$ /* use wcoj */
4 $M_1(x,y) \leftarrow R^+$ adj matrix, $M_2(y,z) \leftarrow S^+$ adj matrix
5 $M \leftarrow M_1 \times M_2$ /* matrix multiplication */
6 $T \leftarrow T \cup \{(a,c) \mid M_{ac} > 0\}$
7 **return** T

Intuitively, the "light" values are handled by standard join techniques, since they will not result in a large intermediate result before the projection. On the other hand, since the "heavy" values will cause a large output, it is better to compute their result directly using (fast) matrix multiplication.

Example 28. *Consider relation R and S as shown below.*



Suppose $\Delta_1 = \Delta_2 = 2$. Then, all the edges marked in red (green) form relation $R^-(S^-)$. $R^- \bowtie S$ and $R \bowtie S^-$ can now be evaluated using any worst-case optimal algorithm. The remaining edges consist of heavy values. Thus, we construct matrices M_1 and M_2 encoding all heavy tuples. The resulting matrix product M shows all the heavy output tuples with their corresponding counts.

Correctness. Consider an output tuple (a,c). If there exists no b such that $(a,b) \in R$ and $(c,b) \in S$, then such a pair cannot occur in the output since it will not occur in $R^- \bowtie S, R \bowtie S^-$ or M. Now suppose that (a,c) has at least one witness b such that $(a,b) \in R$ and $(c,b) \in S$. If b is light in relation R or S, then at least one of (a,b) or (c,b) will be included in R^- or S^- and the output tuples will be discovered in the join of $R^- \bowtie S$ or $R \bowtie S^-$. Similarly, if the degree of a or c is at most Δ_2 in relation R or S respectively, the output tuple will be found in $R^- \bowtie S$ or $R \bowtie S^-$. Otherwise, a, b, c are heavy values so M_1 and M_2 matrix will contain an entry for (a,b) and (b,c) respectively.

Analysis. We now provide a runtime analysis of the above algorithmTo and discuss how to optimally choose Δ_1, Δ_2.

We first bound the running time of the first step. To compute the full join result (before projection), a worst-case optimal algorithm needs time $O(N_R + N_S + |\texttt{OUT}_\bowtie|)$, where $|\texttt{OUT}_\bowtie|$

is the size of the join. The main observation is that the size of the join is bounded by $N_S \cdot \Delta_1 + |\texttt{OUT}| \cdot \Delta_2$. Hence, the running time of the first step is $O(N_R + N_S \cdot \Delta_1 + |\texttt{OUT}| \cdot \Delta_2)$.

To bound the running time of the second step, we need to bound appropriately the dimensions of the two rectangular matrices that correspond to the subrelations R^+, S^+. Indeed, the heavy x-values for R^+ are at most N_R/Δ_2, while the heavy y-values are at most N_S/Δ_1. This is because $|\mathbf{dom}(y)| \leq N_S$. Hence, the dimensions of the matrix for R^+ are $(N_R/\Delta_2) \times (N_S/\Delta_1)$. Similarly, the dimensions of the matrix for S^+ are $(N_S/\Delta_1) \times (N_S/\Delta_2)$. The matrices are represented as two-dimensional arrays and can be constructed in time $C = \max\{N_R/\Delta_2 \cdot N_S/\Delta_1, N_S/\Delta_1 \cdot N_S/\Delta_2\}$ by simply iterating over all possible heavy pairs and checking whether they form a tuple in the input relations. Thus, from Lemma 1 the running time of the matrix multiplication step is $M(\frac{N_R}{\Delta_2}, \frac{N_S}{\Delta_1}, \frac{N_S}{\Delta_2})$. Summing up the two steps, the cost of the algorithm is in the order of:

$$N_R + N_S \ \Delta_1 + |\texttt{OUT}|\Delta_2 + M\big(\frac{N_R}{\Delta_2}, \frac{N_S}{\Delta_1}, \frac{N_S}{\Delta_2}\big) + C \tag{6.1}$$

Using the above formula, one can plug in the formula for the matrix multiplication cost and solve to find the optimal values for Δ_1, Δ_2. We show how to do this in Section 6.3.

In the next part, we provide a theoretical analysis for the case where matrix multiplication is achievable with the theoretically optimal $\omega = 2$ for the case where $N_R = N_S = N$. Observe that the matrix construction cost C is of the same order as $M(\frac{N_R}{\Delta_2}, \frac{N_S}{\Delta_1}, \frac{N_S}{\Delta_2})$ even when $\omega = 2$, since β is the smallest of the three terms $N_R/\Delta_2, N_S/\Delta_1, N_S/\Delta_2$. Thus, it is sufficient to minimize the expression

$$f(\Delta_1, \Delta_2) = N + N \cdot \Delta_1 + |\texttt{OUT}| \cdot \Delta_2 + \frac{N^2}{\Delta_2 \min\{\Delta_1, \Delta_2\}}$$

while ensuring $1 \leq \Delta_1, \Delta_2 \leq N$.

The first observation is that for any feasible solution $f(x, y)$ where $x > y$, we can always improve the solution by decreasing the value of Δ_1 from x to y. Thus, w.l.o.g. we can impose the constraint $1 \leq \Delta_1 \leq \Delta_2 \leq N$.

Case 1. $|\texttt{OUT}| \leq N$. Since $\Delta_1 \leq \Delta_2$, we have $f(\Delta_1, \Delta_2) = N \cdot \Delta_1 + |\texttt{OUT}| \cdot \Delta_2 + N^2/\Delta_2 \cdot \Delta_1$. To minimize the running time we equate $\partial f/\partial \Delta_1 = N - N^2/(\Delta_2 \Delta_1^2) = 0$ and $\partial f/\partial \Delta_2 = \texttt{OUT} - N^2/(\Delta_1 \Delta_2^2) = 0$. Solving this system of equations gives that the critical point has $\Delta_1 = |\texttt{OUT}|^{1/3}$, $\Delta_2 = N/|\texttt{OUT}|^{2/3}$. Since $|\texttt{OUT}| \leq N$, this solution is feasible, and it can be verified that it is the minimizer of the running time, which becomes

$$N + N \cdot |\texttt{OUT}|^{1/3}$$

Case 2. $|\texttt{OUT}| > N$. For this case, there is no critical point inside the feasible region, so we will look for a minimizer at the border, where $\Delta_1 = \Delta_2 = \Delta$. This condition gives us

$f(\Delta) = (N + |\texttt{OUT}|) \cdot \Delta + N^2/\Delta^2$, with minimizer $\Delta = \big(2N^2/(N + |\texttt{OUT}|)\big)^{1/3}$. The runtime then becomes

$$O(N^{2/3} \cdot |\texttt{OUT}|^{2/3})$$

We can summarize the two cases with the following result.

Lemma 17. *Assuming that the exponent in matrix multiplication is $\omega = 2$, the query $\ddot{Q}$ can be computed in time*

$$O(|D| + |D|^{2/3} \cdot |\mathit{OUT}|^{1/3} \cdot \max\{|D|, |\mathit{OUT}|\}^{1/3})$$

Lemma 16 implies a running time of $O(|D| \cdot |\texttt{OUT}|^{1/2})$ for $\ddot{Q}$, which is strictly worse compared to the running time of the above lemma *for every output size* $|\texttt{OUT}|$.

Remark. For the currently best known value of $\omega = 2.37$, the running time is $O(|D|^{0.83} \cdot |\texttt{OUT}|^{0.589} + |D| \cdot |\texttt{OUT}|^{0.41})$.

Optimality. The algorithm is worst-case optimal (up to constant factor) for $|\texttt{OUT}| = \Theta(N^2)$. The running time becomes $O(N^2)$ which matches the lower bound $|\texttt{OUT}|$, since we require at least that much time to enumerate the output.

Comparing with previous results. We now discuss the result in [AP09], which uses matrix multiplication to give a running time of $\tilde{O}(|D|^{0.862} \cdot |\texttt{OUT}|^{0.408} + |D|^{2/3} \cdot |\texttt{OUT}|^{2/3})$. We point out an error in their analysis that renders their claim incorrect for the regime where $|\texttt{OUT}| < N$.

In order to obtain their result, the authors make a split of tuples into light and heavy, and obtain a formula for running time in the order of $N\Delta_b + |\texttt{OUT}|\Delta_{ac} + M\big(\frac{N}{\Delta_{ac}}, \frac{N}{\Delta_b}, \frac{N}{\Delta_{ac}}\big)$, where Δ_b, Δ_{ac} are suitable degree thresholds. Then, they use the formula from [HP98] for the cost of matrix multiplication, where $M(x, y, x) = x^{1.84} \cdot y^{0.533} + x^2$. However, this result can be applied only when $x \geq y$, while the authors apply it for regimes where $x < y$. (Indeed, if say $x = N^{0.3}$ and $y = N^{0.9}$, then we would have $M(x, y, x) = N^{1.03}$, which is smaller than the input size $N^{1.2}$.) Hence, the running time analysis is valid only when $N/\Delta_{ac} \geq N/\Delta_b$, or equivalently $\Delta_b \geq \Delta_c$. Since the thresholds are chosen such that $N\Delta_b = |\texttt{OUT}|\Delta_{ac}$, it means that the result is correct *only in the regime where* $|\mathit{OUT}| \geq N$. In other words, when the output size is smaller than the input size, the running time formula from [AP09] is not applicable.

In the case where $\omega = 2$, the cost formula from [HP98] becomes $M(x, y, x) = x^2$, and [AP09] gives an improved running time of $\tilde{O}(N^{2/3} \cdot |\texttt{OUT}|^{2/3})$. Again, this is applicable only when $|\texttt{OUT}| \geq N$, in which case it matches the bound from Lemma 17. Notice that for $|\texttt{OUT}| < N^{1/2}$ the formula would imply a deterministic sublinear time algorithm.

6.1.2 The Star Query

We now generalize the result to the star query Q_k^*. As before, we assume that all tuples that do not contribute to the join output have already been removed.

Algorithm. The algorithm is parametrized by two integer constants $\Delta_1, \Delta_2 \geq 1$. We partition each relation R_i into three parts, R_i^+, R_i^- and $R_i^\diamond$:

$$\begin{aligned} R_i^- &= \{R_i(a,b) \mid |\sigma_{x_i=a} R_i(x_i, y)| \leq \Delta_2\} \\ R_i^\diamond &= \{R_i(a,b) \mid |\sigma_{y=b} R_j(x_j, y)| \leq \Delta_1, \text{ for each } j \in [k] \setminus i\} \\ R_i^+ &= R_i \setminus (R_i^- \cup R_i^\diamond) \end{aligned}$$

In other words, R_i^- contains all tuples with light x, $R_i^\diamond$ contains all tuples with y values that are light in all other relations, and R_i^+ the remaining tuples. The algorithm now proceeds by computing the following result:

1. Compute $R_1 \bowtie \ldots R_j^- \bowtie \ldots R_k$ using any worst-case optimal join algorithm, then project for each $j \in [k]$.
2. Compute $R_1 \bowtie \ldots R_j^\diamond \bowtie \ldots R_k$ using any worst-case optimal join algorithm, then project for each $j \in [k]$.
3. Materialize $R_1^+, \ldots, R_k^+$ as rectangular matrices and use matrix multiplication to compute their product.

Analysis. We assume that all relation sizes are bounded by N. The running time of the first step is $O(|\texttt{OUT}| \cdot \Delta_2)$ since each light value of variable x_i in relation R_i contributes to at least one output result.

For the second step, the key observation is that since y is light in all relations (except possibly R_i), the worst-case join size before projection is bounded by $O(N \cdot \Delta_1^{k-1})$, and hence the running time is also bounded by the same quantity.

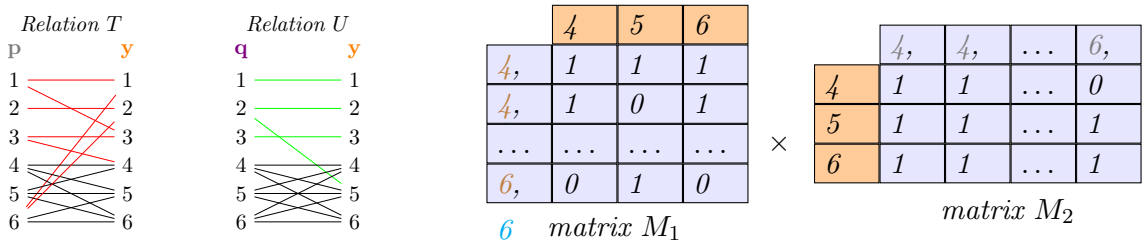
The last step is more involved than simply running matrix multiplication. This is because for each output result formed by heavy x_i values in R_i^+ (say $t = (a_1, a_2, \ldots a_k)$), we need to count the number of y values that connect with each a_i in t. However, running matrix multiplication one at a time between two matrices only tells about the number of connection y values for any two pair of a_i and not all of t. In order to count the y values for all of t together, we divide variables $x_1, \ldots x_k$ into two groups of size $\lceil k/2 \rceil$ and $\lfloor k/2 \rfloor$ followed by creating two adjacency matrices. Matrix V is of size $\left(\frac{N}{\Delta_2}\right)^{\lceil k/2 \rceil} \times \frac{N}{\Delta_1}$ such that

$$V_{(a_1, a_2, \ldots a_{\lceil k/2 \rceil}), b} = \begin{cases} 1, & (a_1, b) \in R_1, \ldots, (a_{\lceil k/2 \rceil}, b) \in R_{\lceil k/2 \rceil} \\ 0, & \text{otherwise} \end{cases}$$

Similarly, matrix W is of size $\left(\frac{N}{\Delta_2}\right)^{\lfloor k/2 \rfloor} \times \frac{N}{\Delta_1}$ such that

$$W_{(a_{\lceil k/2 \rceil+1} \ldots a_k),b} = \begin{cases} 1, & (a_{\lceil k/2 \rceil+1}, b) \in R_{\lceil k/2 \rceil+1}, \ldots, (a_k, b) \in R_k \\ 0, & \text{otherwise} \end{cases}$$

Example 29. *Consider the relations $R(x, y)$ and $S(z, y)$ from previous example and consider relation $T(p, y)$ and $U(q, y)$ as shown below.*



Suppose that we wish to compute the result of star query $Q_4^\star = R(x, y), S(z, y), T(p, y), U(q, y)$. Similar to the previous example, we fix $\Delta_1 = \Delta_2 = 2$. It is easy to verify for this example $R^- = R^\diamond$ and similarly for all other relations. We can now evaluate the join as mentioned in step (1) and (2). Next, we construct the matrices V and W. We divide the variables x, z, p, q into groups, x, z and p, q. V will consist of all heavy (x, z) pairs (9 in total) as one dimension and y as the second dimension. Similarly, W consists of all heavy (p, q) pairs (9 in total) as one dimension and y as the other.

Matrix construction takes time $(N/\Delta_2)^{\lceil k/2 \rceil} \cdot N/\Delta_1$ time in total. We have now reduced step three in computing the matrix product $V \times W^T$. Summing up the cost of all three steps, the total cost is in the order of

$$N \cdot \Delta_1^{k-1} + |\texttt{OUT}| \cdot \Delta_2 + M\big(\big(\frac{N}{\Delta_2}\big)^{\lceil k/2 \rceil}, \frac{N}{\Delta_1}, \big(\frac{N}{\Delta_2}\big)^{\lfloor k/2 \rfloor}\big)$$

Similar to the two-path query, we can find the exact value of Δ_1 and Δ_2 that minimizes the total running cost given a cost formula for matrix multiplication. We conclude this subsection with an illustrative example to show the benefit of matrix multiplication over the combinatorial algorithm.

Example 30. *Let $k = 3$ and $|\texttt{OUT}| = N^{3/2}$. The running time is minimized when*

$$N \cdot \Delta_1^2 = |\texttt{OUT}| \cdot \Delta_2 = M\big(\big(\frac{N}{\Delta_2}\big)^2, \frac{N}{\Delta_1}, \frac{N}{\Delta_2}\big)$$

The first equality gives us $\Delta_1^2 = \sqrt{N} \cdot \Delta_2$. We will choose the thresholds such that, $\Delta_2 < \Delta_1$. This means $\beta = N/\Delta_1$. From the second equality, we get $|\texttt{OUT}| \cdot \Delta_2 = \left(\frac{N}{\Delta_2}\right)^3$. The running time is minimized when $\Delta_2 = N^{\frac{6}{16}}, \Delta_1 = N^{\frac{7}{16}}$, in which case it is $O(N^{\frac{15}{8}})$ which is sub-quadratic (assuming $\omega = 2$). In contrast, Lemma 16 has a worse running time $O(N^2)$.

6.1.3 Boolean Set Intersection

In this setting, we are presented with a workload W containing boolean set intersection (BSI) queries of the form $Q_{ab}() = R(a, y), S(b, y)$ parametrized by the constants a, b. The queries come at a rate of B queries/time unit. In order to service these requests, we can use multiple machines. Our goal is twofold: minimize the number of machines we use, while at the same time minimizing the *average latency*, defined as the average time to answer each query.

Example 31. *The simplest strategy is to answer each request using a separate machine. Computing a single BSI query takes worst-case time $O(N)$, where N is the input size. Hence, the average latency is $O(N)$. At the same time, since queries come at a rate of B queries per time unit, we need $\rho = B \cdot N$ machines to keep up with the workload.*

Our key observation is that, instead of servicing each request separately, we can *batch* requests and compute them all at once. To see why this can be beneficial, suppose that we batch together C queries. Then, we can group all pairs of constants (a, b) to a single binary relation $T(x, z)$ of size C, and compute the following conjunctive query:

$$Q_{batch}(x, z) = R(x, y), S(z, y), T(x, z).$$

Here, R, S have size N, and T has size C. The resulting output will give the subset of the pairs of sets that indeed intersect. The above query can be computed by applying a worst-case optimal algorithm and then performing the projection: this will take $O(N \cdot C^{1/2})$ time. Hence, the average latency for a request will be $\frac{C}{B} + \frac{N}{C^{1/2}}$.

To get even lower latency, we can apply a variant of the AYZ algorithm [AYZ97] that uses fast matrix multiplication. The algorithm works as follows. For x, y, z values with degrees less than some threshold $\Delta \geq 1$, we perform the standard join with running time $O(N \cdot \Delta)$. For the remaining values, we express relations R, S as rectangular matrices of dimensions $\frac{C}{\Delta} \times \frac{N}{\Delta}$ and $\frac{N}{\Delta} \times \frac{C}{\Delta}$ respectively. We compute the matrix product, and then intersect the result with T to obtain the final output. The running time for this step is $M(\frac{C}{\Delta}, \frac{N}{\Delta}, \frac{C}{\Delta})$, which for $\omega = 2$ becomes $O(S \cdot N/\Delta^2)$. To minimize the running time for both steps, we choose $\Delta = C^{1/3}$, and thus the running time becomes $O(N \cdot C^{1/3})$.

Using the above algorithm, we can show the following.

Proposition 18. *Let W be a workload of queries of the form $Q_{ab}() = R(a, y), S(b, y)$ at the rate of B access requests per second. Then, assuming the exponent of matrix multiplication is $\omega = 2$, we can achieve an average latency of $O(N^{3/5}/B^{2/5})$ using $\rho = (B \cdot N)^{3/5}$ machines.*

Proof. Using batch size C, we can process C queries in time $O(N \cdot C^{1/3})$. Hence, the average latency in this case is $O\left(\frac{N}{C^{2/3}} + \frac{C}{\mathsf{B}}\right)$. To minimize the latency, we choose $C = (B \cdot N)^{3/5}$, in

Algorithm 10: `SizeAware` [DTL18]

Input: Indexed sets $R = \{R_1, \ldots, R_m\}$ and c

Output: Unordered SSJ result

1 *degree threshold* $x = \texttt{GetSizeBoundary}(R, c), \mathcal{L} \leftarrow \emptyset$
2 $R = R_l \cup R_h$ /* partition sets into light and heavy */
3 *Evaluate* $R \bowtie R_h$ *and enumerate result*
4 **foreach** $r \in R_l$ **do**
5 **foreach** *c-subset* r_c *of* r **do**
6 $\mathcal{L}[r_c] = \mathcal{L}[r_c] \cup r$
7 **foreach** $l \in \mathcal{L}[r_c]$ **do**
8 *enumerate every set pair in* l *if not output already*

which case we obtain an average latency of $O(N^{3/5}/B^{2/5})$. Then, the number of machines needed to service the workload is $(B \cdot N)/C^{2/3} = (B \cdot N)^{3/5}$. □

Observe that the above proposition strictly improves the number of machines compared to the baseline approach of Example 31. However, the average latency is smaller only for $B \leq N^{3/2}$, otherwise, it is larger.

In our experiments, we use the requests in the batch to filter the relations R and S, and then compute the 2-path query using Algorithm 9.

6.2 Speeding Up SSJ and SCJ

In this section, we will see how to use Algorithm 9 to speed up SSJ and SCJ.

***Unordered* SSJ.** We will first briefly review the state-of-the-art algorithm from [DTL18] called `SizeAware`. Algorithm 10 describes the size-aware set similarity join algorithm. The key insight is to identify the degree threshold x and partition the input sets into light and heavy. All heavy sets that form an output pair are enumerated by a sort-merge join. All light sets are processed by generating all possible c-sized subsets and then building an inverted index over it that allows for enumerating all light output pairs. x is chosen such that the cost of processing heavy and light sets is equal to each other.

We propose three key modifications that give us the new algorithm called `SizeAware++`. First, observe that $\mathcal{J}_H = R \bowtie R_h$ (line 3) is a natural join and requires $N \cdot N/x$ operations (recall that $|R_h| = N/x$ in the worst-case) even if the join output is smaller. Thus, Algorithm 9 is applicable here directly. This strictly improves the theoretical worst-case complexity of Algorithm 10 whenever $|\mathcal{J}_H| < N^2/x$ for all c.

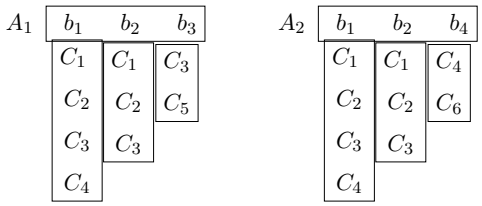
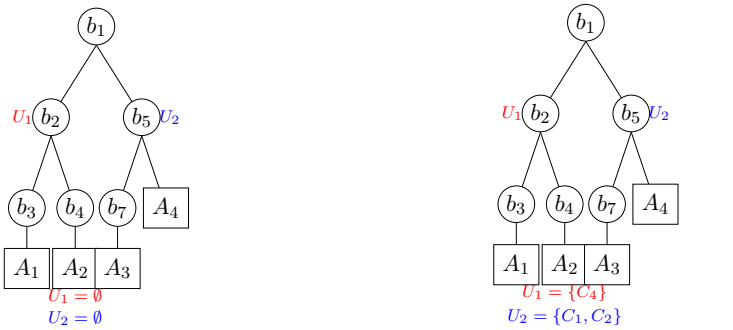


Figure 6.1: Example instance showing inverted lists



(a) Before materialization

(b) After materialization

Figure 6.2: Materialization in prefix tree

The second modification is to deal with high duplication when enumerating all light pairs using the inverted index $\mathcal{L}[r_c]$. The key observation is that line 8 is also performing a brute-force operation by going over all possible pairs and generating the full join result. This step takes $|\mathcal{J}_L| = \sum_{r_c} |\mathcal{L}[r_c]|^2$ time. If the final output is smaller than $|\mathcal{J}_L|$, then we can do better by using a matrix multiplication based algorithm.

The final observation relates to optimizing the expansion of light nodes (line 3 in Algorithm 9). Recall that the algorithm expands all light y values. Suppose we have $R(x, y)$ and $S(z, y)$ relations indexed and sorted according to variable order in the schema. Let $\mathcal{L}[b] = \{c \mid (c, b) \in S(z, y)\}$ denote the inverted index for relation S. The time required to perform the deduplication for a fixed value for x (say a) is $T = \sum_{b:(a,b)\in R} |\mathcal{L}[b]|$. This is unavoidable for overlap $c = 1$ in the worst case. However, it is possible that for $c > 1$, the output size is much smaller than T. In other words, deduplication step is expensive when the overlap between $\mathcal{L}[b]$ for different b is high. The key idea is to reuse computation across multiple a if there is a shared prefix and high overlap. We illustrate the key idea with the following example.

Example 32. *Consider the instance shown in Figure 6.1 with* $\{b_1, \ldots, b_7\}$ *as the possible keys for inverted index* $\mathcal{L}[b]$ *and sets* $A_1, \ldots A_4$ *as shown in the prefix tree. We use the length of inverted list* $\mathcal{L}[b]$ *as the key for sorting the input sets descending order. Suppose overlap* $c = 2$*. After merging inverted list for* b_1, b_2*, we know that* C_1, C_2, C_3 *are present at least two times across all of the lists. At this point, we materialize two things: (i)* $O_1 = \{C_1, C_2, C_3\}$

as output and, (ii) $U_1 = \mathcal{L}[b_1] \cup \mathcal{L}[b_2] \setminus O_1$ *at the node* b_2 *in the tree. We then continue with merging the other lists. When we start the enumeration for* A_2*, we know that the lists* b_1, b_2 *have already been processed and we can simply use the stored output. For* b_4*, we can go over its content and check whether the value is present in* U_1*. Similarly, for sets* A_3 *and* A_4 *that share* b_1, b_5 *as a prefix,* U_2 *stores the union of* $\mathcal{L}[b_1]$ *and* $\mathcal{L}[b_5]$ *after removing the sets that have appeared at least two times. For* A_1, A_2*, simply performing a merge step requires* $9 + 9 = 18$ *operations in total. However, if reusing the computation, we require only* 9 *operations:* 7 *for* A_1 *(merging* b_1 *and* b_2*) and* 2 *for merging inverted list of* b_4 *with stored* U_1*.*

The global sort order for all b is the length of the inverted list $\mathcal{L}[b]$. This encourages more computation reuse since bigger lists will give larger output and merging those repeatedly is expensive. Since a global sort order has been defined, we can construct a prefix tree and store the output and list union in the prefix tree. This technique will provide the largest gain when input sets A_i have a significant overlap. There also exists a tradeoff between the space requirement and computation reuse. Storing the output and list union at every node in the prefix tree increases the materialization requirement. The space usage can be controlled by limiting the depth at which the output and list union is stored. This can avoid excessive materialization when space is limited. The three optimizations in `SizeAware++` together can deliver speedups up to an order of magnitude over `SizeAware` for single-threaded implementation. Next, we highlight the important aspects regarding parallelization of SSJ and why `SizeAware` is not as amenable to parallelization as we would it to be. Partitioning joins $\mathcal{J}_H$ is straightforward since all parallel tasks require no synchronization and access the input data in a read-only manner. Parallelizing $\mathcal{J}_L$ in SizeAware is harder because of two reasons: (i) generating the c-sized subsets requires coordination since a given subset can be generated by multiple small sets; (ii) once the subsets have been generated, a given output pair (r, s) can be connected to multiple c-subsets. This means that each parallel task needs to coordinate to deduplicate multiple results across different c-subsets. On the other hand, using matrix multiplication allows for coordination-free parallelism as the matrix can be partitioned easily and each parallel task requires no interaction with each other. We show how this is achieved in Section 6.4. Note that `SizeAware++` also suffers from the same drawback as it is also generating the c-sized subsets which can be expensive. For dense datasets, using matrix multiplication and filtering the join result to find the similar set pairs is the fastest technique and also benefits the most from parallelization.

***Ordered* SSJ.** In this part, we look at the problem of enumerating SSJ in decreasing order of set similarity. Ordered enumeration of output pairs can be done by first generating the output and then sorting it. Note that the processing of light sets in Algorithm 10 (and consequently `SizeAware++`) is not amenable to finding the set pair with the largest intersection. Once

an output pair has been identified, we still need to enumerate over elements in the sets to identify the exact intersection size. On the other hand, our matrix multiplication based join provides a count that can be used for sorting.

SCJ. SCJ algorithms [BMGT16] typically prune away most of the set pairs that are surely not contained within each other. This acts as a blocking filter. For the remaining set pairs, the verification step performs a sort-merge join to verify if containment holds for either of the sets i.e we perform a merge join for all set pairs that need to be verified. However, the verification step can be slow if the overlap between sets is high (because of multiple replicas) or the average inverted index size is large. For these cases, we can get a significant speedup by simply evaluating the join-project result. This approach is most beneficial when the set containment join result is close to the join-project result. Further, the majority of SCJ algorithms do not use the power of parallel computation. PIEJoin [KRS+16] is the first and the only algorithm that addresses parallel SCJ. Since join processing is highly parallelizable, computing SCJ via join-project output benefits from parallel computation as well.

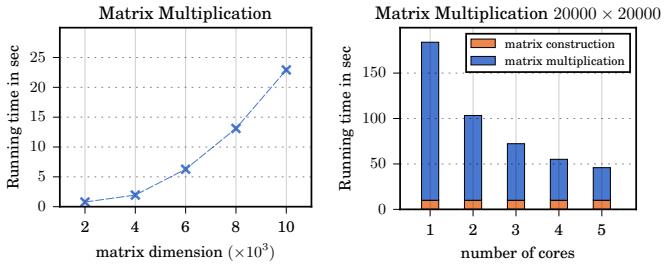
6.3 Cost-Based Optimization

In this section, we deep dive into the challenges of making our framework practical and how we can fine-tune the knobs to minimize the running time.

Estimating output size. So far, we have not discussed how to estimate for $|\mathsf{OUT}|$. We derive an estimate in the following manner. First, it is simple to show that the following holds for $\ddot{Q}$: $|\mathbf{dom}(x)| \leq |\mathsf{OUT}| \leq \min\{|\mathbf{dom}(x)|^2|, |\mathsf{OUT}_{\bowtie}|\}|$ and $|\mathsf{OUT}_{\bowtie}| \leq \mathsf{N} \cdot \sqrt{|\mathsf{OUT}|}$. Thus, a reasonable estimate for $|\mathsf{OUT}|$ is the geometric mean of $\max\{|\mathbf{dom}(x)|, (|\mathsf{OUT}_{\bowtie}|/\mathsf{N})^2\}$ and $\min\{|\mathbf{dom}(x)|^2|, |\mathsf{OUT}_{\bowtie}|\}|$. If $|\mathsf{OUT}_{\bowtie}|$ is not *much larger* than N, then the full join size is also reasonable estimate. We return to this point later in the discussion of the optimizer.

Indexing relations. Join processing by applying worst-case optimal join algorithms is possible only if all relations are indexed over the variables. This means each relation will be stored once for every index order that is required. For a binary relation $R(x, y)$, this would mean storing the relation indexed by all values of x as key and a sorted list of values for y and vice-versa. This can be accomplished in $O(|D| \log |D|)$ time after removing all tuples that do not join. During this pass, it is also straightforward to compute the size of the full join result (i.e., before the projection). Additionally, we create the following indexes:

1. For variable x and degree threshold δ, an index that tells us the deduplication effort when performing set union (i.e $\sum_{\text{light } a} \sum_{b:(a,b)\in R} |\mathcal{L}[b]|$) for all values of x with degree $\leq \delta$. We call this index $\mathsf{sum}(x_\delta)$. Similarly for all values of y with degree $\leq \delta$, $\mathsf{sum}(y_\delta) = \sum_{\text{light } b} |\mathcal{L}[b]|^2$.



(a) Single core scalability

(b) Multi core scalability

Figure 6.3: Matrix Multiplication Running Time

2. For the projected out variable y and degree threshold δ, an index that counts the number of x connected to all y values with degree $\leq \delta$. We call this index $\mathsf{cdfx}(y_\delta)$.

3. For each variable (say w), an index that tells us the number of values for w with degree $\leq \delta$. We call this index $\mathsf{count}(w_\delta)$.

