ADVANCING JOIN ALGORITHMS FOR REAL-WORLD QUERIES

by

Hangdong Zhao

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2025

Date of final oral examination: April 14, 2025

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

Paraschos Koutris, Associate Professor, Computer Sciences

Aws Albarghouthi, Associate Professor, Computer Sciences

Xiangyao Yu, Assistant Professor, Computer Sciences

Jin-Yi Cai, Professor, Computer Sciences and Mathematics

© Copyright by Hangdong Zhao 2025

All Rights Reserved

To my family

ACKNOWLEDGMENTS

It was a stroke of luck that brought me to Paris Koutris, my advisor. Database research was a big leap for me, coming from a background of pure mathematics. I started not knowing what a single JOIN was; and yet, here I am, writing up this dissertation on precisely that. Over the years, Paris has made this transition so much easier for me and he has been the best teacher I could ever hope for—guiding me with wisdom through the inevitable ups and downs of research. I was a slow researcher, but Paris was always patient, supportive and encouraging in every single dicussion. I have lost count of the times he grounded my scattered ideas with rigorous reasoning, or asked just the right question that cut to the heart of the problem. His clarity of thought, his humility, and his calm presence will always be a model I aspire to follow. For all this and so much more, I am deeply grateful to Paris.

It was truly my privilege to have Aws Albarghouthi, Xiangyao Yu, and Jin-Yi Cai as my other committee members for this dissertation—thank you for your thoughtful feedback, your warm encouragement, and the time and care you devoted to my work. Your questions challenged me in the best way, urging me to probe deeper into my ideas, speak with precision, and envision the larger picture. I am especially grateful for your nomination for the Outstanding Graduate Student Award—a gesture of faith that I will always treasure, all the more so given that I asked for it at the very last moment. I would also like to extend my heartfelt thanks to Tom Reps for offering such insightful reflections on my dissertation, from his profound expertise in program analysis. I am deeply grateful to AnHai Doan for the one-on-one sessions we had as I approached the last stage of PhD, where he shared not only wisdom about research, careers, and life, but also many humorous anecdotes that I always find myself laughing about.

I owe a tremendous debt of gratitude to Microsoft Gray Systems Lab (GSL) for the generous funding support during the latter half of my Ph.D. At GSL, I was reminded never to underestimate the power of theoretical research—and to always seek ways to bring it into practice. It was there that Venkatesh Emani first took a chance on me, offering me a research assistantship and taught the rich interplay between databases and artificial intelligence. Later, during my first-ever internship, I met the best intern mentor I could have hoped for: Rana Alotaibi, who filled those months with sparkling ideas, worried about me when I struggled, and cheered on every small step forward. As graduation approached—in all the chaos and uncertainty that came with it—I found steady support from Yuanyuan Tian, my future manager. Thank you for your unwavering encouragement during those last months when it mattered most. Beyond them, I was lucky to meet so many others at Microsoft—Sergiy Matusevych, Bailu Ding, Jesús Camacho Rodríguez, Ernesto Cervantes Juárez, Joyce Cahoon, Nico Bruno, Carlo Curino, among many others. From a research assistantship, to an internship, to now a full-time role, my story at Microsoft has been filled with so many rewarding moments and friendships.

Looking back, being part of Paris's research lab was one of the best pieces of my Ph.D. experience—surrounded by students whose intellect was matched only by their kindness. Shaleen Deep, Xiating Ouyang, Zhiwei Fan, Austen Z. Fan, and Simon Frisk—thank you for making research such a shared and memorable adventure. I loved the way we could bounce ideas off each other, and how we could always count on each other before deadlines. I have learned so much from each of you, not just in research, but in how to show up with curiosity, humor, and heart. Go Team Paris!!

Another great source of fun during my Ph.D. was the chance to travel, explore, and meet incredible minds across the database community—through conferences, workshops, seminars, and many mini-conversations in between. I feel truly fortunate to have had the chance to chat, brainstorm, and even work alongside many of you. I would especially like to thank Sudeepa Roy, Xiao Hu, Jignesh Patel, Kristopher Micinski, Kirk Pruhs, Sam Arch, Remy Wang, Kyle Deeds, Goetz Graefe, Hung Q. Ngo, Dan Suciu, Val Tannen, and many others. Every discussion—whether a formal presentation, a coffee shop brainstorming, or an impromptu hallway chat—left an imprint on how I reason about research,

made me even prouder to be part of this community. I will always be grateful for the inspiration and friendship you have shared.

Madison has always been a freezing place in winters—but over the years, it grew warmer and warmer in my heart, all thanks to my friends. You were the sparkles that brightened my gloomy days, the laughter that echoed in my quiet moments, and the warmth that melted away the cold. You taught me how to cook, how to drive, encouraged me at the gym, and shared your stories with me. I am so grateful to have met you all: Song Bian, Minghao Yan, Abigale Kim, Yifei Yang, Xufeng Cai, Mu Cai, Bobbi Yogatama, Ling Zhang, Wenjie Hu, Kevin Gaffney, Zhenghong Yu, Minh Phan, Skylar Hou, Mingchen Ma, Yusen Liu, Xuefeng Du, Yang Guo, Konstantinos Kanellis, Johannes Freischuetz, Chenhao Ye, Guanzhou Hu, Suyan Qu, Yunjia Zhang, Shawn Zhong, Yuhao Zhang, among many others. Thank you for the countless dinners, stories, and gossips we shared—each one a memory I will always treasure.

I would like to thank my family. I can not thank enough of them. My parents, who have always been my biggest supporters, even from halfway across the globe. Because of the pandemic, long distances, and life's many vicissitudes, I have not been able to see you for a long time. Yet every step of this adventure abroad, I felt your presence, your quiet strength, and your firm faith in me. I have carried your love with me across every mile, through every late night, and onto every shining stage where I presented my milestone. I hope I have made you proud—and I cannot wait to give you the biggest hug when we meet again.

Finally, I would like to thank my partner, my anchor and my light, Ting Cai. From the moment we fall in love, she has been by my side through every twist and turn. I felt incredibly blessed to have her as my love, my confidant, and my best friend. In the stretches of our Ph.D., when the snow grew heavy and the deadlines loomed large, it was her voice that calmed my anxieties and her optimism that lifted the weight off my shoulders. Together, we watched sunsets fade into night and sunrises break the dawn, shared our dreams and fears, and traveled to so many awesome places. She gives me every reason to work hard for the future we wish for, and every reason to savor the quiet, precious moments we deserve. Wherever life leads us next, I am grateful—and faithful—that we will walk through it all, hand in hand, heart to heart.

TABLE OF CONTENTS

Page

ABSTRACT x

1 Introduction 1

1.1 Current Progress 2
1.2 Motivations and Contributions 3
1.3 Chapter Organization and Publications 8

2 Background 10

2.1 Relations and Queries 10
2.2 Join Algorithms 11
2.2.1 Yannakakis Algorithm 12
2.2.2 Tree Decompositions 13
2.2.3 Data Partitioning 15
2.2.4 PANDA Algorithm 16
2.3 Algebraic Semantics 20
2.4 Datalog Basics 22

3 Related Work 23

3.1 Extensions of Join Algorithms 23
3.2 Leveraging Statistical Information 24
3.3 Enumeration-delay Algorithms 25
3.4 Lower Bounds for Join Queries 26

4 Join with Projections 28

4.1 Preliminaries and Notation 31
4.2 Projection Width 32
4.3 Main Result 35
4.3.1 Output-sensitive Yannakakis 35

Page

4.3.2 Our Algorithm . . . 38
4.4 Extension to Aggregation . . . 44
4.5 Applications . . . 44
4.5.1 Path Queries . . . 45
4.5.2 Hierarchical Queries . . . 45
4.5.3 General Queries . . . 46
4.6 Lower Bounds . . . 46
4.7 Related Work . . . 50
4.8 Conclusion . . . 50

5 Join with Negations . . . 51

5.1 Introduction . . . 51
5.1.1 CQs with Negation . . . 51
5.1.2 FAQ with Negation . . . 52
5.1.3 Lower Bounds . . . 53
5.1.4 Extensions . . . 54
5.2 Preliminaries . . . 54
5.3 Signed Acyclicity . . . 56
5.3.1 Signed Leaves . . . 56
5.3.2 Elimination Sequences . . . 60
5.4 Enumeration of Full $\mathsf{CQ}^{\neg}$ . . . 62
5.4.1 Preprocessing phase . . . 64
5.4.2 Enumeration phase . . . 68
5.4.3 A Comprehensive Example . . . 68
5.4.4 Correctness Proof of Full $\mathsf{CQ}^{\neg}$ . . . 71
5.5 Enumeration of $\mathsf{NestFAQ}^{\neg}$ . . . 73
5.5.1 $\mathsf{NestFAQ}^{\neg}$ Queries . . . 73
5.5.2 Enumeration Algorithm: An Overview . . . 76
5.5.3 The Refactoring Algorithm . . . 77
5.5.4 The Aggregation Algorithm . . . 84
5.5.5 The Oracle-construction Algorithm . . . 88
5.5.6 Enumeration of Full $\mathsf{NestFAQ}^{\neg}$ . . . 97

Page

5.5.7 Putting Everything Together . . . 99
5.6 Lower Bounds . . . 102
5.6.1 Lower Bounds for $\mathsf{CQ}^{\neg}$ . . . 102
5.6.2 Lower Bounds for $\mathsf{FAQ}^{\neg}$ . . . 103
5.6.3 An Unconditional Lower Bound for $\mathsf{FAQ}^{\neg}$ . . . 103
5.7 Difference of CQs . . . 104
5.8 $\mathsf{CQ}^{\neg}$ Beyond Linear Time . . . 105
5.9 Conclusion . . . 106

6 Join with Access Patterns . . . 107

6.1 Background . . . 109
6.1.1 CQs with Access Patterns . . . 110
6.1.2 Problem Statement . . . 111
6.2 Partially Materialized Tree Decompositions . . . 112
6.2.1 Online Yannakakis for PMTDs . . . 114
6.3 General Framework . . . 117
6.3.1 2-Phase Disjunctive Rules . . . 118
6.3.2 Preprocessing Phase . . . 119
6.3.3 Online Phase . . . 120
6.3.4 Correctness of the General Framework . . . 121
6.4 Constructing the Tradeoffs . . . 122
6.5 Algorithms for 2-phase Disjunctive Rules . . . 125
6.5.1 Background . . . 125
6.5.2 A 2-phase Algorithmic Paradigm . . . 126
6.5.3 Constructing Tradeoffs as Optimization under Constraints . . . 128
6.6 The 2-phase PANDA Algorithm . . . 130
6.6.1 Joint Shannon-flow Inequalities . . . 130
6.6.2 An Augmentation of the PANDA Algorithm . . . 137
6.6.3 The Algorithm . . . 139
6.6.4 Analysis of the 2-phase PANDA Algorithm . . . 142
6.7 Applications . . . 144
6.7.1 Tradeoffs for k-Set Intersection . . . 144

Page

6.7.2 Tradeoffs via Fractional Edge Covers . . . 145
6.7.3 Tradeoffs via Tree Decompositions . . . 147
6.7.4 Tradeoffs for k-Reachability . . . 152
6.8 Related Work . . . 154
6.9 Conclusion . . . 155

7 Join with Recursion I: An Algorithmic Framework . . . 156

7.1 Preliminaries . . . 158
7.1.1 Semirings and Their Properties . . . 158
7.1.2 Datalog over Semirings . . . 159
7.2 Framework and Main Results . . . 160
7.2.1 Grounding Generation . . . 161
7.2.2 Grounding Evaluation . . . 164
7.2.3 Applications . . . 164
7.3 Grounding Evaluation . . . 165
7.3.1 2-canonical Grounding . . . 165
7.3.2 Finite-rank Semirings . . . 167
7.3.3 Absorptive Semirings with Total Order . . . 168
7.4 Grounding of Acyclic Datalog . . . 169
7.5 Grounding of General Datalog . . . 173
7.6 Related Work . . . 177
7.7 Conclusion . . . 178

8 Join with Recursion II: A System Design . . . 179

8.1 Background . . . 181
8.1.1 Datalog Basics . . . 181
8.1.2 Datalog Evaluation . . . 182
8.1.3 Differential Dataflow . . . 183
8.2 System Design . . . 185
8.3 Logic Fusion . . . 186
8.4 Join Plan Optimization . . . 188
8.4.1 Structural Cost Model . . . 189

Page

8.4.2 Search Strategy 189
8.4.3 Plan Execution 191
8.5 Making Datalog Robust 193
8.6 Subplan Sharing 195
8.7 Boolean Specialization 196
8.8 Extensibility 197
8.9 Benchmarking and Experiments 198
8.9.1 Runtime Summary 200
8.9.2 CPU and Memory Usage 201
8.9.3 Parallel Scalability 202
8.9.4 Join Planning and Robust Execution 203
8.10 Conclusion 205

9 Conclusion and Future Outlook 206

LIST OF REFERENCES 208

ABSTRACT

Join queries lie at the heart of relational data systems. Traditionally, relational engines execute these queries by decomposing them into sequences of binary joins, guided by cost-based optimizers. While this paradigm has dominated for decades, it increasingly struggles to meet the demands of contemporary workloads—characterized by large-scale, skewed data, imprecise or missing statistics, and increasingly rich but intricate query semantics. These challenges expose both theoretical and practical limitations of the established approach, often resulting in brittle query optimizers and significantly suboptimal performance.

In parallel, the research community has made substantial progress in developing join algorithms with strong theoretical guarantees. However, these algorithms are often designed under idealized assumptions such as purely join-based queries or uniformly sized relations—divorced from the broader realities of real-world queries. In practice, relational queries often involve additional constructs like projections, antijoins, or group-by aggregation, operate over skewed data, or must respect a handful of integrity constraints. In other emerging scenarios—such as multi-query optimization, incremental query maintenance, or recursive, loop-containing queries—existing techniques frequently fall short. These settings further complicate the landscape and demand a more nuanced understanding of the interplay between query structure and data characteristics.

This dissertation seeks to bring these join algorithms closer to the practical needs. It presents a suite of novel join algorithms tailored to complex, real-world queries—offering principled performance guarantees without sacrificing generality. Along the way, it contributes new insights and design principles that inform how future query optimizers can be made more robust, adaptable, and interpretable. Collectively, these contributions pave the way toward a new generation of query optimization—firmly grounded in theory and effective in practice.

Chapter 1

Introduction

First introduced by Codd [Cod70] as a foundational operator in relational algebra, join ($\bowtie$) is the now bread and butter of modern database systems. Joins act as the glue that binds disparate datasets, enabling systems to extract meaningful insights, generate personalized recommendations, and support real-time decisions [FHM05, DRH11, ZXW+16, DCZ+16]. At any given moment, millions of join queries are running silently behind the scenes—powering applications from smartphones to data warehouses—across industries such as business, healthcare, and finance.

Since the pioneering System R project [SAC+79], most relational engines execute join queries as a sequence of binary (i.e. pairwise) joins, organized by an execution plan. Constructing such a plan involves choosing both join ordering and physical strategies (e.g., hash join [BSFN24], sort-merge join), a process that is essential for efficient execution. While substantial advances have been made in system-level techniques—such as parallelism [BTAÖ13, LBKN14], vectorized execution [LKP+18], and query compilation [RPE+17, NK15]—the core join algorithm remains largely unchanged. Most database engines continue to rely on cost-based optimizers, which estimate the cost of alternative plans using pre-collected statistics, and select the plan projected to be cheapest. However, this approach hinges on high-quality statistics—an assumption that often fails in practice. Maintaining high-quality statistics is expensive; the data may be skewed, stale, noisy [FKKV25], or simply misleading [KDOS24]. As a result, even mature query optimizers frequently choose suboptimal plans, leading to orders-of-magnitude performance degradations [LGM+15]. To compensate, some commercial systems—such as IBM DB2 and Microsoft SQL Server—employ rule-based heuristics as alternatives or supplements to full cost-based reasoning [PHH92, Gra95]. While effective in specific cases, such heuristics are brittle, difficult to generalize across diverse workloads, and prone to unintended regressions. Over time, these systems tend to accumulate hundreds of heuristics, resulting in optimizers that are increasingly involved, opaque, and unmaintainable.

As data volumes continue to grow, join queries have become a primary bottleneck, often consuming a disproportionate share of execution time and system resources [Cha98]. The limitations of

traditional binary join plans become especially apparent at scale, exposing their asymptotic inefficiencies [FBS+20, WWS23]. At the same time, modern applications demand increasingly complex queries that go far beyond a handful of joins. Streaming systems must respond to data that arrives continuously and unpredictably, typically without ahead-of-time statistics. Graph queries introduce skewed data distributions, intricate join patterns such as cycles and iterative joins. When external logic is embedded within relational engines—like user-defined functions [LBC+24]—optimizers lose visibility into data properties and control flow.

These challenges call for new join algorithms, either built atop existing ones or designed from first principles, that are geared towards real-world queries and workloads. This dissertation explores join algorithms beyond the binary join paradigm, and presents techniques that offer stronger theoretical guarantees, greater interpretability, and adaptability across these modern contexts.

1.1 Current Progress

From a theoretical perspective, join algorithms have long stood at the heart of database theory, with a rich class of literature dating back to the 1970s; see [KDFZ25] for a summary. A foundational milestone was established by Yannakakis [Yan81], who introduced the first linear-time algorithm for evaluating acyclic join queries—those whose join graphs are tree-shaped. This result has since become a cornerstone of the field. Building on Yannakakis, Bagan et al. [BDG07] later established that this acyclicity is both a sufficient and necessary condition for linear-time join algorithms in the worst case. These insights sparked extensive studies into more involved join topologies, i.e. those involving cyclic join patterns.

To generalize beyond acyclicity, researchers turned to *tree decompositions* [RS84, GLS99], which offer a principled way to measure how closely a join graph resembles a tree. This paved the way for the development of worst-case optimal join algorithms (WCOJ), such as GenericJoin [NPRR18], Leapfrog Triejoin [Vel12] and FreeJoin [WWS23], which match the well-established join output size bound—commonly known as the AGM bound [AGM08]. These algorithms were later refined using more nuanced (width) measures like the *fractional hypertree width*, to capture the intrinsic difficulty of evaluating queries in addition to output sizes. Unlike traditional binary joins, WCOJ algorithms adopt an *attribute-at-a-time* strategy, incrementally building intermediate results one attribute at a time rather than one relation at a time. To support this fine-grained execution model, they rely on specialized data structures, such as hash tries, for efficient access and traversal.