All indexes can be built in linear time by storing the sorted vector containing the true distribution of values present in the relation. Then, given a δ, we can binary search over the vector to find the exact count (sum).

Matrix multiplication cost. A key component of all our techniques is matrix multiplication. Lemma 1 states the complexity of performing multiplication and also includes the cost of creating the matrices. However, in practice, this could be a significant overhead both in terms of memory consumption and time required. Further, the scalability of matrix multiplication implementation itself is subject to matrix size, the underlying linear algebra framework, and hardware support (vectorization, SIMD instructions, multithreading support, etc.) To minimize the running time, we need to take into consideration all system parameters to estimate the optimal threshold Δ values.

Symbol	Description
T_s	avg time for sequential access in `std::vector`
T_m	avg time for allocating 32 bytes of memory
co	number of cores available
$\hat{M}(u, v, w, co)$	estimate of time required to multiply matrices of dimension $u \times v$ and $v \times w$ using co cores
T_I	avg time for random access and insert in `std::vector`

Table 6.1: Symbol definitions.

Algorithm 11: Cost Based Optimizer

Output: degree threshold Δ_1, Δ_2

Estimate full join result $|\mathit{OUT}_{\bowtie}|$ *and* $|\mathit{OUT}|$
if $|\mathit{OUT}_{\bowtie}| \leq 20 \cdot N$ **then**
  use worst-case optimal join algorithm
$\mathsf{t}_{\mathsf{light}} \leftarrow |\mathtt{OUT}_{\bowtie}|, \mathsf{t}_{\mathsf{heavy}} \leftarrow 0, \mathsf{prev}_{\mathsf{light}} \leftarrow \infty, \mathsf{prev}_{\mathsf{heavy}} \leftarrow 0, \Delta_1 = N$
while true **do**
  $\mathsf{prev}_{\mathsf{light}} \leftarrow \mathsf{t}_{\mathsf{light}}, \mathsf{prev}_{\mathsf{heavy}} \leftarrow \mathsf{t}_{\mathsf{heavy}}$
  $\mathsf{prev}_{\Delta_1} \leftarrow \Delta_1, \mathsf{prev}_{\Delta_2} \leftarrow \Delta_2$
  $\Delta_1 \leftarrow (1-\epsilon)\Delta_1$
  $\Delta_2 \leftarrow N \cdot \Delta_1 / |\mathtt{OUT}|$
  $\mathsf{t}_{\mathsf{light}} \leftarrow T_I \cdot \mathsf{sum}(y_{\Delta_1}) + \cdot T_I \cdot \mathsf{sum}(x_{\Delta_2}) +$
    $T_m \cdot |\mathbf{dom}(x)| + T_s \cdot \mathsf{cdfx}(y_{\Delta_1}) \cdot |\mathbf{dom}(x)|$
  $u, v, w \leftarrow \#\text{heavy } x, y, z \text{ values using } \mathsf{count}(w_\delta)$
  $\mathsf{t}_{\mathsf{heavy}} \leftarrow \hat{M}(u, v, w, co) + T_m \cdot (u \cdot v + u \cdot w)$
  if $\mathsf{prev}_{\mathsf{light}} + \mathsf{prev}_{\mathsf{heavy}} \leq \mathsf{t}_{\mathsf{light}} + \mathsf{t}_{\mathsf{heavy}}$ **then**
    return $\mathsf{prev}_{\Delta_1}, \mathsf{prev}_{\Delta_2}$

Algorithm 11 describes the cost-based optimizer used to find the best degree thresholds to minimize the running time. To simplify the description, we describe the details for the case of $\ddot{Q}$ where $R = S$ (i.e., a self join). If the full join result is not much larger than the size of the input relation, then we can simply use any worst-casethe optimal join algorithm. For our experiments, we set the upper bound for $|\mathtt{OUT}_{\bowtie}|$ to be at most $20 \cdot N$. Beyond this point, we begin to see the benefit of using matrix multiplication for join-project computation.

To find the best possible estimates for Δ_1, Δ_2, we employ binary search over the value of Δ_2. In each iteration, we increase or decrease its value by a factor of $(1-\epsilon)$ where ϵ is a constant [1]. Once we fix the value of Δ_1 and Δ_2, we can query our precomputed index structure to find the exact number of operations that will be performed for all light y values and all light x values. Then, we find the number of heavy remaining values and get the estimate for time required to compute the matrix product. At the beginning of the next iteration, we compare the new time estimates with the previous iteration. If the new total time is larger than that of the previous iteration, we stop the process and use the last computed values as the degree thresholds. The entire process terminates in worst-case $O(\log^2 N)$ steps.

So far, we have not discussed how to estimate $\hat{M}(u, v, w, co)$. Since this quantity is system-dependent, we precompute a table that stores the time required for different values of

[1] We fix $\epsilon = 0.95$ for our experiments

u, v, w, co. As a brute-force computation for all possible values is very expensive to store and compute, we store the time estimate for $\hat{M}(p, p, p, co)$ for $p \in \{1000, 2000, \ldots, 20000\}, co \in [5]$. Then, given an arbitrary u, v, w, co, we can extrapolate from the nearest estimate available from the table. This works well since Eigen implements the naive $O(n^3)$ (with optimizations) algorithm that offers predictable running time. Figure 6.3a shows scalability of Eigen as the input matrix size increases. Since Eigen makes heavy use of SIMD instructions and vectorization, the running time displays a near quadratic growth rather than cubic for dimensions up to 5000×5000, beyond which the running time growth becomes cubic.

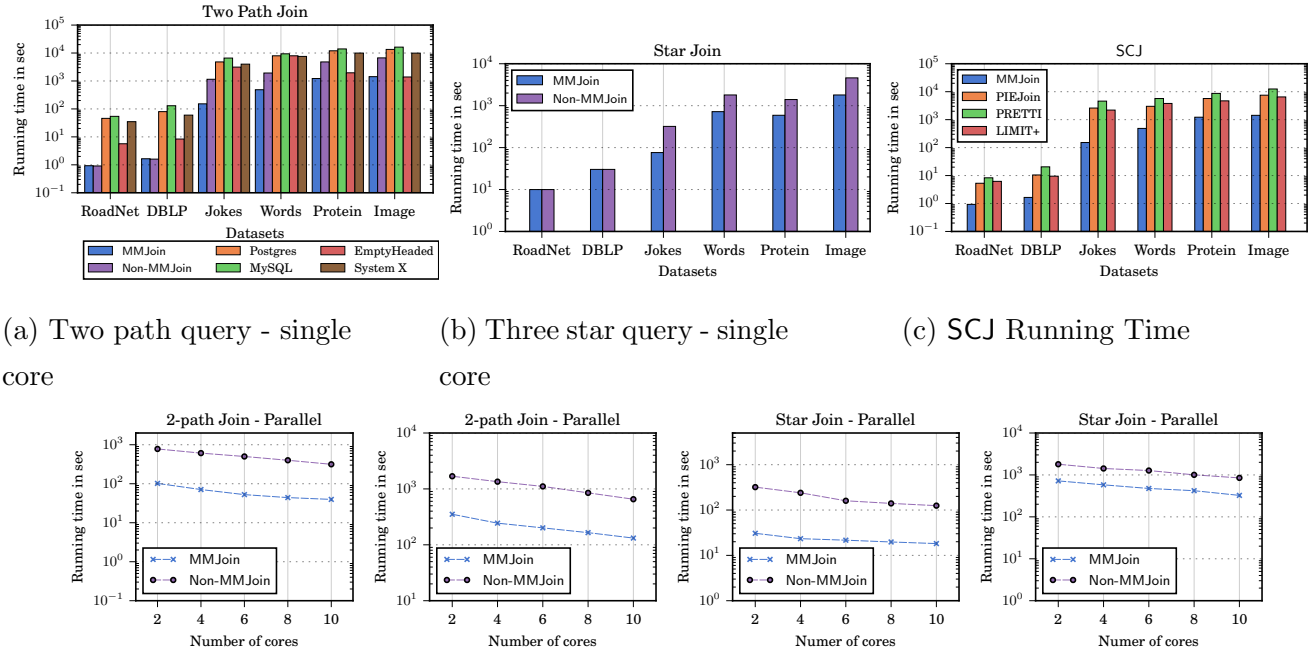
6.4 System Implementation

We implement our techniques in C++ as a standalone library. To perform matrix multiplication, we use Eigen [GJ+10] with Intel MKL [int09] as the underlying linear algebra framework. We choose Eigen for its ease of use and its seamless support for parallelization, even though other frameworks such as `MATLAB` are faster. Intel MKL offers two different functions for performing matrix operations: `SGEMM` and `DGEMM`. `SGEMM` allows low precision real arithmetic while `DGEMM` is for high precision arithmetic. This also makes `DGEMM` **3x** slower than `SGEMM` for the same operation being performed. We use floating point matrices everywhere rather than double precision or integer matrices for better performance.

At this point, we also wish to draw the attention of the reader towards some low-level details of the SSJ and SCJ implementation. Since the goal is to output the result in arbitrary order, both implementations enumerate the result without storing any of the output. Enumerating the result in (say) decreasing order of similarity size or containment size will require storing the output and sorting it before providing the user with a pointer to the result. We implement this in the straightforward way by sorting `std::vector` containing the output. Next, we describe the details of deduplication in our implementation for the case of unordered enumeration.

Deduplication. Since matrix multiplication deduplicates the output for all heavy values, we only need to handle deduplication for the remaining output tuples. The straightforward way to deduplicate is to use a hashmap. However, this has two disadvantages: (i) The memory for hashmap needs to be reserved upfront. This is critical to ensure that there is no resizing (and rehashing of the keys already present) of the hashmap at any point; (ii) upfront reservation would require $|\texttt{OUT}|$ amount of memory for deduplication, which is expensive both in terms of time and memory.

```
1            std::vector<int> y_light; // all light y values
2            std::unordered_map<int, set> R_xy; // indexed relation
3            std::unordered_map<int, set> R_yx; // indexed relation
4            std::vector<int> dedup(N); // reserving N memory
5            for(auto x : [N]) {
6                    dedup.assign(N,0);
```



(a) Two path query - single core

(b) Three star query - single core

(c) SCJ Running Time

(d) Jokes - multi core

(e) Words - multi core

(f) Jokes - multi core

(g) Words - multi core

Figure 6.4: Join Processing for two path and star query

```
7                       for(auto y : y_light) {
8                               if(R_xy[x].find(y) != R_xy[x].end()) {
9                                       for(auto z : R_yx[y]) {
10                                              dedup.at(z) += 1;
11                                              if (dedup.at(z) == 1) {
12                                                      std::cout << x << z;
13                                              }
14                                      }
15                              }
16                      }
17              }
```

The code snippet above shows the join for all light y values, which is the estimate used in line 11 of Algorithm 11. Line 4 reuses the `dedup` vector to check that a given z values has already been output or not. This is possible because we have fixed an x value and then merge all the z values reachable from y that are connected to x. Since the above approach involves random access over the `dedup` vector, it can be easily an order of magnitude more expensive than serial access if the vector does not fit in the L1 cache. An alternative approach is to deduplicate by appending all reachable z values, followed by sorting to deduplicate. For our experiments, we choose the best of the two strategies, depending on the number of elements that need to be deduplicated and the domain size of variables.

Parallelization. The single-threaded execution of all algorithms easily reaches several hours when faced with gigabyte-sized data sets and thus, parallel processing becomes necessary. Eigen parallelizes the matrix multiplication part in a coordination-free way, allowing both parts of our implementation to be highly parallelizable. Figure 6.3b shows the running time of matrix multiplication as the number of cores increases. The speedup obtained is near-linear as the resources available increase. This is possible because each core calculates the matrix product of a partition of data and requires no interaction with the other tasks.

6.5 Experimental Evaluation

In this section, we empirically evaluate the performance of our algorithms. The main goal of the section is fourfold:

1. Empirically verify the speed-up obtained for the 2-path and star queries using the algorithm from Section 6.1 compared to Postgres, MySQL, EmptyHeaded [ALT+17] and Commercial database X.

2. Evaluate the performance of the two-path query against `SizeAware` and `SizeAware++` for unordered and ordered SSJ.

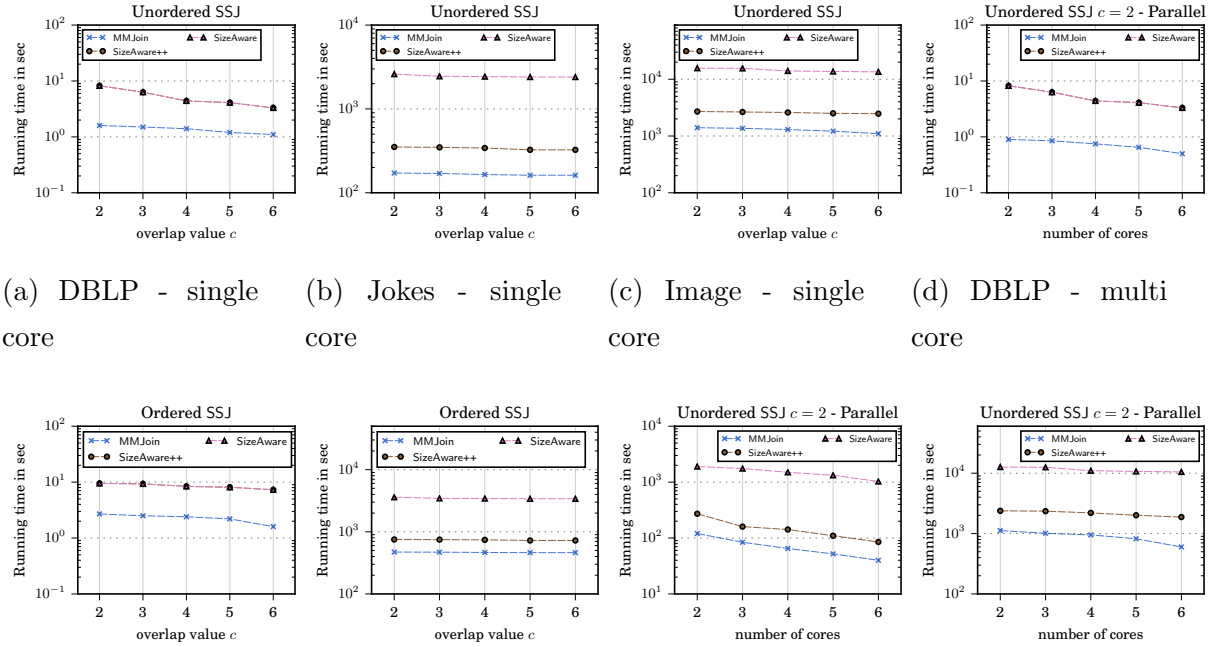
3. Evaluate the performance of the 2-path query against three state-of-the-art algorithms, namely PIEJoin [KRS+16], LIMIT+ [BMGT16], PRETTI for SCJ.

4. Validate the batching technique for boolean set intersection.

All experiments are performed on a machine with Intel Xeon CPU E5-2660@2.6GHz, 20 cores, and 150 GB RAM. Unless specified, all experiments are single-threaded implementations. We use the open-source implementation of each algorithm. For all experiments, we focus on self-join i.e all relations are identical. All C++ code is compiled using clang 8.0 with `-Ofast` flag and all matrix multiplication related code is additionally compiled with `-mavx -mfma -fopenmp` flags for multicore support. Each experiment is run 5 times and we report the running time by averaging three values after excluding the slowest and the fastest runtime.

6.5.1 Datasets

We conduct experiments on six real-world datasets from different domains. DBLP [dbl] is a bibliography dataset from DBLP representing authors and papers. RoadNet [roa] is road network of Pennsylvania. Jokes [jok] is a dataset scraped from Reddit where each set is a joke and there is an edge between a joke and a word if the work is present in the joke. Words [wor] is a bipartite graph between documents and the lexical tokens present in them.



(a) DBLP - single core (b) Jokes - single core (c) Image - single core (d) DBLP - multi core

(e) DBLP - single core (f) Jokes - single core (g) Jokes - multi core (h) Image - multi core

Figure 6.5: Unordered and Ordered SSJ

Image [ima] dataset is a graph where each image is connected to a feature attribute if the image contains the corresponding attribute and Protein [pro] refers to a bipartite graph where an edge signifies interaction between two proteins. Table 6.2 shows the main characteristics of the datasets. DBLP and RoadNet are examples of sparse datasets whereas the other four are dense datasets.

6.5.2 Simple Join Processing

In this part, we evaluate the running time for the two queries: $\ddot{Q}$ and $Q_3^\star$. To extract the maximum performance from Postgres, we use PGTune to set the best configuration parameters. This is important to ensure that the query plan does not perform nested loop inner joins. For all datasets, we create a hash index over each variable to ensure that the optimizer can choose the best query plan. We manually verify that the query plan generated by PostgreSQL (and MySQL) when running these queries chooses HashJoin or MergeJoin. For X, we allow up to 1TB of disk space and supply query hints to make sure that all of the CPU, RAM is available for query execution.

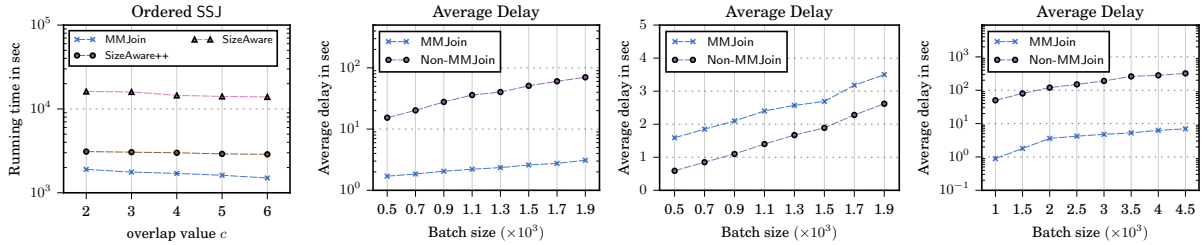
Figure 6.4a shows the run time for different algorithms on a single core. MySQL and Postgres have the slowest running time since they evaluate the full query join result and

Dataset	$\|R\|$	No. of sets	$\|\mathbf{dom}\|$	Avg set size	Min set size	Max set size
DBLP	10M	1.5M	3M	6.6	1	500
RoadNet	1.5M	1M	1M	1.5	1	20
Jokes	400M	70K	50K	5.7K	130	10K
Words	500M	1M	150K	500	1	10K
Protein	900M	60K	60K	15K	50	50K
Image	800M	70K	50K	11.4K	10K	50K

Table 6.2: Dataset Characteristics

then deduplicate. DBMS X performs marginally better than MySQL and Postgres. Non-matrix multiplication (denoted Non-MMJoin) join based on Lemma 16 is the second-best algorithm. Matrix multiplication based join (denoted MMJoin) is the fastest on all datasets except RoadNet and DBLP, where the optimizer chooses to compute the full join. A key reason for the huge performance difference between MMJoin and other algorithms is that deduplication by computing the full join result requires either sorting the data or using hash tables, both of which are expensive operations. In particular, using hash tables requires rehashing of entires every time the hash table increases. Similarly, sorting the full join result is expensive since the full join result can be orders of magnitude larger than the projection query result. Matrix multiplication avoids this since worst-case optimal joins can efficiently process the light part of the input and matrix multiplication is space-efficient due to its implicit factorization of the output formed by heavy values. Remarkably, EmptyHeaded performs comparably to MMJoin for Jokes dataset and outperforms MMJoin slightly on the Image dataset. This is because the Image dataset exclusively contains a dense component where every output is close to a clique. Since EmptyHeaded is designed as a linear algebra engine like Intel MKL, the performance is very similar. Figure 6.4d and 6.4e show the performance of the combinatorial and non-combinatorial algorithm as the number of cores increases. Both algorithms show a speed-up. We omit MySQL and Postgres since they do not allow for multicore processing of single queries.

Next, we turn to the star query on three relations. For this experiment, we take the largest sample of each relation so that the result can fit in the main memory and the join finishes in a reasonable time. Figure 6.4b shows the performance of the combinatorial and non-combinatorial join on a single core. All other engines (except EmptyHeaded) failed to finish in 15000 seconds except on RoadNet and DBLP. EmptyHeaded performed similarly to MMJoin on Protein and Image datasets but not on other datasets. Figure 6.4f and 6.4g show the performance in a multicore setting for Jokes and Words datasets. Once again, matrix multiplication performs better than its combinatorial version across all experiments.



(a) Image - single core (b) Jokes (c) Words (d) Image

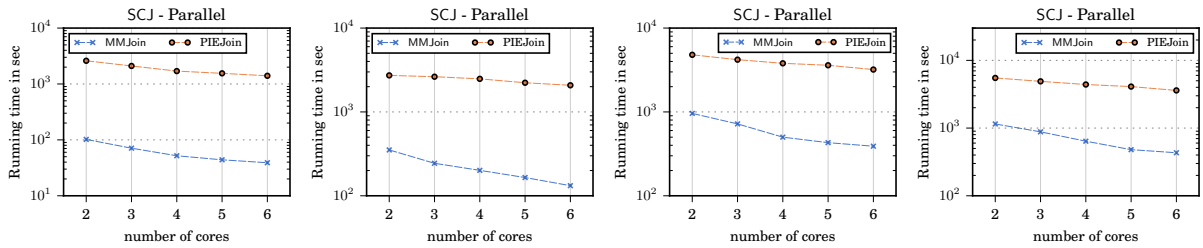
Figure 6.6: Ordered SSJ and minimizing average delay

6.5.3 Set Similarity

In this section, we look at set similarity (SSJ). For both settings below, we materialize the output at all nodes in the prefix tree. We will compare the performance of MMJoin, SizeAware and SizeAware++. We begin with the unordered setting.

***Unordered* SSJ.** Figure 6.5a, 6.5b and 6.5c show the running time of MMJoin, SizeAware and SizeAware++ on a single core for DBLP, Jokes and Image dataset respectively. Since DBLP is a sparse dataset with small set sizes, MMJoin is the fastest and both SizeAware and SizeAware++ are marginally slower due to the optimizer cost. For Jokes and Image datasets, SizeAware is the slowest algorithm. This is because both the light and heavy processing have a lot of deduplication to perform. SizeAware++ is an order of magnitude faster than SizeAware since it uses matrix multiplication but is slower than MMJoin because it still needs to enumerate the c-subsets before using matrix multiplication. MMJoin is the fastest as it is output sensitive and performs the best in a setting with many duplicates. Next, we look at the parallel version of unordered SSJ. Figure 6.5d, 6.5g and 6.5h show the results for multi core settings. For each experiment, we fix the overlap to $c = 2$. Observe that MMJoin join and SizeAware++ are more scalable than SizeAware. This is because the light sets processing of SizeAware cannot be done in parallel while matrix multiplication based deduplication can be performed in parallel.

***Ordered* SSJ.** Recall that for ordered SSJ, our goal is to enumerate the set pairs in descending order of set similarity. Thus, once the set pairs and their overlap is known, we need to sort the result using overlap as the key. Figure 6.5e and 6.5f show the running time for single-threaded implementation of ordered set similarity. Compared to the unordered setting, the extra overhead of materializing the output and sorting the result increases the running time for all algorithms. For SizeAware, there is an additional overhead of finding the overlap for all light sets as well. Both MMJoin and SizeAware++ maintain their advantage similar to the unordered setting.



(a) Jokes dataset (b) Words - multi core (c) Protein - multi core (d) Image - multi core

Figure 6.7: Unordered Parallel SCJ

Impact of optimizations. Recall that SizeAware++ contains three main optimizations - processing heavy sets using MMJoin, processing light sets via MMJoin and using prefix based materialization for computation sharing. Figure 6.8 shows the effect of switching on various optimizations. NO-OP denotes all optimizations switched off. The running time is shown as a percentage of the NO-OP running time (100%). Light denotes using two-path join on only light sets identified by SizeAware but not using the prefix optimizations. Heavy includes the Light optimizations switched on plus two-path join processing on the heavy sets but prefix based optimization is still switched off. Finally, Prefix switches on materialization of the output in prefix tree on top of Light and Heavy. As the figure shows, both Light and Heavy optimizations together improve the running time by an order of magnitude and Prefix further improves by a factor of 5×.

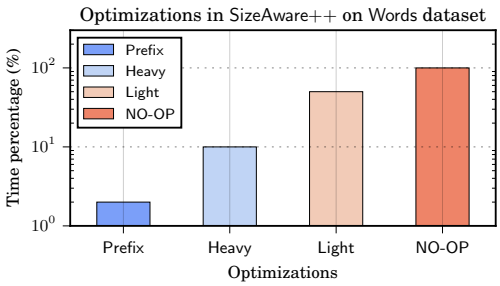


Figure 6.8: SSJ - Impact of optimizations on Words

6.5.4 Set Containment

In this section, we evaluate the performance of different set containment join algorithms. Figure 6.4c shows the running time of PIEJoin, PRETTI and LIMIT+. For all SCJ algorithms, we use the infrequent sort order, choose a limit value of two for LIMIT+ and run the variant where the output is materialized (instead of just simply counting its size).

Once again join processing yields the fastest running time since the join output is a superset of the set containment join result and except RoadNet and DBLP, the join-project result and SCJ result is close to each other. Since the average set size is large for most datasets, SCJ algorithms need to perform expensive verification operations. For the parallel setting, Figure 6.7a, 6.7b, 6.7c and 6.7d show the performance of PIEJoin vs. MMJoin. PIEJoin does not scale as well as MMJoin as it is sensitive to data distribution and choice of partitions chosen by the heuristic in the algorithm.

6.5.5 Boolean Set Intersection

In this part of the experiment, we look at the boolean set intersection scenario where queries are arriving in an online fashion. The arrival rate of queries is set to $B = 1000$ queries per second and our goal is to minimize the average delay metric as defined in Section 6.1.3. The workload is generated by sampling each set pair uniformly at random. We run this experiment for 300 seconds for each batch size and report the mean average delay metric value. Figure 6.6b, 6.6c and 6.6d show the average delay for the three datasets at different batch sizes. Recall that the smaller batch size we choose, the more processing units are required. For the Jokes dataset, Non-MMJoin has the smallest average delay of $\approx 1s$ when $S = 10$. In that time, we collect a further 1000 requests, which means that there is a need for 100 parallel processing units. On the other hand, MMJoin achieves a delay of $\approx 2s$ at batch size 900. Thus, we need only 3 parallel processing units in total to keep up with the workload while sacrificing only a small penalty in latency. For the Image dataset, MMJoin can achieve average delay of $1s$ at $S = 1000$ queries while Non-MMJoin achieves $50s$ at the same batch size. This shows that matrix multiplication is useful for achieving a smaller latency using fewer resources, in line with the theoretical prediction. For the Words dataset, most of the sets have a small degree. Thus, the optimizer chooses to evaluate the join via the combinatorial algorithm. This explains the in-sync behavior of average delay for both algorithms. Note that MMJoin is marginally slower because of the overhead of the optimizer ($\leq 2s$).

Chapter 7

Ranked Enumeration of Conjunctive Query Results

In this chapter, we will focus on join processing for queries in the presence of ranking functions. For many data processing applications, enumerating query results according to an order given by a ranking function is a fundamental task. For example, [YAG+18, CLZ+15] consider a setting where users want to extract the top patterns from an edge-weighted graph, where the rank of each pattern is the sum of the weights of the edges in the pattern. Ranked enumeration also occurs in `SQL` queries with an `ORDER BY` clause [QCS07, ISA+04]. In the above scenarios, the user often wants to see the first k results in the query as quickly as possible, but the value of k may not be predetermined. Hence, it is critical to construct algorithms that can output the first tuple of the result as fast as possible, and then output the next tuple in the order with a very small *delay*. In this article, we study the algorithmic problem of enumerating the result of a Conjunctive Query (CQ, for short) against a relational database where the tuples must be output in the order given by a ranking function.

The simplest way to enumerate the output is to materialize the result $\texttt{OUT}$ and sort the tuples based on the score of each tuple. Although this approach is conceptually simple, it requires that $|\texttt{OUT}|$ tuples are materialized; moreover, the time from when the user submits the query to when she receives the first output tuples is $\Omega(|\texttt{OUT}| \cdot \log |\texttt{OUT}|)$. Further, the space and delay guarantees do not depend on the number of tuples that the user wants to see. More sophisticated approaches to this problem construct optimizers that exploit properties such as the monotonicity of the ranking function, allowing for join evaluation on a subset of the input relations (see [IBS08] and references within). Despite the significant progress, all of the known techniques suffer from large worst-case space requirements, no dependence on k, and provide no formal guarantees on the delay during enumeration, with the exception of a few cases where the ranking function is of a special form. Fagin et al. [FLN03] initiated a long line of study related to aggregation over *sorted lists*. However, [FLN03] and subsequent works also suffer from the above-mentioned limitations as we do not have the materialized output $Q(D)$ that can be used as sorted lists.

In this chapter, we construct algorithms that remedy some of these issues. We present novel algorithms in the same framework as described in section 2.4: the *preprocessing phase*, where the system constructs a data structure that can be used later and the *enumeration phase* when the results are generated. The additional constraint imposed on the enumeration phase is that the results generated must be sorted according to a given ranking function. All of our algorithms aim to minimize the time of the preprocessing phase and guarantee a *logarithmic delay* $O(\log |D|)$ during enumeration. Although we cannot hope to perform efficient ranked enumeration for an arbitrary ranking function, we show that our techniques apply for most ranking functions of practical interest, including lexicographic ordering, and sum (also product or max) of weights of input tuples among others.

Example 33. *Consider a weighted graph G, where an edge (a, b) with weight w is represented by the relation $R(a, b, w)$. Suppose that the user is interested in finding the (directed) paths of length 3 in the graph with the lowest score, where the score is a (weighted) sum of the weights of the edges. The user query in this case can be specified as: $Q(x, y, z, u, w_1, w_2, w_3,) = R(x, y, w_1), R(y, z, w_2), R(z, u, w_3)$ where the ranking of the output tuples is specified for example by the score $5w_1 + 2w_2 + 4w_3$. If the graph has N edges, the naïve algorithm that computes and ranks all tuples needs $\Omega(N^2 \log N)$ preprocessing time. We show that it is possible to design an algorithm with $O(N)$ preprocessing time, such that the delay during enumeration is $O(\log N)$. This algorithm outputs the first k tuples by materializing $O(N+k)$ data, even if the full output is much larger.*

The problem of ranked enumeration for CQs has been studied both theoretically [KS06, CS07, OZ15b] and practically [YAG+18, CLZ+15, BOZ12]. Theoretically, [KS06] establishes the tractability of enumerating answers in sorted order with polynomial delay (combined complexity), albeit with suboptimal space and delay factors for two classes of ranking functions. [YAG+18] presents an anytime enumeration algorithm restricted to acyclic queries on graphs that uses $\Theta(|\texttt{OUT}| + |D|)$ space in the worst case, has a $\Theta(|D|)$ delay guarantee, and supports only simple ranking functions. As we will see, both of these guarantees are suboptimal and can be improved upon.

Ranked enumeration has also been studied for the class of lexicographic orderings. In [BDG07a], the authors show that *free-connex acyclic CQs* can be enumerated in constant delay after only linear time preprocessing. Here, the lexicographic order is chosen by the algorithm and not the user. Factorized databases [BOZ12, OZ15b] can also support constant delay ranked enumeration, but only when the lexicographic ordering agrees with the order of the query decomposition. In contrast, our results imply that we can achieve a logarithmic delay with the same preprocessing time for *any* lexicographic order.

Our Contribution. For ranking results of full (projection free) CQs, we show how to obtain logarithmic delay guarantees with small preprocessing time. We summarize our technical contributions below:

1. Novel Algorithms for Full CQs. Our main contribution (Theorem 19) is a novel algorithm that uses query decomposition techniques in conjunction with the structure of the ranking function. The preprocessing phase sets up priority queues that maintain partial tuples at each node of the decomposition. During the enumeration phase, the algorithm materializes the output of the subquery formed by the subtree rooted at each node of the decomposition *on-the-fly*, in sorted order according to the ranking function. To define the rank of the partial tuples, we require that the ranking function can be *decomposed* with respect to the particular decomposition at hand. Theorem 19 then shows that with $O(|D|^{\texttt{fhw}})$ preprocessing time, where `fhw` is the *fractional hypertree width* of the decomposition, we can enumerate with delay $O(\log |D|)$. We then discuss how to apply our main result to commonly used classes of ranking functions. Our work thoroughly resolves an open problem stated at the Dagstuhl Seminar 19211 [BKPS19] on ranked enumeration (see Question 4.6).

2. Extending results to UCQs. We propose two extensions of Theorem 19 that improve the preprocessing time to $O(|D|^{\texttt{subw}})$, a polynomial improvement over Theorem 19 where `subw` is the *submodular width* of the query Q. The result is based on a simple but powerful application of the main result that can be applied to any full UCQ Q combined with the `PANDA` algorithm proposed by Abo Khamis et al. [AKNS17].

3. Nearly-Tight Lower Bounds. Finally, we show lower bounds (conditional and unconditional) for our algorithmic results. In particular, we show that subject to a popular conjecture, the logarithmic factor in delay cannot be removed. Additionally, we show that for two particular classes of ranking functions, we can characterize for which acyclic queries it is possible to achieve logarithmic delay with linear preprocessing time, and for which it is not.

Organization. We overview the prior work in the space of ranked enumeration in Section 7.1. Section 7.3 contains the algorithm and the proofs for our main result. We study the extension of the algorithm to UCQs in Section 7.4. We conclude this chapter with lower bounds in Section 7.5.

7.1 Related Work

Top-k ranked enumeration of join queries has been studied extensively by the database community for both certain [LCIS05, QCS07, ISA+04, LSCI05] and uncertain databases [RDS07, ZLGZ10]. Most of these works exploit the monotonicity property of scoring functions, building offline indexes and integrate the function into the cost model of the query optimizer to

bound the number of operations required per answer tuple. We refer the reader to [IBS08] for a comprehensive survey of top-k processing techniques discovered before 2008. More recent work [CLZ+15, GGY+14] has focused on enumerating *twig-pattern* queries over graphs. Our work departs from this line of work in two aspects: *(i)* use of novel techniques that use query decompositions and clever tricks to achieve strictly better space requirement and formal delay guarantees; *(ii)* our algorithms apply to arbitrary hypergraphs as compared to simple graph patterns over binary relations. Most closely related to our setting is [KS06] and a line of work initiated by [YAG+18]. [KS06] uses an adaptation of Lawler-Murty's procedure to incrementally compute ordered answers of full acyclic CQs. However, that work was mainly focused on studying the combined complexity of the problem. Further, since the goal was to obtain polynomial delay guarantees, the authors did not attempt to obtain the best possible delay guarantees. This line of work was further extended to parallel setting [GKS11] and also when the data is incomplete [KS07].

The other line of work was initiated by Yang et al. [YAG+18] who presented a novel anytime algorithm, called KARPET, for enumerating *homomorphic tree patterns* with worst-case delay and space guarantees where the ranking function is the sum of weights of input tuples that contribute to an output tuple. KARPET is an any-time algorithm that generates candidate output tuples with different scores and sorts them incremental via a priority queue. However, the candidate generation phase is expensive (which translates to linear delay guarantees) and can be improved substantially, as we show in this article. [YRLG18] made the further connection that KARPET can be extended to arbitrary full CQs (including cycles) by considering different tree decompositions. This connection was concretely established in concurrent work [TAG+] that built upon [YAG+18, YRLG18] to obtain logarithmic delay guarantees using a dynamic programming approach combined with Lawler's procedure [Law72]. Moreover, [TAG+] also conducted an extensive empirical evaluation to demonstrate the benefit of improved delay guarantees in practice. Very recently, the authors were also able to extend their results to theta-joins as well [TGR21a]. For a more detailed overview of the prior work on the topic of ranked enumeration, we refer the reader to [TGR20, TAG+].

Rank aggregation algorithms. Top-k processing over ranked lists of objects has a rich history. The problem was first studied by Fagin et al. [Fag02, FLN03] where the database consists of N objects and m ranked streams, each containing a ranking of the N objects to find the top-k results for coordinate monotone functions. The authors proposed Fagin's algorithm (FA) and Threshold algorithm (TA), both of which were shown to be instance optimal for database access cost under sorted list access and random access model. This model would apply to our setting only if $Q(D)$ is already computed and materialized. More importantly, TA can only give $O(N)$ delay guarantee using $O(N)$ space. [NCS+01] extended

the problem setting to the case where we want to enumerate top-k answers for t-path query. The first proposed algorithm J^* uses an iterative deepening mechanism that pushes the most promising candidates into a priority queue. Unfortunately, even though the algorithm is instance optimal with respect to number of sorted access over each list, the delay guarantee is $\Omega(|\texttt{OUT}|)$ with space requirement $S = \Omega(|\texttt{OUT}|)$. A second proposed algorithm J^*_{PA} allows random access over each sorted list. J^*_{PA} uses a dynamic threshold to decide when to use random access over other lists to find joining tuples versus sorted access but does not improve formal guarantees.

Query enumeration. The notion of constant delay query enumeration was introduced by Bagan, Durand, and Grandjean in [BDG07a]. In this setting, preprocessing time is supposed to be much smaller than the time needed to evaluate the query (usually, linear in the size of the database), and the delay between two output tuples may depend on the query, but not on the database. This notion captures the *intrinsic hardness* of the query structure. For an introduction to this topic and an overview of the state-of-the-art, we refer the reader to the survey [Seg13b, Seg15a]. Most of the results in existing works focus only on lexicographic enumeration of query results where the ordering of variables cannot be arbitrarily chosen. Transferring the static setting enumeration results to under updates has also been a subject of recent interest [BKS18, BKS17b].

Factorized databases. Following the landmark result of [OZ15b] which introduced the notion of using the logical structure of the query for efficient join evaluation, a long line of research has benefited from its application to learning problems and broader classes of queries [BOZ12, BKOZ13, OS16, DK18, KNOZ20b, DHK20, DHK21]. The core idea of factorized databases is to convert an arbitrary query into an acyclic query by finding a query decomposition of small width. This width parameter controls the space and pre-processing time required to build indexes allowing for constant delay enumeration. We build on top of factorized representations and integrate ranking functions in the framework to enable enumeration beyond lexicographic orders.

7.2 Ranking Functions

Consider a natural join query Q and a database D. Our goal is to enumerate all the tuples of $Q(D)$ according to an order that is specified by a *ranking function*. In practice, this ordering could be specified, for instance, in the `ORDER BY` clause of a `SQL` query.

Formally, we assume a total order $\succeq$ of the valuations θ over the variables of Q. The total order is induced by a ranking function `rank` that maps each valuation θ to a number $\texttt{rank}(\theta) \in \mathbb{R}$. In particular, for two valuations θ_1, θ_2, we have $\theta_1 \succeq \theta_2$ if and only if $\texttt{rank}(\theta_1) \geq \texttt{rank}(\theta_2)$. Throughout the article, we will assume that `rank` is a computable function that

takes times linear in the input size to the function. We present below two concrete examples of ranking functions.

Example 34. *For every constant $c \in \mathbf{dom}$, we associate a weight $w(c) \in \mathbb{R}$. Then, for each valuation θ, we can define $\texttt{rank}(\theta) := \sum_{x \in \mathcal{V}} w(\theta(x))$. This ranking function sums the weights of each value in the tuple.*

Example 35. *For every input tuple $t \in R_F$, we associate a weight $w_F(t) \in \mathbb{R}$. Then, each valuation θ, we can define $\texttt{rank}(\theta) = \sum_{F \in \mathcal{E}} w_F(\theta[x_F])$ where x_F is the set of variables in F. In this case, the ranking function sums the weights of each contributing input tuple to the output tuple t (we can extend the ranking function to all valuations by associating a weight of 0 to tuples that are not contained in a relation).*

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

Definition 4 (Decomposable Ranking)**.** *Let $\texttt{rank}$ be a ranking function over $\mathcal{V}$ and $S \subseteq \mathcal{V}$. We say that $\texttt{rank}$ is* S-decomposable *if there exists a total order for all valuations over S, such that for every valuation φ over $\mathcal{V} \setminus S$, and any two valuations θ_1, θ_2 over S we have:*

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

We say that a ranking function is *totally decomposable* if it is S-decomposable for every subset $S \subseteq \mathcal{V}$, and that it is *coordinate decomposable* if it is S-decomposable for any singleton set. Additionally, we say that it is *edge decomposable* for a query Q if it is S-decomposable for every set S that is a hyperedge in the query hypergraph. We point out here that totally decomposable functions are equivalent to monotonic orders as defined in [KS06].

Example 36. *The ranking function $\texttt{rank}(\theta) = \sum_{x \in \mathcal{V}} w(\theta(x))$ defined in Example 34 is totally decomposable, and hence also coordinate decomposable. Indeed, pick any set $S \subseteq \mathcal{V}$. We construct a total order on valuations θ over S by using the value $\sum_{x \in S} w(\theta(x))$. Now, consider valuations θ_1, θ_2 over S such that $\sum_{x \in S} w(\theta_1(x)) \geq \sum_{x \in S} w(\theta_2(x))$. Then, for any valuation φ over $\mathcal{V} \setminus S$ we have:*

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

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

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

where $\mathbf{dom} = \{-1, 1\}$*, and define* $\mathsf{rank}(\theta) := \sum_{i=1}^{d} \theta(x_i) \cdot \theta(y_i)$*. This ranking function corresponds to taking the inner product of the input tuples if viewed as vectors. The total order for* $\mathbf{dom}$ *is* $-1 \prec 1$*. It can be shown that for* $d = 2$*, the function is not* $\{x_1, x_2\}$*-decomposable. For instance, if we define* $\theta_1 = (1, -1) \succeq \theta_2 = (1, 1)$*, then for* $\varphi = (-1, -1)$ *we get* $\mathsf{rank}(1, -1, -1, -1) = \mathsf{rank}(\theta_1 \circ \varphi) > \mathsf{rank}(1, 1, -1, -1) = \mathsf{rank}(\theta_2 \circ \varphi)$ *but if we define* $\varphi = (1, -1)$*, then we get* $\mathsf{rank}(1, 1, 1, -1) = \mathsf{rank}(\theta_1 \circ \varphi) < \mathsf{rank}(1, -1, 1, -1) = \mathsf{rank}(\theta_2 \circ \varphi)$*. This demonstrates that the ranking function is not independent of valuations over* $\{y_1, y_2\}$ *and thus, the function does not satisfy the definition of decomposability.*

Definition 5. *Let* $\boldsymbol{rank}$ *be a ranking function over a set of variables* $\mathcal{V}$*, and* $S, T \subseteq \mathcal{V}$ *such that* $S \cap T = \emptyset$*. We say that* $\boldsymbol{rank}$ *is* T-decomposable conditioned on S *if for every valuation* θ *over* S*, the function* $\mathsf{rank}_\theta(\varphi) := \mathsf{rank}(\theta \circ \varphi)$ *defined over* $\mathcal{V} \setminus S$ *is* T*-decomposable.*

The next lemma connects the notion of conditioned decomposability with decomposability.

Lemma 18. *Let* $\boldsymbol{rank}$ *be a ranking function over a set of variables* $\mathcal{V}$*, and* $T \subseteq \mathcal{V}$*. If* $\boldsymbol{rank}$ *is* T*-decomposable, then it is also* T*-decomposable conditioned on* S *for any* $S \subseteq \mathcal{V} \setminus T$*.*

Proof. We need to show that for every valuation π over S, $\mathtt{rank}(\pi \circ \Phi \circ \theta)$ is T-decomposable where Φ is defined over $U = \mathcal{V} \setminus (S \cup T)$ and θ is defined over T. We use the same total order for θ as used for T-decomposability. Let $\theta_1 \succeq \theta_2$, and consider any valuation Φ over U. Define the valuation φ over $\mathcal{V} \setminus T$ such that $\varphi[S] = \pi$ and $\varphi[U] = \Phi$. Then,

$$\begin{aligned}
\theta_1 \succeq \theta_2 &\Rightarrow \mathtt{rank}(\varphi \circ \theta_1) \geq \mathtt{rank}(\varphi \circ \theta_2) \\
&\Leftrightarrow \mathtt{rank}(\varphi[S] \circ \varphi[U] \circ \theta_1) \geq \mathtt{rank}(\varphi[S] \circ \varphi[U] \circ \theta_2) \\
&\Leftrightarrow \mathtt{rank}(\pi \circ \Phi \circ \theta_1) \geq \mathtt{rank}(\pi \circ \Phi \circ \theta_2)
\end{aligned}$$

Step 1 follows from the definition of T-decomposable. Step 2 and 3 compute the restriction of φ to S and U. □

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

Definition 6 (Compatible Ranking)**.** *Let* $\mathcal{T}$ *be a rooted tree decomposition of hypergraph* $\mathcal{H}$ *of a natural join query. We say that a ranking function is* compatible with $\mathcal{T}$ *if for every node* t *it is* $(\mathcal{B}_t^{\prec} \setminus \boldsymbol{key}(t))$*-decomposable conditioned on* $\boldsymbol{key}(t)$*.*

Example 37. *Consider the join query* $Q(x, y, z) = R(x, y), S(y, z)$*, and the ranking function from Example 35,* $\boldsymbol{rank}(\theta) = w_R(\theta(x), \theta(y)) + w_S(\theta(y), \theta(z))$*. This function is not* $\{z\}$*-decomposable, but it is* $\{z\}$*-decomposable conditioned on* $\{y\}$*.*

Consider a decomposition of the hypergraph of Q that has two nodes: the root node r with $\mathcal{B}_r = \{x, y\}$, and its child t with $\mathcal{B}_t = \{y, z\}$. Since $\mathcal{B}_t^{\prec} = \{y, z\}$ and $\texttt{key}(t) = \{y\}$, the condition of compatibility holds for node t. Similarly, for the root node $\mathcal{B}_t^{\prec} = \{x, y, z\}$ and $\texttt{key}(t) = \{\}$, hence the condition is trivially true as well. Thus, the ranking function is compatible with the decomposition.

7.2.1 Problem Parameters

Given a natural join query Q and a database D, we want to enumerate the tuples of $Q(D)$ according to the order specified by `rank`. We will study this problem in the enumeration framework similar to that of [Seg15b], where an algorithm can be decomposed into two phases:

- a **preprocessing phase** that takes time T_p and computes a data structure of size S_p,
- an **enumeration phase** that outputs $Q(D)$ with no repetitions. The enumeration phase has full access to any data structures constructed in the preprocessing phase and can also use additional space of size S_e. The *delay* δ is defined as the maximum time to output any two consecutive tuples (and also the time to output the first tuple, and the time to notify that the enumeration has been completed).

It is straightforward to perform ranked enumeration for any ranking function by computing $Q(D)$, storing the tuples in an ordered list, and finally enumerating by scanning the ordered list with constant delay. This simple strategy implies the following result.

Proposition 19. *Let Q be a natural join query with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. Let $\mathcal{T}$ be a tree decomposition with fractional hypertree-width $\texttt{fhw}$, and $\texttt{rank}$ be a ranking function. Then, for any input database D, we can preprocess D in time $T_p = O(\log |D| \cdot |D|^{\texttt{fhw}} + |Q(D)|)$ and space $S_p = O(|Q(D)|)$, such that for any k, we can enumerate the top-k results of $Q(D)$ with delay $\delta = O(1)$ and space $S_e = O(1)$*

The drawback of Proposition 19 is that the user will have to wait for $\Omega(|Q(D)| \cdot \log |Q(D)|)$ time to even obtain the first tuple in the output. Moreover, even when we are interested in a few tuples, the whole output result will have to be materialized. Instead, we want to design algorithms that minimize the preprocessing time and space, while guaranteeing a small delay δ. Interestingly, as we will see in Section 7.5, the above result is essentially the best we can do if the ranking function is completely arbitrary; thus, we need to consider reasonable restrictions of `rank`.

To see what it is possible to achieve in this framework, it will be useful to keep in mind what we can do in the case where there is no ordering of the output.

Theorem 18 (due to [OZ15b]). *Let Q be a natural join query with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. Let $\mathcal{T}$ be a tree decomposition with fractional hypertree-width $\texttt{fhw}$. Then, for any input database D, we can pre-process D in time $T_p = O(|D|^{\texttt{fhw}})$ and space $S_p = O(|D|^{\texttt{fhw}})$ such that we can enumerate the results of $Q(D)$ with delay $\delta = O(1)$ and space $S_e = O(1)$*

For acyclic queries, $\texttt{fhw} = 1$, and hence the preprocessing phase takes only linear time and space in the size of the input.

7.3 Main Result

In this section, we present our first main result.

Theorem 19 (Main Theorem). *Let Q be a natural join query with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. Let $\mathcal{T}$ be a fixed tree decomposition with fractional hypertree-width $\texttt{fhw}$, and $\texttt{rank}$ be a ranking function that is compatible with $\mathcal{T}$. Then, for any database D, we can preprocess D with*

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

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

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

In the above theorem, the preprocessing step is independent of the value of k: we perform the same preprocessing if the user only wants to obtain the first tuple or all tuples in the result. However, if the user decides to stop after having obtained the first k results, the space used during enumeration will be bound by $O(k)$. We should also note that all of our algorithms work in the case where the ordering of the tuples/valuations is instead expressed through a `comparable` function that, given two valuations, returns the larger.

It is instructive to compare Theorem 19 with Theorem 18, where no ranking is used when enumerating the results. There are two major differences. First, the delay δ has an additional logarithmic factor. As we will discuss later in Section 7.5, this logarithmic factor is a result of doing ranked enumeration, and it is most likely unavoidable. The second difference is that the space S_e used during enumeration blows up from constant $O(1)$ to $O(|Q(D)|)$ in the worst case (when all results are enumerated).

In the remainder of this section, we will present a few applications of Theorem 19, and then prove the theorem.

7.3.1 Applications

We show here how to apply Theorem 19 to obtain algorithms for different ranking functions.

Vertex-Based Ranking. A vertex-based ranking function over $\mathcal{V}$ is of the form: $\texttt{rank}(\theta) := \bigoplus_{x \in \mathcal{V}} f_x(\theta(x))$ where f_x maps values from **dom** to some set $U \subseteq \mathbb{R}$, and $\langle U, \oplus \rangle$ forms a *commutative monoid*. Recall that this means that $\oplus$ is a binary operator that is commutative, associative, and has an identity element in U. We say that the function is monotone if $a \geq b$ implies that $a \oplus c \geq b \oplus c$ for every c. Such examples are $\langle \mathbb{R}, + \rangle$, $\langle \mathbb{R}, * \rangle$, and $\langle U, max \rangle$, where U is bounded.

Lemma 19. *Let* ***rank*** *be a monotone vertex-based ranking function over $\mathcal{V}$. Then,* ***rank*** *is totally decomposable, and hence compatible with any tree decomposition of a hypergraph with vertices $\mathcal{V}$.*

Proof. Pick any set $S \subseteq \mathcal{V}$ and let $\theta^\star$ be the valuation over $\mathcal{V} \setminus S$ such that for every x, $f_x(\theta^\star(x)) = e$, where e is the identity element of the monoid. Suppose that $\texttt{rank}(\theta^\star \circ \theta_1) \geq \texttt{rank}(\theta^\star \circ \theta_2)$ for valuations θ_1, θ_2 over S establishing a total order $\theta_1 \succeq \theta_2$. This implies that

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

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

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

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

Tuple-Based Ranking. Given a query hypergraph $\mathcal{H}$, a tuple-based ranking function assigns for every valuation θ over the variables x_F of relation R_F a weight $w_F(\theta) \in U \subseteq \mathbb{R}$. Then, it takes the following form: $\texttt{rank}(\theta) := \bigoplus_{F \in \mathcal{E}} w_F(\theta[x_F])$ where $\langle U, \oplus \rangle$ forms a *commutative monoid*. In other words, a tuple-based ranking function assigns a weight to each input tuple, and then combines the weights through $\oplus$.

Lemma 20. *Let* ***rank*** *be a monotone tuple-based ranking function over $\mathcal{V}$. Then,* ***rank*** *is compatible with any tree decomposition of a hypergraph with vertices $\mathcal{V}$.*

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

Proposition 20. *Let Q be a natural join query with optimal fractional hypertree-width* ***fhw****. Let* ***rank*** *be a ranking function that can be either (i) monotone vertex-based, (ii) monotone tuple-based. Then, for any input D, we can pre-process D in time $T_p = O(|D|^{\textbf{fhw}})$ and space $S_p = O(|D|^{\textbf{fhw}})$ such that for any k, we can enumerate the top-k results of $Q(D)$ with $\delta = O(\log |D|)$ and $S_e = O(\min\{k, |Q(D)|\})$*

For instance, if the query is acyclic, hence $\texttt{fhw} = 1$, the above theorem gives an algorithm with linear preprocessing time $O(|D|)$ and $O(\log |D|)$ delay.

Lexicographic Ranking. A typical ordering of the output valuations is according to a *lexicographic order*. In this case, each $\mathbf{dom}(x)$ is equipped with a total order. If $\mathcal{V} = \{x_1, \ldots, x_k\}$, a lexicographic order $\langle x_{i_1}, \ldots, x_{i_\ell} \rangle$ for $\ell \leq k$ means that two valuations θ_1, θ_2 are first ranked on x_{i_1}, and if they have the same rank on x_{i_1}, then they are ranked on x_{i_2}, and so on. This ordering can be naturally encoded by first taking a function $f_x : \mathbf{dom}(x) \to \mathbb{R}$ that captures the total order for variable x, and then defining $\texttt{rank}(\theta) := \sum_x w_x f_x(\theta(x))$, where w_x are appropriately chosen constants. Since this ranking function is a monotone vertex-based ranking, Proposition 20 applies here as well.

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

Proposition 21 (due to [OZ15b, BOZ12])**.** *Let Q be a natural join query with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, and $\langle x_{i_1}, \ldots, x_{i_\ell} \rangle$ a lexicographic ordering of the variables in $\mathcal{V}$. Let $\mathcal{T}$ be a tree decomposition with fractional hypertree-width $\texttt{fhw-lex}$ such that $\langle x_{i_1}, \ldots, x_{i_\ell} \rangle$ forms a prefix in the topological ordering of the variables in the decomposition. Then, for any input database D, we can pre-process D with $T_p = O(|D|^{\texttt{fhw-lex}})$ and $S_p = O(|D|^{\texttt{fhw-lex}})$ such that results of $Q(D)$ can be enumerated with delay $\delta = O(1)$ and space $S_e = O(1)$.*

In other words, if the lexicographic order "agrees" with the tree decomposition (in the sense that whenever x_i is before x_j in the lexicographic order, x_j can never be in a bag higher than the bag where x_i is), then it is possible to get an even better result than Theorem 19, by achieving constant delay $O(1)$, and constant space S_e. However, given a tree decomposition, Theorem 19 applies for any lexicographic ordering - in contrast to Theorem 21. As an example, consider the join query $Q(x, y, z) = R(x, y), S(y, z)$ and the lexicographic ordering $\langle z, x, y \rangle$. Since $\texttt{fhw} = 1$, our result implies that we can achieve $O(|D|)$ time preprocessing with delay $O(\log |D|)$. On the other hand, the optimal width of a tree decomposition that agrees with $\langle z, x, y \rangle$ is $\texttt{fhw-lex} = 2$; hence, Theorem 21 implies $O(|D|^2)$ preprocessing time and space. Thus, variable orderings in a decomposition fail to capture the additional challenge of user-chosen lexicographic orderings. It is also not clear whether further restrictions on variable orderings in Theorem 21 are sufficient to capture ordered enumeration for other ranking functions (such as sum).

Bounded Ranking. A ranking function is *c-bounded* if there exists a subset $S \subseteq \mathcal{V}$ of size $|S| = c$, such that the value of $\texttt{rank}$ depends only on the variables from S. A c-bounded ranking is related to c-determined ranking functions [KS06]: c-determined implies c-bounded, but not vice versa. For c-bounded ranking functions, we can show the following result:

Proposition 22. *Let Q be a natural join query with optimal fractional hypertree-width* `fhw`*. If* `rank` *is a c-bounded ranking function, then for any input D, we can pre-process D in time $T_p = O(|D|^{\texttt{fhw}+c})$ and space $S_p = O(|D|^{\texttt{fhw}+c})$ such that for any k, we can enumerate the top-k results of $Q(D)$ with $\delta = O(\log |D|)$ and $S_e = O(\min\{k, |Q(D)|\})$*

Proof. Let $\mathcal{T}$ by the optimal decomposition of Q with fractional hypertree-width `fhw`. We create a new decomposition $\mathcal{T}'$ by simply adding the variables S that determine the ranking functions in all the bags of $\mathcal{T}$. By doing this, the width of the decomposition will grow by at most an additive factor of c. To complete the proof, we need to show that `rank` is compatible with the new decomposition.

Indeed, for any node in $\mathcal{T}'$ (with the exception of the root node) we have that $S \subseteq \texttt{key}(t)$. Hence, if we fix a valuation over $\texttt{key}(t)$, the ranking function will output the same score, independent of what values the other variables take. □

7.3.2 The Algorithm for the Main Theorem

At a high level, each node t in the tree decomposition will materialize incrementally all valuations over $\mathcal{B}_t^{\prec}$ that satisfies the query that corresponds to the subtree rooted at t. We do not store explicitly each valuation θ over $\mathcal{B}_t^{\prec}$ at every node t, but instead we use a simple recursive structure $C(v)$ that we call a *cell*. If t is a leaf, then $C(\theta) = \langle \theta, [], \perp \rangle$, where $\perp$ is used to denote a null pointer. Otherwise, suppose that t has n children $t_1, \ldots, t_n$. Then, $C(\theta) = \langle \theta[\mathcal{B}_t], [p_1, \ldots, p_n], q \rangle$, where p_i is a pointer to the cell $C(\theta[\mathcal{B}_{t_i}^{\prec}])$ stored at node t_i, and q is a pointer to a cell stored at node t (intuitively representing the "next" valuation in the order). It is easy to see that, given a cell $C(\theta)$, one can reconstruct θ in constant time (dependent only on the query). Additionally, each node t maintains one hash map $\mathfrak{Q}_t$, which maps each valuation u over $\texttt{key}(\mathcal{B}_t)$ to a *priority queue* $\mathfrak{Q}_t[u]$. The elements of $\mathfrak{Q}_t$ are cells $C(\theta)$, where θ is a valuation over $\mathcal{B}_t^{\prec}$ such that $u = \theta[\texttt{key}(\mathcal{B}_t)]$. The priority queues will be the data structure that performs the comparison and ordering between different tuples. We will use an implementation of a priority queue (e.g., a Fibonacci heap [CLRS09b]) with the following properties: (i) we can insert an element in constant time $O(1)$, (ii) we can obtain the min element (top) in time $O(1)$, and (iii) we can delete the min element (pop) in time $O(\log n)$.

Notice that it is not straightforward to rank the cells according to the valuationsbottom-upsince the ranking function is defined over all variables $\mathcal{V}$. However, here we can use the fact that the ranking function is compatible with the decomposition at hand. For each variable $x \in \mathcal{V}$, we designate some value from $\mathbf{dom}(x)$ as $v^\star(x)$. Given a fixed valuation u over $\texttt{key}(\mathcal{B}_t)$, we will order the valuations θ over $\mathcal{B}_t^{\prec}$ that agree with u according to the score: $\texttt{rank}(v_t^\star \circ \theta)$ where $v_t^\star = (v^\star(x_1), v^\star(x_2), \ldots, v^\star(x_p))$ is a valuation over the variables $S = \mathcal{V} \setminus \mathcal{B}_t^{\prec}$ with size p. The key intuition is that the compatibility of the ranking function

with the decomposition implies that the ordering of the tuples implied by the cells in the priority queue $\mathfrak{Q}_t[u]$ will not change if we replace $v_t^\star$ with any other valuation. Thus, the comparator can use $v_t^\star$ to calculate the score which is used by the priority queue internally. We next discuss the *preprocessing* and *enumeration* phase of the algorithm.

Algorithm 12: Preprocessing Phase

```
1  foreach t ∈ V(T) do
2  |  materialize the bag B_t
3  full reducer pass on materialized bags in T
4  forall t ∈ V(T) in post-order traversal do
5  |  foreach valuation θ in bag B_t do
6  |  |  u ← θ[key(B_t)]
7  |  |  if Q_t[u] is NULL then
8  |  |  |  Q_t[u] ← new priority queue          /* ranking function uses v_t^⋆ */
9  |  |  ℓ ← []
   |  |  /* ℓ is a list of pointers                                              */
10 |  |  foreach child s of t do
11 |  |  |  ℓ.INSERT(Q_s[θ[key(B_s)]].TOP())
12 |  |  Q_t[u].INSERT(⟨θ, ℓ, ⊥⟩) /* ranking function uses θ, ℓ, v_t^⋆ to calculate score used
   |  |     by priority queue                                                    */
```

Preprocessing. Algorithm 12 consists of two steps. The first step works exactly as in the case where there is no ranking function: each bag $\mathcal{B}_t$ is computed and materialized, and then we apply a full reducer pass to remove all tuples from the materialized bags that will not join in the final result. The second step initializes the hash map with the priority queues for every bag in the tree. We traverse the decomposition in a bottom up fashion (post-order traversal), and do the following. For a leaf node t, notice that the algorithm does not enter the loop in line 10, so each valuation θ over $\mathcal{B}_t$ is added to the corresponding queue as the triple $\langle \theta, [], \bot \rangle$. For each non-leaf node t, we take each valuation v over $\mathcal{B}_t$ and form a valuation (in the form of a cell) over $\mathcal{B}_t^\prec$ by using the valuations with the largest rank from its children (we do this by accessing the top of the corresponding queues in line 10). The cell is then added to the corresponding priority queue of the bag. Observe that the root node r has only one priority queue, since $\texttt{key}(r) = \{\}$.

Example 38. *As a running example, we consider the natural join query* $Q(x, y, z, p) = R_1(x, y), R_2(y, z), R_3(z, p), R_4(z, u)$ *where the ranking function is the sum of the weights of each input tuple. Consider the following instance* D *and decomposition* $\mathcal{T}$ *for our running example.*

R_1

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

R_2

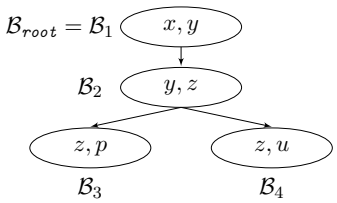
id	$\mathbf{w_2}$	**y**	**z**
1	*1*	*1*	*1*
2	*1*	*3*	*1*

R_3

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

R_4

id	$\mathbf{w_3}$	**z**	**u**
1	*1*	*1*	*1*
2	*5*	*1*	*2*



For the instance shown above and the query decomposition that we have fixed, relation R_i covers bag $\mathcal{B}_i, i \in [4]$. Each relation has size $N = 2$. Since the relations are already materialized, we only need to perform a full reducer pass, which can be done in linear time. This step removes tuple $(3, 1)$ from relation R_2 as it does not join with any tuple in R_1.