A more recent and unifying approach is the PANDA algorithm [KNS17], which reimagines join algorithms through the lens of information theory. PANDA formulates the join problem as a search for a proof in an entropy-based inequality system, guiding the construction of join plans with formal

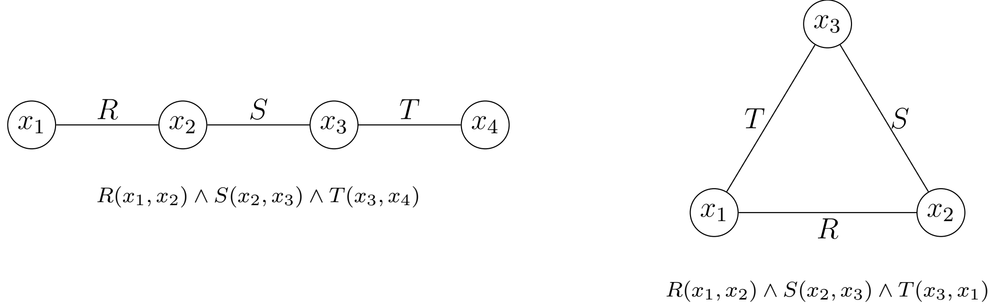


Figure 1.1: Join graphs of a path (left) and a triangle (right) join query.

guarantees on worst-case optimality. A key insight of PANDA is data partitioning: It divides the input data into partitions based on data characteristics and applies different join strategies to each partition. This enables PANDA to adapt to data skew and tailor its behavior to the local structure of the data, leading to an even tighter width measure, called the *submodular width* [Mar13].

In Chapter 2, we will delve deeper into these foundational algorithms and concepts, setting the stage for the new contributions developed in this dissertation.

1.2 Motivations and Contributions

We use two queries in Figure 1.1 to illustrate existing join algorithms, discuss their limitations, and motivate this dissertation. These examples are expressed in the standard notation for conjunctive queries (formally defined in Chapter 2), where $\wedge$ denotes conjunction (i.e., join), and R, S, and T represent input relations. We assume for now that all base relations contain $O(|\mathcal{D}|)$ tuples, where $|\mathcal{D}|$ is the size of the input database, and let $|\texttt{OUT}|$ denote the size of the join output. It is easy to see that the path query (Figure 1.1 left) has a path-like join graph, which is acyclic and can be evaluated in time $O(|\mathcal{D}| + |\texttt{OUT}|)$ using the Yannakakis algorithm [Yan81] (see Theorem 2.1). The path query (left) has an acyclic join graph, which admites executions in time $O(|\mathcal{D}| + |\texttt{OUT}|)$ using the Yannakakis algorithm [Yan81] (see Theorem 2.1). In contrast, the triangle query (Figure 1.1 right) has a cyclic join graph. As such, it requires advanced algorithmic techniques such as WCOJ or the PANDA algorithm. These algorithms attain runtime bounds of $O(|\mathcal{D}|^{3/2} + |\texttt{OUT}|)$ (see Theorem 2.3), where 3/2 is the submodular width of the query. Notably, this runtime is a substantial improvement over any binary join plans, which would take $O(|\mathcal{D}|^2 + |\texttt{OUT}|)$ time in the worst case.

While the techniques surveyed in Section 1.1 (and discussed in much greater depth in Chapter 2) have made significant strides in join algorithms—especially in their ability to optimize complicated (cyclic) join patterns—they are still predominantly confined to pure equi-joins. These algorithms often fall short when applied to real-world workloads that involve richer query constructs, such as

projections, negation, aggregations, among others. In practice, queries are rarely composed of joins alone. Instead, they typically form part of broader query pipelines that interleave joins with various logical and algebraic operators, challenging the assumptions underpinning current join algorithms. In such settings, the theoretical guarantees of join algorithms can degrade significantly, and the algorithms themselves may become even infeasible to apply.

This dissertation is motivated by the widening gap between the theoretical advances in join algorithms and the realities of real-world queries. It seeks to close this gap by developing new algorithms and abstractions that extend the strengths of existing join algorithms to richer and more expressive query classes. Our **contributions** of this dissertation are summarized as follows.

Join with projections (Chapter 4). The first contribution is a new algorithm for *join-project* and *join-aggregate* queries. For example, the path query in Figure 1.2 (left) is now augmented with a projection operator (Π_{x_1,x_4}), which selects only attributes x_1 and x_4 from the join. This is much more common in practice, as users often only want a subset of the attributes from the full join.

Unfortunately, the classical Yannakakis algorithm is not well-suited for handling projections—it immediately degrades to a polynomial runtime of $O(|\mathcal{D}| \cdot |\texttt{OUT}|)$ when applied directly. To address this, we propose an *output-sensitive algorithm* that integrates the data partioning technique into the Yannakakis framework. This results is the first improvement to the Yannakakis algorithm in nearly four decades, achieving a runtime of $O(|\mathcal{D}| \cdot |\texttt{OUT}|^{1-\epsilon})$, where $\epsilon > 0$ is a constant determined by the join-project query topology. In this very case (Figure 1.2, left), it runs in time $O(|\mathcal{D}| \cdot |\texttt{OUT}|^{1/2})$, i.e. $\epsilon = 1/2$. We further show that our algorithm can easily extend to general *join-aggregate queries* with much better guarantees via the algebraic lens of *semirings* (formally defined in Chapter 2).

Join with negations (Chapter 5). The second contribution is a linear-time characterization for *join-aggregate queries with negations*. We present a notion called *signed-acyclicity*, a generalization of classic acyclicity, under which we propose a linear-time algorithm that, perhaps surprisingly, transforms negations into range queries and leverages efficient algorithms for range sums.

As a characterization result, we also show that queries whose graphs are not signed-acyclic—such as the example in Figure 1.2 (right)—do not admit linear-time evaluation. Nevertheless, our algorithm provides key insights that lead to significantly improved strategies for handling antijoins. In Figure 1.2 (right)—a common open-triangle query in personalized recommendations [HW23]—the presence of $\neg T$ (i.e., an antijoin) typically forces existing query planners into a rigid strategy: first joining $R \bowtie S$, then applying the antijoin with $\neg T$. In contrast, our approach introduces a novel technique that pushes antijoins below joins, enabling more flexible and efficient query plans. In this open-triangle case, our method achieves a runtime of $O(|\mathcal{D}|^{3/2} + |\texttt{OUT}|)$, matching the best-known runtime of the triangle query using WCOJ algorithms. See Example 5.12 for the details.

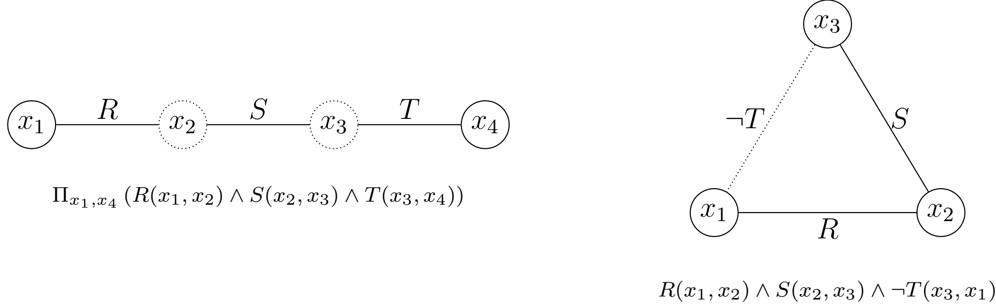


Figure 1.2: Join graphs of a path (left) and an open-triangle (right) join query. Dotted nodes x_2, x_3 indicate that the attributes are not necessary for the output (i.e. projected away). The dotted edge $\neg T$ indicates that the join is an antijoin.

Join with access patterns (Chapter 6). The third contribution is a general algorithmic framework for handling *conjunctive queries with access patterns* (or CQAP for short). These workloads arise naturally in multi-query optimization contexts, such as SQL prepare statements[1], where users provide query templates containing placeholders for attribute values. Continuing from Figure 1.2 (left), consider a query template referred to as the 3-Reachability CQAP:

$$\text{3-Reachability}(x_1, x_4 | x_1, x_4) \leftarrow R(x_1, x_2) \wedge S(x_2, x_3) \wedge T(x_3, x_4) \tag{1.1}$$

Here, the query template is provided ahead of time for preprocessing and the actual values of x_1 and x_4 are specified at runtime by users (i.e., during the online phase). The query engine must efficiently respond to each instantiated request:

$$\Pi_{a_1,a_4}(R(a_1, x_2) \wedge S(x_2, x_3) \wedge T(x_3, a_4)) \tag{1.2}$$

A fundamental challenge is the co-design of materializations and answering spanning two distinct phases: preprocessing and online phase. Figure 1.3 illustrates this two-phase process. An efficient algorithm should avoid re-executing the join (e.g. (1.2)) entirely for each online request, yet also refrain from fully materializing the entire join result (e.g. (1.1)) at preprocessing, as this can be prohibitively large. This naturally introduces space-time tradeoffs: trading materialization during preprocessing against response time during request execution. Notably, our framework unifies many previously isolated algorithmic problems under the CQAP abstraction, including k-Set Disjointness and k-Reachability [CP10a, GKLP17].

Our main contribution is an unified framework that systematically explores space-time tradeoffs for any CQAPs. The framework leverages tree decompositions [GGS14, Mar13] and PANDA [KNS17].

[1] https://en.wikipedia.org/wiki/Prepared_statement

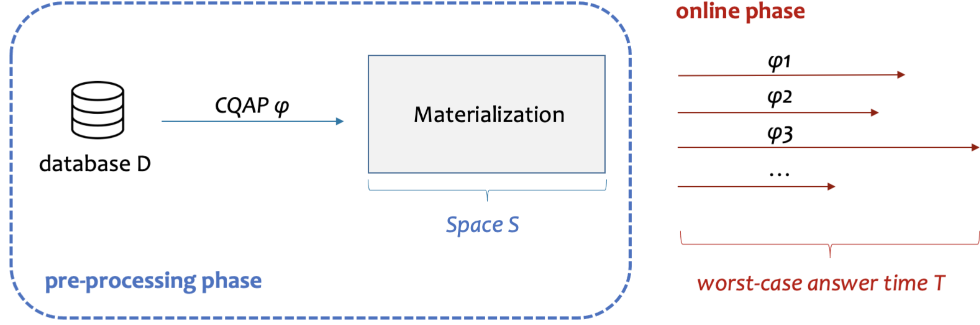


Figure 1.3: The preprocessing phase (left) and online phase (right) of a CQAP. The preprocessing phase materializes succinct data structures, while the online phase executes user request queries.

We introduce the novel notion of partially-materialized tree decompositions, which augments classical tree decompositions to represent which components are materialized during preprocessing and which are executed on-demand. Similar to PANDA, we use entropic inequalities to guide the pattern of tasks across the two phases. This approach yields a pair of inequality systems—one for preprocessing and one for online answering—which can be jointly optimized to explore a continuum of tradeoffs. We apply our framework to problems such as k-Reachability, recovering known bounds and, in many cases, strictly improving them. Our findings refute long-standing conjectures about the optimality of space-time tradeoffs for k-Reachability.

To concretely illustrate, consider again the 3-Reachability CQAP. A naive strategy—fully materializing the join—requires $O(|\mathcal{D}|^2)$ space but achieves constant-time online answering. Conversely, computing the query from scratch (e.g., via Yannakakis) minimizes space usage to $O(|\mathcal{D}|)$ but incurs $O(|\mathcal{D}| + |\texttt{OUT}|)$ query time. Our approach smoothly interpolates between these extremes. For example, as depicted in Figure 1.4, by using $|\mathcal{D}|^{7/5}$ space, we can reduce the worst-case query time to $|\mathcal{D}|^{2/5}$—strictly improving prior tradeoffs.

Join with recursion (Chapter 7). The fourth (and final) contribution is a general algorithm and a full-fledged system prototype for evaluating *recursive queries*. We study this through the lens of Datalog [AHV95]—a declarative query language that adds recursion to relational algebra in a concise syntax, while maintaining elegant fixpoint semantics. These features have made Datalog a popular object of study in both database theory and systems [GHLZ13, MTKW18]. More recently, Abo Khamis et al. [KNP+22a] proposed Datalog$^\circ$, a principled extension of Datalog over semirings that generalizes recursive join-aggregations (e.g. shortest paths) and identifies algebraic properties that governs convergence.

Our contributions are two-fold (both from theoretical and system perspectives):

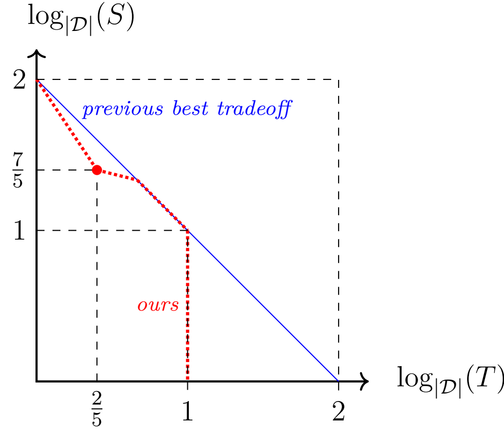


Figure 1.4: Space-time tradeoffs for 3-Reachability CQAP on a log scale. The blue line recovers the previous best tradeoff, while the red dotted curve represents our new tradeoff.

- (Algorithm) Despite extensive research, general-purpose algorithms for efficient Datalog evaluation remain elusive. Prior efforts have either focused on restricted Datalog fragments (e.g., chain Datalog [Yan90], program analysis [MP21]) or provided only coarse-grained complexity results for the general case. In the first part, we ask the following question: given any Datalog program over a naturally-ordered semiring[2], what is the best possible evaluation algorithm? To this end, we propose a general two-phase framework for evaluating Datalog over semirings. The first phase translates the program into an equivalent system of equations (termed a *grounding*), and the second phase computes its fixpoint over the semiring. Our key insight is to minimize the size of the grounding by exploiting modern join algorithms—such as Yannakakis and PANDA. In the second phase, we develop near-optimal fixpoint algorithms for a broad family of semirings. This results in a set of new algorithms with much tighter runtime guarantees. As a special case of our framework, it generalizes the Dijkstra algorithm.

- (System) While the existing Datalog-based systems such as Soufflé [SJSW16], Flix [MYL16], RecStep [FZZ+19] have achieved practical success in domains like program analysis [SJSW16, SEV16] and graph processing [LGS13, FZZ+19], they often face trade-offs between flexibility and efficiency. For example, Soufflé is a high-performance engine purpose-built for program analysis, but its domain-specific designs limit its general-purpose capabilities and scalability. RecStep, in contrast, takes a more modular approach by layering Datalog atop traditional databases for ease of deployment, but this significantly complicates the integration of Datalog-specific optimizations—particularly those that span multiple iterations.

[2] A requirement for convergence, formally defined in Chapter 2

Our system, FlowLog, seeks the best of both worlds. It harnesses efficient off-the-shelf database primitives (such as well-established join implementations) while retaining fine-grained controls over optimizations. Inspired by the grounding technique, our architecture uses an intermediate representation (IR) to decouple the logical structure of Datalog from its physical execution. This IR allows the system to reason about and apply standard database optimizations such as logic fusion—techniques that significantly reduce resource consumptions. A core challenge in recursive query processing is volatility: the same Datalog program can exhibit drastically different execution behavior under varying data distributions, join orderings, or across different iterations. Collecting accurate statistics is usually infeasible, as inputs for future iterations may only materialize during prior ones. To address this, we implement a lightweight Datalog query optimizer that targets the worst-case—avoiding materialization blow-ups and brittle join plans. In particular, we use database theory techniques such as *sideways information passing* to contruct worst-case optimized plans. FlowLog is built atop the differential dataflow framework [MMII13], enabling automatic efficiency for both batch and incremental recursive workloads. Our evaluation shows that FlowLog consistently outperforms state-of-the-art systems—including Soufflé, RecStep, and DDlog —across diverse benchmarks, while preserving a simple and modular design.

1.3 Chapter Organization and Publications

This dissertation is organized as follows.

- Chapter 2 surveys the current landscape of join algorithms, the core insights and limitations. We set up the notation and definitions that will be used throughout the dissertation.
- Chapter 3 presents related work on join algorithms beyond the ones discussed in Chapter 2.
- Chapter 4 presents the output-sensitive Yannakakis algorithm that extends traditional join algorithms to incorporate arbitrary projections and aggregations. This chapter contains materials from our recent PODS 2025 paper [DZFK24], which was accomplished while being an Research Assistant funded by Microsoft.
- Chapter 5 establishes a linear-time characterization for join queries with negations and aggregations, and introduces matching algorithms and lower bounds for such class of queries. Chapter 5 is based on our PODS 2024 paper [ZFOK23].
- Chapter 6 pivots to join queries under access patterns. We design a general algorithmic framework that leverages PANDA to guide the strategic materialization of query sub-expressions,

supporting efficient evaluation across access requests. This chapter contains our PODS 2023 paper [ZDK23a].

- Chapter 7-8 study recursive queries through the lens of Datalog. Chapter 7 introduces a novel grounding-based algorithm that is broadly applicable to recursive queries and provides strong runtime guarantees. Chapter 8 presents a Datalog system prototype incorporating a modular, which seamlessly integrates modern join algorithms and shows consistent performance gains over existing Datalog implementations. Chapter 7 contains our PODS 2024 paper [ZDK+24a].
- Chapter 9 discusses future directions and concludes this dissertation.

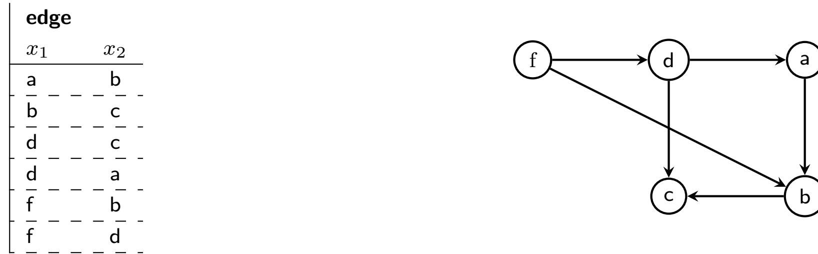
Chapter 2

Background

In this chapter, we set up the basic notations and definitions used in this dissertation.