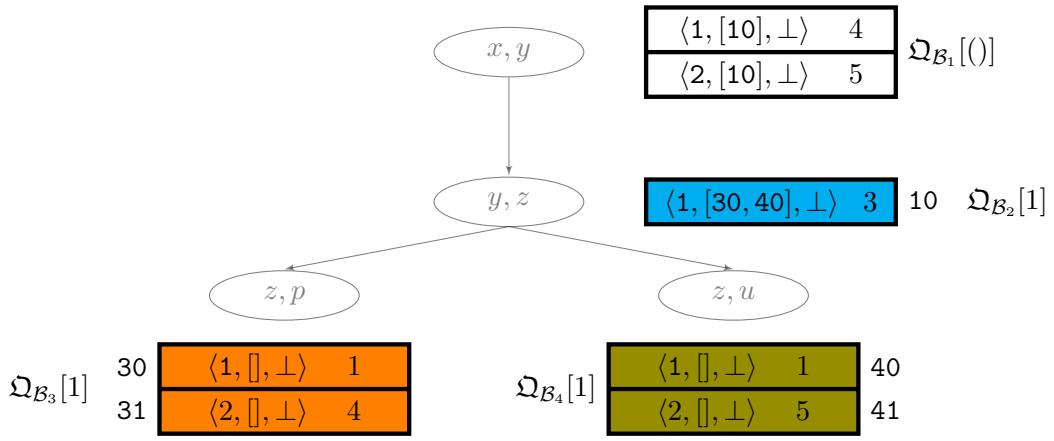
Figure 7.1a shows the state of priority queues after the pre-processing step. For convenience, θ in each cell $\langle \theta, [p_1, \ldots, p_k], \mathsf{next} \rangle$ is shown using the primary key of the tuple and pointers p_i and next *are shown using the address of the cell it points to. The cell in a memory location is followed by the partial aggregated score of the tuple formed by creating the tuple from the pointers in the cell recursively. For instance, the score of the tuple formed by joining $(y = 1, z = 1) \in R_2$ with $(z = 1, p = 1)$ from R_3 and $(z = 1, u = 1)$ in R_4 is $1 + 1 + 1 = 3$[1] (shown as* ⟨1, [30, 40], ⊥⟩ 3 *in the figure). The pointer addresses* 30 *and* 40 *refer to the topmost cell in the priority queue for $\mathcal{B}_3$ and $\mathcal{B}_4$. Each cell in every priority queue points to the top element of the priority queue of child nodes that it joins with. Note that since both tuples in R_1 join with the sole tuple from R_2, they point to the same cell.*

Lemma 21. *The runtime of Algorithm 12 is $O(|D|^{fhw})$. Moreover, at the end of the algorithm, the resulting data structure has size $O(|D|^{fhw})$.*

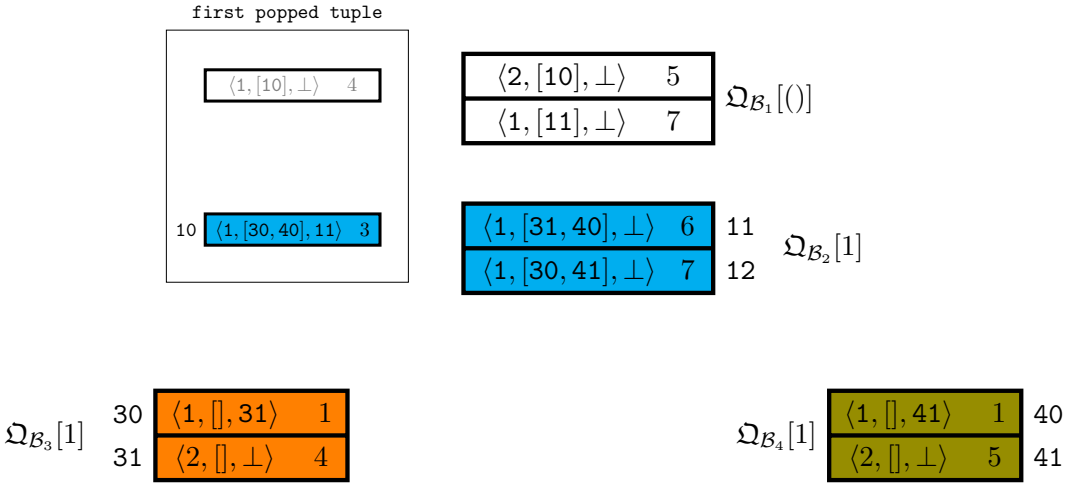
Proof. It is known that the materialization of each bag can be done in time $O(|D|^{\mathtt{fhw}})$, and the full reducer pass is linear in the size of the bags [Yan81]. For the second step of the preprocessing algorithm, observe that for each valuation in a bag, the algorithm performs only a constant number of operations (the number of children in the tree plus one), where each operation takes a constant time (since insert and top can be done in $O(1)$ time for the priority queue). Hence, the second step needs $O(|D|^{\mathtt{fhw}})$ time as well.

Regarding the space requirements, it is easy to see that the data structure uses only constant space for every valuation in each bag, hence the space is bounded by $O(|D|^{\mathtt{fhw}})$. □

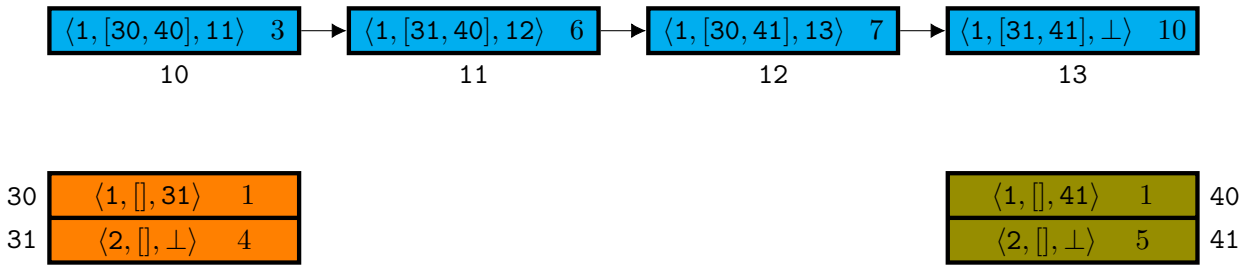
[1]Note that since our ranking function is sum, we use $v^\star(x) = 0$ for each variable. This allows us to only look at the partial score of tuples that join in a particular subtree.



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



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



(c) The materialized output stored at subtree rooted at $\mathcal{B}_2$ after enumeration is complete (`OUT`$(\mathcal{B}_2^{\prec})$)

Figure 7.1: Preprocessing and enumeration phase for Example 7. Each cell is assigned a memory addressed (written next to the cell). Pointers in cells are populated with the memory address of the cell they are pointing to. Cells are color coded according to the bag (white for root bag, blue for $\mathcal{B}_2$, orange for $\mathcal{B}_3$ and olive for $\mathcal{B}_4$.)

Algorithm 13: Enumeration Phase

```
1  procedure ENUM()
2      while 𝔒_r[()] is not empty do
3          output 𝔒_r[()].TOP()
4          TOPDOWN(𝔒_r[()].TOP(), r)
5  procedure TOPDOWN(c, t)
6      /* c = ⟨θ, [p_1, ..., p_k], next⟩ */
7      u ← θ[key(B_t)]
8      if next == ⊥ then
9          b ← 𝔒_t[u].POP()
10         while b == 𝔒_t[u].TOP() do
11             𝔒_t[u].POP()                    /* remove duplicate cells (at most a constant) */
12         foreach child t_i of t do
13             p'_i ←TOPDOWN(*p_i, t_i)
14             if p'_i ≠ ⊥ then
15                 𝔒_t[u].INSERT(⟨θ, [p_1, ..., p'_i, ... p_k], ⊥⟩)   /* insert new candidate(s);
                       priority queue uses ranking function and θ, ℓ, v*_t to calculate the score */
16         if t is not the root then
17             next ← ADDRESS_OF(𝔒_t[u].TOP())
18     return next
```

Enumeration. Algorithm 13 presents the algorithm for the enumeration phase. The heart of the algorithm is the procedure TOPDOWN(c, t). The key idea of the procedure is that whenever we want to output a new tuple, we can simply obtain it from the top of the priority queue in the root node (node r is the root node of the tree decomposition). Once we do that, we need to update the priority queue by popping the top, and inserting (if necessary) new valuations in the priority queue. This will be recursively propagated in the tree until it reaches the leaf nodes. Observe that as the new candidates are being inserted, the next pointer of cells c at some node of the decomposition are being updated by pointing to the topmost element in the priority queue of its children. This chaining materializes the answers for the particular bag that can be reused.

Example 39. *Figure 7.1b shows the state of the data structure after one iteration in* ENUM(). *The first answer returned to the user is the topmost tuple from* $\mathfrak{O}_{\mathcal{B}_1}[()]$ *(shown in top left of the figure). Cell* ⟨1, [10], ⊥⟩ 4 *is popped from* $\mathfrak{O}_{\mathcal{B}_1}[()]$ *(after satisfying if condition on line 5 since* next *is* ⊥*). We recursively call* TOPDOWN *for child node* $\mathcal{B}_2$ *and cell* ⟨1, [30, 40], ⊥⟩ 3 *as the function argument (since that is the cell at memory address* 10*). The* next *for this cell*

is also $\perp$ *and we pop it from* $\mathfrak{Q}_{\mathcal{B}_2}[1]$. *At this point,* $\mathfrak{Q}_{\mathcal{B}_2}[1]$ *is empty. The next recursive call is for* $\mathcal{B}_3$ *with* $\langle 1, [], \perp \rangle \quad 1$ *(cell at memory adress* 30*). The least ranked tuple but larger than* $\langle 1, [], \perp \rangle \quad 1$ *in* $\mathfrak{Q}_{\mathcal{B}_3}[1]$ *is the cell at address* 31 . *Thus,* next *for* $\langle 1, [], \perp \rangle \quad 1$ *is updated to* 31 *and cell at memory address* 31 *(* $\langle 2, [], \perp \rangle \quad 4$ *) is returned which leads to creation and insertion of* $\langle 1, [31, 40], \perp \rangle \quad 6$ *cell in* $\mathfrak{Q}_{\mathcal{B}_2}[1]$. *Similarly, we get the other cell in* $\mathfrak{Q}_{\mathcal{B}_2}[1]$ *by recursive call for* $\mathcal{B}_4$. *After both the calls are over for node* $\mathcal{B}_2$, *the topmost cell at* $\mathfrak{Q}_{\mathcal{B}_2}[1]$ *is cell at memory address* 11 *,which is set as the* next *for* $\langle 1, [30, 40], \perp \rangle \quad 3$ *(changing into* $\langle 1, [30, 40], 11 \rangle \quad 3$ *), terminating one full iteration.*

Let us now look at the second iteration of ENUM(). *The tuple returned is top element of* $Q_{\mathcal{B}_1}[()]$ *which is* $\langle 2, [10], \perp \rangle \quad 5$. *However, the function* TOPDOWN() *with* $\langle 2, [10], \perp \rangle \quad 5$ *does not recursively go all the way down to leaf nodes. Since* $\langle 1, [30, 40], 11 \rangle \quad 3$ *already has* next *populated, we insert* $\langle 2, [11], \perp \rangle \quad 5$ *in* $Q_{\mathcal{B}_1}[()]$ *completing the iteration. This demonstrates the benefit of materializing ranked answers at each node in the tree. As the enumeration continues, we are materializing the output of each subtree on-the-fly that can be reused by other tuples in the root bag.*

New candidates are inserted into the priority queue using the logic on line 15 of Algorithm 13. Given a bag s with k children $s_1, \ldots, s_k$ and a cell c, the algorithm increments the pointers $p_1, \ldots p_k$ one at a time while keeping the remaining pointers fixed. Observe that for the tuple $c.\theta$, the candidate generation logic will enumerate the cartesian product $\times_{i \in [k]}(\sigma_{\texttt{key}(\mathcal{B}_{s_i})=c.\theta[\texttt{key}(\mathcal{B}_{s_i})]}\texttt{OUT}(\mathcal{B}_{s_i}^{\prec}))$. Here, $\texttt{OUT}(\mathcal{B}_j^{\prec})$ is the ranked materialized output of subtree rooted at $\mathcal{B}_j$, and the selection condition filters only those tuples that agree with $c.\theta$ on the key variables of the children bags. Indeed as Figure 7.1b shows, initially, only $\langle 1, [30, 40], \perp \rangle \quad 3$ was present in $\mathfrak{Q}_{\mathcal{B}_2}[1]$ but it generated two cells $\langle 1, [31, 40], \perp \rangle \quad 6$ and $\langle 1, [30, 41], \perp \rangle \quad 7$. When these two cells are popped, they will increment pointers and both of them will generate $\langle 1, [31, 41], \perp \rangle \quad 10$. Thus, cartesian product of R_3 and R_4 is enumerated. This also shows that while each cell can generate k new candidates, in the worst-case, each cell may also be generated k times and inserted into the priority queue. These duplicates are removed by the procedure in the while loop on line 11[2]. Since the query size is a constant, we may only need to pop a constant number of times in the worst-case and thus affects the delay guarantee only by a constant factor.

Lemma 22. *Algorithm 13 enumerates* $Q(D)$ *with delay* $\delta = O(\log |D|)$.

Proof. To show the delay guarantee, it suffices to prove that procedure TOPDOWN takes $O(\log |D|)$ time when called from the root node, since getting the top element from the priority queue at the root node takes only $O(1)$ time.

[2]One could also avoid adding duplicates by storing all inserted tuples in a parallel hash set and checking against it before insertion.

Indeed, TOPDOWN traverses the tree decomposition recursively. The key observation is that it visits each node in $\mathcal{T}$ exactly once. For each node, if next is not $\perp$, the processing takes time $O(1)$. If next $== \perp$, it will perform a constant number of pops – with cost $O(\log |D|)$ – and a number of inserts equal to the number of children. Thus, in either case the total time per node is $O(\log |D|)$. Summing up over all nodes in the tree, the total time until the next element is output will be $O(\log |D|) \cdot |V(\mathcal{T})| = O(\log |D|)$. □

We next bound the space S_e needed by the algorithm during the enumeration phase.

Lemma 23. *After Algorithm 13 has enumerated k tuples, the additional space used by the algorithm is $S_e = O(\min\{k, |Q(D)|\})$.*

Proof. The space requirement of the algorithm during enumeration comes from the size of the priority queues at every bag in the decomposition. Since we have performed a full reducer pass over all bags during the preprocessing phase, and each bag t stores in its priority queues all valuations over $\mathcal{B}_t^{\prec}$, it is straightforward to see that the sum of the sizes of the priorities queues in each bag is bounded by $O(|Q(D)|)$.

To obtain the bound of $O(k)$, we observe that for each tuple that we output, the TOPDOWN procedure adds at every node in the decomposition a constant number of new tuples in one of the priority queues in this node (equal to the number of children). Hence, at most $O(1)$ amount of data will be added in the data structure between two consecutive tuples are output. Thus, if we enumerate k tuples from $Q(D)$, the increase in space will be $k \cdot O(1) = O(k)$. □

Chaining of cells. Observe that as TOPDOWN is called recursively, the next is continuously being updated. This chaining is critical to achieving good delay guarantees. Intuitively, chaining of cells at a bag allows materialization of the join result of the subquery rooted at that bag in sorted order (previously referred to as $\texttt{OUT}(\mathcal{B}_j^{\prec})$). This implies that repeated computation is not being performed and cells at the parent of a bag can re-use the sorted materialization. For example, figure 7.1c shows the eventual sequence of pointers at node $\mathcal{B}_2$ which is the ranked materialized output of the subtree rooted at $\mathcal{B}_2$. The pointers between cells are added to emphasize the chained order. The reader can observe that the score for the cells highlighted in blue is also in increasing order.

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

Lemma 24. *Algorithm 13 enumerates $Q(D)$ in order according to* rank.

Proof. We will prove our claim by induction on post-order traversal of the decomposition and use the compatibility property of the ranking function with the decomposition at hand. We will show that the priority queue for each node s gives the output in the correct order which in turn populates $\texttt{OUT}(\mathcal{B}_s^{\prec})$ correctly. Recall that $\texttt{OUT}(\mathcal{B}_s^{\prec})$ is the ranked materialized

output of subtree rooted at $\mathcal{B}_s$. Since $\texttt{OUT}(\mathcal{B}_s^{\prec})$ is a linked list, we will frequently use the notation $\texttt{OUT}(\mathcal{B}_s^{\prec})[\ell]$ to find the cell at ℓ^{th} location in the list. Given a cell c created at node s we will use $\texttt{output}(c)$ to denote the tuple formed over $\mathcal{B}_{\mathcal{B}_s}^{\prec}$ by traversing all the pointers recursively and finding all valuations for the variables.

Base Case. Correctness for ranked output of $\texttt{OUT}(\mathcal{B}_s)$ for leaf node s is trivial as the leaf node tuples are already materialized and do not require any join processing. Let $\phi^\star$ be the valuation over $\mathcal{V} \setminus \mathcal{B}_s$ according to definition of decomposability. We insert each valuation θ over node s with score $\texttt{rank}(\phi^\star \circ \theta)$. Since $\texttt{rank}$ is compatible with the decomposition, it follows that if $\texttt{rank}(\phi^\star \circ \theta_2) \geq \texttt{rank}(\phi^\star \circ \theta_1)$ such that $\theta_1[\texttt{key}(s)] = \theta_1[\texttt{key}(s)]$, then $\theta_2 \succeq \theta_1$, thus recovering the correct ordering for tuple in s. Since leaf nodes are the base relations and each tuple is inserted into the priority queue, the pop operation will arrange them in order and also chain the materialization correctly.

Inductive Case. Consider some node s in post-order traversal with children $s_1, \ldots s_m$. By the induction hypothesis, the ordering of $\texttt{OUT}(\mathcal{B}_{s_i}^{\prec})$ for each value of $\texttt{key}(s_i)$ is correctly generated. Let θ be a fixed tuple materialized in the relation at node s and let $u = \theta[\mathcal{B}_s]$. Observe that the preprocessing phase creates a cell for θ whose pointer list $[p_1, \ldots p_m]$ is the cell at location 0 of the materialized list of all m children $\texttt{OUT}(\mathcal{B}_{s_i}^{\prec})$. We claim that this is the least ranked tuple that can be formed over $\mathcal{B}_s^{\prec}$. Indeed, if any pointer p_i points to any other cell (say c_i') at some location greater than 0 in the list $\texttt{OUT}(\mathcal{B}_{s_i}^{\prec})$, we can create a smaller ranked tuple by changing p_i to point to the first cell in the list (denoted c_i^0). This is directly a consequence of the compatibility of the ranking function and the induction hypothesis which tells us that $\texttt{output}(c_i^0) \preceq \texttt{output}(c_i')$ since $\texttt{OUT}(\mathcal{B}_{s_i})$ is generated correctly. Let $\phi^\star$ be a valuation over $\mathcal{V} \setminus \mathcal{B}_s^{\prec}$. Then,

$$\texttt{rank}(\theta \circ \phi^\star \circ \texttt{output}(c_1') \circ \ldots \texttt{output}(c_i') \cdots \circ \texttt{output}(c_m')) \geq$$
$$\texttt{rank}(\theta \circ \phi^\star \circ \texttt{output}(c_1') \circ \ldots \texttt{output}(c_i^0) \cdots \circ \texttt{output}(c_m'))$$

because $\texttt{rank}(\texttt{output}(c_i')) \geq \texttt{rank}(\texttt{output}(c_i^0))$ and $\texttt{rank}$ is $\mathcal{B}_{s_i}^{\prec} \setminus \texttt{key}(s_i)$-decomposable conditioned on $\texttt{key}(s_i)$. Note that no sibling of s_i has any common variables other than the key variables which have already been fixed. This proves that the first tuple returned by the priority queue at node s will be correct.

Next, we proceed to show that the correctness for an arbitrary step in the execution. Suppose c is the last cell popped at line 9. From line 12-14, one may observe that a new candidate is pushed into priority queue for key by incrementing pointers to $\texttt{OUT}(\mathcal{B}_{s_i}^{\prec})$ one at a time for each child bag $\mathcal{B}_{s_i}$, while keeping the remainder of tuple (including $\texttt{key}(s_i)$) fixed (line 14). Let $c.p_i$ point to cell $\texttt{OUT}(\mathcal{B}_{s_i}^{\prec})[\ell_i]$ which will be denoted by c_i. Then, the m candidates generated by the logic will contain pointers that point to the following locations

$$\begin{aligned}\mathcal{L} = \ell_1 + 1, \ell_2, \ell_3, \ldots, \ell_m,\\ \ell_1, \ell_2 + 1, \ell_3, \ldots, \ell_m,\\ \ell_1, \ell_2, \ell_3 + 1, \ldots, \ell_m,\\ \ldots\\ \ell_1, \ell_2, \ell_3, \ldots, \ell_m + 1\end{aligned}$$

Suppose that the next smallest cell that must be popped is $c^{\succ}$ and has pointer list that points to $\ell_i^{\succ}$ location in $\texttt{OUT}(\mathcal{B}_{s_i}^{\prec})$. We need to show that $c^{\succ}$ is one of the cells with pointer locations as shown in $\mathcal{L}$ or is already in the priority queue. For the sake of contradiction, suppose there is a cell c' with $c'.p_i = \mathsf{address\_of}(\texttt{OUT}(\mathcal{B}_{s_i}^{\prec})[\ell_i'])$, $i \in [m]$ that is the next smallest but is neither in $\mathcal{L}$, nor present in the priority queue. In other words, $\texttt{rank}_\phi(\mathsf{output}(c')) < \texttt{rank}_\phi(\mathsf{output}(c^{\succ}))$ for any valuation ϕ over $\mathcal{V} \setminus \mathcal{B}_s^{\prec}$. We will show that such a scenario will violate compatibility of ranking function. There are three possible scenarios regarding domination of $\ell_i^{\succ}$ and ℓ_i'.

1. $\ell_i^{\succ} < \ell_i'$ for each $i \in [m]$. This scenario implies that $\texttt{rank}(\phi \circ \mathsf{output}(c^{\succ})) \leq \texttt{rank}(\phi \circ \mathsf{output}(c'))$. Indeed, we have that

$$\begin{aligned}&\texttt{rank}(\phi \circ \theta \circ \mathsf{output}(c_1^{\succ}) \circ \ldots \mathsf{output}(c_m^{\succ}))\\ &\qquad\leq \texttt{rank}(\phi \circ \theta \circ \mathsf{output}(c_1') \circ \mathsf{output}(c_2^{\succ}) \circ \ldots \mathsf{output}(c_m^{\succ}))\\ &\qquad\leq \texttt{rank}(\phi \circ \theta \circ \mathsf{output}(c_1') \circ \mathsf{output}(c_2') \circ \ldots \mathsf{output}(c_m^{\succ}))\\ &\qquad\ldots\\ &\qquad\leq \texttt{rank}(\phi \circ \theta \circ \mathsf{output}(c_1') \circ \mathsf{output}(c_2') \circ \ldots \mathsf{output}(c_m'))\end{aligned}$$

   Each inequality is a successive application of $(\mathcal{B}_{s_i}^{\prec} \setminus \texttt{key}(s_i))$-decomposability since $\mathsf{output}(c_i') \succeq \mathsf{output}(c_i^{\succ})$ from the ordering correctness of each bag s_i.

2. $\ell_i^{\succ} > \ell_i'$ for each $i \in [m]$. This scenario would mean that c' was generated before c given our candidate generation logic (line 9-12), violating our assumption that c' has not been enumerated yet or inserted into the priority queue. Observe that each candidate in $\mathcal{L}$ dominates the pointer locations of the cell that generated them. Thus, successive applications of the logic on c' will generate $c^{\succ}$ eventually. Since the preprocessing phase creates a cell with pointer locations 0, generating $c^{\succ}$ must happen

after c' because the pointers are advanced one at a time and thus cannot go directly from $\ell'_i - 1$ to $\ell'_i + 1$ (thus skipping ℓ'_i) without violating the correctness of priority queue.

3. ℓ'_i and ℓ are incomparable. It is easy to see that all candidates in $\mathcal{L}$ dominate pointer locations of c but are incomparable to each other. Also, the only way to generate new candidate tuples is line 12-14. Thus, if c' is not in the priority queue, there are two possibilities. Either there is some cell c'' in the priority queue that is dominated by c' and thus, $\texttt{rank}_\varphi(\texttt{output}(c'')) \leq \texttt{rank}_\varphi(\texttt{output}(c'))$. c'' will eventually generate c' via a chain of cells that successively dominate each other. As c was popped before c'', it follows that $\texttt{rank}_\varphi(\texttt{output}(c)) \leq \texttt{rank}_\varphi(\texttt{output}(c'')) \leq \texttt{rank}_\varphi(\texttt{output}(c'))$, a contradiction to our assumption that c' has a smaller rank than c. The second possibility is that there is no such c'', which will mean that c and c' are generated in the same for loop line 12. But this would again mean that c' is in the priority queue. Thus, both these cases violate one of our assumptions made. Since the priority queue in the preprocessing phase is initialize with a cell with pointers to location 0 of all $\texttt{OUT}(\mathcal{B}^{\prec}_{s_i})$, every answer tuple in the cartesian product $\times_{i\in[m]}(\sigma_{\texttt{key}(\mathcal{B}_{s_i})=c.\theta[\texttt{key}(\mathcal{B}_{s_i})]}\texttt{OUT}(\mathcal{B}^{\prec}_{s_i}))$ will be enumerated by successive pointer increments.

Therefore, it cannot be the case that $\texttt{rank}_\varphi(\texttt{output}(c')) < \texttt{rank}_\varphi(\texttt{output}(c))$ which proves the ordering correctness for node s. Since the output $\texttt{OUT}(\mathcal{B}^{\prec}_{s})$ is populated using this ordering form priority queue, it is also materialized (chaining of cells at line 17) in ranked order.

□

7.4 Extensions

In this section, we describe two extensions of Theorem 19 and how it can be used to further improve the main result.

7.4.1 Ranked Enumeration of UCQs

We begin by discussing how ranked enumeration of full UCQs can be done. The first observation is that given a full UCQ $\varphi = \varphi_1 \cup \ldots \varphi_\ell$, if the ranked enumeration of each φ_i can be performed efficiently, then we can perform ranked enumeration for the union of query results. This can be achieved by applying Theorem 19 to each φ_i and introducing another priority queue that compares the score of the answer tuples of each φ_i, pops the smallest result, and fetches the next smallest tuple from the data structure of φ_i accordingly. Although each $\varphi_i(D)$ does not contain duplicates, it may be the case that the same tuple is generated by multiple φ_i. Thus, we need to introduce a mechanism to ensure that all tuples with the same weight are enumerated in a specific order. Fortunately, this is easy

to accomplish by modifying Algorithm 13 to enumerate all tuples with the same score in lexicographic increasing order. The choice of lexicographic ordering as a tie-breaking criterionThe comparison is not the only valid choice. As long as the ties are broken consistently, other ranking functions can also be used. This ensures that tuples from each φ_i also arrive in the same order. Since each φ_i is enumerable in ranked order with delay $O(\log |D|)$ and the overhead of the priority queue is $O(\ell)$ (priority queue contains at most one tuple from each φ_i), the total delay guarantee is bounded by $O(\ell \cdot \log |D|) = O(\log |D|)$ as the query size is a constant. The space usage is determined by the largest fractional hypertree-width across all decompositions of subqueries in φ. This immediately leads to the following result.

Theorem 20. *Let $\varphi = \varphi_1 \cup \ldots \varphi_\ell$ be a full UCQ. Let $\mathtt{fhw}$ denote the fractional hypertree-width of all decompositions across all CQs φ_i, and $\mathtt{rank}$ be a ranking function that is compatible with the decomposition of each φ_i. Then, for any input database D, we can pre-process D in time and space,*

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

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

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

Algorithm 14 shows the enumeration algorithm. It outputs one output tuple t in every iteration and line 12-line 13 pop out all duplicates of t in the queue. Recall that since $Q = Q_1 \cup \ldots Q_\ell$, there can be at most ℓ duplicates for some constant ℓ. $\texttt{ENUM}_i()$ is the invocation of $\texttt{ENUM}()$ procedure from Algorithm 13 on query Q_i.

The comparison function for priority queues in Algorithm 13 for each subquery Q_i of Q is modified in the following way. Consider two tuples t_1 and t_2 with schema $(x_1, x_2, \ldots, x_n)$ and scores $\texttt{rank}(t_1)$ and $\texttt{rank}(t_2)$ respectively.

Comparison function in Algorithm 15 compares t_1 and t_2 based on the ranking function and tie breaks by using the lexicographic ordering of the two tuples. This ensures that all tuples with the same score arrive in a fixed from $\texttt{ENUM}_i()$ procedure of each subquery Q_i.

7.4.2 Improving The Main Result

Although Theorem 20 is a straightforward extension of Theorem 19, it is powerful enough to improve the pre-processing time and space of Theorem 19 by using Theorem 20 in conjunction with *data-dependent* tree decompositions. It is well known that the query result for any CQ can be answered in time $O(|D|^{\mathtt{fhw}} + |Q(D)|)$ time and this is asymptotically tight [AGM13]. However, there exists another notion of width known as the *submodular width* (denoted $\mathtt{subw}$) [Mar13]. It is also known that for any CQ, it holds that $\mathtt{subw} \leq \mathtt{fhw}$. Recent work by Abo Khamis et al. [AKNS17] presented an elegant algorithm called PANDA

Algorithm 14: Preprocessing and Enumeration Phase

```
1  procedure PREPROCESS
2      Apply Algorithm 12 to all Q_i
3      QUEUE ← ∅
4      for i ∈ {1, ..., ℓ} do
5          QUEUE.PUSH(ENUM_i())  /* Initialize QUEUE with smallest candidate for each Q_i */
6  procedure ENUM()
7      while QUEUE is not empty do
8          t ← QUEUE.POP()
9          output t                                  /* Suppose t came from subquery Q_i */
10         QUEUE.PUSH(ENUM_i())                      /* Push the next candidate for Q_i */
11         while QUEUE.TOP() == t do
12             QUEUE.POP()                           /* drain the queue of duplicate t */
               /* Suppose duplicate t came from subquery Q_j                          */
13             QUEUE.PUSH(ENUM_j())                  /* Push the next candidate for Q_j */
```

Algorithm 15: Comparison function for priority queues

```
   /* Returns the smaller ranked tuple of t_1 and t_2; break ties in lexicographic ordering */
1  procedure COMPARE (t_1, t_2)
2      if rank(t_1) < rank(t_2) then
3          return t_1
4      if rank(t_2) < rank(t_1) then
5          return t_2
6      foreach i ∈ {n, n − 1, ..., 1} do
7          if π_{x_i}(t_1) < π_{x_i}(t_2) then
8              return t_1
9          if π_{x_i}(t_2) < π_{x_i}(t_1) then
10             return t_2
```

that constructs multiple decompositions by partitioning the input database to minimize the intermediate join size result. PANDA computes the output of any full CQ in time $O(|D|^{\texttt{subw}} \cdot \log |D| + |\texttt{OUT}|)$. In other words, PANDA takes a CQ query Q and a database D as input and produces multiple tree decompositions in time $O(|D|^{\texttt{subw}} \cdot \log |D|)$ such that each answer tuple is generated by at least one decomposition. The number of decompositions depends only on the size of the query and not on D. Thus, when the query size is a constant, the number of decompositions constructed is also a constant. We can now apply

Theorem 20 by setting φ_i as the tree decompositions produced by PANDA to get the following result. [DK21] describes the details of the enumeration algorithm and the tie-breaking comparison function.

Theorem 21. *Let φ be a natural join query with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, submodular width* ***subw****, and* ***rank*** *be a ranking function that is compatible with each tree decomposition of φ. Then, for any input database D, we can pre-process D in time and space,*

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

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

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

7.5 Lower Bounds

In this section, we provide evidence for the near optimality of our results.

7.5.1 The Choice of Ranking Function

We first consider the impact of the ranking function on the performance of ranked enumeration. We start with a simple observation that deals with the case where `rank` has no structure and can be accessed only through a black box that, given a tuple/valuation, returns its score: we call this a *black box* [3] ranking function. Note that all of our algorithms work under the black box assumption.

Proposition 23. *Let Q be a natural join query, and* ***rank*** *a black box ranking function. Then, any enumeration algorithm on a database D needs $\Omega(|Q(D)|)$ calls to* ***rank****– and worst case $\Omega(|D|^{\rho^*})$ calls – in order to output the smallest tuple.*

Indeed, if the algorithm does not examine the rank of an output tuple, then we can always assign a value to the ranking function such that the tuple is the smallest one. Hence, in the case where there is no restriction on the ranking function, the simple result in Proposition 19 that materializes and sorts the output is essentially optimal. Thus, it is necessary to exploit the properties of the ranking function to construct better algorithms. Unfortunately, even for natural restrictions of ranking functions, it is not possible to do much better than the $|D|^{\rho^*}$ bound for certain queries.

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

[3] Black box implies that the score $\texttt{rank}(\theta)$ is revealed only upon querying the function.

Lemma 25. *Consider the query* $Q(x_1, y_1, x_2, y_2) = R(x_1, y_1), S(x_2, y_2)$ *and let* `rank` *be a black box coordinate decomposable ranking function. Then, there exists an instance of size* N *such that the time required to find the smallest tuple is* $\Omega(N^2)$.

Proof. We construct an instance D as follows. For every variable we use the domain $\{a_1, \ldots, a_N\}$, which we equip with the order $a_1 < a_2 < \cdots < a_N$. Then, every tuple in R and S is of the form (a_i, a_{N-i+1}) for $i = 1, \ldots, N$. Similarly, every tuple in S is of the form (a_i, a_{N-i+1}). The result of the query Q on D has size N^2.

To construct a family of coordinate decomposable ranking functions, we consider all ranking functions that are monotone w.r.t. the order of the domain $\{a_1, \ldots, a_N\}$ for every variable.

We will show that any two tuples in $Q(D)$ are incomparable, in the sense that neither tuple dominates the other in all variables. Indeed, consider two distinct tuples $t_1 = (a_i, a_{N-i+1}, a_j, a_{N-j+1})$, and $t_2 = (a_k, a_{N-k+1}, a_\ell, a_{N-\ell+1})$. For the sake of contradiction, suppose t_1 dominates t_2. Then, we must have $i \geq k$ and $N - i + 1 \geq N - k + 1$, giving $i = k$. Similarly, $j = \ell$. But this contradicts our assumption that $t_1 \neq t_2$.

Therefore, a ranking function from our family can assign an arbitrary score to the N^2 tuples without violating the coordinate decomposability. This is because coordinate decomposability only imposes a condition on the ranking of tuples where one dominates the other. For any non-dominating tuple pair, the ranking function is free to assign any value as the score. Thus, any algorithm that does not examine all N^2 tuples can miss the smallest, which gives us the desired lower bound. □

Lemma 25 shows that for coordinate decomposable functions, there exist queries where obtaining constant (or almost constant) delay requires the algorithm to spend superlinear time during the preprocessing step. Given this result, the immediate question is to see whether we can extend the lower bound to other CQs.

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

Theorem 22. *Consider a full acyclic query* Q *and a coordinate decomposable black box ranking function. There exists an algorithm that enumerates the result of* Q *in ranked order with* $O(\log |D|)$ *delay guarantee and* $T_p = O(|D|)$ *preprocessing time if and only if there are no atoms* R *and* S *in* Q *such that* $\mathsf{vars}(R) \setminus \mathsf{vars}(S) \geq 2$ *and* $\mathsf{vars}(S) \setminus \mathsf{vars}(R) \geq 2$.

Proof. Suppose that Q does not satisfy the condition of the theorem. Then, there are atoms R and S in Q with $\mathsf{vars}(R) \setminus \mathsf{vars}(S) \geq 2$ and $\mathsf{vars}(S) \setminus \mathsf{vars}(R) \geq 2$. This means that we can apply the same construction as Lemma 25 to show that the preprocessing time must be $\Omega(|D|^2)$ in order to achieve $O(\log |D|)$ delay.

For the other direction, we will construct a compatible decomposition $\mathcal{T}$ for a Q that satisfies the condition. Pick the relation with the largest number of variables as the root of the decomposition. Each remaining relation forms a single bag that is connected to the root.

The main observation is that each child bag of the root contains at most one variable that is not in the root. Indeed, suppose that there exists one child bag with at least two variables not present in the root. But the root bag must also contain at least two variables in addition to the common variables. This would violate the condition of the theorem, which is a contradiction.

Equipped with this observation, we can see that the constructed tree is indeed a valid decomposition of $\texttt{fhw} = 1$. Additionally, we can show the compatibility of $\mathcal{T}$. The root bag is $\mathcal{B}_{\texttt{r}}$-decomposable by definition since $\texttt{key}(\mathcal{B}_{\texttt{r}}) = \{\}$. Let u_i be the unique variable in bag $\mathcal{B}_i$. Then, $\mathcal{T}$ is u_i-decomposable conditioned on $\texttt{key}(\mathcal{B}_i)$. This follows from the definition of coordinate decomposable. Thus, Theorem 19 is applicable. □

For example, the query $Q(x, y, z) = R(x, y), S(y, z)$ satisfies the condition of Theorem 22, while the Cartesian product query defined in Lemma 25 does not.

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

Definition 7. *Given a connected hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, a path P in $\mathcal{H}$ from vertex x_1 to x_{s+1} is a vertex-edge alternate set $x_1E_1x_2E_2 \dots x_sE_sx_{s+1}$ such that $\{x_i, x_{i+1}\} \subseteq E_i (i \in [s])$ and $x_i \neq x_j, E_i \neq E_j$ for $i \neq j$. Here, s is the length of the path P. The distance between any two vertices u and v, denoted $d(u, v)$, is the length of the shortest path connecting u and v. The* diameter *of a hypergraph, $\mathsf{dia}(\mathcal{H})$, is the maximum distance between all pairs of vertices.*

Theorem 23. *Consider a full connected acyclic join query Q and a black box edge decomposable ranking function. Then, there exists an algorithm that enumerates the result of Q in ranked order with $O(\log |D|)$ delay and $T_p = O(|D|)$ preprocessing time if and only if $\mathsf{dia}(Q) \leq 3$.*

Proof. To prove the hardness result, suppose that $\mathsf{dia}(Q) \geq 4$. We construct a database instance D and a family of edge decomposable ranking functions such that any algorithm must make $\Omega(|D|^2)$ calls to the ranking function to output the first tuple in $Q(D)$. Indeed, since $\mathsf{dia}(Q) \geq 4$, there exists a path $x_1R_1y_1S_1zS_2y_2R_2x_2$ in the hypergraph of Q. Since this path cannot be made shorter, this implies that $x_1 \notin S_1, S_2, R_2$; $y_1 \notin S_2, R_2$; $z \notin R_1, R_2$; $y_2 \notin R_1, S_1$ and $x_2 \notin R_1, S_1, S_2$.

Figure 7.3 shows how the construction for the database instance D; all variables not depicted take the same constant value. For the family of ranking functions, we consider all

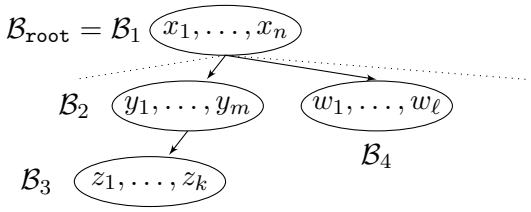


Figure 7.2: Query decomposition example with depth 2

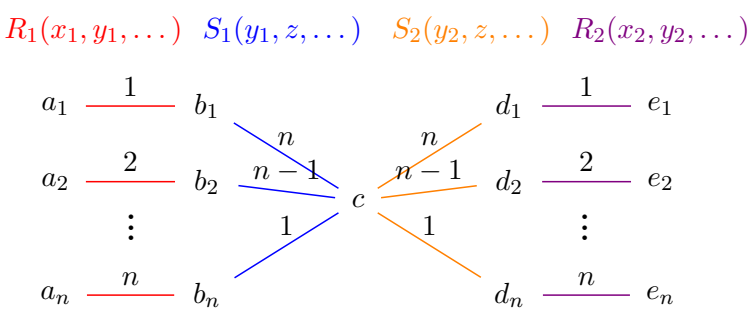


Figure 7.3: Database instance D for Theorem 23. Each edge is color-coded by the relation it belongs to. Values over the edges denote the weight assigned to each tuple.

functions that are monotone with respect to the order of the tuples as depicted in Figure 7.3. Using the same argument as in Lemma 25, it is easy to see that any correct algorithm must examine the rank of $\Omega(n^2)$ tuples tocolor-codeda find the smallest one.

For the other direction, we first show that if $\mathsf{dia}(Q) \leq 3$, then we can construct a tree decomposition with $\texttt{fhw} = 1$ and depth at most one. Indeed, any such decomposition is compatible with any edge decomposable function, and hence Theorem 19 is applicable with $\texttt{fhw} = 1$. In the remainder of the proof, we will show how to find such a decomposition.

Among all tree decompositions with $\texttt{fhw} = 1$ (there exists at least one), let $\mathcal{T}$ be the one with the smallest depth. We will show that it has depth one. Suppose not; then, there is at least one bag that is not a child of the root bag. Figure 7.2 shows an example of a query decomposition where bag $\mathcal{B}_3$ is not a neighbor of $\mathcal{B}_1$. Note that $\mathcal{B}_3 \not\subseteq \mathcal{B}_2, \mathcal{B}_2 \not\subseteq \mathcal{B}_3, \mathcal{B}_2 \not\subseteq \mathcal{B}_1, \mathcal{B}_1 \not\subseteq \mathcal{B}_2, \mathcal{B}_4 \not\subseteq \mathcal{B}_1, \mathcal{B}_1 \not\subseteq \mathcal{B}_4$ (otherwise, bags can be merged without changing the $\texttt{fhw}$ and there will be a decomposition of depth one, a contradiction). Thus, there must exist at least one variable in $\mathcal{B}_3$ (say u) that does not exist in $\mathcal{B}_2$, and hence not in any of $\mathcal{B}_1, \mathcal{B}_4$ since this is a valid decomposition; similarly there exists one variable in $\mathcal{B}_4$ (say v) that is unique to it. This implies that the distance between u and v is at least 4, which is a contradiction to the fact that $\mathsf{dia}(Q) \leq 3$. □

For example, $Q(x, y, z, w) = R(x, y), S(y, z), T(z, w)$ has diameter 3, and thus we can enumerate the result with linear preprocessing time and logarithmic delay for any edge decomposable ranking function. On the other hand, for the 4-path query $Q(x, y, z, w, t) = R(x, y), S(y, z), T(z, w), U(w, t)$, it is not possible to achieve this.

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

7.5.2 Beyond Logarithmic Delay

Next, we examine whether the logarithmic factor that we obtain in the delay of Theorem 19 can be removed for ranked enumeration. In other words, is it possible to achieve constant delay enumeration while keeping the preprocessing time small, even for simple ranking functions? To reason about this, we need to describe the $X + Y$ *sorting* problem.

Given two lists of n numbers, $X = \langle x_1, x_2, \ldots, x_n \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, we want to enumerate all n^2 pairs (x_i, y_j) in ascending order of their sum $x_i + y_j$. This classic problem has a trivial $O(n^2 \log n)$ algorithm that materializes all n^2 pairs and sorts them. However, it remains an open problem whether the pairs can be enumerated faster in the RAM model. Fredman [Fre76] showed that $O(n^2)$ comparisons suffice in the nonuniform linear decision tree model, but it remains open whether this can be converted into an $O(n^2)$-time algorithm in the real RAM model. Steiger and Streinu [SS95] gave a simple algorithm that takes $O(n^2 \log n)$ time while using only $O(n^2)$ comparisons.

Conjecture 4 ([BCD+06, DO05])**.** $X + Y$ sorting *does not admit an* $O(n^2)$ *time algorithm.*

In our setting, $X + Y$ sorting can be expressed as enumerating the output of the cartesian product $Q(x, y) = R(x), S(y)$, where relations R and S correspond to the sets X and Y respectively. The ranking function is $\texttt{rank}(x, y) = x + y$. Conjecture 4 implies that it is not possible to achieve constant delay for the cartesian product query and the sum ranking function; otherwise, a full enumeration would produce a sorted order in time $O(n^2)$.

Chapter 8

Ranked Enumeration of Conjunctive Queries with Projections

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

Example 40. *Consider the DBLP dataset as a single relation $R(A, B)$, indicating that A is an author of paper B. Given an author a, the function h-index(a) returns the h-index of a. A popular analytical task asks to find all co-authors who authored at least one paper together. Additionally, the pairs of authors should be returned in decreasing order of the sum of their h-indexes, since users are only interested in the top-100 results. The following SQL query captures this task.*

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

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

Importance of joins with projections. Join queries containing projections appear in several practical applications such as recommendation systems [FLLQ19, LFZ19], similarity search [YSN+12], and network reachability analysis [EL05, Bir08]. In fact, as Manegold et al. [MKB09] remarked, joins in real-life queries almost always come with projections over certain attributes. Matrix multiplication [AP09], path queries (equivalent to sparse matrix multiplication), and reachability queries [GHL+13] are all examples of join-project queries that have widespread applications in linear and relational algebra. Other data models such as SPARQL [PAG09] also support the projection operator and evaluation of join-project queries has been a subject of research, both theoretically [AG11] and practically [CFZ07].

In fact, as SPARQL supports `ORDER BY`/`LIMIT` operator, ranked enumeration for queries (that include projections) and top-k over knowledge bases in the SPARQL model has also been explicitly studied recently [LBBA16, CRW21]. As many practical SPARQL evaluation systems [HS05, SM13] evaluate queries using RDBMS, it is important to develop efficient algorithms for such queries in the relational model. Similarly, [XD17b] argued that since a large fraction of the data of interest resides in RDBMS, efficient execution of graph queries (such as path and reachability queries that contain projections and ranking) using RDBMS as the backend is immensely useful. In the relational setting, join-project queries also appear in the context of probabilistic databases (see Section 2.3 in [DS07]). This motivates us to develop efficient algorithms, both in theory and practice, that address the challenge of incorporating the ranked enumeration paradigm for join-project queries.

Prior Work. Efficient evaluation of join queries in the presence of ranking functions has been a subject of intense research in the database community. Recent work [TGR20, DK21, YAG+18, CLZ+15, TGR21b] has made significant progress in identifying optimal algorithms for enumerating query results in ranked order. In each of these works, the key idea is to perform on-the-fly sorting of the output via the use of priority queues by taking into account the query structure. [CLZ+15] considered the problem of top-k tree matching in graphs and proposed optimal algorithms by combining Lawler's procedure [Law72] with the ranking function. [TGR20] introduced multiple dynamic programming algorithms that lazily populate the priority queues. [YAG+18] took a different approach where all possible candidates were eagerly inserted into the priority queues and [DK21] generalized these ideas to present a unified theory of ranked enumeration for full join queries. Very recently, [TGR21b] was able to extend some of these results to non-equi-joins as well. The performance metric for enumerating query results is the *delay* [BDG07a], defined as the time difference between any two consecutive answers. Prior work was able to obtain logarithmic delay guarantees, which were shown to be optimal. However, all prior work in this space suffers from one fundamental limitation: it assumes that the join query is full, i.e. there are no projections involved. In fact, [TGR21b] explicitly remarks that in presence of projections, the strong guarantees obtained for full queries do not hold anymore. Their suggestion to handle this limitation is to convert the query with projections into a projection-free result, i.e., they materialize the join query result, apply the projection filter, and then rank the resulting output. However, this conversion requires an expensive materialization step. For instance, Example 40 requires worst-case $\Omega(|D|^2)$ time and space (here $|D|$ is the size of the database) to materialize the join result of the query (without any ordering) before providing it as input for sorting. An alternate approach is to modify the weights of the tuples/attribute values to allow re-use of existing algorithms. As we show later, this approach also does not fare any better and

requires enumerating the full output of the join query, which can be polynomially slower than the optimal solution.

On the practical side, all RDBMS and graph processing engines evaluate join-project queries in the presence of ranking functions by performing three operations in serial order: (*i*) materializing the result of the full join query, (*ii*) de-duplicating the query result (since the query has `DISTINCT` clause), and (*iii*) sorting the de-duplicated result according to the ranking function. The first step in this process is a show-stopper. Indeed, the size of the full join query result can be orders of magnitude larger than the size of the final output after applying projections and de-duplicating it. Thus, the materialization and the de-duplication step introduce significant overhead since they are blocking operators. Further, if the user is interested in only a small fraction of the ordered output (top-k with small k[1]), the user still has to wait until the entire query completes even to see the top-ranked result.

Our Contribution and Key Ideas. In this paper, we initiate the study of ranked enumeration over join-project queries. We focus on two important ranking functions: `SUM` ($f(x,z) = x + z$) and `LEXICOGRAPHIC` ($f(x,z) = x, z$) for two reasons. First, both of these functions are very useful in practice [IBS08]. Second, extending the algorithmic ideas to other functions, such as `MIN`, `MAX`, `AVG` and circuits that use sum and products, is quite straightforward. More specifically, we make three contributions.

1. Enumeration with Formal Delay Guarantees. Our first main result shows that for any *acyclic* query (the most common fragment of queries in practice [BMT20]) with arbitrary projection attributes, it is possible to develop efficient enumeration algorithms.

Theorem 24. *For an acyclic join-project query Q, an instance D and a ranking function* rank $\in \{$*SUM*, *LEXICOGRAPHIC*$\}$*, the query result $Q(D)$ can be enumerated according to* rank *with worst-case delay $O(|D| \log |D|)$, after $O(|D|)$ preprocessing time.*

This result implies that top-k results in $Q(D)$ can be enumerated in $O(k|D| \log |D|)$ time. Theorem 24 is able to recover the prior results [TGR20] for ranked enumeration of full queries as well. We are also able to generalize it to arbitrary join-project queries which may contain cycles using the idea of *generalized query decompositions*, as well as *union of queries*, which is a strictly more expressive class of queries. The key idea of our algorithm is to develop multiway join plans [NPRR12] by exploiting the properties of join trees. Embedding the priority queues in the join tree strategically allows us to generate the sorted output on-the-fly and avoid the binary join plans that all state-of-the-art systems use. Further, since we formulate the problem in terms of delay guarantees, it allows our techniques to be limit-aware: for small k, the answering time is also small.

[1] the value of k in the `LIMIT` clause is not known until the user submits the query.

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

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

Note that $Q_m^\star = \pi_r(Q^*_{A_1,\ldots,A_m})$. Analysis of query logs has shown that star queries (both ranked and unranked) are the most important class of queries seen in practice that form over 90% of all non-trivial queries in real-world settings [BMT20]. The main motivation of studying this setting is as follows: for many data analysis pipelines, queries are asked repeatedly by users or accessed by some downstream tasks. In this case, it is desirable to spend some time preprocessing the input database to ensure that the repeated executions of the query are as fast as possible. The goal of the preprocessing phase is to compute a space-efficient intermediate data structure, which is used by the query answering algorithm to enumerate the query results according to an order given by a ranking function as fast as possible, and ideally, with delay guarantees.

Theorem 25. *For a star join-project query $Q_m^\star$, an instance D, and a ranking function* rank $\in \{$*SUM*, *LEXICOGRAPHIC*$\}$*, the query result $Q(D)$ can be enumerated according to* rank *with worst-case delay $O\left(|D|^{1-\epsilon} \log |D|\right)$, using $O\left(|D|^{1+(m-1)\epsilon}\right)$ preprocessing time and $O\left(|D|^{m(1-\epsilon)}\right)$ space, for any $0 \leq \epsilon \leq 1$.*

Theorem 25 enables users to carefully control the space usage, preprocessing time, and delay. For both Theorem 24 and Theorem 25, we can show that the delay guarantee is optimal subject to a conjecture about the running time of star join-project queries.

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

algorithm uses the additional structure of lexicographical ordering and can execute queries $2-3\times$ faster than the sum function. For queries with unions and cycles, our algorithm maintains its performance improvement over the baselines.

Organization. We overview the prior work for our problem in Section 8.1. Section 8.2 recalls some of the useful notation from previous chapters. The algorithm for acyclic CQs is presented in Section 8.3 which is extended to incorporate a trade-off for star queries in Section 8.4. We show how to handle cyclic queries in Section 8.5. Finally, we conclude the chapter with a comprehensive experimental evaluation in Section 8.6.

8.1 Related Work

Top-k. Top-k ranked enumeration of full join queries has been studied extensively by the database community for both certain [LCIS05, QCS07, ISA+04, LSCI05, APV11, IBS08, BMS+06, TPK+03] and uncertain databases [RDS07, ZLGZ10]. Most of these works exploit the monotonicity property of scoring functions, building offline indexes and integrating the function into the cost model of the query optimizer to bound the number of operations required per answer tuple. We refer the reader to [IBS08] for a comprehensive survey. We note that none of these works consider join-project queries. While many algorithms have used the idea of priority queues for efficient sorting, none of them allow for projections, which was an open problem. Ours is the first work to consider the ranked enumeration of join-project queries.

Rank aggregation algorithms. Top-k processing over ranked lists of objects has a rich history. The problem was first studied by Fagin et al. [Fag02, FLN03] where the database consists of N objects and m ranked streams, each containing a ranking of the N objects to find the top-k results for coordinate monotone functions. The authors proposed Fagin's algorithm (FA) and Threshold algorithm (TA), both of which were shown to be instance optimal for database access cost under sorted list access and random access model. A key limitation of these works is that it expects the input to be materialized, i.e., $Q(D)$ must already be computed and stored for the algorithm to perform random access. This is prohibitively expensive since the space requirement is huge.

Unranked enumeration of query results. Recent work by Kara et al. [KNOZ20a] showed that for a small but important fragment of CQs known as hierarchical queries, it is possible to obtain a trade-off between preprocessing and delay guarantees. Importantly, this result is applicable even in the presence of arbitrary projects. However, the authors did not investigate how to add ranking because adding priority queues at a different location in the join tree leads to different complexities. In fact, a follow-up work [DHK21] showed that the same unranked enumeration could be performed with better delay guarantees under certain settings. Our work considers the class of CQs with arbitrary projections and we are also able to

extend the main result to UCQs, an even broader class of queries. Naturally, our algorithm automatically recovers the existing results for full CQs as well [DK21, TAG+], in addition to the first extensive empirical evidence on how ranked enumeration can be performed for CQs containing projections beyond free-connex queries. Acyclic Boolean queries can be evaluated optimally in linear time (data complexity) by the Yannakakis algorithm [Yan81].

Factorization and Aggregation. Factorized databases [BOZ12, OZ15a, CO15] exploit the distributivity of product over union to represent query results compactly and generalize the results on bounded `fhw`to the non-Boolean case [OZ15a]. [AKNR16] captures a wide range of aggregation problems over semirings. Factorized representations can also enumerate the query results with constant delay according to lexicographic orders of the variables [BKOZ13]. For that to work, the lexicographic order must "agree" with the factorization order. However, it was shown in [DK21] that the algorithm for lexicographic ordering is not optimal. Further, since all prior work in this space uses the concept of a variable ordering, adding projections to the query forces the building of a GHD that can materialize the entire join query result, which is expensive and an unavoidable drawback.

Ranked enumeration. Both Chang et al. [CLZ+15] and Yang et al. [YAG+18] provide any-k algorithms for *graph queries* instead of the more general CQs; Kimelfeld and Sagiv [KS06] give an any-k algorithm for acyclic queries with polynomial delay. Recent work on ranked enumeration of MSO logic over words is also of particular interest [BGJR21]. None of these existing works give any non-trivial guarantees for CQs with projections. Ours is the first work in this space that provides non-trivial guarantees.

8.2 Preliminaries

In this section, we will present our results for queries containing projections. First, for the benefit of the reader, we recall some of the definitions from Section 2 that will be used extensively. We will focus on the class of *join-project queries*, which are defined as

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

Here, each relation has schema $R_i(\mathbf{A}_i)$, where $\mathbf{A}_i$ is an ordered set of attributes. Let $\mathbb{A} = \mathbf{A}_1 \cup \mathbf{A}_2 \cup \cdots \cup \mathbf{A}_m$. The projection operator $\pi_{\mathbf{A}}$ only keeps a subset of the attributes from $\mathbb{A}$. The join we consider is *natural join*, where tuples from two relations can be joined if they share the same value on the common attributes. A join-project query is *full* if $\mathbf{A} = \mathbb{A}$. Unlike prior work on ranked enumeration, we place no restriction on the set of attributes in the projection operator. For simplicity of presentation, we do not consider selections; these can be easily incorporated into our algorithms. As an example, the SQL query in Example 40 corresponds to the following query: $\pi_{A,B}(R_1(A, C) \bowtie R_2(B, C))$. For tuple t, we will use the shorthand $t[A]$ to denote $\pi_A(t)$.

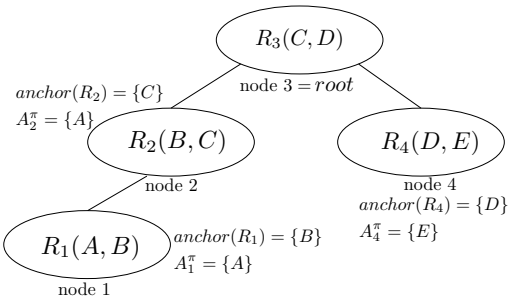


Figure 8.1: Illustration of join tree for a join-project query $Q = \pi_{A,E}(R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D) \bowtie R_4(D, E))$.

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

Example 41. *Consider a join-project query $Q = \pi_{A,E}(R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D) \bowtie R_4(D, E))$ under the ranking function* `SUM` *defined over attributes A, E. In other words, for every output tuple t, the score of the tuple is $t[A] + t[E]$. Figure 8.1 shows the join tree for the query. We fix R_3 as the root with R_2 as the left child and R_4 (a leaf node) as the right child. R_1, as a leaf node, is also the only child of R_2.*

8.2.1 Ranking Functions

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

Example 42. *Consider a function* $w : \mathbf{dom}(A) \to \mathbb{R}$ *for any attribute* $A \in \mathbf{A}$*. For each query result* t*, we define its rank as* $\texttt{rank}(t) = \sum_{A \in \mathbf{A}} w(t[A])$*, the total sum of the weights over all attributes in* $\mathbf{A}$*.*

We will focus on SUM and LEXICOGRAPHIC in this paper. We note that both functions are instantiations of a more general class of *decomposable functions* [DK21]. The ideas introduced for SUM and LEXICOGRAPHIC are readily applicable to more complicated functions including products, a combination of sum and products, etc.

8.2.2 Problem Parameters

Given a join-project query Q and a database D, an enumeration query asks to enumerate the tuples of $Q(D)$ according to some specific ordering defined by `rank`. We study this problem in a similar framework as [Seg15b], where an algorithm is decomposed into:

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

Prior work [DK21] has shown that for acyclic joins without projections, there exists an algorithm with $T_p = S_p = O(|D|)$ that can achieve $\delta = O(\log |D|)$ delay under ranking. However, the problem of ranked enumeration when projections are involved is wide open.

Using Existing Algorithms. One possible solution to the problem is to set the weights of non-projection attributes to 0. This will ensure that for SUM function, only the projection attributes are considered in the ranking and existing algorithms for full join queries could be used. However, this proposal gives poor delay guarantees and is as expensive as enumerating the full join result. For example, for the four path query in Example 41, the output of the query could be constant in size but the full join can be as large as $\Omega(|D|^2)$ which is prohibitively expensive, but our algorithm would only require $O(|D|)$ in this case. In general, a join with ℓ relations may require as much as $\Omega(|D|^{\ell-1})$ time to output the smallest tuple.

Given a black box algorithm $\mathcal{A}$ for ranked enumeration of full join queries, we can obtain an algorithm for ranked enumeration of join-project queries. algorithm 16 shows how $\mathcal{A}$ can be used for enumeration of join-project queries. The key idea here is that $\mathcal{A}$ can assign a weight of zero to all values of non-projection attributes. This will guarantee that tuples enumerated by $\mathcal{A}$ will be sorted only over the sum of attribute values of $\mathbf{A}$. The while

Algorithm 16: UseExistingAlgorithm

Input : Join-project query Q, ranking function SUM and database D

Output: $Q(D)$ in ranked order

1 $Q' \leftarrow$ full query obtained by dropping the projection operator from Q

2 Provide Q', ranking function SUM that assigns weight zero to all values of attributes $\mathbb{A} \setminus \mathbf{A}$, and D to $\mathcal{A}$ for pre-processing;

3 last $\leftarrow \emptyset$

4 **while** $\mathcal{A}$.hasNext() **do**

5 $\quad o \leftarrow \mathcal{A}$.getNext() /* enumerate the answer from $\mathcal{A}$ */

6 $\quad$ **if** last $\neq o[\mathbf{A}]$ **then**

7 $\quad\quad$ output $o[\mathbf{A}]$, last $\leftarrow o[\mathbf{A}]$

8 **return** /* Enumeration complete */

loop contains a variable that stores the last tuple output $\mathbf{A}$. This helps in ensuring that no duplicates are output for the join-project query. Intuitively, algorithm 16 is performing a ranked group by over the attributes $\mathbf{A}$ and outputs an answer when the grouping value changes. We now show that there exist queries where the dealy for this algorithm could be large. Consider the query

$$Q = \pi_{X_1}(\bowtie_{i \in [\ell]} R_i(X_i, Y))$$

.

Suppose that each relation R_i has N attribute value for X_i that is connected one Y attribute value $y^\star$ for each $i \in \{1, \ldots, \ell\}$. Then, the smallest answer of QD) will be output $N^{\ell-1}$ times by $\mathcal{A}$ on line 5. This is because the join of $R_2 \bowtie \cdots \bowtie R_\ell$ has a size of $N^{\ell-1}$. This directly implies that the delay guarantee is $\Omega(N^{\ell-1})$.

8.3 General acyclic queries

We first describe the main algorithm of enumerating acyclic join-project queries for SUM ordering in subsection 8.3.1, followed by a specialized algorithm for LEXICOGRAPHIC ordering in subsection 8.3.2. Before we describe the algorithm, we introduce two key data structures that will be used: *cell* and *priority queues*.

Definition 8. *A cell, denoted as $c = \langle t, [p_1, \ldots, p_k], q \rangle$, is a vector consisting of three values: (i) a tuple $t \in R_i$ for node i in the join tree $\mathcal{T}$, (ii) an array of pointers $[p_1, \ldots, p_k]$ where the ℓ^{th} pointer points to a cell defined for ℓ^{th} child of node i in $\mathcal{T}$, (iii) a pointer q that can only point to another cell defined for node i.*

Given a cell c defined for node i, one can reconstruct the tuple over $\mathbf{A}_i^\pi$in constant time (dependent only on the query size, which is a constant) by traversing the pointers recursively.

We will use output(c) to denote the utility method that performs this task. Note that the time and space complexity of creating a cell is $O(1)$ since the size of the query and the database schema are assumed to be constant. This implies that we only need to insert/access a constant number of entries in the vector representing a cell. Similarly, output(c) also takes $O(1)$ time since the join tree size is a constant.

Priority queue. A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. The space complexity of a priority queue containing $|S|$ elements is $O(|S|)$. We will use an implementation of a priority queue (e.g., a Fibonacci heap [FT87]) with the following properties: (i) an element can be inserted in $O(1)$ time, (ii) the min element can be obtained in $O(1)$ time, and (iii) the min element can be popped and deleted in $O(\log |D|)$ time. We will use the priority queue in conjunction with a cell in the following way: for two cells c_1 and c_2, the priority queue uses $\texttt{rank}(\texttt{output}(\texttt{c}_1))$ and $\texttt{rank}(\texttt{output}(\texttt{c}_2))$ in the comparator function to determine the relative ordering of c_1 and c_2. If $\texttt{rank}(\texttt{output}(\texttt{c}_1)) = \texttt{rank}(\texttt{output}(\texttt{c}_2))$, then we break ties according to the lexicographic order of $\texttt{output}(\texttt{c}_1)$ and $\texttt{output}(\texttt{c}_2)$. The choice of lexicographic ordering is not driven by any specific consideration; as long as the ties are broken consistently, we can use other tie-breaking criteria too. Once again, the comparator function only takes a $O(1)$ time to compare since the ranking function $\texttt{rank}(\texttt{output}(\texttt{c}))$ can be evaluated in constant time.