2.1 Relations and Queries

A *relation* (or *table*) has a finite set of tuples, each of the same length and carries some structural information. As an example, the edge relation (see Figure 2.1) encodes a directed graph. The rows such as (a, b) are the *tuples/facts*; and the columns (e.g. x_1, x_2) are the *attributes/schema/variables* of a relation and the length of the attributes is called the *arity* of the relation (e.g. 2 in this case). The size of a relation is the number of tuples it contains, denoted as $|\mathsf{edge}|$ (e.g. $|\mathsf{edge}| = 6$).



edge

x_1	x_2
a	b
b	c
d	c
d	a
f	b
f	d

Figure 2.1: An example relation **edge** that encodes the directed graph on the right.

A *database instance* (or a *database* for short) $\mathcal{D}$ is a finite set of relations. The size of a database is the total number of tuples in the database, denoted as $|\mathcal{D}| = \sum_{R \in \mathcal{D}} |R|$. We denote $\mathsf{dom}(\mathcal{D})$ as the set of all constants that occur in $\mathcal{D}$, also called the *active domain* of $\mathcal{D}$.

Join queries are typlically analyzed from the lens of *conjunctive queries* (CQ) in database theory. A *conjunctive query* Q uses the following formulation:

$$Q(\mathbf{x}_F) \leftarrow \bigwedge_{K \in \mathcal{E}} R_K(\mathbf{x}_K) \tag{2.1}$$

where $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is its associated *hypergraph* $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = [n] = \{1, \ldots, n\}$ and $\mathcal{E}$ is a set of hyperedges; each hyperedge is a subset of $[n]$. The relations (or *atoms*) of the query are $R_K(\mathbf{x}_K), K \in \mathcal{E}$, where $\mathbf{x}_K = (x_j)_{j \in K}$ is the schema of atom R_K, for any $K \subseteq [n]$.

The variables $\mathbf{x}_F$ ($F \subseteq [n]$) in the head of the query are the *free* variables (or *head* variables). We denote the *result* (or *output*) of a query Q on a database $\mathcal{D}$ as $Q(\mathcal{D})$, which is yet another table of schema $\mathbf{x}_F$ that contains all tuples that satisfy the query. We use $|\texttt{OUT}|$ or $|Q(\mathcal{D})|$ to denote the number of tuples in $Q(\mathcal{D})$, i.e. output size of the query. A CQ is *full* if $F = [n]$ (i.e. a *full join*), and it is *Boolean* if $F = \emptyset$, simply denoted as $Q()$. Boolean queries merely checks the existence of some output tuples and $|Q(\mathcal{D})| = 1$. In most cases, we consider fixed queries (i.e. of constant size), except for Chapter 5 where we consider the query to be part of the runtime measure.

Example 2.1 (three-hops)**.** The following is a conjunctive query that asks for all the (x_1, x_4) pairs such that there exists a (x_2, x_3) such that (x_1, x_2), (x_2, x_3), and (x_3, x_4) are all in the edge relation:

$$\mathsf{three\text{-}hops}(x_1, x_4) \leftarrow \mathsf{edge}(x_1, x_2), \mathsf{edge}(x_2, x_3), \mathsf{edge}(x_3, x_4). \tag{2.2}$$

The Boolean and full versions of this query are as follows:

- The Boolean version checks whether there exists *any* path of length three in the graph:

$$\mathsf{three\text{-}hops\text{-}Bool}() \leftarrow \mathsf{edge}(x_1, x_2),\ \mathsf{edge}(x_2, x_3),\ \mathsf{edge}(x_3, x_4).$$

- The full version returns the entire 4-tuple (x_1, x_2, x_3, x_4) for each such three-hop paths:

$$\mathsf{three\text{-}hops\text{-}full}(\mathbf{x}_{[4]}) \leftarrow \mathsf{edge}(x_1, x_2),\ \mathsf{edge}(x_2, x_3),\ \mathsf{edge}(x_3, x_4).$$

Example 2.2 (triangles)**.** The following is a conjunctive query that asks for all (directed) triangles in the graph encoded by edge:

$$\mathsf{triangles}(x_1, x_2, x_3) \leftarrow \mathsf{edge}(x_1, x_2), \mathsf{edge}(x_2, x_3), \mathsf{edge}(x_3, x_1), \tag{2.3}$$

where x_1, x_2, x_3 are the free variables.

2.2 Join Algorithms

In this dissertation, we use the standard word-RAM model with $O(\log |\mathcal{D}|)$-bit words and unit-cost operations [CLRS22] for all complexity results. $\widetilde{O}$ is a shorthand that hides poly-logarithmic factors in the size of the input. A fundamental theme of database theory is efficient CQ evaluation, posing the foundational question:

Given a CQ and a database $\mathcal{D}$, what is the fastest join algorithm that computes $Q(\mathcal{D})$?

In practice, binary joins—i.e., joining two relations at a time—are the most commonly used strategy. However, binary join plans often incur sub-optimal asymptotic runtime, even for simple multi-way joins. In fact, in the case of three-hops-Bool (Example 2.1), any binary join sequence would require $O(|\mathcal{D}|^2)$ time, e.g. if asking for $\Pi_{x_1,x_3}(\mathsf{edge}(x_1,x_2) \bowtie \mathsf{edge}(x_2,x_3))$ first; whereas the Yannakakis algorithm [Yan81] evaluates the same query in $O(|\mathcal{D}|)$ time. The following section surveys such significant algorithmic advances for join processing over the past few decades.

2.2.1 Yannakakis Algorithm

An initial breakthrough of join algorithms is the Yannakakis algorithm [Yan81], which applies to the class of *α-acyclic* joins, such as three-hops-Bool. A CQ Q is α-acyclic if we can construct a tree $\mathcal{T}$ such that the nodes are atoms and for every variable, say x_1, the atoms that contains x_1 are connected in $\mathcal{T}$; $\mathcal{T}$ is then a *join tree* of Q. We will use the term *acyclic* for short except for Chapter 5 where we introduce other types of acyclicity. The join tree for three-hops is a simple path $\mathsf{edge}(x_1,x_2) - \mathsf{edge}(x_2,x_3) - \mathsf{edge}(x_3,x_4)$. An α-acyclic join satisfies many desirable properties and among them, Yannakakis algorithm (Algorithm 1) showed the following.

Theorem 2.1 (Yannakakis [Yan81])**.** *Let Q be an acyclic* CQ *and $\mathcal{D}$ be an input database. Then, if Q is Boolean or full, we can evaluate the query in time $O(|\mathcal{D}| + |\mathit{OUT}|)$.*

In Algorithm 1, we root a given join tree arbitrarily and denote it $(\mathcal{T}, \chi, r)$, where $\mathcal{T}$ is the tree itself, r is the designated root and χ is a mapping function that maps each atom (i.e. node) to its underlying schema. Hence the atom for a node $t \in V(\mathcal{T})$ can be identified conveniently as $R_{\chi(t)}$ of schema $\mathbf{x}_{\chi(t)}$. Recall that F is the free variables of the query and we use F_s to denote the free variables occurring in the subtree rooted at s (including free variables in $\chi(s)$ itself).

The while-loop in Algorithm 1 essentially corresponds to a bottom-up join-project plan following the join tree. The key part of the algorithm is the *full reducer* at Line 1, which is a preprocessing step after which each of the intermediate join result $T_{\chi(s)\cup F_s}$ can never be larger than the input and output sizes. The full reducer consists of two passes of linear-time semijoins: first bottom-up and then top-down; and it ensures that every input tuple that does not participate in the final query results is removed (i.e. semijoin-reduced).

Subsequent advancements extend Yannakakis algorithm to *free-connex* queries [BDG07]. A CQ Q with free variables F is called *free-connex acyclic* it is α-acyclic and the hypergraph $([n], \mathcal{E} \cup \{F\})$ is also α-acyclic. On a high level, it augments the Yannakakis algorithm to allow a very retricted pattern of projections (i.e. F being free-connex).

Algorithm 1: Yannakakis Algorithm [Yan81]

Input : acyclic query $Q(\mathbf{x}_F)$, database instance $\mathcal{D}$, rooted join tree $(\mathcal{T}, \chi, r)$

Output: $Q(\mathcal{D})$

1 $\mathcal{D} := (R_{\chi(s)})_{s \in V(\mathcal{T})} \leftarrow$ **apply** a full reducer for $\mathcal{D}$
2 $K \leftarrow$ a queue of $V(\mathcal{T})$ following a post-order traversal of $\mathcal{T}$
3 **while** $K \neq \emptyset$ **do**
4     $s \leftarrow K.pop()$
5   **if** *s is a leaf in $\mathcal{T}$* **then**
6         $T_{\chi(s)} = R_{\chi(s)}(\mathbf{x}_{\chi(s)})$
7   **else**
8         $T_{\chi(s) \cup F_s} = \Pi_{\chi(s) \cup F_s}\left(R_{\chi(s)} \wedge \left(\bigwedge_{(s,t) \in E(\mathcal{T})} \Pi_{F_t \cup (\chi(s) \cap \chi(t))} T_{\chi(t)}\right)\right)$
9         $\chi(s) \leftarrow \chi(s) \cup F_s$
10 **return** $\Pi_F(T_{\chi(r)})$

Theorem 2.2 (Bagan et al. [BDG07])**.** *Let Q be a free-connex acyclic* CQ *and $\mathcal{D}$ be an input database. Then, we can evaluate the query in time $O(|\mathcal{D}| + |\texttt{OUT}|)$. This tractability result does not hold for queries that are not free-connex acyclic, under some popular lower-bound conjectures.*

Theorem 2.2 established a linear time characterization of CQs, i.e. a CQ is free-connex acyclic if and only if it can be evaluated in linear time. The characterization excludes cyclic queries like triangles (Example 2.2); and acyclic ones having arbitrary projections, e.g. three-hops (Example 2.1) with x_1 and x_4 as free variables.

2.2.2 Tree Decompositions

Beyond acyclicity, designing join algorithms for cyclic queries—such as the triangle query (Example 2.2)—has been a long-standing focus in database theory. Recent advancements point toward a unifying strategy: reducing to acyclicity by pre-computing carefully chosen intermediate joins. These intermediate results can then act as new input relations for a final, reduced join that is acyclic by construction. The challenge now lies in judiciously selecting which intermediate joins to compute so as to avoid an explosion in intermediate result sizes. This leads naturally to the concept of *tree decompositions*, which provide a principled way to factorize the query hypergraph and guide the construction of efficient join plans.

Definition 2.1 (Tree Decomposition)**.** A *tree decomposition* of a hypergraph $\mathcal{H} = ([n], \mathcal{E})$ is a pair $(\mathcal{T}, \chi)$ where $\mathcal{T}$ is a tree and $\chi : V(\mathcal{T}) \to 2^{[n]}$ maps each node t of the tree to a subset $\chi(t)$ of vertices such that

(1) every hyperedge $F \in \mathcal{E}$ is a subset of some $\chi(t)$, $t \in V(\mathcal{T})$,

(2) for every vertex $v \in [n]$, the set $\{t \mid v \in \chi(t)\}$ is a non-empty (connected) sub-tree of $\mathcal{T}$. This is called the *running intersection property*.

The sets $e = \chi(t)$ are called the *bags* of the tree decomposition. A tree decomposition for Q is *free-connex* if it is also a tree decomposition of the hypergraph $([n], \mathcal{E} \cup \{F\})$. A query is said to be (free-connex) α-acyclic if there is a (free-connex) tree decomposition such that every bag $\chi(t)$ of the tree decomposition corresponds uniquely to an atom $R_{\chi(t)}$. Such a tree decomposition degenerates to a *join tree* of the query (and the query has to be acyclic).

A tree decomposition suggests a join algorithm in the sense that we first compute intermediate joins at each bag; then apply the Yannakakis algorithm for the final, now ayclic, join. The guarantees of such algorithms often depend on the choice of tree decomposition, which affects both the size and runtime of the intermediate joins. To quantify this, various cost measures have been proposed, such as the *generalized hypertree width* [GLS99] and tree width [GGS14, RS84].

Example 2.3. Consider the 4-cycle (full) CQ

$$Q_\diamond(\mathbf{x}_{1234}) \leftarrow R_{12}(\mathbf{x}_{12}) \wedge R_{23}(\mathbf{x}_{23}) \wedge R_{34}(\mathbf{x}_{34}) \wedge R_{14}(\mathbf{x}_{14}).$$

There are two tree decompositions (see Figure 2.2), both having a generalized hypertree width 2. If using the left tree decomposition, the intermediate joins $T_{123} = R_{12} \bowtie R_{23}$ and $T_{134} = R_{14} \bowtie R_{34}$ costs $O(|\mathcal{D}|^2)$ time (and size) each, then for $T_{123} \bowtie T_{134}$, we apply the Yannakakis algorithm in time $O(|\mathcal{D}|^2 + |\mathtt{OUT}|)$ (by Theorem 2.1).

Figure 2.2: Two rooted tree decompositions for the 4-cycle query (Example 2.3). Annotations such as T_{123} denote the intermediate join results at the corresponding bags, e.g. $T_{123} = R_{12} \bowtie R_{23}$.

2.2.3 Data Partitioning

Up to this point, we have discussed join algorithms that are essentially join-project plans—possibly having semijoins or bushy structures. In contrast, Abo Khamis et al. [KNS17] introduced a novel approach through the PANDA algorithm, which incorporates a *partitioning operator* that splits a relation into two or more disjoint parts. The key insight behind data partitioning is that different subsets of a relation may exhibit different statistical properties—such as varying value frequencies across attributes—and thus may respond differently to query plans. This observation unlocks the possibility to optimize queries by dividing the input into meaningful partitions and applying a tailored query plan to each, rather than treating the relation as a uniform whole.

Partitioning benefits join algorithms in two ways when considering tree decompositions. First, it enables efficient algorithms to join atoms in each bag of a decomposition beyond just binary joins. The groundbreaking *worst-case optimal joins algorithms* [Vel12, NPRR18, WWS23] show a refined cost measure of intermediate joins per-bag (over the generalized hypertree width), i.e. the *fractional hypertree width*. This measure can be defined using the concept of submodular functions.

Definition 2.2 (Submodularity)**.** A function $f : 2^{[n]} \mapsto \mathbb{R}_+$ is a non-negative *set function* on $[n]$ ($n \geq 1$). The set function is

- *monotone* if $f(X) \leq f(Y)$ whenever $X \subseteq Y$, and
- *submodular* if $f(X \cup Y) + f(X \cap Y) \leq f(X) + f(Y)$, for all $X, Y \subseteq [n]$

A non-negative, monotone, submodular set function h such that $h(\emptyset) = 0$ is a *polymatroid*. Let Q be a CQ associated with a hypergraph $\mathcal{H} = ([n], \mathcal{E})$. Let Γ_n be the set of all polymatroids h on $[n]$ such that $h(K) \leq 1$ for all $K \in \mathcal{E}$. The *fractional hypertree width* of Q is

$$\mathsf{fhw}(Q) := \min_{(\mathcal{T},\chi)\in\mathcal{F}} \max_{h\in\Gamma_n} \max_{t\in V(\mathcal{T})} h(\chi(t)) \tag{2.4}$$

where $\mathcal{F}$ is the set of all *non-redundant* tree decompositions of φ. A tree decomposition is *non-redundant* if no bag is a subset of another. Abo Khamis et al. [KNS17] proved that non-redundancy ensures that $\mathcal{F}$ is finite, hence the outer minimum is well-defined.

Intuitively, the fractional hypertree width captures the worst-case size of the largest bag (the inner max), assuming the best possible tree decomposition is chosen (the outer min). This quantity directly governs the worst-case complexity of the query, as shown in the following theorem.

Theorem 2.3 ([NPRR18, KNS17])**.** *Let Q be a* CQ *with fractional hypertree width* $\mathsf{fhw}(Q)$*. If Q is full or Boolean, we can evaluate Q in time $O(|\mathcal{D}|^{\mathsf{fhw}(Q)} + |\mathit{OUT}|)$, where $|\mathit{OUT}|$ is the output size.*

Example 2.4. Consider the following triangle query

$$Q_\Delta(x_1, x_2, x_3) \leftarrow R_{12}(x_1, x_2), R_{23}(x_2, x_3), R_{13}(x_1, x_3). \tag{2.5}$$

which subsumes the $\mathsf{triangles}$ query (Example 2.2) as a special case. Take a query plan that attempts to first join R_{12} and R_{23}. The problem of this strategy is that in the worst case its output size can be $O(N^2)$, when R_{12}, R_{23} have a single value for attribute x_2 that repeat $O(N)$ times. To address this, we will split R_{12} into two parts: the *light* part R_{12}^L has the tuples where a value of x_2 appears $\leq \sqrt{N}$ times, and the heavy part R_{12}^H otherwise. It turns out that we can now join R^L and S with a much smaller blowup of the intermediate size: $|R_{12}^L \bowtie R_{23}| \leq N \cdot \sqrt{N} = N^{3/2}$. After this join, we can semijoin with R_{13} for the first output. But for R_{12}^H we need to follow a different join plan: we will instead enforce $R_{12}^H \bowtie R_{13}$ first. The key idea here is that there can be at most $\sqrt{N}$ values of x_2 that appear in R_{12}^H, hence $|R_{12}^H \bowtie R_{13}| \leq \sqrt{N} \cdot N = N^{3/2}$. By unioning both outputs, we end the join algorithm in time $O(N^{3/2})$. Indeed, the query has a trivial tree decomposition of putting all three atoms in one bag, having fractional hypertree width of 3/2.

A second benefit of data partitioning arises in using multiple tree decompositions (i.e., multiple join plans). Consider $Q_\diamond$ as an example. As Example 2.3 points out, either tree decomposition leads to a worst-case intermediate join size of $O(|\mathcal{D}|^2)$—indeed, its fractional hypertree width (fhw) is 2. To go beyond, we we must exploit multiple tree decompositions and partition the data such that each decomposition guides the plan of a subpart of the data. Yet, it is now highly non-trivial since not only do we need to find the best possible tree decomposition for each subpart of data, we also need to determine a partitioning scheme across tree decompositions. Abo Khamis et al. [KNS17] proposed the PANDA algorithm that achieves this goal, which is the state-of-the-art join algorithm for most Boolean or full CQs. More precisely, it attains the more nuanced *submodular width* proposed by [Mar10].

Definition 2.3 (Definition 2.2 continued)**.** Following the definition of submodular functions, we can define the *submodular width* as follows. The *submodular width* of a CQ Q is

$$\mathsf{subw}(Q) := \max_{h \in \Gamma_n} \min_{(\mathcal{T}, \chi) \in \mathcal{F}} \max_{t \in V(\mathcal{T})} h(\chi(t)), \tag{2.6}$$

and it is easy to see that $\mathsf{subw}(Q) \leq \mathsf{fhw}(Q)$ for any CQ Q. For $Q_\diamond$, the submodular width is 3/2.

2.2.4 PANDA Algorithm

This section briefly summarizes the PANDA algorithm [KNS17].

Theorem 2.4 ([KNS17]). *Let Q be a CQ with submodular width $\mathsf{subw}(Q)$. If Q is full or Boolean, we can evaluate Q in time $\widetilde{O}(|\mathcal{D}|^{\mathsf{subw}(Q)} + |\mathit{OUT}|)$, where $|\mathit{OUT}|$ is the output size and $\widetilde{O}$ hides a polynomial factor with respect to the input size $|\mathcal{D}|$.*

A core mission of PANDA is to find a data partitioning scheme among decompositions while for each bag of every tree decomposition, joining the assigned data pieces efficiently. If assuming every bag is has a corresponding materialized view, the join algorithm would be running one Yannakakis algorithm per decomposition, and union the results into one. PANDA addresses the mission through the concept of disjunctive rules and Shannon-flow inequalities.

Definition 2.4 (Disjunctive Rules). A disjunctive rule has the exact body of a CQ, while the head is a disjunction of (intermediate) output relations $T_B(\mathbf{x}_B)$, which we call *targets*. Let $\mathsf{B} \subseteq 2^{[n]}$ be a non-empty set, then a *disjunctive rule* ρ has the formulation:

$$\rho: \quad \bigvee_{B \in \mathsf{B}} T_B(\mathbf{x}_B) \leftarrow \bigwedge_{K \in \mathcal{E}} R_K(\mathbf{x}_K). \tag{2.7}$$

Given a database instance $\mathcal{D}$, a *model* of ρ is a tuple $(T_B)_{B \in \mathsf{B}}$ of relations, one for each target, such that the logical implication indicated by (2.7) holds. More precisely, for any tuple $\mathbf{a}$ that satisfies the body, there is a target $T_B \in (T_B)_{B \in \mathsf{B}}$ such that $\Pi_B(\mathbf{a}) \in T_B$. The *size* of a model is defined as the maximum size of its output relations and the *output size* of a disjunctive rule ρ, denoted as $|\rho|$, is defined as the minimum size over all models.

PANDA considers the set of all non-redundant tree decompositions of the query and to construct one disjunctive rule, it selects one bag per tree decomposition to be one of the targets. It constructs every possible disjunctive rule ρ in such a way, where the targets are the bags of chosen from the tree decompositions, one per decomposition (see Example 2.5).

Example 2.5. Consider the query $Q_\diamond$ (Example 2.3). The two tree decompositions in Figure 2.2 yield four disjunctive rules, each having two targets. The four disjunctive rules are:

$$\begin{aligned}
\rho_1: &\quad T_{134}(\mathbf{x}_{134}) \vee T_{234}(\mathbf{x}_{234}) \leftarrow R_{12}(\mathbf{x}_{12}), R_{23}(\mathbf{x}_{23}), R_{34}(\mathbf{x}_{34}), R_{14}(\mathbf{x}_{14}) \\
\rho_2: &\quad T_{134}(\mathbf{x}_{134}) \vee T_{124}(\mathbf{x}_{124}) \leftarrow R_{12}(\mathbf{x}_{12}), R_{23}(\mathbf{x}_{23}), R_{34}(\mathbf{x}_{34}), R_{14}(\mathbf{x}_{14}) \\
\rho_3: &\quad T_{123}(\mathbf{x}_{123}) \vee T_{234}(\mathbf{x}_{234}) \leftarrow R_{12}(\mathbf{x}_{12}), R_{23}(\mathbf{x}_{23}), R_{34}(\mathbf{x}_{34}), R_{14}(\mathbf{x}_{14}) \\
\rho_4: &\quad T_{123}(\mathbf{x}_{123}) \vee T_{124}(\mathbf{x}_{124}) \leftarrow R_{12}(\mathbf{x}_{12}), R_{23}(\mathbf{x}_{23}), R_{34}(\mathbf{x}_{34}), R_{14}(\mathbf{x}_{14})
\end{aligned} \tag{2.8}$$

Intuitively, a model is a consequence of splitting the input and joining corresponding pieces into bags selected from tree decompositions, e.g. a model of ρ_1, say (T_{134}, T_{234}), puts part of the data into T_{134} (the first bag of the left decomposition) and the other into T_{234} (the first bag of the right decomposition).

A desirable property of the disjunctive rules is that: if we can find a model of each disjunctive rule, then we can recover the query result $Q_\diamond$ by first unioning the targets, e.g. T_{134} from ρ_1 and ρ_2, then applying the Yannakakis algorithm on every decomposition, and finally collect partial results from each Yannakakis run as output. The formal proofs of correctness can be found in Abo Khamis et al. [KNS17].

Now the question becomes how to find a model smallest possible of a disjunctive rule efficiently, to which PANDA introduces *Shannon-flow inequalities*. A Shannon-flow inequality essentially conveys a size and runtime guarantees on the models. We will use Q_Δ (Example 2.4) as an example here as it only has one disjunctive rule as there is only one tree decomposition with one bag. The disjunctive rule is the query itself.

Definition 2.5 (Shannon-flow Inequalities)**.** Following Definition 2.2 of submodular functions, the following inequality (let $h(Y|X) = h(Y) - h(X)$)

$$\sum_{X \subset Y \subseteq [n]} \delta_{Y|X} \cdot h(Y|X) := \sum_{X \subset Y \subseteq [n]} \delta_{Y|X} \cdot (h(Y) - h(X)) \geq \sum_{\emptyset \neq Z \subseteq [n]} \lambda_{Z|\emptyset} \cdot h(Z|\emptyset), \tag{2.9}$$

is called a *Shannon-flow inequality* if it holds for any polymatroid function $h \in \Gamma_n$ and all $\delta_{Y|X}$ and $\lambda_{Z|\emptyset}$ are some non-negative rational coefficients, i.e., $\delta_{Y|X}, \lambda_{Z|\emptyset} \in \mathbf{Q}_+$. To concisely represent Shannon-flow inequalities, we define the conditional polymatroid as in [KNS17].

Definition 2.6 (Conditional Polymatroids)**.** Let $\mathcal{C} \subseteq 2^{[n]} \times 2^{[n]}$ denote the set of all pairs (X, Y) such that $\emptyset \subseteq X \subset Y \subseteq [n]$. A vector $\boldsymbol{h} \in \mathbf{R}_+^{\mathcal{C}}$ has coordinates indexed by pairs $(X, Y) \in \mathcal{C}$ and we denote the corresponding coordinate value of $\boldsymbol{h}$ by $h(Y|X)$. A vector $\boldsymbol{h}$ is called a *conditional polymatroid* if and only if there is a polymatroid h such that $h(Y|X) = h(Y) - h(X)$; and, we say that the polymatroid h defines the conditional polymatroid $\boldsymbol{h}$. In particular, $\boldsymbol{h} = (h(Y|X))_{(X,Y)\in\mathcal{C}}$.

If for (2.9), we ask $\lambda_{Z|X} = 0$ for any $\emptyset \neq X \subset Z \subseteq [n]$, then each of $\{\delta_{Y|X}\}, \{\lambda_{Y|X}\}$ can be interpreted as vectors over (X, Y) pairs, where $\emptyset \subseteq X \subset Y \subseteq [n]$. We denote them as $\boldsymbol{\delta}, \boldsymbol{\lambda}$, respectively. Thus, the Shannon-flow inequality (2.9) can be re-written into an inequality on conditional polymatroids, i.e. the $\mathbf{Q}_+^{\mathcal{C}}$ space, as $\langle \boldsymbol{\delta}, \boldsymbol{h} \rangle \geq \langle \boldsymbol{\lambda}, \boldsymbol{h} \rangle$, where $\langle \cdot, \cdot \rangle$ denotes dot product.

Definition 2.7 (Proof Sequence)**.** A major contribution from [KNS17] says that any Shannon-flow inequality $\langle \boldsymbol{\delta}, \boldsymbol{h} \rangle \geq \langle \boldsymbol{\lambda}, \boldsymbol{h} \rangle$ can be proved by just applying the following 4 rules:

(R1) submodularity rule	$h(I \cup J|J) - h(I|I \cap J) \leq 0,$	$I \perp J$
(R2) monotonicity rule	$-h(Y|\emptyset) + h(X|\emptyset) \leq 0,$	$X \subset Y$
(R3) composition rule	$h(Y|\emptyset) - h(Y|X) - h(X|\emptyset) \leq 0,$	$X \subset Y$
(R4) decomposition rule	$-h(Y|\emptyset) + h(Y|X) + h(X|\emptyset) \leq 0,$	$X \subset Y$

where $I \perp J$ means $I \nsubseteq J$ and $J \nsubseteq I$. $(R1), (R2)$ come exactly from the submodularity and monotonicity properties of polymatroids; $(R3), (R4)$ simply follow from the definition of $h(Y|X) = h(Y) - h(X)$. All the rules can be vectorized over all (X, Y) pairs, where $\emptyset \subseteq X \subset Y \subseteq [n]$. For every $I \perp J$, we define a vector $\mathbf{s}_{I,J}$, and for every $X \subset Y$, we define three vectors $\mathbf{m}_{X,Y}, \mathbf{c}_{X,Y}, \boldsymbol{d}_{Y,X}$ such that the linear rules above can be written using dot-products:

(R1) submodularity rule	$\langle \mathbf{s}_{I,J}, \boldsymbol{h} \rangle \leq 0,$	$I \perp J$
(R2) monotonicity rule	$\langle \mathbf{m}_{X,Y}, \boldsymbol{h} \rangle \leq 0,$	$X \subset Y$
(R3) composition rule	$\langle \mathbf{c}_{X,Y}, \boldsymbol{h} \rangle \leq 0,$	$X \subset Y$
(R4) decomposition rule	$\langle \boldsymbol{d}_{Y,X}, \boldsymbol{h} \rangle \leq 0,$	$X \subset Y$

Proof sequences provide a constructive way to justify Shannon-flow inequalities. More significantly, they yield a step-by-step join-project-partition plan that can be used to evaluate a disjunctive rule efficiently—both in time and output size—in $\widetilde{O}(|\mathcal{D}|^{\mathsf{subw}(Q)})$, where $\mathsf{subw}(Q)$ denotes the submodular width of the query Q. More precisely, each step in the proof sequence maps to a relational operation: monotonicity rules translate to projections, composition rules correspond to joins, and decomposition rules become the partitioning steps. The submodularity rule is essentially a no-op, altering the proof structure without touching the data. The following result (also Theorem 2 in [WY22]) is a direct consequence of Theorem B.12 and Proposition B.13 from [KNS17], and underpins the correctness and efficiency of the PANDA algorithm.

Theorem 2.5 ([KNS17])**.** *For any Shannon-flow inequality $\langle \boldsymbol{\delta}, \boldsymbol{h} \rangle \geq \langle \boldsymbol{\lambda}, \boldsymbol{h} \rangle$ such that $\|\boldsymbol{\lambda}\|_1 = 1$, there is a proof sequence of length $O(poly(2^n))$.*

Theorem 2.5 implies that there is a proof sequence for a Shannon-flow inequality that is exponentially long in the number of variables n in the query (assumed to be fixed), but it is of length independent of the size of the database.

Example 2.6 (Example 2.4 continued)**.** The following is a Shannon-flow inequality for the triangle query (Example 2.4):

$$\frac{1}{2}h(12) + \frac{1}{2}h(23) + \frac{1}{2}h(13) \geq h(123),$$

which implies a size and runtime bound of $O(|\mathcal{D}|^{3/2})$ since $h(12), h(23), h(13) \leq \log|\mathcal{D}|$ encodes the input size constraints. The inequality can be proved by the following proof sequence:

$$\begin{aligned}
\frac{1}{2}h(12) + \frac{1}{2}h(23) + \frac{1}{2}h(13) &\geq \frac{1}{2}h(12) + \frac{1}{2}h(23) + \frac{1}{2}h(123|2) && \textit{(submodularity)}\\
&\geq \frac{1}{2}h(2) + \frac{1}{2}h(12|2) + \frac{1}{2}h(23) + \frac{1}{2}h(123|2) && \textit{(decomposition)}\\
&\geq \left(\frac{1}{2}h(2) + \frac{1}{2}h(123|2)\right) + \left(\frac{1}{2}h(123|23) + \frac{1}{2}h(23)\right) && \textit{(submodularity)}\\
&\geq h(123) && \textit{(composition)}
\end{aligned}$$

We can see that the proof sequence hints exactly the join algorithm outlined in Example 2.4, where the first parenthesis indicates the heavy join of R_{12}^H and R_{13}, and the second parenthesis indicates the light join of R_{12}^L and R_{23}. Indeed, both parentheses use the composition rule.

2.3 Algebraic Semantics

A CQ assumes the relational set semantics where a tuple either appears in a relation or not. Many real-world applications demand alternative semantics for join queries: SQL uses bag semantics (and so a tuple may duplicate in a table). In other scenarios, we may want to group-by aggregate over join results—such as sums, max, min, or counts. Remarkably, many of these diverse semantics can be captured by a single, elegant algebraic structure of *semirings*.

Definition 2.8 (Semiring)**.** A tuple $\sigma = (\Sigma, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is a (commutative) *semiring* if $\oplus$ and $\otimes$ are binary operators over Σ for which:

1. $(\Sigma, \oplus, \mathbf{0})$ is a commutative monoid with additive identity $\mathbf{0}$ (i.e., $\oplus$ is associative and commutative, and $a \oplus 0 = a$ for all $a \in D$);
2. $(\Sigma, \otimes, \mathbf{1})$ is a (commutative) monoid with multiplicative identity $\mathbf{1}$ for $\otimes$;
3. $\otimes$ distributes over $\oplus$, i.e., $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ for $a, b, c \in \Sigma$; and
4. $a \otimes \mathbf{0} = \mathbf{0}$ for all $a \in \Sigma$.

Such examples include the Boolean semiring $\mathbb{B} = (\{\mathsf{false}, \mathsf{true}\}, \vee, \wedge, \mathsf{false}, \mathsf{true})$, the natural numbers semiring $(\mathbb{N}, +, \cdot, 0, 1)$, and the *tropical semiring* $\mathsf{Trop}^+ = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$. A semiring is *idempotent* (also called a *dioid*) if $\oplus$ is idempotent, i.e., $a \oplus a = a$ for all $a \in \Sigma$. In this dissertation, we restrict our attention to commutative semirings, and will henceforth refer to them as *semirings*.

Green et al. [GKT07] introduced the seminal idea of using annotations of a semiring to reason about data provenance over joins. In this framework, every relation R_K is modeled as an *σ-relation*,

or a *factor*, where every tuple is annotated by an element from the semiring domain Σ. Tuples not in the relation are annotated by $\mathbf{0} \in \Sigma$ implicitly. Equivalently, a factor R_K can be viewed as a function—hence the name *factor*—that maps tuples to their annotations, i.e. $R_K : \mathsf{Dom}(\mathbf{x}_K) \to \Sigma$. Following the convention in [AKNR16], we assume all σ-relations are represented via the *listing representation*: each factor R_K is stored as a table of pairs $\langle \mathbf{a}_K, R_K(\mathbf{a}_K)\rangle$, where $\mathbf{a}_K$ is a tuple of the relation and $R_K(\mathbf{a}_K) \in \Sigma$ is the *weight* of the tuple. In set-theoretic contexts, we may also abuse the notation R_K to denote the set of tuples of schema $\mathbf{x}_K$ explicitly stored in the factor table.

Standard (set-based) relations correspond to $\mathbb{B}$-relations where false or true annotations indicate the absence or presence of a tuple in the relation. Over the natural numbers semiring $(\mathbb{N}, +, \cdot, 0, 1)$, the annotation encodes the multiplicity of a tuple, reflecting bag semantics. The semantic of a join can be naturally lifted to semirings by interpreting projection as $\oplus$ and join as $\otimes$. Abo Khamis et al. [AKNR16] introduced *functional aggregate queries* (FAQ) that express join-aggregate queries through semirings. A FAQ φ over a semiring σ has the following formulation:

$$\varphi(\mathbf{x}_F) \leftarrow \bigoplus_{\mathbf{x}_{[n]\setminus F}} \bigotimes_{K \in \mathcal{E}} R_K(\mathbf{x}_K), \tag{2.10}$$

where (i) $([n], \mathcal{E})$ is the *associated hypergraph* of φ (the hyperedges $\mathcal{E} \subseteq 2^{[n]}$), (ii) $\mathbf{x}_F$ are the head variables ($F \subseteq [n]$), and (iii) each R_K is an input σ-relation of schema $\mathbf{x}_K$ and we use $\mathcal{D} = (R_K)_{K \in \mathcal{E}}$ to denote an input database instance (but they are now σ-relations). Acyclicity for FAQs is identical to CQs, through its associated hypergraph. Similar to $Q(\mathcal{D})$, we use $\varphi(\mathcal{D})$ to denote the query result of $\varphi(\mathbf{x}_F)$ evaluated with input $\mathcal{D}$, i.e. the resulting σ-relation of schema $\mathbf{x}_F$.

Example 2.7. Consider the following FAQ over the natural numbers semiring $(\mathbb{N}, +, \cdot, 0, 1)$, derived from Example 2.1:

$$\mathsf{three\text{-}hops\text{-}Cnt}() \leftarrow \bigoplus_{\mathbf{x}_{[4]}} \mathsf{edge}(x_1, x_2) \otimes \mathsf{edge}(x_2, x_3) \otimes \mathsf{edge}(x_3, x_4). \tag{2.11}$$

Unlike three-hops-Bool that detects the existence of a three-hop path, three-hops-Cnt counts the total number of such paths in the graph.