8.3.1 General Algorithm

Algorithm 17: PREPROCESSACYCLIC

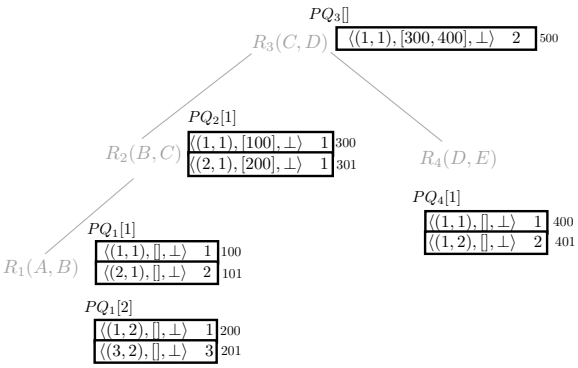
Input : Input query Q, database instance D; join tree $\mathcal{T}$; ranking function rank.
Output: Priority Queues PQ

```
1 foreach R_i ∈ T in post order traversal do
2     foreach t ∈ R_i do
3         u ← π_{key(R_i)}(t);
4         if PQ_i[u] does not exist then
5             PQ_i[u] ← ∅;      /* Initialize a priority queue */
6         L ← ∅;
7         foreach R_j is the child of R_i do
8             L.insert(PQ_j[π_{key(R_j)}(u)].top());
9     PQ_i[u].insert(⟨t, L, ⊥⟩);
```

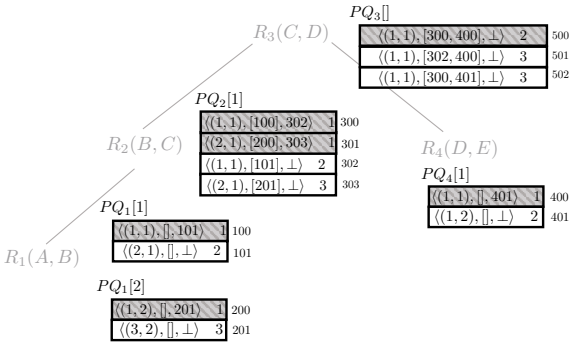
In this section, we present the algorithm for Theorem 24. At a high level, each node i in the join tree will materialize, in an incremental fashion, all tuples over the attributes $\mathbf{A}_i^{\pi} \cup \texttt{key}(R_i)$ in sorted order. To efficiently store the materialized output, we will use the cell data structure. Since we need to sort the materialized output, each node in the join tree maintains

a set of priority queues indexed by $\pi_{\texttt{key}(R_i)}(u), u \in R_i$. The values of the priority queue are the cells of node i. For example, given the join tree from Example 41, node 2 containing R_2 will incrementally materialize the sorted result of the subquery $\pi_{C,A}(R_2(B,C) \bowtie R_1(A,B))$ that is indexed by the values $\pi_C(R_2(B,C))$ since $\mathbf{A}_2^{\pi} = \{A\}$ and $\texttt{key}(R_2) = \{C\}$. Note that there may be multiple possible join trees for a given acyclic query. Our algorithm applies to all join trees. In fact, any node in the join tree can be chosen as the root without any impact on the time and space complexity.

Preprocessing Phase. We begin by describing the algorithm for preprocessing in algorithm 17. We assume that a join tree has been fixed and the input instance D does not contain any dangling tuples, i.e., tuples that will not contribute to the join; otherwise, we can invoke the Yannakakis algorithm [Yan81] to remove all dangling tuples. We initialize a set of empty priority queues for every node in the join tree. We proceed in a bottom-up fashion and perform the following steps. For each leaf relation $R_i \in \mathcal{T}$, we create a cell $\langle t, [], \bot \rangle$ for each tuple $t \in R_i$ and insert it into $\mathsf{PQ}_i[\pi_{\texttt{key}(R_i)}(t)]$. For each non-leaf relation $R_j \in \mathcal{T}$, we create a cell for $t \in R_j$, which points to the top of the priority queue in each child node of R_j that can be joined with t. This cell is then added to the priority queue $\mathsf{PQ}_i[\pi_{\texttt{key}(R_i)}(t)]$. Note that we only have one priority queue for the root relation r since $\texttt{key}(r) = \{\}$ by definition.



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



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

Figure 8.2: Example to demonstrate the preprocessing and enumeration phase of the general algorithm

Example 43. *Continuing with the 4-path query running example, consider the following instance D as shown below.*

Algorithm 18: EnumAcyclic

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

Output: $Q(D)$ in ranked order

```
1  procedure Enum()
2      last ← ∅;
3      while PQ_r[∅] ≠ ∅ do
4          o ← PQ_r[∅].top();
5          if IS_EQUAL(o, last) = false then
6              print output(o), last ← o;                    /* new output found */
7          Topdown (o, r);
8  procedure Topdown(c, j) /* c = ⟨t, [p_1, ..., p_k], next⟩ */
9      u ← π_{key(R_j)}(c.t);
10     if c.next = ⊥ then
11         while true do
12             temp ← pop(PQ_j[u]);
13             foreach R_i is a child of R_j do
14                 p'_i ← Topdown(c.p_i, i) ;
15                 if p'_i ≠ ⊥ then
16                     PQ_j[u].insert(⟨t, [c.p_1, ..., p'_i, ... c.p_k], ⊥⟩)
17             if R_j is not the root then
18                 c.next ← addressof(PQ_j[u].top());
19             if IS_EQUAL(temp, PQ_j[u].top()) = false then break
20     return c.next;
21 procedure IS_EQUAL(c_1, c_2)
22     if rank(output(c_1)) ≠ rank(output(c_2)) then return false;
23     foreach A ∈ A do
24         if output(c_1)[A] < output(c_2)[A] then return false;
25     return true;
```

R_1

A	B
1	1
2	1
1	2
3	2

R_2

B	C
1	1
2	1

R_3

C	D
1	1
1	2

R_4

D	E
1	1
1	2

As we saw before, Figure 8.1 shows the join tree along with the anchor attributes in each relation. Figure 8.2a shows the state of priority queues after the preprocessing step. After the

full reducer pass, tuple $(1, 2)$ *is removed from* R_3 *because no join tuple can be formed using it. Then, we start constructing the cells for each node starting with the leaf nodes. Since* B *is the anchor for relation* R_1*, we create two priority queues* $\mathsf{PQ}_1[1]$ *and* $\mathsf{PQ}_1[2]$*. For* $\mathsf{PQ}_1[1]$*, we create the cells for tuples* $(1, 1)$ *and* $(2, 1)$*. For convenience, the cells are followed by the partially aggregated score. Consider relation* $R_2(B, C)$*. The cell for tuple* $(1, 1)$ *in* $\mathsf{PQ}_2[1]$ *points to the top of* $\mathsf{PQ}_1[1]$ *(shown as pointer with address* 100*). The root bag consists of a single tuple entry that points to the cells at locations* 300 *and* 400*. The output tuple that can be formed by the root bag is* $(A = 1, E = 1)$*.*

Enumeration Phase. We describe the enumeration procedure in algorithm 18. The high-level idea is to output answers by repeatedly popping elements from the root priority queue. It may be possible that multiple tuples of the root priority queue output the same final result. To deduplicate answers, we compare the answer at the current top of the priority queue with the previous answer (line 5) and output it only if they are different. Then, we invoke the procedure TOPDOWN to insert new candidates into the priority queue. This procedure will be recursively propagated over the join tree until it reaches the leaf nodes. Observe that once the new candidates have been inserted, the next pointer of a cell is updated by pointing to the topmost element in the priority queue. This chaining materializes the answers for a particular node that can be reused and is key to avoiding repeated computation.

Example 44. *Continuing our running example, Figure 8.2b shows the state of the priority queues after one complete iteration of procedure* `Enum()`*. We first pop the only element in the root priority queue and note that the output tuple* $(A = 1, E = 1)$ *is enumerated. Then we call* `TOPDOWN` *with cell at memory* 500 *and root (node* 3*) as arguments (denoted as* `TOPDOWN`$(*500, 3)$*). The* next *for the cell is* $\perp$ *so we pop the cell at* 500 *from the priority queue (shown as greyed out in the figure) and recursively call* `TOPDOWN`$(*300, 2)$*. The cell at memory location* 300 *has* $\mathsf{next} = \perp$*. Therefore, we enter the while loop, pop the cell and recursively call* `TOPDOWN`$(*100, 1)$*. We have now reached the leaf node. The anchor attribute value for cell at* 100 *is* $u = 1$*, so we pop the current cell from* $\mathsf{PQ}_1[1]$ *(greyed out cell at* 100*), find the next candidate at the top of* $\mathsf{PQ}_1[1]$ *(which is cell at* 101*), chain it to the cell at* 100 *by assigning* $\mathsf{next} = 101$ *and return the cell at* 101 *to the parent. When the program control returns from the recursive call back to node* 2*, we create a new cell (at memory address* 302*) that points to* 101 *and insert it into the priority queue. However, observe that the cell at memory location* 301 *also generates* $A = 1$*, a duplicate since cell at* 300 *also generated it. This is where the equality check at line 19 comes in. Since both cells at* 300 *and* 301 *generate the same value, we also pop off the cell at* 301 *in the subsequent while loop iteration, find its next candidate, and create the cell at* 303*, and insert it into the priority queue. This ensures that all elements in* $\mathsf{PQ}_2[1]$ *generating the same* A *value are removed, ensuring no duplicates at the root level. Finally, the control returns to the root level* `TOPDOWN` *call. The recursive call*

to the right child (node 4*) create a new cell* 401 *and we insert two cells at the root priority queue, cell* 501 *and* 502 *that correspond to output tuple* $(A = 2, E = 1)$ *and* $(A = 1, E = 2)$ *respectively.*

We are now ready to formally prove Theorem 24.

Lemma 26. *The delay guarantee of* EnumAcyclic *is at most* $O(|D| \log |D|)$.

Proof. To prove the delay guarantee, we analyze the worst-case running time that can happen in each iteration. Without loss of generality, we assume that there exists at least one non-anchor projection attribute in all leaf nodes (otherwise, we can simply remove the node from the join tree after the full reducer pass).

Let us first analyze the procedure `ENUM`. The root priority queue (denoted PQ_r) can contain at most $|D|$ entries for each output result. Indeed, in the worst-case, the output tuple emitted to the user on line 6 may join with every tuple in the root node relation. Therefore, in the worst-case, for every tuple outputted to the user, we invoke the `TOPDOWN` procedure $|D|$ times.

We now argue that `TOPDOWN` takes at most $O(|D| \log |D|)$ time. There are two key observations to be made. First, `TOPDOWN` takes $O(1)$ time if $c.\mathsf{next}$ is already populated. Intuitively, this means that in a previous iteration, there was a recursive call made to `TOPDOWN` that populated the next and we are now simply reusing the computation. Thus, for every tuple output to the user, it suffices to count how many times `TOPDOWN` is invoked in total, such that next is not empty. We say that such invocations of `TOPDOWN` are *non-trivial*. Our claim is that the number of non-trivial `TOPDOWN` calls is at most $O(|D|)$. Consider the set of all anchor attribute values for some node j (denoted $\mathcal{L}_{R_j}$). We fix some $u \in \mathcal{L}_{R_j}$. In the worst-case, $\texttt{TOPDOWN}(c, j)$ such that $u = \pi_{\texttt{key}(R_j)}(c.t)$ is invoked by all tuples t' in the parent relation of node j such that $\pi_{\texttt{key}(R_j)}(t') = u$ but only the first call will be non-trivial since the first call populates the $c.\mathsf{next}$. Therefore, it follows that for each $u \in \mathcal{L}_{R_j}$, there is at most one non-trivial call to $\texttt{TOPDOWN}(c, R_j)$ for a fixed node R_j. Since the size of the join tree is at most $O(1)$, the total number of non-trivial calls over all nodes in the join tree is $\sum_j \sum_{u \in \mathcal{L}_{R_j}} 1 = \sum_j |D| = O(|D|)$. Finally, we perform one pop operation and a constant number of inserts into the priority queue in every non-trivial invocation which adds another $\log |D|$ factor in the running time, giving us the claimed delay guarantee of $O(|D| \log |D|)$. □

Lemma 27. PreprocessAcyclic *running in* $O(|D| \log |D|)$ *time, generates a data structure of size* $O(|D|)$.

Proof. For a node $R_i \in \mathcal{T}$, we initialize an empty priority queue for each possible value in $\texttt{key}(R_i)$. The for loop on line 7 takes $O(1)$ time as each node has $O(1)$ children in the decomposition and all operations on line 8-12 also take $O(\log |D|)$ time. Overall, since each node has at most $O(|D|)$ tuples, the claim readily follows. □

Lemma 28. EnumAcyclic *enumerates the query result $Q(D)$ in ranked order correctly.*

Proof. We will prove our claim by induction on height of the tree. We will show that at each relation R, the priority queue $\mathsf{PQ}_R[u]$ correctly computes the answers over all projection attributes in the subtree rooted at R (denoted $\mathbf{A}_R^\pi$) in ranked order.

Base Case. Correctness for ranked output of leaf relations is trivial. For all tuples t in the relation at a leaf node with the same anchor attribute value u, $\mathsf{PQ}_R[u]$ contains all tuples t and the priority queue (whose implementation is assumed to be correct) will pop out $t[\mathbf{A}_R^\pi]$ in the correct order since the score $\mathsf{rank}(t[\mathbf{A}_R^\pi])$ is used as the priority queue comparator function and all projection attributes $\mathbf{A}_R^\pi$ are present in t already. It also follows that $R[\mathbf{A}_R^\pi]$ will be materialized by chaining the cells popped from the priority queue since the leaf relation already contains all the attributes in $\mathbf{A}_R^\pi$.

Inductive Case. Consider a relation R with children relations $R_1, \ldots R_s$. Let c be the cell that was input to the TopDown procedure call under consideration. Let $u = c.t[\mathtt{key}(R)]$ and $u_i = c.t[\mathtt{key}(R_i)]$. Let $t_\pi = \mathsf{output}(c_{\mathsf{next}})$ be the next tuple and let c_{next} be the cell that is to be returned to the parent of R by the recursive call to R. Our goal is to show that t_π is generated correctly and c_{next} is inserted in $\mathsf{PQ}_R[u]$. This will guarantee the correct ordering of the tuples over attribute set $\mathbf{A}_R^\pi$. From the induction hypothesis, we have that the ranked output, over the attributes $\mathbf{A}_i^\pi$, from each R_i is generated correctly. For the sake of contradiction, suppose that t'_π is the next smallest candidate, i.e. $\mathsf{rank}(t'_\pi) < \mathsf{rank}(t_\pi)$.

Lines 13-16 in algorithm 18 generates s possible candidate cells (and is the only way to generate new candidates) that could generate t_π. Let $\hat{c}_i$ be the cell returned from the recursive call to TopDown for relation R_i. Then, there are s possible cells that are generated by TopDown for relation R whose array of pointers will be one of:

$$\hat{c}_1, c_2, c_3, \ldots, c_s,$$
$$c_1, \hat{c}_2, c_3, \ldots, c_s,$$
$$\ldots,$$
$$c_1, c_2, c_3, \ldots, \hat{c}_s$$

Here, c_i is i^{th} pointer in the pointer array of c and $\mathsf{rank}(\mathsf{output}(\hat{c}_i)) \geq \mathsf{rank}(\mathsf{output}(c_i))$ since the priority queue of the R_i will generate answers in increasing order (from the induction hypothesis). Assume for the sake of contradiction that $c'_{\mathsf{next}} = \langle c.t, [c'_1, c'_2, \ldots, c'_s], \perp \rangle$ is the cell such that $t'_\pi = \mathsf{output}(c'_{\mathsf{next}})$ and is not one of the s candidate cells generated above. Since t'_π has a smaller score than t_π, it follows that c'_{next} has not been inserted in the priority queue yet (otherwise the priority queue will return c'_{next} correctly). We will show that such a scenario will violate the monotonicity property of the sum ranking function.

Case 1. $\mathsf{rank}(\mathsf{output}(c_i)) \geq \mathsf{rank}(\mathsf{output}(c'_i))$ **for** $i \in [s]$**.** This scenario implies that t'_π has been generated before t_π, which would violate our assumption that t'_π has not been inserted in the priority queue. Indeed, since $\mathsf{rank}(\mathsf{output}(c_i)) \geq \mathsf{rank}(\mathsf{output}(c'_i))$ and lines 13-16 in algorithm 18 generate cells that are monotonically increasing, there exists a sequence of generations that generate c_{next} from c'_{next} which would mean that c'_{next} was present in the priority queue at some point.

Case 2. $\mathsf{rank}(\mathsf{output}(c_i)) < \mathsf{rank}(\mathsf{output}(c'_i))$ **for** $i \in [s]$**.** This scenario implies that $\sum_i \mathsf{rank}(\mathsf{output}(c_i)) = \mathsf{rank}(t_\pi) < \sum_i \mathsf{rank}(\mathsf{output}(c'_i)) = \mathsf{rank}(t'_\pi)$ which contradicts our assumption that t'_π has a smaller score than t_π.

Case 3. $\mathsf{rank}(\mathsf{output}(c_i))$ **and** $\mathsf{rank}(\mathsf{output}(c'_i))$ **are incomparable.** If c'_{next} has not been inserted in the priority queue yet, there are two possible scenarios. Either c'_{next} will be generated by some cell c''_{next} that is already in the priority queue. Clearly, $\mathsf{rank}(\mathsf{output}(c'_{\mathsf{next}})) > \mathsf{rank}(\mathsf{output}(c''_{\mathsf{next}})) > \mathsf{rank}(\mathsf{output}(c_{\mathsf{next}}))$ using the same reasoning as **Case 1** which contradicts our assumption. The second possibility is that c'_{next} and c_{next} are generated in the same loop. But this would again imply that c'_{next} is in the priority queue, a contradiction to our assumption.

Therefore, it cannot be the case that $\mathsf{rank}(t'_\pi) < \mathsf{rank}(t_\pi)$ which proves that for relation R, all tuples generated over the attributes $\mathbf{A}_R^\pi$ are in the correct order and consequently, the claim holds for all relations in the join tree, including the root.

In the second part of the proof, we will show that every result enumerated must belong to $Q(D)$. This is a direct consequence of the properties of the join tree. Indeed, if some tuple $t \notin Q(D)$ is enumerated, that would violate the join condition on some attribute $X \in \mathbf{A}$. Next, we will show that every query result in $Q(D)$ will be enumerated using induction on the join tree. In the base case, for some leaf node R in $\mathcal{T}$, every query result over attributes $\mathbf{A}_R^\pi$ will be enumerated from `Topdown` because the relation is materialized. By the induction hypothesis, for each $i \in [s]$,

$$Q_i = (\bowtie_{j \in \text{ nodes in subtree rooted at } i} R_j)[\mathbf{A}_i^\pi]$$

is enumerated completely. For each tuple t in R, lines 13-16 guarantees that,

$$((\times_{i \in [s]} Q_i) \bowtie t)[\mathbf{A}_R^\pi]$$

must be inserted into $\mathsf{PQ}[\pi_{\mathtt{key}(R)}(t)]$, and thus enumerated. In other words, the projection of cartesian product between each subquery rooted at R_i (i.e. Q_i as defined above) and t over $\mathbf{A}_R^\pi$ is generated. This implies that the subquery induced by the subtree rooted at R enumerates all join result over $\mathbf{A}_R^\pi$. In the preprocessing phase, all tuples in R are put into the entries of $\mathsf{PQ}[\pi_{\mathtt{key}(R)}(t)]$, thus every query result in $Q(D)$ will necessarily be enumerated. $\square$

Together, the above lemmas establish Theorem 24. We also now outline how we can recover prior results for full queries from [TGR20]. The key observation is that when the query is full, the while loop always terminates after a constant number of operations. This is because no two answer tuples over $\mathbf{A}_R^\pi$ are the same, i.e., the attribute values for any two answer tuples cannot be identical. Further, any answer tuple at R is inserted at most a constant number of times. This guarantees that `IS_EQUAL` procedure will return false after popping an answer a constant number of times and the while loop terminates. Thus, in the worst-case, `TOPDOWN` visits each relation of the join tree at most once and each iteration takes only $O(\log |D|)$ time due to the priority queue operations.

Free-connex queries. A join query Q is said to be free-connex if Q is acyclic and the hypergraph containing edges of Q and a new edge containing only the projection variables is also acyclic. Free-connex is an interesting class of queries that allows constant delay enumeration for a variety of problems [BDG07a, CK19]. If a join-project query is free-connex, then our algorithm achieves $O(\log |D|)$ delay enumeration after linear time preprocessing. It can be shown that for free-connex queries, all projection attributes are connected and must be at the top of the tree and all non-projection attributes are connected and at the bottom of the tree. This observation allows us to remove all relations that do not contain any projection attributes or relations where only anchor attributes are projection attributes, transforming the free-connex query into a full join query.

8.3.2 Improvement for Lexicographic Ranking

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

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

Enumeration Phase. Given an attribute order of output attributes $\mathbf{A} = \{A_1, A_2, \cdots, A_m\}$, we start by fixing the minimum value in $\mathbf{dom}(A_1)$ as a_1. Then, we perform the two-phase semi-joins to remove tuples that cannot be joined with value a_1, and find the values in $\mathbf{dom}(A_2)$ that survive after semi-joins, denoted as $\mathcal{L}_{A_2}(a_1)$. Similarly, we fix the minimum value in $\mathcal{L}_{A_2}(a_1)$ as a_2, and perform the two-phase semi-joins for finding the values in

$\mathbf{dom}(A_3)$ that can be joined with both a_1, a_2. We continue this process until all attributes in $\mathbf{A}$ have been fixed, and end up with enumerating such a query result (with fixed values). Then, we backtrack and continue the process until all values in attribute A_1 are exhausted.

Algorithm 19: EnumAcyclicLexi(t, $\mathcal{L}$, i)

Input : Input query Q, database D

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

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

Algorithm 19 takes as input an acyclic query Q, an database D, an integer $i \in \{1, \cdots, m\}$, a tuple t defined over attributes $A_1, \cdots, A_{i-1}$, and a set of values $\mathcal{L} \subseteq \mathbf{dom}(A_i)$ that can be joined with t in D. The original problem can be solved by invoking EnumAcyclicLexi($\emptyset, \mathbf{dom}(A_1), 1$) for sorted $\mathbf{dom}(A_1)$.

Lemma 29. EnumAcyclicLexi *enumerates $Q(D)$ correctly in lexicographic order with delay guarantee $O(|D|)$ after preprocessing time $T_p = O(|D| \log |D|)$ and space complexity $O(|D|)$.*

Proof. We will prove that the algorithm enumerates the query result in the correct lexicographic order using induction on the parameter i in algorithm 19.

Base case. $i = 1$**.** Note that since $\mathbf{dom}(A_1)$ is sorted t', will be initialized with $a \in \mathbf{dom}(A_1)$ in sorted order which guarantees that all output tuples with a as the attribute value for A_1 will be enumerated before tuples with a' as the attribute value for A_1 if $a \leq a'$, thus respecting the lexicographic order for A_1.

Inductive case. $i = k$**.** From the induction hypothesis, we get that the algorithm will enumerate correctly for the first $k-1$ attributes including A_{k-1}. From the previous iteration, $\mathcal{L}'$ is set to be the sorted list of attribute values for A_k that join with $R_k(A_k, A_{k+1})$. Therefore, it follows that for the recursive call where $i = k$, t' will have a attribute values set in sorted order for A_k, ensuring that all answers containing $a = \pi_{A_k}(t')$ will be enumerated before answers containing a' as attribute value for A_k for all $a \leq a'$. Thus, the argument holds for $i = m$ as well, concluding the proof.

Next, we analyze the time complexity. The preprocessing phase takes $O(|D| \log |D|)$ time in total for sorting and building hashing tables for each relation. Observe that after fixing a value in $\mathbf{dom}(A_i)$, it takes at most $O(|D|)$ time to perform the semi-joins and find the set of candidate values in $\mathbf{dom}(A_{i+1})$. Whenever i reaches m, a valid query result will be

Algorithm 20: PREPROCESSSTAR

Input : Input star query $Q^\star_m$, ranking function rank and database D; degree threshold $\delta \geq 1$

Output: Heavy output $\mathcal{O}^H$ and priority queue PQ

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

enumerated. As the query size, as well as m, is a constant, the delay between two consecutive query results is at most $O(|D|)$, which improves Lemma 26 by a log factor. □

8.4 Star Queries

In this section, we present a specialized data structure for the *star query*, which is represented as: $Q^\star_m = \pi_{\mathbf{A}}(R_1(A_1, B) \bowtie R(A_2, B) \bowtie \cdots \bowtie R_m(A_m, B))$. where $\mathbf{A} = \{A_1, \cdots, A_m\}$. All relations in a star query join on exactly the same attribute(s). In this following, we present a specialized data structure on ranked enumeration for $Q^\star_m$ in Section 8.4.1, and prove the optimality in Section 8.4.2.

8.4.1 The Algorithm

Consider the star query $Q^\star_m$, a database D and a ranking function RANK. Now we present a data structure for Theorem 25.

Preprocessing Phase. Without loss of generality, assume that there is no dangling tuples in D. Moreover, if $\mathbf{A}$ does not include an attribute A, we can remove efficiently R_i using a semi-join. We first fix a degree threshold $\delta \geq 1$ (whose value will be determined later). For each $i \in \{1, 2, \cdots, m\}$, a value $a_i \in \mathbf{dom}(A)$ is *heavy* if it has degree larger than δ in R_i, i.e., $|\sigma_{A=a_i}(R_i)| \geq \delta$, and *light* otherwise. A tuple $t = (a_i, b) \in R_i$ is *heavy* if a_i is heavy. For R_i, let R_i^H, R_i^L be the set of heavy and light tuples in R_i. An output $t = (a_1, a_2, \ldots, a_m) \in Q^\star_m(D)$ is *heavy* if a_i is heavy in R_i for each $i \in \{1, 2, \cdots, m\}$, and *light* otherwise. In this way,

Algorithm 21: EnumStar

Input : Star query $Q^\star_m$, ranking function rank and database D; Output of $\mathcal{O}^H$ and priority queue PQ

Output: $Q^\star_m(D)$ in ranked order

```
while PQ ≠ ∅ do
    t ← PQ.pop();
    output t;                                   /* enumerate the result */
    if t ∉ O^H then
        i ← smallest positive index such that π_{A_j}(t) is heavy for all j < i and π_{A_i}(t)
          is light;
        next ← EnumAcyclic(Q_i, T_i);
        PQ.insert(next);
    else PQ.insert(O^H.pop());
```

we can divide the output $Q^\star_m(D)$ into $\mathcal{O}^H$ and $\mathcal{O}^L$, containing all heavy and light output tuples separately. In the preprocessing phase, our goal is to materialize all heavy output tuples ($\mathcal{O}^H$) ordered by `RANK`. Details are described in algorithm 20. We compute $\mathcal{O}^H = \pi_{\mathbf{A}} \left(R_1^H \bowtie R_2^H \bowtie \cdots \bowtie R_m^H\right)$ by invoking the Yannakakis algorithm [Yan81], and then sort $\mathcal{O}^H$ by `rank`. Next, we insert the smallest query result from $\mathcal{O}^H$ into the priority queue. Then, we define m different subqueries as $Q_i = \pi_{\mathbf{A}} \left(R_1^H \bowtie \cdots \bowtie R_{i-1}^H \bowtie R_i^L \bowtie R_{i+1} \bowtie \cdots \bowtie R_m\right)$ where tuples in relation R_j are heavy for any $j < i$ and tuples in relation R_i are light. For such Q_i, we consider a join tree $\mathcal{T}_i$ in which R_i is the root and all other relations are children of R_i. We preprocess a data structure for Q_i with $\mathcal{T}_i$, by invoking Algorithm 17.

Enumeration Phase. As described in algorithm 21, the high-level idea in the enumeration is to perform a $(m+1)$-way merge over $\mathcal{O}^H$ and Q_i's. Specifically, we maintain a priority queue PQ with one entry for each subquery Q_i and one entry for $\mathcal{O}^H$. Once the smallest element is extracted from PQ (say t generated by Q_i), we extract the next smallest candidate from Q_i (if there is any) and insert it into PQ. Moreover, finding the smallest candidate output result from $\mathcal{O}^H$ is trivial since $\mathcal{O}^H$ has been materialized in a sorted way in the preprocessing phase. We conclude this subsection with the formal statement of the result.

Lemma 30. *algorithm 20 runs in time $T = O(|D| \cdot (|D|/\delta)^{m-1})$ and requires space $S = O((|D|/\delta)^m)$. algorithm 21 correctly enumerates the result of the query in ranked order with delay $O(|D| \log |D|/\delta)$.*

Proof. ***Time and Space complexity.*** First, we analyze its time complexity. Computing data statistics for each relation R_i takes $O(|D|)$ time. As the degree threshold is δ, there are at most $O(\frac{|D|}{\delta})$ heavy values for each A_i. The size of $\mathcal{O}^H$ can be bounded by $O(|D| \cdot (\frac{|D|}{\delta})^{m-1})$,

as well as the time cost for computing $\mathcal{O}^H$ by the Yannakakis algorithm. At last, initializing data structures for each Q_i takes linear time in terms of $O(|D|)$. As the number of such queries is $O(m)$, the time cost for line 3-6 is $O(|D| \cdot m)$. Overall, the time complexity of Algorithm 20 is $O(|D| \cdot (\frac{|D|}{\delta})^{m-1})$. For the space usage, storing $\mathcal{O}^H$ takes $O((\frac{|D|}{\delta})^m)$ space since its size can be bounded by $O((\frac{|D|}{\delta})^m)$ using the AGM bound.

Delay guarantee. Note that min-extraction and update of the priority queue takes $O(\log |D|)$ time. The expensive part is the invocation of EnumAcyclic for each Q_i. In the join tree $\mathcal{T}_i$, each node contains at least one projection attribute. Plugging in Lemma 26 and the observation that each y value has bounded degree, the delay of enumerating query result from Q_i is at most the degree of values of attribute A_i, i.e., δ, since tuples in R_i are light in this case, by construction of Q_i. Together, the delay of algorithm 21 is $O(\delta \log |D|)$. By setting $\delta = |D|^{1-\epsilon}$ for arbitrary constant $0 < \epsilon < 1$, we can achieve the result in Theorem 25.

Correctness. Recall that O^H is already materialized and sorted order and since the enumeration of each subquery Q_i is performed using EnumAcyclic (which enumerated the query result correctly), the output of each Q_i is enumerated in the correct order. It remains to be shown that EnumStar can also combine the answers from O^H and Q_i correctly. Suppose that t was the last answer output to the user. If t was generated by some Q_i (which can be checked in constant time), then we find the next smallest candidate from Q_i and insert it into PQ. Otherwise, if $t \in O^H$, we find the next tuple in the materialized output and insert it into PQ. Thus, since the smallest candidates for each Q_i and O^H are present in the PQ, the smallest answer that must be output to the user necessarily has to be from the priority queue. Thus, the priority queue performs the $m + 1$-way merge correctly. This concludes the proof. □

8.4.2 Tradeoff Optimality

We next present conditional optimality for our trade-off achieved in Theorem 25. Before showing the proof, we first revisit a result on unranked evaluation for $Q^\star_m$ in [AP09]:

Lemma 31 ([AP09])**.** *There exists a combinatorial*[2] *algorithm that can evaluate $Q^\star_m$ on any database D in time $O\left(|D| \cdot |Q^\star_m(D)|^{1-\frac{1}{m}}\right)$.*

which was presented over a decade ago without any improvement since then. Thus, it is not unreasonable to conjecture that Lemma 31 is optimal. Based on the conjectured optimality of Lemma 31, we can show the following result for unranked enumeration.