At first glance, FAQ may appear more involved than standard CQ. However, most join algorithms discussed in this chapter extend naturally to FAQ with minimal changes! Indeed, the Yannakakis algorithm applies directly to acyclic FAQ and retains identical linear-time characterizations as in the CQ case (Theorem 2.1 and Theorem 2.2) [AKNR16]. The InsideOut algorithm [AKNR16], which generalizes worst-case optimal join algorithms, further shows that Theorem 2.3 continues to hold for cyclic FAQs. For idempotent semirings (or dioids), Zhao et al. [ZDK+24b] showed that the PANDA algorithm can be modified to attain identical complexity results as Theorem 2.4.

2.4 Datalog Basics

Beyond CQs, Datalog is a powerful yet elegant declarative query language for expressing recursive queries over databases. We review standard Datalog [AHV95] that consists of a finite set of *rules* over a set of *extensional* and *intensional* relations (termed as EDBs and IDBs respectively, henceforth). EDBs correspond to relations in a given database, each comprising a set of EDB tuples/facts, whereas IDBs are derived by the rules. The head atom of each rule is an IDB, and the body consists of zero or more EDBs and IDBs as a conjunctive query (2.1) defining how the head IDB is derived. That is, a rule represents a logical implication: if the body is true, so is the head of the rule. We illustrate with the textbook example of transitive closure on a binary EDB R denoting edges in a directed graph, a single IDB T, and two rules:

$$T(x_1, x_2) \leftarrow R(x_1, x_2). \tag{2.12}$$

$$T(x_1, x_2) \leftarrow T(x_1, x_3) \wedge R(x_3, x_2). \tag{2.13}$$

In standard Datalog, IDBs are derived given EDBs (or equivalently, evaluation proceeds over the Boolean semiring). Similar to conjunctive queries, Datalog evaluation can be generalized to semirings [KNP+22a]. In this setting, both EDBs and IDBs are annotated with elements from a semiring σ—that is, they are now σ-relations. This generalization enables the expression of recursive aggregation patterns, such as All-Pairs Shortest Paths (APSP) when evaluating (2.12) over the tropical semiring Trop^+. However, evaluating such programs requires careful attention to their convergence properties. Formal definitions and convergence guarantees are discussed in Chapter 7.

From a practical standpoint, many Datalog engines have been developed and studied [SJSW16, MYL16, SYI+16, FZZ+19, FMK22, KNP+22b, SGM22, RB19]. To support real-world applications, these systems typically extend the Datalog language with additional constructs such as constraints, stratified negation [AG94], and arithmetic operations. We will present a detailed discussion of these extensions and systems in Chapter 7.

Chapter 3

Related Work

In this chapter, we discuss some related work to this dissertation.

3.1 Extensions of Join Algorithms

Recent research on join algorithms has increasingly focus on extending classical join algorithms towards more expressive query constructs. This shift aligns closely with the goal of this dissertation.

These work have introduced join algorithms that go beyond traditional heuristic-based optimizations. For example, recent research [WY23, KCM+20] introduced join algorithms capable of handling inter-relational predicates that span multiple joins such as $R_{12}(x_1, x_2) \wedge R_{23}(x_2, x_3) \wedge (x_1 \leq x_3)$. Koutris et al. [KMRS17] addresses similar queries, but for disequality predicates such as $x_1 \neq x_3$, which also resist traditional strategies like predicate pushdown. In both cases, the proposed algorithms introduce new foundational techniques that address the shortcomings of classical approaches.

More recently, researchers extend the Yannakakis algorithm to handle top-k queries [TGR20, DHK22], where the goal is to return only the k highest-ranked tuples according to a scoring function. These approaches prioritize early pruning and rank-aware evaluation to avoid materializing the full join output. Wang et al. [HW23] introduced the idea of *difference pushdown* for evaluating differences of join queries, where the difference operator is pushed down to the join level instead of applying it after evaluating both subqueries. We will revisit this idea in Chapter 5, where we generalize it within a broader framework for handling negation in join queries. Additionally, recent advances in consistent query answering [FKOW23, KOW21, KOW24] enhance modern join algorithms to work over inconsistent datasets. Their method ensures that the results remain consistent and interpretable despite underlying data irregularities.

Across these works, a unifying theme is the development of new algorithmic principles that are theoretically sound and practically effective. These contributions often result in orders-of-magnitude performance improvements and set the stage for modern query optimization techniques.

3.2 Leveraging Statistical Information

Classic join algorithms—such as the Yannakakis algorithm [Yan81] and worst-case optimal joins (WCOJ) [NPRR18]—typically reason about performance using only two parameters: the input database size $|\mathcal{D}|$ and the of the output $|\texttt{OUT}|$. While these algorithms provide worst-case optimal guarantees under this model, it often fails to capture the nuances of real-world data. This is largely because the input and output sizes alone do not capture the full structure or statistical properties of the database instance. A query optimizer that uses more fine-grained statistical information can often select plans that significantly outperform worst-case optimal ones in practice—even if those plans are not worst-case optimal.

Consider again the case of the triangle query (Figure 1.1, right) and now suppose that we know that x_2 is a primary key in the relation $S(x_2, x_3)$. If we reason solely based on $|\mathcal{D}|$ and $|\texttt{OUT}|$, a WCOJ algorithm runs in time $O(|\mathcal{D}|^{3/2} + |\texttt{OUT}|)$. However, if we exploit the primary key constraint and first join $R(x_1, x_2)$ and $S(x_2, x_3)$, the intermediate join result would have size $O(|\mathcal{D}|)$, and the total runtime could be tightened to $O(|\mathcal{D}| + |\texttt{OUT}|)$. This simple example illustrates how integrity constraints—such as primary keys—can be used to guide the optimizer toward significantly more efficient plans.

Primary keys are just one type of useful statistical information. Others include cardinality constraints (i.e. relation sizes), functional dependencies, foreign-key constraints, and more generally, degree constraints [KNS17]. A degree constraint states that for disjoint attribute sets $\mathbf{x}_A$ and $\mathbf{x}_B$ in a relation R, the number of $\mathbf{x}_A$ values associated with any fixed $\mathbf{x}_B$ value is bounded by some constant $d > 0$. Degree constraints subsume both cardinality constraints and functional dependencies, and are naturally integrated into PANDA [KNS17], and many our own contributions (Chapter 6 and Chapter 7), which supports join-project-partition plans with tighter runtime guarantees and refined query plans.

Recent research has taken this idea even further by proposing join algorithms that leverage richer classes of statistical properties:

- Degree sequences [DSBC23] represent the distribution of frequencies (i.e., how many $\mathbf{a}_A$ values exist per $\mathbf{a}_B$) rather than just a single upper bound.

- Norm bounds [KNOS24, ZMK+25] introduce ℓ_p-norms to compactly capture these frequency distributions.

- Partition constraints [DM25] generalize degree constraints and incorporate notions of degeneracy from graph theory.

These more expressive statistical models enable more accurate cost estimation and allow for join plans that are better tailored to the instance at hand. However, they come with a tradeoff: collecting and maintaining fine-grained statistics is more expensive, and may be infeasible in some cases. Consequently, a central open problem remains—how to strike the right balance between the granularity (and type) of statistical information and the overall efficiency of join algorithms. This question lies at the heart of narrowing the gap between theoretical and practical query optimization.

3.3 Enumeration-delay Algorithms

The efficiency of an algorithm is typically measured by the total time it takes to produce the output. This approach is natural when the output is a single scalar value—such as an aggregate. However, the query output is often a relation consisting of many tuples, many times significantly larger than the input data. This disparity invites a more fine-grained approach to measuring the runtimes of join algorithms.

In fact, a join algorithm for a query Q can be viewed as running in two distinct phases:

- **Pre-processing phase:** The algorithm first reads the input instance $\mathcal{D}$ and constructs an intermediate data structure $\mathcal{L}$. The objective in this phase is to minimize the total preprocessing time or space, ideally linear in $|\mathcal{D}|$.

- **Enumeration phase:** After preprocessing, the algorithm begins emitting the result tuples from $\mathcal{L}$. The key metric here is the maximum time between producing any two consecutive output tuples, including the time to produce the first one and the time to detect the end of the output. This quantity is termed as the *delay*.

The ideal goal is to achieve *constant-delay enumeration*, where the time between two consecutive outputs is $O(1)$ [Seg13]. This is a strong guarantee, as it implies that if only k tuples of the output are needed, the total cost is $O(|\mathcal{D}|+k)$. Furthermore, when the output may be reused multiple times, we can store just the intermediate structure $\mathcal{L}$—rather than materializing the full output [ND].

It turns out that we can modify Yannakakis algorithm (Theorem 2.1) such that we can have the ideal linear preprocessing time and constant-delay enumeration [BDG07].

Theorem 3.1 (Bagan et al. [BDG07])**.** *Let Q be a free-connex acyclic* CQ *and $\mathcal{D}$ be a database instance. Then, we can enumerate $Q(\mathcal{D})$ with linear preprocessing time $O(|\mathcal{D}|)$ and constant delay.*

In fact, leveraging this result with the PANDA framework, we can extend constant-delay enumeration to arbitrary join queries. The resulting algorithm achieves preprocessing time of $\widetilde{O}(|\mathcal{D}|^{\mathsf{subw}})$ and still guarantees constant delay.

The idea of intermediate (succinct) data structures for fast enumeration naturally leads to the concept of *factorized representations* of query results [OZ15, OS16]. One major advantage of factorization is that it often supports post-processing tasks—such as filtering and aggregation—directly on the compact representation, without decompressing it [SOC16, KNN+18].

Enumeration-delay Trade-offs. While constant-delay enumeration is desirable, it often requires storing a potentially large intermediate structure. In scenarios where memory is constrained or when preprocessing time must be minimized, we may want to trade a slightly higher delay for lower preprocessing cost and reduced space usage. Recent work, such as our own contributions in Chapter 6, explores these tradeoffs systematically for various classes of join queries [KNOZ20, Yao82, DHK23, DLT23b, KNOZ23b]. These tradeoff points enrich the design space of join algorithms and provide more flexibility for practical deployment.

Ranked Enumeration. Another practical perspective is *ranked enumeration*, where output tuples are produced in a specific order determined by a ranking function. For example, users might wish to see tuples ranked by the sum of their numeric attributes or by lexicographic order. In this setting, the goal remains to minimize the delay between consecutive outputs.

Recent results show that, for most useful ranking functions over acyclic full joins, we can achieve linear preprocessing time and $O(\log|\mathcal{D}|)$ delay [TAG+20, DK21]. This means that ranked enumeration adds only a logarithmic overhead compared to unordered enumeration. Moreover, these techniques have been generalized to accommodate queries with projections [DHK22, CTG+23].

3.4 Lower Bounds for Join Queries

An orthogonal direction that complements the design of efficient join algorithms is the study of *lower bounds*. Despite its fundamental significance, progress on lower bounds for join processing has been limited and has only gained traction in recent years. One promising approach for establishing lower bounds is through fine-grained complexity. This method begins with a problem that is widely believed to be intractable and attempts to reduce join query evaluation to it. A starting point in this is the following two conjectures for the k-clique problem:

Definition 3.1 (Boolean k-Clique Conjecture)**.** There is no real $\epsilon > 0$ such that computing the k-clique problem (with $k \geq 3$) over the Boolean semiring in an (undirected) n-node graph requires time $O(n^{k-\epsilon})$ using a combinatorial algorithm.

Definition 3.2 (Min-Weight k-Clique Conjecture)**.** There is no real $\epsilon > 0$ such that computing the k-clique problem (with $k \geq 3$) over the tropical semiring in an (undirected) n-node graph with integer edge weights can be done in time $O(n^{k-\epsilon})$.

Recent work [FKZ23] has leveraged the min-weight k-clique conjecture to derive conditional lower bounds for evaluating Booleam CQ over the tropical semiring. The authors introduce a parameter known as the *clique embedding power* of a join query, denoted clemb, and proves a lower bound of $\Omega(n^{\mathsf{clemb}})$. It is known that $\mathsf{clemb} \leq \mathsf{subw}$, and in fact it was recently shown that this is tight for joins on binary relations with $\mathsf{subw} < 2$ [BG24]. As an example, this tells us that our WCOJ algorithm is optimal for the triangle query (Figure 1.1, right), i.e. $\mathsf{clemb} = \mathsf{subw} = 3/2$ and hence a lower bound of $\Omega(|\mathcal{D}|^{3/2})$ for the triangle query. For example, this result implies that our worst-case optimal join (WCOJ) algorithm is optimal for the triangle query (Figure 1.1, right), where $\mathsf{clemb} = \mathsf{subw} = 3/2$, leading to a lower bound of $\Omega(|\mathcal{D}|^{3/2})$.

A second approach to proving lower bounds examines simpler computational models. Recent work [FKZ24] has investigated *semiring circuits and formulas*, which aim to characterize the minimal number of join and union operations required to compute a query's output—reflecting the inherent computational difficulty of the query. For example, joining a tuple (a, b) with (b, c) on b counts as a single join operation. In this framework, lower bounds on the size of semiring circuits and formulas are established using the notion of *entropic width*, leading to bounds on the minimum circuit size of the form $\Omega(|\mathcal{D}|^{\mathsf{entw}})$ [KNS17]. It is known that $\mathsf{entw} \leq \mathsf{subw}$, but whether this inequality is ever strict remains an open question. Similar to the fine-grained complexity approach, existing results are currently limited to the tropical semiring, and generalizing them to other semirings or broader query classes remains an active area of research.

Despite recent progress, many gaps remain. In particular, existing lower bounds predominantly apply to Boolean queries or full conjunctive queries and do not cover more expressive queries involving projections, negations, or modern query constructs. In this dissertation, alongside proposing new join algorithms for such queries, we also develop new lower bounds where possible—extending the above two frameworks to accommodate broader query features and patterns.

Chapter 4

Join with Projections

The seminal work of Yannakakis [Yan81] showed that for a database $\mathcal{D}$ of size $|\mathcal{D}|$ and any acyclic full or Boolean conjunctive query (CQ) $Q(\mathbf{x}_F)$ with output $\mathtt{OUT} = Q(\mathcal{D})$, we can evaluate the query result in time $O(|\mathcal{D}| + |\mathtt{OUT}|)$. This result was later extended by Bagan et al. [BDG07] to the broader class of free-connex acyclic queries, providing a linear-time characterization of such CQs. However, when applying the Yannakakis algorithm to CQs with arbitrary free variables $\mathbf{x}_F$ (i.e., arbitrary projections), the runtime deteriorates to a polynomial bound of $O(|\mathcal{D}| + |\mathcal{D}| \cdot |\mathtt{OUT}|)$.

Beyond acyclic queries, the PANDA algorithm [KNS17] provide worst-case guarantees for any full or Boolean CQ. Abo Khamis et al. [KNS17] showed that using the PANDA algorithm along with the Yannakakis framework, any CQ can be evaluated in $\tilde{O}(|\mathcal{D}| + |\mathcal{D}|^{\mathsf{subw}} \cdot |\mathtt{OUT}|)$ time[1]. Notably, the PANDA algorithm itself uses the Yannakakis algorithm as the final step for join evaluation once the cyclic query has been transformed into a set of acyclic queries. Berkholz and Schweikardt [BS19] proposed a measure known as the *free-connex submodular width* that allows additional enumeration related guarantees and thus, can evaluate certain classes of CQs efficiently.

Despite its fundamental importance, no known improvements have been made to Yannakakis result until very recently. In an elegant result, Hu [Hu24a] showed that using fast matrix multiplication (i.e., a matrix multiplication algorithm that multiplies two $n \times n$ matrices in $O(n^{\omega})$ time, $\omega < 3$), we can evaluate any acyclic CQ in $O(|\mathcal{D}| + |\mathtt{OUT}| + |\mathcal{D}| \cdot |\mathtt{OUT}|^{5/6})$ (if $\omega = 2$, which gives the strongest result in their framework). The key idea is to use the star query as a fundamental primitive that benefits from fast matrix multiplication, an insight also explored by prior work [AP09, DHK20], and evaluate any acyclic CQ in a bottom-up fashion by repeatedly applying the star query primitive. However, Hu's result has two principal limitations. First, the algorithm is non-combinatorial in nature, an undesirable property from a practical standpoint. Second, the algorithm does not support general aggregations (a.k.a. FAQs cf. Section 4.4), a common use case in $\mathtt{SQL}$ [Dat89]. It is also open whether

[1] $\tilde{O}$ notation hides a polylogarithmic factor in terms of $|\mathcal{D}|$.

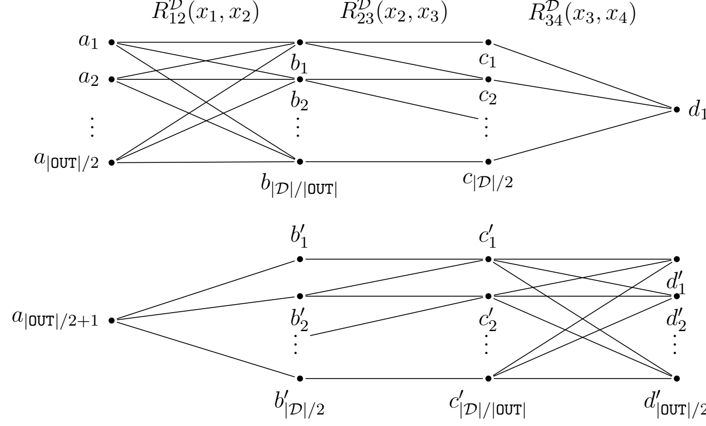


Figure 4.1: Database instance $\mathcal{D}$ showing relational instances for relations $R_{12}(x_1, x_2), R_{23}(x_2, x_3)$, and $R_{34}(x_3, x_4)$ for the three path query.

the upper bound obtained is optimal. Therefore, the question of whether Hu's upper bound can be improved and whether the improvement can be made using a combinatorial[2] algorithm remains open.

In this chapter, we show that it is possible to evaluate any acyclic CQ Q combinatorially in time $O(|\mathcal{D}| + |\mathtt{OUT}| + |\mathcal{D}| \cdot |\mathtt{OUT}|^{1-\epsilon})$ where $0 < \epsilon \leq 1$ is a query-dependent constant. An key class of queries that we consider is the path query $P_k(x_1, x_{k+1}) \leftarrow R_{12}(x_1, x_2) \wedge R_{23}(x_2, x_3) \wedge \cdots \wedge R_{k,k+1}(x_k, x_{k+1})$. To give the reader an overview of our key ideas, we show our techniques on the three path query $P_3(x_1, x_4)$.

Example 4.1. Consider the database instance as shown in Figure 4.1 for the $P_3(x_1, x_4)$ query. This instance was also used by Hu to show the limitations of Yannakakis algorithm. For the relation instance $R_{23}^{\mathcal{D}}$, each b_i has a degree $|\mathtt{OUT}|/2$ and is connected to $c_{1+(i-1)\cdot|\mathtt{OUT}|/2}, c_{2+(i-1)\cdot|\mathtt{OUT}|/2}, \ldots, c_{i\cdot|\mathtt{OUT}|/2}$. Similarly, each c'_i is connected to $b'_{1+(i-1)\cdot|\mathtt{OUT}|/2}, b'_{2+(i-1)\cdot|\mathtt{OUT}|/2}, \ldots, b'_{i\cdot|\mathtt{OUT}|/2}$ and has a degree $|\mathtt{OUT}|/2$. For any $1 \leq |\mathtt{OUT}| \leq |\mathcal{D}|$, Hu showed that any join order chosen by the Yannakakis algorithm for evaluating the query on the specific instance must incur a materialization cost of $\Omega(|\mathcal{D}| \cdot |\mathtt{OUT}|)$. However, their argument assumes that the entire relation is used to do the join. We demonstrate an algorithm for this query that bypasses the assumption. First, partition $R_{12}^{\mathcal{D}}(x_1, x_2)$ into $R_{12}^{\mathcal{D},H}(x_1, x_2)$ and $R_{12}^{\mathcal{D},L}(x_1, x_2)$ based on the degree threshold of x_2 values. In particular, for the instance under consideration, suppose we fix $\Delta = 2$. We define:

$$R_{12}^{\mathcal{D},L}(x_1, x_2) = \{t \in R_{12}^{\mathcal{D}} \mid |\sigma_{x_2 = t(x_2)} R_{12}^{\mathcal{D}}| \leq \Delta\}$$

[2]Although combinatorial algorithm does not have a formal definition, it is intuitively used to mean that the algorithm does not use any algebraic structure properties. A key property of combinatorial algorithms is that they are practically efficient [WW18]. Nearly all practical/production join algorithms known to date are combinatorial in nature.

Let $R_{12}^{\mathcal{D},H}(x_1,x_2) = R^{\mathcal{D}} \setminus R_{12}^{\mathcal{D},L}(x_1,x_2)$. Now, the original query can be answered by unioning the output of $P_3^H = \Pi_{x_1,x_4}(R_{12}^{\mathcal{D},H}(x_1,x_2) \wedge R_{23}^{\mathcal{D}}(x_2,x_3) \wedge R_{34}^{\mathcal{D}}(x_3,x_4))$ and $P_3^L = \Pi_{x_1,x_4}(R_{12}^{\mathcal{D},L}(x_1,x_2) \wedge R_{23}^{\mathcal{D}}(x_2,x_3) \wedge R_{34}^{\mathcal{D}}(x_3,x_4))$. Both of the join queries can be evaluated in $O(|\mathcal{D}|)$ time, regardless of the value of $|\texttt{OUT}|$. For both queries, we first apply a semijoin filter to remove all tuples from the input that do not participate in the join, a linear time operation. P_3^L is evaluated by first joining $R_{12}^{\mathcal{D},L}(x_1,x_2) \wedge R_{23}^{\mathcal{D}}(x_2,x_3)$ using any standard join algorithm, and projecting the output on variables x_1, x_3. The materialized intermediate result is then joined with $R_{34}^{\mathcal{D}}(x_3,x_4)$ and projected on x_1, x_4. P_3^H is evaluated in the opposite order by first joining $R_{23}^{\mathcal{D}}(x_2,x_3) \wedge R_{34}^{\mathcal{D}}(x_3,x_4)$ using any standard join algorithm, and projecting the output on variables x_2, x_4. The materialized intermediate result is then joined with $R_{12}^{\mathcal{D},H}(x_1,x_2)$, and finally projected on x_1, x_4.

Building upon the idea in Example 4.1, we show that by carefully partitioning the input and evaluating queries in a specific order, it is possible to improve the running complexity of join evaluation for a large class of CQs.

Our Contribution In this chapter, we introduce a combinatorial algorithm for evaluating any CQ Q. We introduce a recursive algorithm that generalizes the Yannakakis algorithm and to our knowledge, it has led to the first provable improvement of the Yannakakis algorithm after almost four decades! Our results are easily extensible to support aggregations and commutative semirings. To characterize the runtime of CQs, we also present a new simple width measure that we call the *projection width*, $\texttt{pw}(Q)$, that closely relates to existing width measures known for acyclic queries. As an example, our generalized algorithm, when applied to the path query $P_k(x_1, x_{k+1})$, runs in time $O(|\mathcal{D}| + |\texttt{OUT}| + |\mathcal{D}| \cdot |\texttt{OUT}|^{1-1/k})$. Rather surprisingly, this result—which is combinatorial and hence assumes $\omega = 3$—is already polynomially better than Hu's previous result for $k \leq 5$ even if we assume that $\omega = 2$ to obtain the strongest possible result in their setting. At the heart of our main algorithm is a technical result that allows tighter bounding of the cost of materializing intermediate results when executing Yannakakis algorithm. We show that for path queries, we can further refine the runtime to $O(|\mathcal{D}| + |\texttt{OUT}| + |\mathcal{D}| \cdot |\texttt{OUT}|^{1-1/\lceil (k+1)/2 \rceil})$ [DZFK24]; and for $k = 3$, this matches the lower bound of $\Omega(|\mathcal{D}| + |\texttt{OUT}| + |\mathcal{D}| \cdot \sqrt{|\texttt{OUT}|})$ that holds for evaluating any P_k for $k \geq 2$.

Further, we establish tightness of our results by demonstrating a matching lower bound on the running time for a restricted subclass of acyclic and cyclic CQs. Our lower bounds are applicable to both join processing (aka Boolean semiring) and FAQs.

4.1 Preliminaries and Notation

Conjunctive Queries We use standard CQ definitions defined in Chapter 2, through its associated hypergraph $\mathcal{H}$. For this chapter, we will identify the relational instance[3] for relation R_K in database $\mathcal{D}$ using $R_K^{\mathcal{D}}$. We will say that a variable $x \in V(\mathcal{H})$ that is present in at least two relations is a *join variable*, while a variable that is present in exactly one relation is *isolated*.

Tree Decomposition We follow standard definitions of tree decompositions $(\mathcal{T}, \chi)$ as defined in Chapter 2. The sets $e = \chi(t)$ are called the *bags* of the tree decomposition.

More specifically in this chapter, we use $\chi^{-1}(e)$ to recover the node of the tree with bag e. A query is said to be *α-acyclic*[4] if there exists a tree decomposition such that every bag $\chi(t)$ of the tree decomposition corresponds uniquely to an input atom $R_{\chi(t)}$. Such a tree decomposition is the *join tree* of the query. It is known that an α-acyclic query can be evaluated in time $O(|\mathcal{D}| + |\mathtt{OUT}|)$. For simplicity, we will sometimes refer to the node of a join tree through the relation assigned to the node (or the relational instance) since there is a one-to-one mapping. A *rooted tree decomposition* is a tree decomposition that is rooted at some node $r \in V(\mathcal{T})$, denoted by $(\mathcal{T}, \chi, r)$. Different choices of r change the orientation of the tree. We will use $\mathcal{L}(\mathcal{T}) \subseteq V(\mathcal{T})$ to denote the set of leaf nodes of a rooted tree decomposition and $\mathcal{I}(\mathcal{T}) = V(\mathcal{T}) \setminus \mathcal{L}(\mathcal{T})$ as the set of internal nodes (i.e. all non-leaf nodes) of the rooted tree decomposition. For any node $t \in V(\mathcal{T})$ in a rooted tree decomposition, we use F_t to denote the set of free variables that appear in the subtree rooted at t (including free variables in $\chi(t)$). The subtree rooted at t is denoted via $\mathcal{T}_t$.

Recall that a CQ Q with free variables F is called *free-connex acyclic* it is α-acyclic and the hypergraph $([n], \mathcal{E} \cup \{F\})$ is also α-acyclic.

Tuples and Operators A tuple v over a set of variables $\mathbf{x}_K$ is a total function that maps each variable $x \in \mathbf{x}$ to a value in dom. Given a tuple v defined over $\mathbf{x}$, and a set of variables $\mathcal{S} \subseteq \mathbf{x}$, $t(\mathcal{S})$ is the restriction of t onto $\mathcal{S}$. For a relation R_K over variables $\mathbf{x}_K$, $\mathcal{S} \subseteq \mathbf{x}_K$, and a tuple $s = v(\mathcal{S})$, we define $\sigma_{\mathcal{S}=s}(R_K) = \{t \mid t \in R_K^{\mathcal{D}} \wedge t(\mathcal{S}) = s\}$ as the set of tuples in $R_K^{\mathcal{D}}$ that agree with s over variables in $\mathcal{S}$, and $\Pi_{\mathcal{S}}(R_K) = \{t(\mathcal{S}) \mid t \in R_K^{\mathcal{D}}\}$ as the set of restriction of the tuples in $R_K^{\mathcal{D}}$ to the variables in $\mathcal{S}$. The output or result of evaluating a CQ Q over $\mathcal{D}$ (denoted $Q(\mathcal{D})$) can be defined as $\{\Pi_{\mathbf{x}_F}(v(\mathbf{x}_{[n]})) \mid v(\mathbf{x}_K) \in R_K^{\mathcal{D}}, \forall K \in \mathcal{E}\}$. We denote by $\mathtt{OUT}$ the result of running Q over database $\mathcal{D}$ (i.e., $\mathtt{OUT} = Q(D)$) and we use $|\mathtt{OUT}|$ to denote the number of tuples in $Q(D)$.

For relation R_K over variables $\mathbf{x}_K$, a threshold Δ, and a set $\mathcal{S} \subset \mathbf{x}_K$, we say that a tuple $v(\mathcal{S})$ is *heavy* if $|\sigma_{\mathcal{S}=v(\mathcal{S})}(R_K^{\mathcal{D}})| > \Delta$, and *light otherwise*. We will use $d(v, \mathcal{S}, R_K^{\mathcal{D}})$ to denote $|\sigma_{\mathcal{S}=v(\mathcal{S})}(R_K^{\mathcal{D}})|$.

[3]We will frequently refer to the relational instance $R_K^{\mathcal{D}}$ as just relation for the sake of brevity.

[4]Throughout this chapter, we will use acyclic to mean α-acyclic.

The *semijoin* of two relations $R^{\mathcal{D}}_{K_1}$ and $R^{\mathcal{D}}_{K_2}$ (denoted as $R^{\mathcal{D}}_{K_1} \ltimes R^{\mathcal{D}}_{K_2}$) is defined as $\Pi_{K_1}(R^{\mathcal{D}}_{K_1} \wedge R^{\mathcal{D}}_{K_2})$. A *full reducer* [BG81] of a database $\mathcal{D}$ is a finite sequence (that only depends on the schema of the relations in $\mathcal{D}$) of semijoin operations that filters out tuples from the relations in the database $R^{\mathcal{D}}_K$ such that $R^{\mathcal{D}}_K = \Pi_K(\bigwedge_{K \in \mathcal{E}} R^{\mathcal{D}}_K(\mathbf{x}_K))$, i.e., all remaining tuples in the input relations participate in the result of the full join query $\bigwedge_{K \in \mathcal{E}} R^{\mathcal{D}}_K(\mathbf{x}_K)$.

Model of Computation We use the standard RAM model with uniform cost measure. For an instance of size N, every register has length $O(\log N)$. Any arithmetic operation (such as addition, subtraction, multiplication and division) on the values of two registers can be done in $O(1)$ time. A sorting the values of N registers can be done in $O(N \log N)$ time.

4.2 Projection Width

Consider a CQ Q with hypergraph $\mathcal{H}$ and free variables $\mathbf{x}_F$. We first define the *reduced query* of Q, denoted $\mathsf{red}[Q]$ (in [Hu24a], this is called a cleansed query) via Algorithm 2. The while-loop of the procedure essentially follows the GYO algorithm [Mar79, YO79], with the difference that we can remove isolated variables only if they are not free. The reduced query is well-defined because the resulting hypergraph after the while-loop terminates is the same independent of the sequence of vertex and hyperedge removals. If $F = V(\mathcal{H})$, the procedure does not remove any variables (but may potentially remove hyperedges). If $F = \emptyset$, the while-loop is identical to the GYO algorithm and will return the hypergraph $(\emptyset, \{\{\}\})$. We say that Q is *reduced* if $Q = \mathsf{red}[Q]$, i.e., the query cannot be further reduced.

Algorithm 2: Reduced CQ

Input : acyclic Q with hypergraph $\mathcal{H}$ and free variables $\mathbf{x}_F$

Output: $\mathsf{red}[Q]$

```
1 H' ← multihypergraph of H                      /* edges E' in H' are multisets  */
2 while H' has changed do
3     if ∃ isolated variable x ∉ x_F then
4         foreach e ∈ E(H') do
5             e ← e \ {x}                          /* remove x from all hyperedges */
6         V(H') ← V(H) \ {x}                       /* remove x from the vertex set */
7     if ∃ e, f ∈ E(H') such that e ⊆ f then
8         E(H') ← E(H') \ {e}
9 return H', F
```

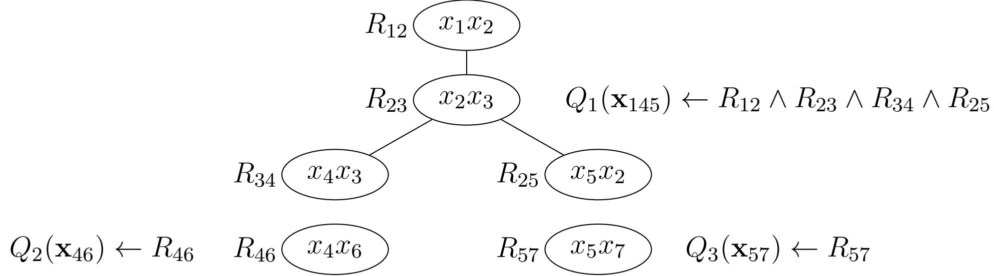


Figure 4.2: Depiction of the graph $G_Q^{\exists}$ and the decomposition of the running example query $Q(\mathbf{x}_{14567}) \leftarrow R_{12}(\mathbf{x}_{12}) \wedge R_{23}(\mathbf{x}_{23}) \wedge R_{34}(\mathbf{x}_{34}) \wedge R_{25}(\mathbf{x}_{25}) \wedge R_{46}(\mathbf{x}_{46}) \wedge R_{57}(\mathbf{x}_{57})$

The following three properties of reduced queries will be essential in this section.

Proposition 4.1. *An acyclic* CQ *Q is free-connex acyclic if and only if all the variables of* red*$[Q]$ are free.*

Proof. Consider an acyclic CQ Q such that all its variables in red$[Q]$ are free. Then, the hypergraph formed by adding a hyperedge containing all free variables is still acyclic, and thus satisfies the definition of free-connex acyclic. Indeed, there exists a join tree for the new hypergraph where the new hyperedge with all free variables is the root and all remaining hyperedges are its children.

For the other direction, consider an acyclic CQ Q such that there exists a non-free variable (say z) in red$[Q]$. Clearly, such a variable is not isolated. Let e_1 and e_2 be two hyperedges that contain z. Let E^* be the hyperedge added to the hypergraph containing all the variables. We claim that the new hypergraph containing edge E^* is no longer acyclic. Indeed, suppose that E^* is the root of the join tree (if there exists a join tree, we can reorient it to make any node as the root). Note that e_1 and e_2 cannot be subsets of each other and thus, one cannot be in the subtree of the other in the join tree. This is because if (say) e_1 contains a free variable (say f) that is not contained in e_2, then having e_1 in the subtree of e_2 will violate the variable connectedness condition for f, as f is present in E^* and e_1 but not in e_2. Thus, e_1 and e_2 must be either be directly connected to E^* since both e_1 and e_2 may contain isolated free variables (all of which are also present in E^*), or they may be connected to the root through a series of intermediate nodes. However, the variable z is not present in E^* as E^* only contains free variables and thus, the connectedness condition cannot be satisfied for variable z. Thus, a join tree cannot exist and the query cannot be free-connex acyclic. □

Proposition 4.2. *Let Q be a reduced acyclic* CQ*, and $\mathcal{T}$ be a join tree of Q. Then, every leaf node of $\mathcal{T}$ has an isolated free variable.*

Proof. Suppose there exists a leaf node t that violates the desired property. First, we establish that $\chi(t)$ contains free variable(s). For the sake of contradiction, let us assume that $\chi(t)$ contains no

free variable. Since all isolated variables have already been removed, it must be the case that all remaining variables in $\chi(t)$ are also present in its parent and thus are join variables. However, such hyperedges are removed by Line 8 of Algorithm 2. Therefore, $\chi(t)$ must contain free variable(s).

Next, suppose that the free variable is not isolated. Then, it must be the case that the free variable is also present in the parent (otherwise it would be isolated). By the same argument as before, since all other variables in $\chi(t)$ are also present in the parent, we get a contradiction since such hyperedges are removed by Algorithm 2. This concludes the proof. $\square$

Proposition 4.3. *Let Q be a reduced acyclic* CQ*, and $\mathcal{T}$ be a join tree of Q. Then, no variable in any node of $\mathcal{T}$ can be isolated and non-free.*

Proposition 4.3 follows directly from the operation performed on Line 5-6 in Algorithm 2.

Second, we define the *decomposition* of a CQ Q following [Hu24a]. Define the graph $G_Q^{\exists}$, where each hyperedge is a vertex, and there is an edge between e, e' if they share a non-free variable. Let $E_1, \dots, E_k$ be the connected components of $G_Q^{\exists}$. Then, the decomposition of Q, denoted $\mathsf{decomp}(Q)$, is a set of queries $\{Q_1, \dots, Q_k\}$, where Q_i is the CQ with hypergraph $(\bigcup_{e \in E_i} e, E_i)$ and free variables $F \cap \bigcup_{e \in E_i} e$. If the decomposition of Q has exactly one query, we say that Q is *existentially connected.*

Definition 4.1 (Projection Width)**.** Consider an acyclic CQ Q. Then, $\mathtt{pw}(Q)$ is the maximum number of relations across all queries in $\mathsf{decomp}(\mathsf{red}[Q])$.