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

Lemma 32. *Consider star query $Q_m^\star$, database D and some constant $\epsilon \in [0,1]$. If there exists an algorithm that supports $O(|D|^{1-\epsilon}\log|D|)$-delay enumeration after $O(|D|^{1+(m-1)\epsilon-\epsilon'})$ preprocessing time for some constant $\epsilon' > 0$, the optimality of Lemma 31 will be broken.*

Proof. For a star query $Q_m^\star$ and database D, assume there is an algorithm that supports $O(|D|^{1-\epsilon}\log|D|)$-delay enumeration after $O(|D|^{1+(m-1)\epsilon-\epsilon'})$ preprocessing time. Then, there is an algorithm evaluating $Q_m^\star(D)$ in (big-Oh of)

$$|D|^{1+(m-1)\epsilon-\epsilon'} + |D|^{1-\epsilon}\log|D| \cdot |Q_m^\star(D)|$$

time. Let $\alpha = m\epsilon - \epsilon'$ for some constant $\epsilon' < \epsilon$. Consider an instance D' for $Q_m^\star$ with output size $N^\alpha/\log N$. The running time of the algorithm is (big-Oh of)

$$\begin{aligned}&|D'|^{1+(m-1)\epsilon-\epsilon'} + |Q_m^\star(D')| \cdot (|D'|^{1-\epsilon}\log|D'|)\\ =&|D'|^{1-\frac{\epsilon'}{m}} \cdot |Q_m^\star(D')|^{(1-\frac{1}{m})}\end{aligned}$$

which breaks the optimality of Lemma 31. □

The lower bound holds for any ranking function. Lemma 32 implies that for star queries, both Theorem 24 and Theorem 25 are optimal. Before concluding this section, we also remark on the question of whether the logarithmic factor that we obtain in the delay guarantee is removable. Prior work [DK21] showed that for the following simple join query $Q = R(x) \bowtie S(y)$ over SUM, there exists no algorithm supporting constant-delay enumeration after linear preprocessing time. Note that this does not rule out a sub-logarithmic delay guarantee, which remains an open problem.

8.5 General queries

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

Theorem 26. *For a join-project query Q, a database instance D and a ranking function* RANK $\in \{$*SUM*, *LEXICOGRAPHIC*$\}$*, the query results $Q(D)$ can be enumerated according to* RANK *with $O(|D|^{fhw}\log|D|)$ delay, after $O(|D|^{fhw}\log|D|)$ preprocessing time.*

We now go one step further and extend our algorithm to queries that are *unions* of join-project queries (UCQs) using an idea introduced by [DK21]. A UCQ query is of the

form $Q = Q_1 \cup Q_2 \cup \cdots \cup Q_m$, where each Q_i is a join-project query defined over the same projection attributes $\mathbf{A}$. Semantically, $Q(D) = \bigcup_i Q_i(D)$. Recent work by Abo Khamis et al. [AKNS17] presents an improved algorithm (called PANDA) that constructs multiple GHDs by partitioning the input database into disjoint pieces and build a GHD for each piece. In this way, the size of materialized subquery can be bounded by $O(|D|^{\texttt{subw}})$, where subw is the *submodular width* [Mar13] of input query Q. Moreover, $\texttt{subw} \leq \texttt{fhw}$ holds generally for query Q, thus improving the previous result on fhw. By using Theorem 24 in conjunction with data-dependent decompositions from PANDA we can immediately obtain the following result:

Theorem 27. *For a join-project query Q, a database instance D and a ranking function* $\texttt{RANK} \in \{\textit{SUM}, \textit{LEXICOGRAPHIC}\}$*, the query results $Q(D)$ can be enumerated according to* RANK *with $O(|D|^{\texttt{subw}} \log |D|)$ delay, after $O(|D|^{\texttt{subw}} \log |D|)$ preprocessing time.*

Example 45. *Consider the 4-cycle (butterfly) query $\pi_{A,C}(R_1(A,B) \bowtie R_2(B,C) \bowtie R_3(C,D) \bowtie R_4(D,A))$ with ranking function $\texttt{RANK}(t) = \pi_A(t) + \pi_C(t)$. With $\texttt{fhw} = 2$, Theorem 27 implies that the query results can be enumerated according to* RANK *with $O(|D|^2 \log |D|)$ delay, after $O(|D|^2)$ preprocessing time. With $\texttt{subw} = \frac{3}{2}$, Theorem 27 implies that query results can be enumerated according to* RANK *with delay $O(|D|^{3/2} \log |D|)$, after $O(|D|^{3/2} \log |D|)$ preprocessing time.*

A note on optimality. The reader may wonder whether the exponent of fhw and subw in Theorem 26 and Theorem 27 are truly necessary. For the triangle query $Q_\triangle(x,y) = R(x,y) \bowtie S(y,z) \bowtie T(z,x)$ which is the simplest cyclic query, $\texttt{fhw} = \texttt{subw} = 3/2$ and even after 30 years, the original AYZ algorithm [AYZ94] that detects the existence of a triangle in $O(|D|^{3/2})$ time still remains the best known combinatorial algorithm. It is widely conjectured [Mar21, AW14, AWY18, KPP16] that there exists no better algorithm. As noted in [AKNS17], the notion of submodular width was suggested as the yardstick for optimality. Indeed, the groundbreaking results by Marx [Mar13] rule out algorithms with better dependence than subw in the exponent for a small class of queries but a general unconditional lower bound still remains out of reach. Thus, any improvement in the exponent would automatically imply a better algorithm for cycle detection since ranked enumeration is at least as hard.

Abboud, Backurs and, Vassilevska Williams presented the Combinatorial $k-$Clique Hypothesis [ABW18].

Definition 9. *Let C be the smallest constant such that a combinatorial algorithm running in $O(n^{Ck/3})$ time exists for detecting a $k-$clique in a n node $O(n^2)$ edge graph, for all sufficiently large constants k. The Combinatorial $k-$Clique Hypothesis states that $C = 3$.*

Next, we recall a cycle detection hypothesis from [LWW18] that depends on the combinatorial $k-$clique hypothesis.

Theorem 28 (Combinatorial Sparse k-Cycle Lower Bound)**.** *Detecting a directed odd $k-$cycle in a graph with n nodes and $m = n^{(k+1)/(k-1)}$ in time $O(m^{(1-\epsilon)2k/(k+1)})$ for any constant $\epsilon > 0$ violates the Combinatorial $k-$Clique Hypothesis.*

Consider an odd $k-$ cycle CQ over binary relations of the form $Q^{\circlearrowleft}(x_1, x_2) = R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie \cdots \bowtie R_k(x_k, x_1)$. Then, we can show the following straightforward result.

Lemma 33. *For some odd k cycle query $Q^{\circlearrowleft}$ with $k \geq 3$, an algorithm that enumerates the query result with delay $O(|D|^{2k/(k+1)-\epsilon})$ after $O(|D|^{2k/(k+1)-\epsilon})$ preprocessing violates the Combinatorial Sparse $k-$Cycle Lower Bound.*

Proof. If the algorithm has delay guarantee $\delta = O(|D|^{2k/(k+1)-\epsilon})$, then in δ time, we can decide whether there exists a cycle or not since the preprocessing time is also $O(|D|^{2k/(k+1)-\epsilon})$, a contradiction. □

Note that the submodular width for the odd $k-$cycle query is $\texttt{subw} = 2 - 2/(k+1) = 2k/(k+1)$. Thus, there exists queries for which the exponential dependence of `subw` in Theorem 27 cannot be avoided assuming the Combinatorial $k-$Clique Hypothesis.

8.6 Experimental Evaluation

In this section, we perform an extensive evaluation of our proposed algorithm. Our goal is to evaluate three aspects: (a) how fast our algorithm is compared to state-of-the-art implementations for both `SUM` and `LEXICOGRAPHIC` ranking functions on various queries and datasets, (b) test the empirical performance of the space-time trade-off in Theorem 25, (c) investigate the performance of our algorithm on various cyclic queries based on different shapes and (d) test the scalability behavior of our algorithm.

8.6.1 Experimental Setup

We use Neo4j 4.2.3 community edition, MariaDB 10.1.47[3] and PostgreSQL 11.12 for our experiments. All experiments are performed on a Cloudlab machine [DRM+19] running Ubuntu 18.04 equipped with two Intel E5-2630 v3 8-core CPUs@2.40 GHz and 128 GB RAM. We focus only on the main memory setting and all experiments run on a single core. Since the join queries are memory intensive, we take special care to ensure that only one DBMS engine is running at a time, restart the session for each query to ensure temp tables in main

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

```
1   DBLP_2hop = SELECT DISTINCT A_1.name, A_2.name FROM Author AS A_1, Author AS
        A_2, AuthorPapers AS AP_1, AuthorPapers as AP_2, Paper AS P WHERE
        AP_1.pid = AP_2.pid AND AP_1.aid = A_1.aid AND AP_2.aid = A_2.aid AND
        P.is_research =true ORDER BY A_1.weight + A_2.weight LIMIT k
2
3   DBLP_3hop = SELECT DISTINCT A.name, P.name FROM Author AS A, Paper AS P,
        AuthorPapers AS AP_1, AuthorPapers as AP_2, AuthorPapers as AP_3 WHERE
        AP_1.pid = AP_2.pid AND AP_2.aid = AP_3.aid AND AP_1.aid = A.aid AND
        AP_3.pid = P.pid AND P.is_research =true ORDER BY A.weight + P.weight LIMIT k
4
5   DBLP_4hop = SELECT DISTINCT A_1.name, A_2.name FROM Author AS A_1, Author AS
        A_2, AuthorPapers AS AP_1, AuthorPapers as AP_2, AuthorPapers as AP_3,
        AuthorPapers as AP_4, Paper AS P_1, Paper AS P_2 WHERE AP_1.pid = AP_2.pid
        AND AP_2.aid = AP_3.aid AND AP_3.pid = AP_4.pid AND AP_3.pid = P_2.pid AND
        AP_1.pid = P_1.pid AND AP_1.aid = A_1.aid AND AP_4.aid = A_2.aid AND P_1.is_research =
        true AND P_2.is_research =true ORDER BY A_1.weight + A2.weight LIMIT k
6
7   DBLP_3star = SELECT DISTINCT A_1.name, A_2.name, A_3.name FROM Author AS A_1,
        Author AS A_2, Author AS A_3, AuthorPapers AS AP_1, AuthorPapers as AP_2,
        AuthorPapers as AP_3, Paper AS P WHERE AP_1.pid = AP_2.pid = AP_3.pid AND
        AP_1.aid = A_1.aid AND AP_2.aid = A_2.aid AND AP_3.aid = A_3.aid AND AP_3.pid = P.pid
         AND P.is_research =true ORDER BY A_1.weight + A_2.weight + A_3.weight LIMIT k
```

Figure 8.3: Network analysis queries for DBLP. Queries for IMDB are defined similarly.

memory are flushed out to avoid any interference, and also monitor that no temp tables are created on the disk. We only keep one database containing a single relation when performing experiments. We switch off all logging to avoid any performance impact. For PostgreSQL and MariaDB, we allow the engines to use the full main memory to ensure all temp tables are resident in the RAM and sorting (if any) happens without any disk IOs by increasing the sort buffer limit. For Neo4j, we allow the JVM heap to use the full main memory at the time of start-up. We also build bidirectional B-tree indexes for each relation ahead of time and create named indexes in Neo4j. All of our algorithms are implemented in C++ and compiled using the GNU C++ 7.5.0 compiler that ships with Ubuntu 18.04. Each experiment is run 5 times and we report the median after removing the slowest and the fastest run.

8.6.1.1 Small-Scale Datasets

We use two real-world small scale datasets for our experiments: the DBLP dataset, containing the relationship between authors and papers, and the IMDB dataset, containing the

relationship between actors, directors, and movies. We use these datasets for two reasons: (i) both datasets are useful and studied extensively in practical problems such as similarity search [YSN+12], citation graph analysis [RT05], and network analysis [EL05, Bir08]. (ii) small-scale datasets allow experiments to finish for all systems allowing us to make a fair comparison and develop a fine-grained understanding. In line with prior work [KGS+20], for each tuple we assign the weight attribute (and add it to the table schema) in two ways: first, we assign a randomly chosen value, and second, logarithmic weights in which the weight of the entity (author and paper in DBLP) v is $\log_2(1 + deg_v)$, where deg_v denotes its degree in the relation. The schema for both datasets is as shown below (underlined attributes are primary keys for the relation):

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

Queries. We consider 4 acyclic join queries as shown in Figure 8.3 for the small-scale datasets, which are commonly seen in practice [SH13, BMT20]. Intuitively, the first three queries find all the top-k weighted 2-hops, 3-hops, and 4-hops reachable attribute pairs within the DBLP network. As remarked by in [SH13, CLYZ18, KSKÇ12], these queries are of immense practical interest (e.g., see Table 4 in [SH13]). Queries for IMDB dataset are defined similarly. In Subsection 8.6.2.2, we also investigate the performance for cyclic queries.

8.6.1.2 Large-Scale Datasets

We also perform experiments on two real-world large-scale relational datasets and one relational benchmark. The first dataset is from the Friendster [LK14] online social network that contains $1.8B$ tuples. In the social network each, the user is associated with multiple groups. The second dataset is the Memetracker [LK14] dataset which describes user generated memes and which users have interacted with the meme. The dataset contains $418M$ tuples. For both Friendster and Memetracker, we use weights for users as the number of groups they belong to and the number of memes they create respectively. Finally, we also use the queries containing a ranking function from the LDBC Social Network Benchmark [EALP+15] with scale factor SF = 10, a publicly available benchmark, to perform scalability experiments.

Queries. For Friendster and Memetracker, we use two popular queries that are used in network analysis. Similar to the DBLP queries, we identify the ranked user pairs in the two-hop and three-hop neighborhoods for all users. The ranking is the sum of the weights of the user pair. These queries have a widespread application in understanding information

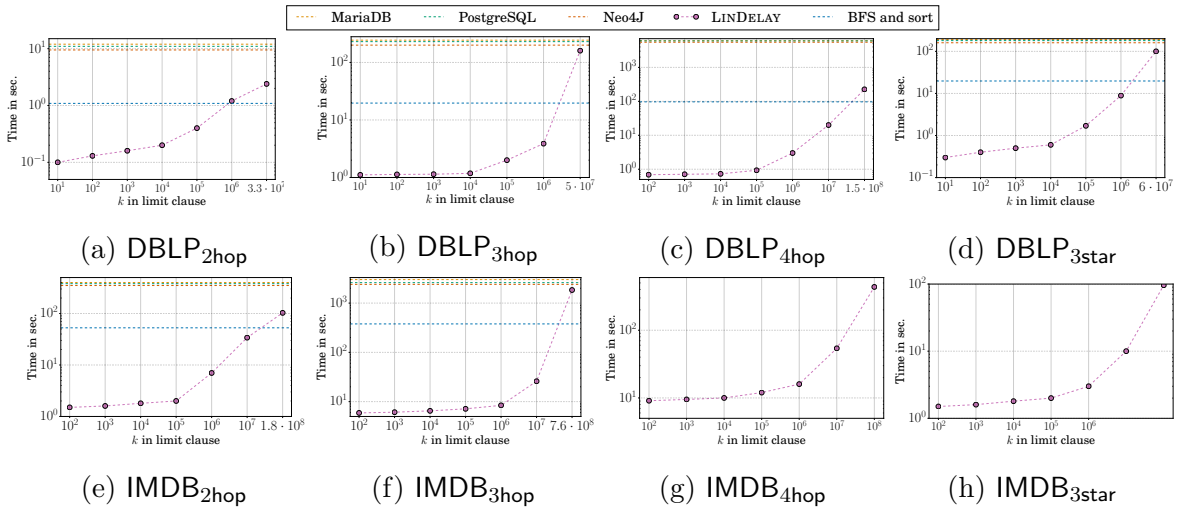
flow in a network [MSGL14] and are used in recommendation systems [FLLQ19, LFZ19]. For the LDBC benchmark, we use the multi-source version of **Q3, Q10** and **Q11**. Each of these queries is a variant of the neighborhood analysis and contains `UNION`.

8.6.2 Small Scale Experiments

In this section, we compare the empirical performance of the algorithm given by Theorem 24 (labeled as LinDelay in all figures) against the baselines for each query. To perform a fair comparison, we materialize the top-k answers in-memory since other engines also do it. However, a strength of our system is that if a downstream task only requires the output as a stream, we can enumerate the result instead of materializing it, which is not possible with other engines.

Sum ordering. Figure 8.4 shows the main results for the DBLP and IMDB datasets when the ranking function is the sum function and the weights are chosen randomly. Let us first review the results for the DBLP dataset. Figure 8.4a shows the running time for different values of k in the limit clause. The first observation is that all engines materialize the join result, followed by deduplicating and sorting according to the ranking function which leads to poor performance for all baselines. This is because all engines treat sorting and distinct clause as blocking operators, verified by examining the query plan. On the other hand, our approach is limit-aware. For small values of k, we are up to two orders of magnitude faster and as the value of k increases, the total running time of our algorithm increases linearly. Even when our algorithm has to enumerate and materialize the entire result, it is still faster than asking the engines for the top-10 results. This is a direct benefit of generating the output in deduplicated and ranked order. As the path length increases from two to three and four paths (Figure 8.4b and Figure 8.4c), the performance gap between existing engines and our approach also becomes larger. We also point out that all engines require a large amount of main memory for query execution. For example, MariaDB requires about 40GB of memory for executing $\mathsf{DBLP}_{\mathsf{4hop}}$. In contrast, the space overhead of our algorithm is dominated by the size of the priority queue. For DBLP dataset, our approach requires a measly 1.3GB, 4GB, 3GB and 2.7GB total space for $\mathsf{DBLP}_{\mathsf{2hop}}, \mathsf{DBLP}_{\mathsf{3hop}}, \mathsf{DBLP}_{\mathsf{4hop}}$ and $\mathsf{DBLP}_{\mathsf{3star}}$ respectively. For $\mathsf{DBLP}_{\mathsf{3hop}}, \mathsf{DBLP}_{\mathsf{4hop}}$ and $\mathsf{DBLP}_{\mathsf{3star}}$, we also implement breadth-first search (BFS) followed by a sorting step using the idea of algorithm 19. As it can be seen from the figures, BFS and sort provide an intermediate strategy which is faster than our algorithm for large values of k but at the cost of expensive materialization of the entire result, which may not be always possible (and is the case for IMDB dataset). However, deciding to use BFS and sort requires knowledge of the output result size, which is unknown apriori and difficult to estimate. For the IMDB dataset, we observe a similar trend of our algorithm displaying superior performance compared to all other baselines. In this case, BFS

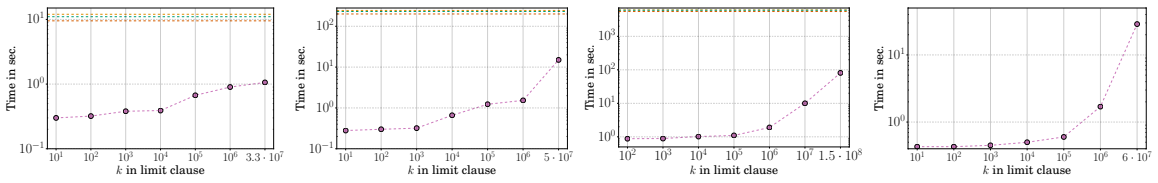
and sorting even for $\mathsf{DBLP}_{\mathsf{4hop}}$ is not possible since the result is almost 0.5 trillion items. For $\mathsf{DBLP}_{\mathsf{3star}}$, none of the engines were able to compute the result after running for 5 hours when main memory ran out. BFS and sort also failed due to the size being larger than the main memory limit. Lastly, Neo4j was consistently the best performing (albeit marginally) engine among all baselines. While there is little scope for rewriting the SQL queries to try to obtain better performance, Neo4j has graph-specific operators such as variable-length expansion. We tested multiple rewritings of the query, which leads to different query plans, to obtain the best performance (although this is the job of the query optimizer), which is finally reported in the figures. Regardless of the rewritings, Neo4j still treats materializing and sorting as a blocking operator which is a fundamental bottleneck.



(a) $\mathsf{DBLP}_{\mathsf{2hop}}$ (b) $\mathsf{DBLP}_{\mathsf{3hop}}$ (c) $\mathsf{DBLP}_{\mathsf{4hop}}$ (d) $\mathsf{DBLP}_{\mathsf{3star}}$

(e) $\mathsf{IMDB}_{\mathsf{2hop}}$ (f) $\mathsf{IMDB}_{\mathsf{3hop}}$ (g) $\mathsf{IMDB}_{\mathsf{4hop}}$ (h) $\mathsf{IMDB}_{\mathsf{3star}}$

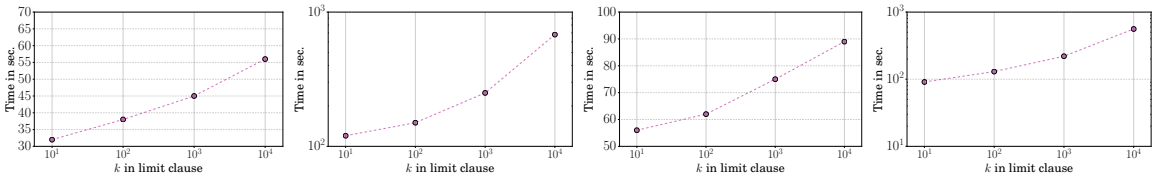
Figure 8.4: Comparing our algorithm with state-of-the-art engines for sum function



(a) $\mathsf{DBLP}_{\mathsf{2hop}}$ (b) $\mathsf{DBLP}_{\mathsf{3hop}}$ (c) $\mathsf{DBLP}_{\mathsf{4hop}}$ (d) $\mathsf{DBLP}_{\mathsf{3star}}$

Figure 8.5: Comparing our linear delay algorithm with state-of-the-art engines for lexicographic function.

Lexicographic ordering. Figures 8.5a, 8.5b, 8.5c and 8.5d show the running time for different values of k in the limit clause for lexicographic ranking function on DBLP (i.e. we replace $\mathsf{A}_1.\mathsf{weight} + \mathsf{A}_2.\mathsf{weight}$ with $\mathsf{A}_1.\mathsf{weight}, \mathsf{A}_2.\mathsf{weight}$ in the `ORDER BY` clause) for random weights. The first striking observation here is that the running time for all baseline engines is identical to that of the sum function. This demonstrates that existing engines are also



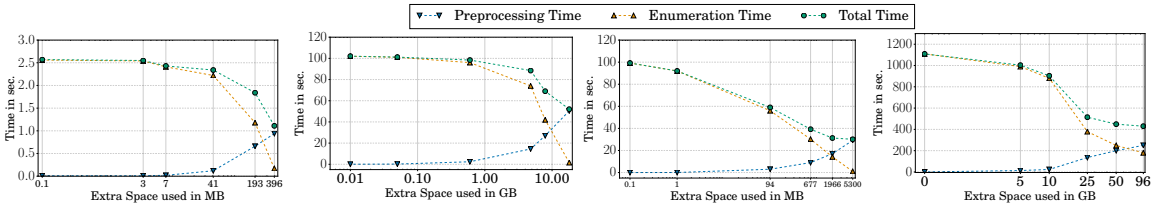
(a) Memetracker 2-neighborhood (b) Memetracker 3-neighborhood (c) Friendster 2-neighborhood (d) Friendster 3-neighborhood

Figure 8.6: LinDelay performance on large-scale datasets

agnostic to the ranking function in the query and fail to take advantage of the additional structure. However, lexicographic functions are easier to handle in practice than the sum because we can avoid the use of a priority queue altogether. This in turn leads to faster running time since the push and pops from the priority queue are expensive due to the logarithmic overhead and need for re-balancing of the tree structure. Thus, we obtain a $2\times$ improvement for lexicographic ordering as compared to the sum function.

Join ordering. At this point, the reader may wonder what is the impact of different join orderings on the query execution time for DBMS engines in the presence of `ORDER BY`. To investigate this, we supply join order hints to each of the engines. We run the queries on all possible join order hints to find the best possible running time. We found that the join order hints had virtually no impact on execution time. For instance, $\mathsf{DBLP}_{\mathsf{4hop}}$ on Neo4J takes $5521.61s$ without any join hints and the best possible join ordering reduces the time to $5418.23s$, a mere 1.8% reduction. This is not surprising since the bottleneck for all engines is the materialization of the unsorted output, which is orders of magnitude larger than the final output and ends up being the dominant cost. In fact, for queries containing only self-joins, join order hints do not have any impact on the query plan because all relations are identical. For instance, all join orderings for the query $Q(x_1, x_2, x_3) = R(x_1, y) \bowtie R(x_2, y) \bowtie R(x_3, y)$ will lead to identical query plans. Further, the number of possible join orderings that may need to be explored is exponential in the number of relations. On the other hand, our algorithm has the advantage of bypassing the materialization due to the delay-based problem formulation and use of multi-way joins.

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



(a) $\mathsf{DBLP}_{\mathsf{2hop}}$ (b) $\mathsf{IMDB}_{\mathsf{2hop}}$ (c) $\mathsf{DBLP}_{\mathsf{3star}}$ (d) $\mathsf{IMDB}_{\mathsf{3star}}$

Figure 8.7: Comparing the preprocessing and enumeration trade-off for sum function when enumerating the entire result

8.6.2.1 Enumeration with Preprocessing

We next investigate the empirical performance of the preprocessing step and its impact on the result enumeration as described by Theorem 25. For all experiments in this section, we fix k to be large enough to enumerate the entire result (which is equivalent to having no limit clause at all).

Sum ordering. Figure 8.7a and Figure 8.7b show the trade-off between space used by the data structure constructed in the preprocessing phase and running time of the enumeration algorithm for $\mathsf{DBLP}_{\mathsf{2hop}}$ and $\mathsf{IMDB}_{\mathsf{2hop}}$ respectively. We show the trade-off for 6 different space budgets but the user is free to choose any space budget in the entire spectrum. As expected, the time required to enumerate the result is large when there is no preprocessing and it gradually drops as more and more results are materialized in the preprocessing phase. The sum of preprocessing time and enumeration time is not a flat line: this is because as an optimization, we do not use priority queues in the preprocessing phase. Instead, we can simply use the BFS and sort algorithm for all chosen nodes which need to be materialized. This is a faster approach in practice as we avoid the use of priority queues but priority queues cannot be avoided for the enumeration phase. We observe similar trend for $\mathsf{DBLP}_{\mathsf{3star}}, \mathsf{IMDB}_{\mathsf{3star}}$ on both datasets as well.

8.6.2.2 Cyclic Queries

We also compare the performance of our algorithm to other systems for cyclic queries. We choose four cyclic queries found commonly in practice inspired by [TGR20]: four-cycle, six-cycle, eight-cycle, and bowtie query (two four cycles joined at a common attribute). Figure 8.8a shows the performance of our algorithm on the DBLP dataset for the sum function. As the table shows, our algorithm can process all queries within 200 seconds, with the bowtie query being the most computationally intensive. In contrast, for $k = 10$ the fastest performing engine Neo4J required 240s (450s) for four-cycle (six cycle). It did not finish execution for eight cycle and bowtie queries due to an out-of-memory error. For the IMDB dataset,

our algorithm was able to process all queries, while Neo4J was not able to process any query (except four-cycle) due to its large memory requirement..

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

(a) Cyclic query performance on the DBLP dataset for different values of k in the LIMIT clause.

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

(b) Scalability for different scale factors (SF) in LDBC

Figure 8.8: Cyclic query performance and scalability for LDBC

8.6.3 Large Scale Experiments and Scalability

In this section, we investigate the performance of our techniques on large-scale datasets. Figure 8.6a and 8.6b shows the time to find the top-k answers for the Memetracker dataset on two neighborhood and three neighborhood queries. Compared to the small-scale datasets, the execution time increases rapidly even for low values of k. This is attributed to the high duplication of answers, which leads to a rapidly increasing priority queue size. *None of MariaDB, Postgres, and Neo4J were able to finish, or even to find the top*-10 *answers, within* 5 *hours in our experiments.* The same trend is also observed for the Friendster dataset as shown in Figure 8.6d and 8.6c. Similar to the small-scale datasets, lexicographic functions were faster than the sum function for our algorithm but DBMS engines were unable to finish query execution.

Scalability. We also conduct scalability experiments on LDBC benchmark queries that contain the `ORDER BY` clause. Figure 8.8b shows the scalability of our algorithm for finding answers of queries **Q3, Q10, Q11**. As the scale factor of the dataset increases, the execution time also increases linearly. For each of these queries, all engines require more than 3 hours to compute the result even for SF = 10 and $k = 10$. This is because of the serial execution plan generated by the engines, which forces the materialization of the unsorted result before sorting and filtering for top-k.

Chapter 9

Conclusions

We conclude the dissertation with a summary of the main results and point at some open problems following our work.

The overall goal of the dissertation is to understand how the structure of the database and query, along with the task at hand, can be used to identify fine-grained materialization strategies to speed up data analytics. In particular, we studied all problems in the setting where we have preprocessing phase that allows us to develop a sophisticated data structure and an enumeration phase that uses the data structure to efficiently complete the task.

We first introduced the novel notion of adorned queries over arbitrary full CQs and developed a data structure that allowed us to tradeoff the space usage with the answering time/delay guarantee of the query. Then, we combined the data structure with tree decompositions to obtain optimal (conjectured) space usage.

Next, using the formalism of adorned queries over Boolean CQs, we showed how existing problems studied in the algorithmic community can be cast into our framework. The advantage of doing so is twofold. First, this insight allows us to recover state-of-the-art results for many problems into a single unified framework. Second, viewing the problems as CQs enables us to apply the results already established in the database community, allowing us to falsify proposed conjectures in the literature by identifying unexpectedly better algorithms. Further, new lower bounds were established for star queries and path queries.

We then pivoted our attention to the more traditional task of join query evaluation. Very few prior works studied the problem of join query evaluation that takes the projection attributes into account. For star and path queries, an important class of queries often seen in practice, we proposed novel algorithms that are sensitive to the output using the novel idea of interleaved query processing. Remarkably, this idea is powerful enough to improve upon state-of-the-art tradeoffs that were recently proposed in the literature. Using fast matrix multiplication, we were further able to push the boundaries and improve the established tradeoffs.

Traditionally, fast matrix multiplication has been thought of as a technique only of theoretical interest due to the large constant involved in the algorithm. However, recent advancement in processor architecture has enabled the development of incredibly fast libraries that have dramatically improved the performance of matrix multiplication. One could view this improvement as a potentially smaller ω, the matrix multiplication exponent. We investigate how matrix multiplication can be used to improve the big Oh running time complexity of star queries. In doing so, we identify two errors in prior work that lead to incorrect results. We fix and generalize the analysis, and show how fast matrix multiplication can be applied to important problems such as set similarity and set containment. Finally, we also undertake an experimental investigation to understand the practical benefits. Our experiments indicate that orders of magnitude performance improvement are possible when the input dataset contains a dense component, a condition that is easy to identify.

The last part of this dissertation is dedicated to the study of ranked enumeration for arbitrary CQs. For full CQs, we thoroughly resolve an open problem stated at the Dagshtuhl Seminar 19211 on ranked enumeration. We obtain logarithmic delay guarantees for a large practical class of ranking functions and provide evidence for the near optimality of our results. Remarkably, we were able to adapt the algorithm to also include arbitrary CQs that may contain projections. Lastly, an extensive experimental evaluation shows that our approach is an order of magnitude faster while using lesser space than state-of-the-art engines. This demonstrates that delay-based problem formulation is not only theoretically appealing but also the optimal solution in practice.