Example 4.2. Consider the query

$$Q^{\star}_{\ell}(x_1, \dots, x_\ell) \leftarrow R_1(x_1, y) \wedge \cdots \wedge R_\ell(x_\ell, y).$$

One can observe that $\mathsf{red}[Q^{\star}_{\ell}] = Q^{\star}_{\ell}$. The decomposition of the reduced query has only one connected component with ℓ atoms, hence $\mathtt{pw}(Q^{\star}_{\ell}) = \ell$.

Example 4.3. Consider the query $Q(\mathbf{x}_{14567}) \leftarrow R_{12}(\mathbf{x}_{12}) \wedge R_{23}(\mathbf{x}_{23}) \wedge R_{34}(\mathbf{x}_{34}) \wedge R_{25}(\mathbf{x}_{25}) \wedge R_{46}(\mathbf{x}_{46}) \wedge R_{57}(\mathbf{x}_{57})$. The query is already reduced since there are no isolated variables or any hyperedges that are contained in another. To get the projection width of the query, Figure 4.2 shows the graph $G_Q^{\exists}$ where each atom of the query becomes a vertex. Note that there is no edge between R_{34} and R_{46} because x_4 is a free variable. This graph has three connected components, and the largest connected component contains four hyperedges, hence $\mathtt{pw}(Q(\mathbf{x}_{14567})) = 4$. Figure 4.2 also shows the three queries corresponding to each connected component of $G_Q^{\exists}$.

We observe that $1 \leq \mathtt{pw}(Q) \leq |E(\mathcal{H})|$, and also that $\mathtt{pw}(Q)$ is always an integer. When Q is full, every hyperedge of the reduced hypergraph forms its own connected component of size one; in this case, $\mathtt{pw}(Q) = 1$. More generally:

Proposition 4.4. *An acyclic* CQ Q *is free-connex acyclic if and only if* $\mathtt{pw}(Q) = 1$*.*

Proof. Suppose that Q is free-connex. From Proposition 4.1, $\mathsf{red}[Q]$ has only free variables. Hence, in the decomposition every hyperedge forms a connected component of size one. Thus, $\mathtt{pw}(Q) = 1$.

For the other direction, suppose that $\mathtt{pw}(Q) = 1$. Let us examine the reduced query $Q' = \mathsf{red}[Q]$. Then, every query in the decomposition of Q' has size one. However, this implies in turn that all variables of Q' are free; indeed, if there exists a non-free variable x, the variable x would be isolated and that would violate the fact that Q' is reduced. From Proposition 4.1, we now have that Q is free-connex acyclic. □

Hu [Hu24a] defined a notion similar to $\mathtt{pw}(Q)$, called free-width, $\mathsf{freew}(Q)$. To compute free-width, we first define the free-width of an existentially connected query to be the size of the smallest set of hyperedges that covers all the isolated variables. Then, $\mathsf{freew}(Q)$ is the maximum freewidth over all queries in the decomposition of Q. It is easy to see that $\mathsf{freew}(Q) \leq \mathtt{pw}(Q)$.

4.3 Main Result

In this section, we describe our main result that builds upon the celebrated Yannakakis algorithm. First, we recall the algorithm and its properties as outlined in Algorithm 1 [Yan81]. The first step of the algorithm is to apply the full reducer which removes all tuples from the input database relations that do not contribute to the output. The main idea of the algorithm is to use a bottom-up evaluation strategy over the join tree. In particular, a node s is processed once each of its children have been processed. Consider node s whose children are all leaves. The key step of the algorithm is to do the join of relational instances corresponding to s and each of its children but projecting the output on only the free variables in the subtree and the variables on node s. The bottom-up process continues until we reach root and final join result is returned.

Yannakakis showed two key properties of Algorithm 1. First, when the processing of a node s is over (i.e. when the algorithm has finished execution of Line 8 for node s), it holds that

$$T^{\mathcal{D}}_{\chi(s)} = \Pi_{F_s \cup \chi(s)}(\wedge_{t \in V(\mathcal{T}_s)} R^{\mathcal{D}}_{\chi(t)})$$

Therefore, when only node r is left in the tree, then $T^{\mathcal{D}}_{\chi(r)} = \Pi_{F_r \cup \chi(r)}(\wedge_{t \in V(\mathcal{T})} R^{\mathcal{D}}_{\chi(t)})$, and thus, $\Pi_F(T^{\mathcal{D}}_{\chi(r)})$ gives the desired result. Second, Yannakakis showed that the entire algorithm takes time $O(|\mathcal{D}| + |\mathcal{D}| \cdot |\mathtt{OUT}|)$, which is the time taken to execute Line 8 in each iteration.

4.3.1 Output-sensitive Yannakakis

This section presents a general lemma that forms the basis of our main result. In particular, we will show that under certain restrictions of the instance and the root of the join tree, Algorithm 1

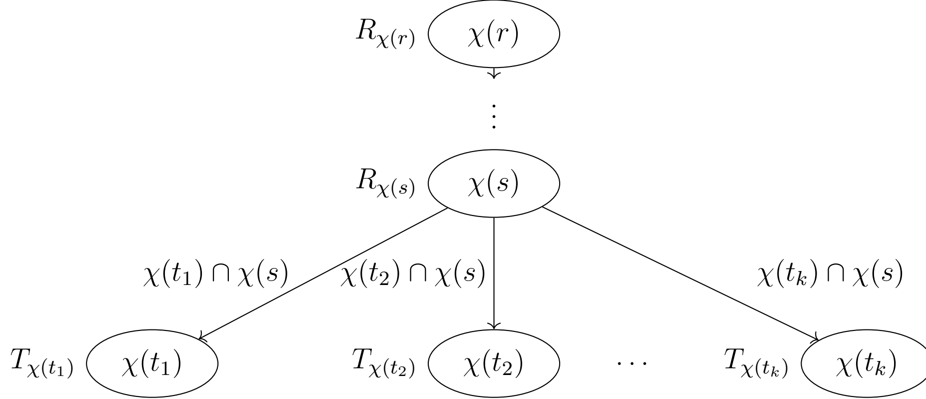


Figure 4.3: A join tree with root node r. Edge labels show the common variables between bag $\chi(s)$ and bag $\chi(t_i)$.

(Yannakakis algorithm) achieves a better runtime by a factor of Δ. In the next section, we will present an algorithm that takes advantage of this observation.

The first condition is that the root node must contain an isolated free variable. This is not necessarily true for any node in the join tree, but we can make it happen by choosing the root to be a leaf (recall that in a reduced instance, every leaf node has an isolated free variable). The second condition is that all tuples in the root node over the join variables are heavy w.r.t. a degree threshold Δ. This is not generally true (unless $\Delta = 1$), so we will need to partition the instance to achieve this requirement. For a join tree $(\mathcal{T}, \chi)$, we define $\chi^{\bowtie}(s) \subseteq \chi(s)$ to return only the join variables of $\chi(s)$

Lemma 4.1. *Let $Q(\mathbf{x}_F)$ be a reduced acyclic CQ, $(\mathcal{T}, \chi, r)$ a rooted join tree of Q, and $\mathcal{D}$ be a database instance. Suppose that*

1. *$R_{\chi(r)}$ has at least one isolated free variable*
2. *for every $v \in R^{\mathcal{D}}_{\chi(r)}$, we have $d(v, \chi^{\bowtie}(r), R^{\mathcal{D}}_{\chi(r)}) > \Delta$ for some integer $\Delta \geq 1$.*

Then, Algorithm 1 runs in time $O(|\mathcal{D}| + (\sum_{t \in \mathcal{I}(\mathcal{T})} |R^{\mathcal{D}}_{\chi(t)}|) \cdot |\mathtt{OUT}|/\Delta)$.

Proof. To prove this result, we will show that the size of the intermediate result that is materialized in Line 8 for an internal node s is bounded by $|\mathtt{OUT}| \cdot (|R^{\mathcal{D}}_{\chi(s)}|/\Delta)$.

Consider the point in the algorithm where we join the node s with its children nodes $t_1, \dots, t_k$, as shown in Figure 4.3. The relational instances assigned to the nodes $t_1, \dots, t_k$ may not necessarily correspond to the base relations $R^{\mathcal{D}}_{\chi(t_i)}$ since a previous iteration of the while loop may have performed a join that led to creation of the intermediate relation $T^{\mathcal{D}}_{\chi(t_i)}$. Let v be a tuple over the variables $\mathbf{x}_{[n]}$ such that $v(\mathbf{x}_F) \in Q(\mathcal{D})$ and for each relation $R^{\mathcal{D}}_K(\mathbf{x}_K)$, it holds that $v(\mathbf{x}_K) \in R^{\mathcal{D}}_K$. We first claim the following inequality:

$$\prod_{i \in [k]} d(v, \chi(t_i) \cap \chi(s), T^{\mathcal{D}}_{\chi(t_i)}) \leq |\mathtt{OUT}|/\Delta \tag{4.1}$$

To show this, let Z be the set of all variables in the nodes $r, t_1, \dots, t_k$ except for the isolated free variables. Consider the join query $Q'(\mathbf{x}_F)$ where the values for all variables except Z have been fixed according to v (i.e. the tuples in the relations are filtered as shown below).

$$Q'(\mathcal{D}) = \Pi_F((R^{\mathcal{D}}_{\chi(r)} \ltimes v(\chi^{\bowtie}(r))) \wedge_{i \in [k]} (T^{\mathcal{D}}_{\chi(t_i)} \ltimes v(\chi^{\bowtie}(t_i))) \wedge_{u \in V(\mathcal{T}) \setminus \{r, t_1, \dots, t_k\}} (R^{\mathcal{D}}_{\chi(u)} \ltimes v)) \tag{4.2}$$

From the definition of v, it holds that $Q'(\mathcal{D}) \subseteq Q(\mathcal{D})$. Next, we claim that $|Q'(\mathcal{D})|$ is exactly:

$$A_v := d(v, \chi^{\bowtie}(r), R^{\mathcal{D}}_{\chi(r)}) \cdot \prod_{i \in [k]} d(v, \chi(t_i) \cap \chi(s), T^{\mathcal{D}}_{\chi(t_i)})$$

To see why this holds, first, observe that the semijoin of v with all relations except for the root node and nodes $t_1, \dots, t_k$ (i.e. all the relations considered by the last conjunct of Equation 4.2) fixes their size to one. This is because of our choice of v that guarantees that $v(\mathbf{x}_F) \in Q(\mathcal{D})$, and thus implies that the tuple formed by restricting v onto the schema of each relation is present in the corresponding relational instance. Intuitively, it means that there exists a *join path* from the root node to the leaf nodes $t_1, \dots, t_k$ since for each intermediate node, there is a tuple formed by restricting v present in the corresponding input relation.

Next, note that $R_{\chi(r)}$ has at least one isolated free variable, and tuple $v(Z)$ fixes values for all join variables in the relation. Similarly, tuple $v(Z)$ also fixes values for all join variables in $R_{\chi(t_i)}$, which are $\chi(t_i) \cap \chi(s)$. Since the query is reduced, Proposition 4.2 guarantees that every leaf node has an isolated free variable. Further, Proposition 4.3 guarantees that for all bags, every variable is either an isolated free variable or a join variable. Therefore, once the join variables of nodes $r, t_1, \dots, t_k$ have been fixed, it is guaranteed that all remaining variables in those nodes are output variables. Since we have already established that for all intermediate relations from root to the leaf nodes t_i, there exists tuples that join with $v(Z)$, it holds that $|Q'(\mathcal{D})| = |R^{\mathcal{D}}_{\chi(r)} \ltimes v(\chi^{\bowtie}(r))| \cdot \Pi_{i \in [k]} |T^{\mathcal{D}}_{\chi(t_i)} \ltimes v(\chi^{\bowtie}(t_i))| = A_v$.

To finish the claim, note that $A_v = |Q'(\mathcal{D})| \leq |Q(\mathcal{D})| = |\mathtt{OUT}|$ and also from assumption (2) of the lemma, we have that $d(v, \chi^{\bowtie}(r), R^{\mathcal{D}}_{\chi(r)}) > \Delta$.

To complete the proof, let $W = \chi(s) \cap (\bigcup_{i \in [k]} \chi(t_i))$ denote the join variables of node s that are also present in some leaf node $t_1, \dots, t_k$. We observe that the size of the intermediate bag

$\Pi_{\chi(s)\cup F_s}(R^{\mathcal{D}}_{\chi(s)} \wedge T^{\mathcal{D}}_{\chi(t_1)} \wedge \cdots \wedge T^{\mathcal{D}}_{\chi(t_k)})$ can be bounded by

$$\begin{aligned}
&\sum_{w \in \Pi_W(R^{\mathcal{D}}_{\chi(s)})} |R^{\mathcal{D}}_{\chi(s)} \ltimes w| \cdot \prod_{i \in [k]} |T^{\mathcal{D}}_{\chi(t_i)} \ltimes w| \\
&= \sum_{w \in \Pi_W(R^{\mathcal{D}}_{\chi(s)})} d(w, W, R^{\mathcal{D}}_{\chi(s)}) \cdot \prod_{i \in [k]} d(w, \chi(t_i) \cap \chi(s), T^{\mathcal{D}}_{\chi(t_i)}) \\
&\leq \frac{|\mathtt{OUT}|}{\Delta} \cdot \sum_{w \in \Pi_W(R^{\mathcal{D}}_{\chi(s)})} d(w, W, R^{\mathcal{D}}_{\chi(s)}) && \text{(using Equation 4.1)} \\
&\leq \frac{|\mathtt{OUT}|}{\Delta} \cdot |R^{\mathcal{D}}_{\chi(s)}| && \text{(Sum of all degrees equates the relation size)}
\end{aligned}$$

Here, the first line bounds the total join size as the sum of sizes of the cartesian product of the relations after semijoin for each fixing of $w \in \Pi_W(R^{\mathcal{D}}_{\chi(s)})$. The first inequality holds since the degree product bound holds for all tuples w. Indeed, since the query is acyclic, once a full reducer has been applied, for each w, there exists a tuple v over $\mathbf{x}_{[n]}$ such that $w = v(W)$, $v(\mathbf{x}_F) \in Q(\mathcal{D})$, and $v(\mathbf{x}_K) \in R^{\mathcal{D}}_K$ for each relation in $\mathcal{D}$. Observe that the time bound requires only using the size of the relation assigned to the parent of the leaf nodes (and not the sizes of the relations assigned to the leaf nodes itself). □

Discussion. Observe that Lemma 4.1 degenerates to the standard bound of Yannakakis algorithm for $\Delta = 1$. However, as we will show in the next part, choosing $\Delta > 1$ can lead to better overall join processing algorithms. It is also interesting to note that the running time bound obtained in Lemma 4.1 can only be achieved when the root of the decomposition is a relation with the two properties as outlined in the statement. It can be shown that no other choice of the root node achieves a time better than $O(|\mathcal{D}| \cdot |\mathtt{OUT}|)$. We note that Lemma 4.1 requires the query to be reduced. Indeed, without the reduced query requirement, relations may contain variables that are neither free and nor join. The presence of such variables renders the join size computation incorrect.

4.3.2 Our Algorithm

In this section, we will present the algorithm for our main result. We will first consider a CQ Q that is existentially connected and reduced.

Algorithm 3 shows the improved procedure. It processes the nodes of the join tree in a leaf-to-root order just like Yannakakis algorithm, i.e., a node s is processed only after each of its children have been processed. However, our algorithm departs in the operations involved in the processing of each node. In particular, consider node s whose children are all leaves. For every leaf node t that is a child of s, we partition the relation $T^{\mathcal{D}}_{\chi}(t)$ assigned to the node into two disjoint partitions (the heavy partition $T^{\mathcal{D},H}_{\chi}(t)$ and the light partition $T^{\mathcal{D},L}_{\chi}(t)$) using a chosen degree threshold. Then, we use

Lemma 4.1 to process $T_{\chi}^{\mathcal{D},H}(t)$ and add the produced output into a set $\mathcal{J}$. This processing is done in lines 15-17 of the while loop. The crucial detail is that the processing of the heavy partition is done by reorienting the join tree to be rooted at t, and then applying Yannakakis in a bottom-up fashion.

Once all heavy sub-relations of the leaf nodes are processed, we are left with all light parts of the relations for the children of node s. At this point, we join the relations in the subtree rooted at s (Line 22) and then remove the nodes for children of s from the join tree $\mathcal{T}$ (Line 24). The step on Line 22 is identical to the one on Line 8 of the Yannakakis algorithm. Note that modifying the structure of the join tree $\mathcal{T}$ by deleting nodes in our algorthm is a departure from Yannakakis algorithm. Although modification of $\mathcal{T}$ is not required for correctness of our algorithm, as we will show later, by controlling the node processing order $\mathcal{K}$ (on Line 5), one can use our algorithm in innovative ways. Therefore, when $\mathcal{K}$ is a subset of $V(\mathcal{T})$ instead of containing all the nodes, it is important to keep the database $\mathcal{D}$ and $\mathcal{T}$ up-to-date to reflect which nodes have been processed. We next state the main result (the proof can be found in the appendix).

Lemma 4.2. *Given an acyclic join query $Q(\mathbf{x}_F)$ that is existentially connected and reduced, database $\mathcal{D}$, and an integer threshold $1 \leq \Delta \leq |\mathcal{D}|$, Algorithm 3 computes the join result $Q(\mathcal{D})$ in time $O(|\mathcal{D}| \cdot \Delta^{k-1} + |\mathcal{D}| \cdot |\mathtt{OUT}|/\Delta)$, where k is the number of atoms in Q.*

Proof. We begin with a simple observation: the only step in the algorithm that modifies the join tree is the join operation on Line 22. In any given iteration of the while loop, once the join on Line 22 is computed, node s becomes a leaf node in the join tree since we delete all its children right after. Then, we get the following.