9.1 Future Work

We describe some directions for future work next.

Unconditional lower bounds. Throughout the dissertation, most of our lower bounds are conditional. For space-time tradeoffs, can we develop unconditional lower bounds, even if for the more restrictive pointer machine model? For ranked enumeration of full and join-project CQs, are there lower bounds that can be obtained for a broader set of ranking functions? Can we prove that the logarithmic and linear delay guarantee for full and join-project CQs respectively, cannot be improved? Is it possible to rule out constant answering time for 2-Set Disjointness using subquadratic space?

More general space-time bounds. The first question is to study the tradeoff between space vs answering time (and delay guarantees) for arbitrary non-boolean 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 better answering time guarantees. It

would be interesting to investigate whether our ideas can be applied to existing algorithms for constructing distance oracles to improve their space requirement.

Ranked Enumeration. There remain several open questions regarding how the structure of ranking functions influences the efficiency of the algorithms. In particular, it would be interesting to find fine-grained classes of ranking functions that are more expressive than totally decomposable, but less expressive than coordinate decomposable. For instance, the ranking function $f(x, y) = |x - y|$ is not coordinate decomposable, but it is piecewise coordinate decomposable on either side of the global minimum critical point for each x valuation. Finally, recent work has made considerable progress in query evaluation under updates. In this setting, the goal is to minimize the update time of the data structure as well as minimize the delay. A simple application of our algorithm is useful here. For any full acyclic query, one can maintain the relations under updates in constant time by updating the hash maps and then applying the preprocessing and enumeration phase of our algorithm. This algorithm gives a linear delay guarantee since the preprocessing phase takes linear time. One could also apply the preprocessing phase of our algorithm after each update to reset all priority queues which makes the update time linear but the enumeration delay can now be $O(\log |D|)$. For join-project queries, the first important problem is to extend our results from the main memory setting to the distributed setting. A key challenge here is to theoretically analyze how to achieve optimal load balancing to minimize the communication cost. Since the cost of I/O must also be taken into account, it becomes important to identify the optimal priority queue storage layout to ensure that access cost is low. It would also be interesting to develop output-balanced algorithms. The second exciting challenge is to incorporate approximation into the ranking. For some applications, it may be sufficient to get an approximately ordered output which could lead to improved running time guarantees. Finally, it would be useful to re-rank the query results when the ranking function is changed by the user and extend our ideas to non-monotone ranking functions.

Reducing Space Requirements. For ranked enumeration, as the number of enumerated answers grows, so does the priority queue size. An interesting question is whether we can improve upon the large use of memory while achieving the same time bounds.

Performance in practice. This dissertation consists of a mix of theoretical and empirical research. We saw in Section 6.5 how matrix multiplication can be useful in practice and Section 8.6 shows the performance improvement over database engines for ranked enumeration of join-project queries. Yet, there is still much room to perform extensive practical research. We would like to convert the space-time tradeoffs developed in Chapter 3 into a practical algorithm that can autotune its space usage on-the-fly based on the changing distribution of the query workload. There likely exists several research challenges that include translating

the theoretical data structure into an efficient one that uses bit manipulation to minimize its memory footprint, tightly integrating it with set intersection algorithms, and combining it with known techniques such as bitmap indexes.

LIST OF REFERENCES

[ABW18] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the current clique algorithms are optimal, so is valiant's parser. *SIAM Journal on Computing*, 47(6):2527–2555, 2018.

[ACFR20] Shimaa Ahmed, Amrita Roy Chowdhury, Kassem Fawaz, and Parmesh Ramanathan. Preech: a system for privacy-preserving speech transcription. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2703–2720, 2020.

[AG11] Renzo Angles and Claudio Gutierrez. Subqueries in sparql. *AMW*, 749:12, 2011.

[Aga14] Rachit Agarwal. The space-stretch-time tradeoff in distance oracles. In *ESA*, pages 49–60. Springer, 2014.

[AGHP11] Rachit Agarwal, P Brighten Godfrey, and Sariel Har-Peled. Approximate distance queries and compact routing in sparse graphs. In *INFOCOM*, pages 1754–1762. IEEE, 2011.

[AGM13] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

[AHS+15] Ibrahim Abdelaziz, Razen Harbi, Semih Salihoglu, Panos Kalnis, and Nikos Mamoulis. Spartex: A vertex-centric framework for rdf data analytics. *Proceedings of the VLDB Endowment*, 8(12):1880–1883, 2015.

[AKKNS20] Mahmoud Abo Khamis, Phokion G Kolaitis, Hung Q Ngo, and Dan Suciu. Decision problems in information theory. In *ICALP*, 2020.

[AKNR16] Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. Faq: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 13–28, 2016.

[AKNS17] Mahmoud Abo Khamis, Hung Q Ngo, and Dan Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 429–444. ACM, 2017.

[ALOR18] Christopher Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. Levelheaded: A unified engine for business intelligence and linear algebra querying. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 449–460. IEEE, 2018.

[ALT+17] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):20, 2017.

[AMF06] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 671–682, 2006.

[AN16] Peyman Afshani and Jesper Asbjørn Sindahl Nielsen. Data structure lower bounds for document indexing problems. In *ICALP 2016Automata, Languages and Programming*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik GmbH, 2016.

[AP09] Rasmus Resen Amossen and Rasmus Pagh. Faster join-projects and sparse matrix multiplications. In *Proceedings of the 12th International Conference on Database Theory*, pages 121–126. ACM, 2009.

[APV11] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Best position algorithms for efficient top-k query processing. *Information Systems*, 36(6):973–989, 2011.

[AW14] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 434–443. IEEE, 2014.

[AWY18] Amir Abboud, Virginia Vassilevska Williams, and Huacheng Yu. Matching triangles and basing hardness on an extremely popular conjecture. *SIAM Journal on Computing*, 47(3):1098–1122, 2018.

[AYZ94] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. In *European Symposium on Algorithms*, pages 354–364. Springer, 1994.

[AYZ97] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

[BB12] Johann Brault-Baron. A negative conjunctive query is easy if and only if it is beta-acyclic. *Computer Science Logic 2012*, page 137, 2012.

[BB13] Johann Brault-Baron. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. PhD thesis, Université de Caen, 2013.

[BCD+06] David Bremner, Timothy M Chan, Erik D Demaine, Jeff Erickson, Ferran Hurtado, John Iacono, Stefan Langerman, and Perouz Taslakian. Necklaces, convolutions, and x+ y. In *European Symposium on Algorithms*, pages 160–171. Springer, 2006.

[BDG07a] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*, pages 208–222. Springer, 2007.

[BDG07b] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, pages 208–222, 2007.

[BGI97] Gautam Bhargava, Piyush Goel, and Balakrishna Ragmavendra Iyer. Enumerating projections in sql queries containing outer and full outer joins in the presence of inner joins, November 11 1997. US Patent 5,687,362.

[BGJR21] Pierre Bourhis, Alejandro Grez, Louis Jachiet, and Cristian Riveros. Ranked enumeration of mso logic on words. *ICDT*, 2021.

[Bir08] Maria Biryukov. Co-author network analysis in dblp: Classifying personal names. In *International Conference on Modelling, Computation and Optimization in Information Systems and Management Sciences*, pages 399–408. Springer, 2008.

[BKOZ13] Nurzhan Bakibayev, Tomáš Kočiskỳ, Dan Olteanu, and Jakub Závodnỳ. Aggregation and ordering in factorised databases. *Proceedings of the VLDB Endowment*, 6(14):1990–2001, 2013.

[BKPS19] Endre Boros, Benny Kimelfeld, Reinhard Pichler, and Nicole Schweikardt. Enumeration in Data Management (Dagstuhl Seminar 19211). *Dagstuhl Reports*, 9(5):89–109, 2019.

[BKS17a] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 303–318. ACM, 2017.

[BKS17b] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 303–318. ACM, 2017.

[BKS18] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering fo+ mod queries under updates on bounded degree databases. *ACM Transactions on Database Systems (TODS)*, 43(2):7, 2018.

[BMGT16] Panagiotis Bouros, Nikos Mamoulis, Shen Ge, and Manolis Terrovitis. Set containment join revisited. *Knowledge and Information Systems*, 49(1):375–402, 2016.

[BMS+06] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Christian Theobalt, and Gerhard Weikum. Io-top-k: Index-access optimized top-k query processing. 2006.

[BMT20] Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large sparql query logs. *The VLDB Journal*, 29(2):655–679, 2020.

[BOZ12] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodnỳ. Fdb: A query engine for factorised relational databases. *Proceedings of the VLDB Endowment*, 5(11):1232–1243, 2012.

[CC62] Abraham Charnes and William W Cooper. Programming with linear fractional functionals. *Naval Research Logistics (NRL)*, 9(3-4):181–186, 1962.

[CCC+20] Prasad Chalasani, Jiefeng Chen, Amrita Roy Chowdhury, Xi Wu, and Somesh Jha. Concise explanations of neural networks using adversarial training. In *International Conference on Machine Learning*, pages 1383–1391. PMLR, 2020.

[CCW+21] Yunang Chen, Amrita Roy Chowdhury, Ruizhe Wang, Andrei Sabelfeld, Rahul Chatterjee, and Earlence Fernandes. Data privacy in trigger-action systems. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 501–518, 2021.

[CFZ07] Olivier Corby and Catherine Faron-Zucker. Implementation of sparql query language based on graph homomorphism. In *International Conference on Conceptual Structures*, pages 472–475. Springer, 2007.

[CG85] Stefano Ceri and Georg Gottlob. Translating sql into relational algebra: Optimization, semantics, and equivalence of sql queries. *IEEE Transactions on software engineering*, (4):324–345, 1985.

[CK19] Nofar Carmeli and Markus Kröll. On the enumeration complexity of unions of conjunctive queries. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 134–148, 2019.

[CL15] Timothy M Chan and Moshe Lewenstein. Clustered integer 3sum via additive combinatorics. In *STOC*, pages 31–40, 2015.

[CLRS09a] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Chapter 8.2, Introduction to algorithms*. MIT press, 2009.

[CLRS09b] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[CLYZ18] Jinpeng Chen, Yu Liu, Guang Yang, and Ming Zou. Inferring tag co-occurrence relationship across heterogeneous social networks. *Applied Soft Computing*, 66:512–524, 2018.

[CLZ+15] Lijun Chang, Xuemin Lin, Wenjie Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. Optimal enumeration: Efficient top-k tree matching. *Proceedings of the VLDB Endowment*, 8(5):533–544, 2015.

[CO15] Radu Ciucanu and Dan Olteanu. Worst-case optimal join at a time. Technical report, Technical report, Oxford, 2015.

[CP10a] Hagai Cohen and Ely Porat. Fast set intersection and two-patterns matching. *Theoretical Computer Science*, 411(40-42):3795–3800, 2010.

[CP10b] Hagai Cohen and Ely Porat. On the hardness of distance oracle for sparse graph. *arXiv preprint arXiv:1006.1117*, 2010.

[CRJ20] Amrita Roy Chowdhury, Theodoros Rekatsinas, and Somesh Jha. Data-dependent differentially private parameter learning for directed graphical models. In *International Conference on Machine Learning*, pages 1939–1951. PMLR, 2020.

[CRW21] Philipp Christmann, Rishiraj Saha Roy, and Gerhard Weikum. Efficient contextualization using top-k operators for question answering over knowledge graphs. *arXiv preprint arXiv:2108.08597*, 2021.

[CS07] Sara Cohen and Yehoshua Sagiv. An incremental algorithm for computing ranked full disjunctions. *Journal of Computer and System Sciences*, 73(4):648–668, 2007.

[CY+11] Rada Chirkova, Jun Yang, et al. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2011.

[CYZ+08] Chen Chen, Xifeng Yan, Feida Zhu, Jiawei Han, and S Yu Philip. Graph olap: Towards online analytical processing on graphs. In *2008 eighth IEEE international conference on data mining*, pages 103–112. IEEE, 2008.

[dbl] DBLP. https://dblp.uni-trier.de/.

[DHK20] Shaleen Deep, Xiao Hu, and Paraschos Koutris. Fast join project query evaluation using matrix multiplication. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1213–1223, 2020.

[DHK21] Shaleen Deep, Xiao Hu, and Paraschos Koutris. Enumeration algorithms for conjunctive queries with projection. In *24th International Conference on Database Theory (ICDT 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[DK17] Shaleen Deep and Paraschos Koutris. Compressed representations of conjunctive query results. *arXiv preprint arXiv:1709.06186*, 2017.

[DK18] Shaleen Deep and Paraschos Koutris. Compressed representations of conjunctive query results. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 307–322. ACM, 2018.

[DK21] Shaleen Deep and Paraschos Koutris. Ranked enumeration of conjunctive query results. *To appear in Joint 2021 EDBT/ICDT Conferences, ICDT '21 Proceedings*, 2021.

[DM14] Arnaud Durand and Stefan Mengel. The complexity of weighted counting for acyclic conjunctive queries. *Journal of Computer and System Sciences*, 80(1):277–296, 2014.

[DM15] Arnaud Durand and Stefan Mengel. Structural tractability of counting of solutions to conjunctive queries. *Theory of Computing Systems*, 57(4):1202–1249, 2015.

[DO05] Erik D Demaine and Joseph O'Rourke. Open problems from cccg 2005. In *Canadian Conference on Computational Geometry*, pages 75–80, 2005.

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

[DS07] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, 2007.

[DSP20] Harshad Deshmukh, Bruhathi Sundarmurthy, and Jignesh M Patel. To pipeline or not to pipeline, that is the question. *arXiv preprint arXiv:2002.00866*, 2020.

[DTL18] Dong Deng, Yufei Tao, and Guoliang Li. Overlap set similarity joins with theoretical guarantees. In *Proceedings of the 2018 International Conference on Management of Data*, pages 905–920. ACM, 2018.

[EALP+15] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 619–630, 2015.

[EL05] Ergin Elmacioglu and Dongwon Lee. On six degrees of separation in dblp-db and more. *ACM SIGMOD Record*, 34(2):33–40, 2005.

[Fag02] Ronald Fagin. Combining fuzzy information: an overview. *ACM SIGMOD Record*, 31(2):109–118, 2002.

[FFG02] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *Journal of the ACM (JACM)*, 49(6):716–752, 2002.

[FLLQ19] Chenyuan Feng, Zuozhu Liu, Shaowei Lin, and Tony QS Quek. Attention-based graph convolutional network for recommendation system. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7560–7564. IEEE, 2019.

[FLN03] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences*, 66(4):614–656, 2003.

[Fre76] Michael L Fredman. How good is the information theory bound in sorting? *Theoretical Computer Science*, 1(4):355–361, 1976.

[Fri04] Ehud Friedgut. Hypergraphs, entropy, and inequalities. *The American Mathematical Monthly*, 111(9):749–760, 2004.

[FT87] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[GGS14] Georg Gottlob, Gianluigi Greco, and Francesco Scarcello. Treewidth and hypertree width. *Tractability: Practical Approaches to Hard Problems*, 1, 2014.

[GGY+14] Manish Gupta, Jing Gao, Xifeng Yan, Hasan Cam, and Jiawei Han. Top-k interesting subgraph discovery in information networks. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 820–831. IEEE, 2014.

[GHL+13] Todd J Green, Shan Shan Huang, Boon Thau Loo, Wenchao Zhou, et al. *Datalog and recursive query processing*. Now Publishers, 2013.

[GHQ95a] Ashish Gupta, Venkatesh Harinarayan, and Dallan Quass. Generalized projections: a powerful approach to aggregation. Technical report, Stanford InfoLab, 1995.

[GHQ95b] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. 1995.

[GJ+10] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[GKLP17] Isaac Goldstein, Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Conditional lower bounds for space/time tradeoffs. In *Workshop on Algorithms and Data Structures*, pages 421–436. Springer, 2017.

[GKS11] Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. Optimizing and parallelizing ranked enumeration. *Proceedings of the VLDB Endowment*, 4(11):1028–1039, 2011.

[GS13] Gianluigi Greco and Francesco Scarcello. Structural tractability of enumerating csp solutions. *Constraints*, 18(1):38–74, 2013.

[GU18] François Le Gall and Florent 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.

[HP98] Xiaohan Huang and Victor Y Pan. Fast rectangular matrix multiplication and applications. *Journal of complexity*, 14(2):257–299, 1998.

[HS05] Stephen Harris and Nigel Shadbolt. Sparql query processing with conventional relational database systems. In *International Conference on Web Information Systems Engineering*, pages 235–244. Springer, 2005.

[HSK98] Jiawei Han, Nebojsa Stefanovic, and Krzysztof Koperski. Selective materialization: An efficient method for spatial data cube construction. In *Pacific-Asia conference on knowledge discovery and data mining*, pages 144–158. Springer, 1998.

[IBS08] Ihab F Ilyas, George Beskales, and Mohamed A Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.

[ima] image. https://cs.stanford.edu/~acoates/stl10/.

[int09] *Intel Math Kernel Library. Reference Manual*. Intel Corporation, Santa Clara, USA, 2009. ISBN 630813-054US.

[ISA+04] Ihab F Ilyas, Rahul Shah, Walid G Aref, Jeffrey Scott Vitter, and Ahmed K Elmagarmid. Rank-aware query optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 203–214. ACM, 2004.

[jok] Jokes. https://goldberg.berkeley.edu/jester-data/.

[JP05] Ravindranath Jampani and Vikram Pudi. Using prefix-trees for efficiently computing set joins. In *International Conference on Database Systems for Advanced Applications*, pages 761–772. Springer, 2005.

[Kaz13] Wojciech Kazana. *Query evaluation with constant delay*. PhD thesis, 2013.

[KGS+20] Mehdi Kargar, Lukasz Golab, Divesh Srivastava, Jaroslaw Szlichta, and Morteza Zihayat. Effective keyword search over weighted graphs. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[KNF+12] Alfons Kemper, Thomas Neumann, Florian Funke, Viktor Leis, and Henrik Mühe. Hyper: Adapting columnar main-memory data management for transactional and query processing. *IEEE Data Eng. Bull.*, 35(1):46–51, 2012.

[KNN+19] Ahmet Kara, Hung Q Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Counting triangles under updates in worst-case optimal time. In *22nd International Conference on Database Theory*, 2019.

[KNOZ20a] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 375–392, 2020.

[KNOZ20b] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 375–392, 2020.

[KOW21] Paraschos Koutris, Xiating Ouyang, and Jef Wijsen. Consistent query answering for primary keys on path queries. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS'21, page 215–232, New York, NY, USA, 2021. Association for Computing Machinery.

[KP19] Tsvi Kopelowitz and Ely Porat. The strong 3sum-indexing conjecture is false. *arXiv preprint arXiv:1907.11206*, 2019.

[KPP16] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3sum conjecture. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 1272–1287. SIAM, 2016.

[KRR13] Tomasz Kociumaka, Jakub Radoszewski, and Wojciech Rytter. Efficient indexes for jumbled pattern matching with constant-sized alphabet. In *ESA*, pages 625–636. Springer, 2013.

[KRS+16] Anja Kunkel, Astrid Rheinländer, Christopher Schiefer, Sven Helmer, Panagiotis Bouros, and Ulf Leser. Piejoin: towards parallel set containment joins. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*, page 11. ACM, 2016.

[KS06] Benny Kimelfeld and Yehoshua Sagiv. Incrementally computing ordered answers of acyclic conjunctive queries. In *International Workshop on Next Generation Information Technologies and Systems*, pages 141–152. Springer, 2006.

[KS07] Benny Kimelfeld and Yehoshua Sagiv. Combining incompleteness and ranking in tree queries. In *International Conference on Database Theory*, pages 329–343. Springer, 2007.

[KS13] Wojciech Kazana and Luc Segoufin. Enumeration of first-order queries on classes of structures with bounded expansion. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 297–308. ACM, 2013.

[KSKÇ12] Onur Küçüktunç, Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. Recommendation on academic networks using direction aware citation analysis. *arXiv preprint arXiv:1205.1143*, 2012.

[Law72] Eugene L Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management science*, 18(7):401–405, 1972.

[LBBA16] Jyoti Leeka, Srikanta Bedathur, Debajyoti Bera, and Medha Atre. Quark-x: An efficient top-k processing framework for rdf quad stores. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 831–840, 2016.

[LCFK21] Jingjie Li, Amrita Roy Chowdhury, Kassem Fawaz, and Younghyun Kim. Kalεido: Real-time privacy control for eye-tracking systems. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[LCIS05] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F Ilyas, and Sumin Song. Ranksql: query algebra and optimization for relational top-k queries. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 131–142. ACM, 2005.

[LDH+08] Cindy Xide Lin, Bolin Ding, Jiawei Han, Feida Zhu, and Bo Zhao. Text cube: Computing ir measures for multidimensional text database analysis. In *2008 Eighth IEEE International Conference on Data Mining*, pages 905–910. IEEE, 2008.

[LFHDB15] Yongming Luo, George HL Fletcher, Jan Hidders, and Paul De Bra. Efficient and scalable trie-based algorithms for computing set containment relations. In *2015 IEEE 31st International Conference on Data Engineering*, pages 303–314. IEEE, 2015.

[LFZ19] Xiaoming Li, Hui Fang, and Jie Zhang. Supervised user ranking in signed social networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 184–191, 2019.

[LHG04] Xiaolei Li, Jiawei Han, and Hector Gonzalez. High-dimensional olap: A minimal cubing approach. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 528–539, 2004.

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[LMNT15] Kasper Green Larsen, J Ian Munro, Jesper Sindahl Nielsen, and Sharma V Thankachan. On hardness of several string indexing problems. *Theoretical Computer Science*, 582:74–82, 2015.

[LPH02] Laks VS Lakshmanan, Jian Pei, and Jiawei Han. Quotient cube: How to summarize the semantics of a data cube. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 778–789. Elsevier, 2002.

[LSCI05] Chengkai Li, Mohamed A Soliman, Kevin Chen-Chuan Chang, and Ihab F Ilyas. Ranksql: supporting ranking queries in relational database management systems. In *Proceedings of the 31st international conference on Very large data bases*, pages 1342–1345. VLDB Endowment, 2005.

[LWW18] Andrea Lincoln, Virginia Vassilevska Williams, and Ryan Williams. Tight hardness for shortest cycles and paths in sparse graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1236–1252. SIAM, 2018.

[Mar13] Dániel Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *Journal of the ACM (JACM)*, 60(6):42, 2013.

[Mar21] Dániel Marx. Modern lower bound techniques in database theory and constraint satisfaction. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 19–29, 2021.

[MBO12] Sudip Misra, Romil Barthwal, and Mohammad S Obaidat. Community detection in an integrated internet of things and social network architecture. In *2012 IEEE Global Communications Conference (GLOBECOM)*, pages 1647–1652. IEEE, 2012.

[MCCJ21] Casey Meehan, Amrita Roy Chowdhury, Kamalika Chaudhuri, and Somesh Jha. A shuffling framework for local differential privacy. *arXiv preprint arXiv:2106.06603*, 2021.

[McM56] B. McMillan. Two inequalities implied by unique decipherability. *IRE Transactions on Information Theory*, 2(4):115–116, December 1956.

[MKB09] Stefan Manegold, Martin L Kersten, and Peter Boncz. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *Proceedings of the VLDB Endowment*, 2(2):1648–1653, 2009.

[MSGL14] Seth A Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. Information network or social network? the structure of the twitter follow graph. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 493–498, 2014.

[NCS+01] Apostol Natsev, Yuan-Chi Chang, John R Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, volume 1, pages 281–290, 2001.

[NL04] Alan Nash and Bertram Ludäscher. Processing unions of conjunctive queries with negation under limited access patterns. In *EDBT*, pages 422–440. Springer, 2004.

[NPRR12] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 37–48. ACM, 2012.

[NRR13] Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2013.

[NZRS] Feng Niu, Ce Zhang, Chris Ré, and Jude Shavlik. Felix: Exploiting specialized subtasks in markov logic networks for higher efficiency and quality.

[OS16] Dan Olteanu and Maximilian Schleich. Factorized databases. *ACM SIGMOD Record*, 45(2):5–16, 2016.

[OVL81] Mark H Overmars and Jan Van Leeuwen. Dynamization of decomposable searching problems yielding good worst-case bounds. In *Theoretical Computer Science*, pages 224–233. Springer, 1981.

[OZ15a] Dan Olteanu and Jakub Závodnỳ. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)*, 40(1):1–44, 2015.

[OZ15b] Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2, 2015.

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3):1–45, 2009.

[PDZ+18] Jignesh M Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. Quickstep: A data platform based on the scaling-up approach. *Proceedings of the VLDB Endowment*, 11(6):663–676, 2018.

[PKZT01] Dimitris Papadias, Panos Kalnis, Jun Zhang, and Yufei Tao. Efficient olap operations in spatial data warehouses. In *International Symposium on Spatial and Temporal Databases*, pages 443–459. Springer, 2001.

[PP06] Anna Pagh and Rasmus 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.

[PR10] Mihai Patrascu and Liam Roditty. Distance oracles beyond the thorup-zwick bound. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 815–823. IEEE, 2010.

[pro] protein. https://string-db.org/cgi/download.pl?sessionId=IBdaKPtZGbl2.

[PTS+17] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*, volume 19, 2017.

[PW18] Fotis Psallidas and Eugene Wu. Smoke: fine-grained lineage at interactive speed. *Proceedings of the VLDB Endowment*, 11(6):719–732, 2018.

[QCS07] Yan Qi, K Selçuk Candan, and Maria Luisa Sapino. Sum-max monotonic ranked joins for evaluating top-k twig queries on weighted data graphs. In *Proceedings of the 33rd international conference on Very large data bases*, pages 507–518. VLDB Endowment, 2007.

[RCGvDMJ] Amrita Roy Chowdhury, Chuan Guo, Laurens van Der Maaten, and Somesh Jha. Eiffel: Ensuring integrity for federated learning.

[RCWH+20] Amrita Roy Chowdhury, Chenghong Wang, Xi He, Ashwin Machanavajjhala, and Somesh Jha. Crypte: Crypto-assisted differential privacy on untrusted servers. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 603–619, 2020.

[RDS07] Christopher Re, Nilesh Dalvi, and Dan Suciu. Efficient top-k query evaluation on probabilistic data. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 886–895. IEEE, 2007.

[roa] RoadNet. https://snap.stanford.edu/data/roadNet-PA.html.

[RS86] Neil Robertson and Paul D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322, 1986.

[RT05] Erhard Rahm and Andreas Thor. Citation analysis of database publications. *ACM Sigmod Record*, 34(4):48–53, 2005.

[SDKN20] Bruhathi Sundarmurthy, Harshad Deshmukh, Paris Koutris, and Jeffrey Naughton. Providing insights for queries affected by failures and stragglers. *arXiv preprint arXiv:2002.01531*, 2020.

[Seg13a] Luc Segoufin. Enumerating with constant delay the answers to a query. In *Joint 2013 EDBT/ICDT Conferences, ICDT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 10–20, 2013.

[Seg13b] Luc Segoufin. Enumerating with constant delay the answers to a query. In *Proceedings of the 16th International Conference on Database Theory*, pages 10–20. ACM, 2013.

[Seg15a] Luc Segoufin. Constant delay enumeration for conjunctive queries. *ACM SIGMOD Record*, 44(1):10–17, 2015.

[Seg15b] Luc Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Record*, 44(1):10–17, 2015.

[SH13] Yizhou Sun and Jiawei Han. Mining heterogeneous information networks: a structural analysis approach. *Acm Sigkdd Explorations Newsletter*, 14(2):20–28, 2013.

[SHK00] Nebojsa Stefanovic, Jiawei Han, and Krzysztof Koperski. Object-based selective materialization for efficient implementation of spatial data cubes. *IEEE Transactions on knowledge and Data Engineering*, 12(6):938–958, 2000.

[SM13] Juan F Sequeda and Daniel P Miranker. Ultrawrap: Sparql execution on relational data. *Journal of Web Semantics*, 22:19–39, 2013.

[SOC16] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*, pages 3–18. ACM, 2016.

[SORK11] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. Probabilistic databases, synthesis lectures on data management. *Morgan & Claypool*, 2011.

[SS95] William L Steiger and Ileana Streinu. A pseudo-algorithmic separation of lines from pseudo-lines. *Inf. Process. Lett.*, 53(5):295–299, 1995.

[TAG+] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *Proceedings of the VLDB Endowment*, 13(9).

[TGR20] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Optimal join algorithms meet top-k. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, volume 13, pages 2659–2665. NIH Public Access, 2020.

[TGR21a] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Beyond equi-joins: Ranking, enumeration and factorization. *Proc. VLDB Endow.*, 14(11):2599–2612, 2021.

[TGR21b] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Beyond equi-joins: Ranking, enumeration and factorization. *arXiv preprint arXiv:2101.12158*, 2021.

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

[Ull85] Jeffrey D. Ullman. Implementation of logical query languages for databases. *ACM Trans. Database Syst.*, 10(3):289–321, September 1985.

[Vel] TL Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm, icdt,(2014).

[WL03] Fang Wei and Georg Lausen. Containment of conjunctive queries with safe negeuration. In *ICDT*, pages 346–360. Springer, 2003.

[wor] words. https://archive.ics.uci.edu/ml/datasets/bag+of+words.

[XD17a] Konstantinos Xirogiannopoulos and Amol 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.

[XD17b] Konstantinos Xirogiannopoulos and Amol 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.

[XD19] Konstantinos Xirogiannopoulos and Amol Deshpande. Memory-efficient group-by aggregates over multi-way joins. *arXiv preprint arXiv:1906.05745*, 2019.

[XKD15] Konstantinos Xirogiannopoulos, Udayan Khurana, and Amol Deshpande. Graphgen: Exploring interesting graphs in relational data. *Proceedings of the VLDB Endowment*, 8(12):2032–2035, 2015.

[XSD17] Konstantinos Xirogiannopoulos, Virinchi Srinivas, and Amol 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.

[YAG+18] Xiaofeng Yang, Deepak Ajwani, Wolfgang Gatterbauer, Patrick K Nicholson, Mirek Riedewald, and Alessandra Sala. Any-k: Anytime top-k tree pattern retrieval in labeled graphs. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 489–498. International World Wide Web Conferences Steering Committee, 2018.

[Yan81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, pages 82–94, 1981.

[YRLG18] Xiaofeng Yang, Mirek Riedewald, Rundong Li, and Wolfgang Gatterbauer. Any-k algorithms for exploratory analysis with conjunctive queries. In *Proceedings of the 5th International Workshop on Exploratory Search in Databases and the Web*, pages 1–3, 2018.

[YSN+12] Xiao Yu, Yizhou Sun, Brandon Norick, Tiancheng Mao, and Jiawei Han. User guided entity similarity search using meta-path selection in heterogeneous information networks. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2025–2029, 2012.

[YZY+18] Jianye Yang, Wenjie Zhang, Shiyu Yang, Ying Zhang, Xuemin Lin, and Long Yuan. Efficient set containment join. *The VLDB Journal—The International Journal on Very Large Data Bases*, 27(4):471–495, 2018.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[ZDP19] Zuyu Zhang, Harshad Deshmukh, and Jignesh M Patel. Data partitioning for in-memory systems: Myths, challenges, and opportunities. In *CIDR*, 2019.

[ZLGZ10] Zhaonian Zou, Jianzhong Li, Hong Gao, and Shuo Zhang. Finding top-k maximal cliques in an uncertain graph. 2010.

[ZLXH11] Peixiang Zhao, Xiaolei Li, Dong Xin, and Jiawei Han. Graph cube: on warehousing and olap multidimensional networks. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 853–864, 2011.