Observation 1. *After every iteration of the while loop, it holds that all relations corresponding to the internal nodes of the join tree have size $O(|\mathcal{D}|)$.*

Let us now bound the time required to process all leaf nodes. Fix a leaf node s. Except the operation on Line 15, all other operations take at most $O(|\mathcal{D}|)$ time (note that $\mathcal{D}$ is also modified in every iteration of the inner loop). We claim that Line 15 takes time $O(|\mathcal{D}| \cdot |\mathtt{OUT}|/\Delta)$. By our choice of root node (which is s) for calling Lemma 4.1, $\mathcal{T}$ is rooted at s. Further, note that the leaves of $(\mathcal{T}, s)$ are the same leaves as $(\mathcal{T}, r)$ (except for s, which is now the root). Further, by Proposition 4.2, it is also guaranteed that the bag of s contains isolated free variable(s). Thus, the conditions of applying Lemma 4.1 are satisfied. Since our choice of threshold for the partition is Δ_s, Lemma 4.1 tells us that the evaluation time required is at most big-O of

$$\left(\sum_{t \in \mathcal{I}(\mathcal{T})} |T_{\chi(t)}^{\mathcal{D}}|\right) \cdot |\mathtt{OUT}|/\Delta_s = (|T_{\chi}^{\mathcal{D}}(s)| + \mathcal{D}) \cdot |\mathtt{OUT}|/\Delta_s = |\mathcal{D}| \cdot |\mathtt{OUT}|/\Delta$$

Algorithm 3: Generalized Yannakakis Algorithm

Input : reduced and existentially connected acyclic query $Q(\mathbf{x}_F)$, instance $\mathcal{D}$, rooted join tree $(\mathcal{T}, \chi)$, $1 \leq \Delta \leq |\mathcal{D}|$

Output : $Q(\mathcal{D}), (\mathcal{T}, \chi), \mathcal{D}$

1 **choose** an arbitrary root r for $\mathcal{T}$

2 $\mathcal{D} := (R^{\mathcal{D}}_{\chi(s)})_{s \in V(\mathcal{T})} \leftarrow$ **apply** a full reducer for $\mathcal{D}$

3 $\mathcal{T}^{IN} \leftarrow$ clone of $\mathcal{T}$ `/* clone of the join tree since we will edit` $\mathcal{T}$ `in-place */`

4 $N \leftarrow |\mathcal{D}|$ `/* storing the size of the input */`

5 $\mathcal{K} \leftarrow$ a queue of $s \in V(\mathcal{T})$ following a post-order traversal of $\mathcal{T}$

6 **while** $\mathcal{K} \neq \emptyset$ **do**

7     $s \leftarrow \mathcal{K}.pop()$

8   **if** *s is a leaf in $\mathcal{T}$* **then**

9   **if** *s is not the root of $\mathcal{T}$* **then**

10   **if** *s is a leaf in $\mathcal{T}^{IN}$* **then**

                `/* identical to Line 6 in Alg 1 and initializes the` $T^{\mathcal{D}}_{\chi(s)}$ `*/`

11                 $T^{\mathcal{D}}_{\chi(s)} = \Pi_{\chi(s)}(R^{\mathcal{D}}_{\chi(s)}(\mathbf{x}_{\chi(s)}))$

12             $\Delta_s \leftarrow \Delta \cdot (|T^{\mathcal{D}}_{\chi(s)}| + N)/N$

13             $T^{\mathcal{D},H}_{\chi(s)} = \{\mathbf{v} \in T^{\mathcal{D}}_{\chi(s)} \mid |\sigma_{\mathbf{v}(\chi^{\bowtie}(s))}(T^{\mathcal{D}}_{\chi(s)})| > \Delta_s\}$, $T^{\mathcal{D},L}_{\chi(s)} = T^{\mathcal{D}}_{\chi(s)} \setminus T^{\mathcal{D},H}_{\chi(s)}$

14             $\mathcal{D}^H_s \leftarrow (\mathcal{D} \setminus T^{\mathcal{D}}_{\chi(s)}) \cup T^{\mathcal{D},H}_{\chi(s)}$

15   **let** $Q_s(\mathcal{D}^H_s)$ be the output of Algorithm 1 with input as $\mathcal{D}^H_s$ and $\mathcal{T}$ rooted at s

16             $\mathcal{J} \leftarrow \mathcal{J} \cup Q_s(\mathcal{D}^H_s)$

17             $\mathcal{D} \leftarrow$ **apply** a full reducer for $(\mathcal{D} \setminus T^{\mathcal{D}}_{\chi(s)}) \cup T^{\mathcal{D},L}_{\chi(s)}$

18             $T^{\mathcal{D}}_{\chi(s)} = \Pi_{\chi(s)}(T^{\mathcal{D},L}_{\chi(s)})$

19   **else**

20             $\mathcal{J} \leftarrow \mathcal{J} \cup \Pi_F(T^{\mathcal{D}}_{\chi(r)})$ `/*` s `is the only node in the tree */`

21   **else**

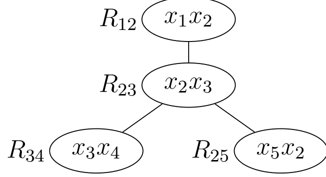
22         $T^{\mathcal{D}}_{\chi(s) \cup F_s} = \Pi_{\chi(s) \cup F_s}\left(T^{\mathcal{D}}_{\chi(s)} \wedge \left(\bigwedge_{(s,t) \in E(\mathcal{T})} \Pi_{F_t \cup (\chi(s) \cap \chi(t))} T^{\mathcal{D}}_{\chi(t)}\right)\right)$

23         $\chi(s) \leftarrow \chi(s) \cup F_s$

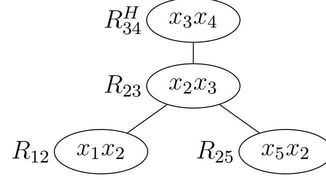
24   **truncate** all children of s and directed edges $(s, t) \in E(\mathcal{T})$ from $\mathcal{T}$

25         $\mathcal{K}$. *push_to_head*(s) `/*` s `became a leaf, process` s `immediately */`

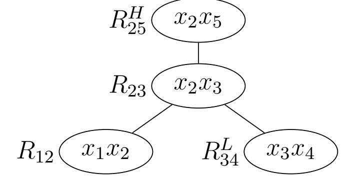
26 **return** $\mathcal{J}, (\mathcal{T}, \chi), \mathcal{D}$ `/* Return output, tree decomposition, and instance */`



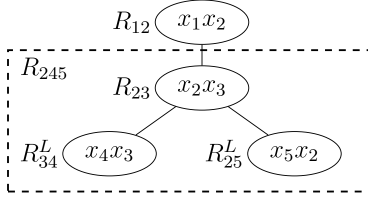
(a) The join tree for Q_1



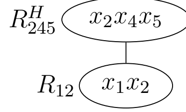
(b) Applying Lemma 4.1 with R^H_{34} as root node.



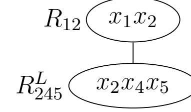
(c) Applying Lemma 4.1 with R^H_{25} as root node.



(d) Joining the relations in subtree rooted at R_{23}



(e) Applying Lemma 4.1 with R^H_{245} as root node.



(f) Joining the remaining relations.

Figure 4.4: The running example query running Algorithm 3. Each figure shows a rooted join tree.

Here, the first equality holds because of Observation 1 which guarantees that only s can have $\Omega(N)$ size. The reader can verify that $\Delta \leq \Delta_s$, and thus, is well-defined.

Next, we will bound the time required for the join operation on Line 22. For any node s, let $\#s$ denote the number of nodes in the subtree rooted at s (including s). We will show by induction that time required for the join is $O(|\mathcal{D}| \cdot \Delta^{\#s-1})$.

Base Case. In the base case, consider the leaf nodes with relations of size $O(|\mathcal{D}|)$. Consider such a leaf ℓ and note that the relation size satisfies $O(|\mathcal{D}| \cdot \Delta^{\#\ell-1}) = O(|\mathcal{D}|)$ since the number of nodes a subtree rooted at a leaf is one (the leaf itself).

Inductive Case. Now, consider a node s that is not a leaf. The size of the relation for its child node t is $O(|\mathcal{D}| \cdot \tau^{\#t-1})$. For t, the degree threshold for partitioning the relation is $\Delta_t = \Delta \cdot (|\mathcal{D}| \cdot \Delta^{\#t-1} + |\mathcal{D}|)/|\mathcal{D}| \leq 2\Delta^{\#t}$. Therefore, by making the same argument as in the proof of Lemma 4.1, we get the degree product as $\Pi_{t \in \text{child of } s} \Delta_t = O(\Delta^{\sum_{t \in \text{child of } s} \#t}) = \Delta^{\#s-1}$. Thus, the join time and the size of $R_{\chi(s)}$ is $O(|\mathcal{D}| \cdot \Delta^{\#s-1})$. Since the tree contains $|\mathcal{T}|$ number of nodes, the total time required is dominated by the last iteration, giving us $O(|\mathcal{D}| \cdot \Delta^{|\mathcal{T}|-1})$.

In each iteration of the while loop, one node of the tree is processed. Thus, the total number of iterations is $|\mathcal{T}| = k$ and the total running time of the algorithm is $O(|\mathcal{D}| \cdot \Delta^{k-1} + |\mathcal{D}| \cdot |\texttt{OUT}|/\Delta)$. $\square$

Example 4.4. We use the query Q_1 shown in Figure 4.4a as an example to show the execution of Algorithm 3. The join tree will be visited in the order $\mathcal{K} = \{\chi^{-1}(\mathbf{x}_{34}), \chi^{-1}(\mathbf{x}_{25}), \chi^{-1}(\mathbf{x}_{23}), \chi^{-1}(\mathbf{x}_{12})\}$.

We first visit node for R_{34} and since it is a leaf node, we apply Lemma 4.1 with the root node as the heavy partition R_{34}^H as shown in Figure 4.4b. Once the heavy partition has been processed, we replace $R_{34}^{\mathcal{D}}$ with $R_{34}^{\mathcal{D},L}$ in the input database (Line 17), which will be used in all subsequent iterations of the algorithm. Next, we process leaf node R_{25} by again calling Lemma 4.1 with heavy partition R_{25}^H as the root. $R_{25}^{\mathcal{D}}$ is then replaced with $R_{25}^{\mathcal{D},L}$ in $\mathcal{D}$.

Now, both leaf nodes have their relations replaced by the light partitions. When we process node R_{23}, a non-leaf relation, we join all relations in the subtree rooted at R_{23} (shown in dashed rectangle in Figure 4.4d). Thus, the query $Q'(\mathbf{x}_{245}) \leftarrow R_{34}^L \wedge R_{25}^L \wedge R_{23}$ is evaluated, the variables in the bag for the node are replaced with $\mathbf{x}_{245}$ and the relation is the output of the query $Q'(\mathbf{x}_{245})$. Leaf nodes R_{25}^L and R_{34}^L are deleted, and R_{245} becomes a leaf node. $\chi^{-1}(\mathbf{x}_{245})$ is added to the front of $\mathcal{K}$ on Line 25. Therefore, in the next iteration, we take the heavy partition of node R_{245}^H and apply Lemma 4.1. Finally, we visit the node for R_{12}, process the join $Q''(\mathbf{x}_{145}) \leftarrow R_{12} \wedge R_{245}^L$ (Figure 4.4f) and the root node bag is modified to $\mathbf{x}_{145}$ with relation as the result $R_{145}^{\mathcal{D}} = Q''(\mathcal{D})$. Node R_{245}^L is deleted. At this point $\mathcal{K} = \emptyset$, and we union $Q''(\mathbf{x}_{145})$ and $\mathcal{J}$ on Line 20. Since the entire tree is now processed, the while loop terminates, and the final result $\mathcal{J}$ is returned.

Finding the optimal threshold Δ. To find the optimal threshold that minimizes the running time of Algorithm 3, we can equate the two terms in the running time expression of Theorem 4.2 to obtain $\Delta = |\texttt{OUT}|^{1/k}$, yielding a running time of $O(|\mathcal{D}| \cdot |\texttt{OUT}|^{1-1/k})$. However, the value of $|\texttt{OUT}|$ is not known apriori. To remedy this issue, we use the *doubling trick* [ACFS00] that was first introduced in the context of multi-armed bandit algorithms. The key idea is to guess the value of $|\texttt{OUT}|$. Suppose the guessed output size is O and let α be a constant that is an upper bound of the constant hidden in the big-O runtime complexity of Algorithm 3. We start with an estimate of $O_1 = 2^0$ in the first round and run the algorithm. If the algorithm does not finish execution in $\alpha \cdot |\mathcal{D}| \cdot O_i^{1-1/k}$ steps, then we terminate the algorithm and pick the new estimate to be $O_{i+1} = 2 \cdot O_i$ and re-run the algorithm. However, if the algorithm finishes, then we have successfully computed the query result. Note that α can be determined by doing an analysis of the program and counting the number operations required for each line. Hence, the algorithm terminates within $\lceil \log(2 \cdot |\texttt{OUT}|) \rceil$ rounds and the total running time is $\alpha \cdot |\mathcal{D}| \cdot \sum_{i \in [\lceil \log_2(2 \cdot |\texttt{OUT}|) \rceil]} O_i^{1-1/k} = \alpha \cdot |\mathcal{D}| \cdot \sum_{i \in [\lceil \log_2(2 \cdot |\texttt{OUT}|) \rceil]} 2^{i \cdot (1-1/k)} = O(|\mathcal{D}| \cdot |\texttt{OUT}|^{1-1/k})$ for any $k \geq 2$. More formally, we show the following result.

Theorem 4.1. *Given a reduced and existentially connected acyclic query $Q(\mathbf{x}_F)$, and a database $\mathcal{D}$, we can compute $Q(\mathcal{D})$ in time $O(|\mathcal{D}| + |\mathcal{D}| \cdot |\texttt{OUT}|^{1-1/k})$, where k is the number of atoms in Q.*

We note that the doubling trick argument for join evaluation has been used in prior works [DTL18, AP09]. This idea can also be applied to the results in [Hu24a], allowing us to shave the polylog

factors in the total running time and removing the need to estimate the output size via sophisticated algorithms.

General CQs. Finally, we discuss what happens for a general acyclic CQ that may not be reduced or existentially connected. In this case, we can relate the runtime of the algorithm to the projection width we defined in the previous section. The main insight here is that once a general acyclic CQ has been reduced and decomposed, we can evaluate each component separately. The query evaluation output of each component can be combined easily since the query can now be viewed as a free-connex acyclic query, whose evaluation is well understood [BDG07]. Note that for a CQ that is reduced and existentially connected, pw is exactly the number of atoms in the query.

Theorem 4.2. *Given an acyclic* CQ Q *and a database* $\mathcal{D}$*, we can compute the output* $Q(\mathcal{D})$ *in time* $O(|\mathcal{D}| + |\mathit{OUT}| + |\mathcal{D}| \cdot |\mathit{OUT}|^{1-1/\mathit{pw}(Q)})$.

Proof. This result follows exactly the argument in [Hu24a], which shows that the runtime of an evaluation algorithm for Q can be reduced to the computation of each query in $\mathsf{decomp}(\mathsf{red}[Q])$. In particular, as a first step it can be shown that we can compute in linear time an instance $\mathcal{D}'$ such that $Q(\mathcal{D}) \leftarrow \mathsf{red}[Q](\mathcal{D}')$; this is done by doing semijoins in the same order as the GYO algorithm.

As a second step, we use Threorem 4.1 to compute each query $Q_i \in \mathsf{decomp}(\mathsf{red}[Q])$ in time $O(|\mathcal{D}| + |\mathcal{D}| \cdot |Q_i(\mathcal{D})|^{1-1/k_i})$, where k_i is the number of atoms in Q_i. Then, we materialize each query and compute $Q(\mathcal{D})$ by doing the join $\bigwedge_i Q_i(\mathcal{D})$. This join query corresponds to a free-connex acyclic CQ, so it can be evaluated in time $O(|\mathcal{D}| + \sum_i |Q_i(\mathcal{D})| + |Q(\mathcal{D})|) = O(|\mathcal{D}| + |\mathtt{OUT}|)$. The desired claim follows from the fact that projection width is defined as the maximum number of atoms in any query Q_i of the decomposition. □

Self-joins. So far, we have assumed that the query does not contain any repeated relations (i.e. no self-joins). However, our framework can handle self-joins as well by performing a few basic transformations. First, if a query contains repeated relations, we make copies of the input relation(s) in the database involved in the self-join, assign a unique relational name to each copy, and use the unique name for each occurrence of the repeated relation to rewrite the query. Then, we order the schema of each relation according to the variable order $[n]$. This operation is straightforward since reordering of the variables in the schema of a relation merely corresponds to shuffling each tuple in the relation to match the reordered schema. Finally, for all atoms $R_K(\mathbf{x}_K), S_K(\mathbf{x}_K), \ldots, V_K(\mathbf{x}_K)$ that have the same schema, we only keep one atom in query (say $R_K(\mathbf{x}_K)$) and modify the instance $R_K^{\mathcal{D}} = R_K^{\mathcal{D}} \cap S_K^{\mathcal{D}} \cap \cdots \cap V_K^{\mathcal{D}}$. Each step takes at most $O(|\mathcal{D}|)$ time and satisfies the formulation of a CQ, and thus our main result can extend to self-joins.

Example 4.5. Consider the query $Q(x_1, x_2, x_4) \leftarrow R(x_1, x_2) \wedge R(x_2, x_1) \wedge S(x_2, x_3) \wedge S(x_3, x_4)$. The query contains a self-join on both R and S. Therefore, we first create two copies of relation

R: $R_1(x_1, x_2)$ and $R_2(x_2, x_1)$; and two copies of S: $S_1(x_2, x_3)$ and $S_2(x_3, x_4)$. The rewritten query becomes $Q(x_1, x_2, x_4) \leftarrow R_1(x_1, x_2) \wedge R_2(x_2, x_1) \wedge S_1(x_2, x_3) \wedge S_2(x_3, x_4)$. Next, we modify the schema of relation R_2 to $R_2(x_1, x_2)$ to order the variables in relation $R_2^{\mathcal{D}}(x_1, x_2)$. Finally, since R_1 and R_2 have the same schema, we compute $R_1^{\mathcal{D}} = R_1^{\mathcal{D}} \cap R_2^{\mathcal{D}}$ and discard R_2. The final rewritten query is $Q(x_1, x_2, x_4) \leftarrow R_1(x_1, x_2) \wedge S_1(x_2, x_3) \wedge S_2(x_3, x_4)$.

Repeated variables in the schema of a relation can also be handled in a linear time preprocessing step by modifying the schema to only have one occurrence of each variable and modifying each tuple in the relational instance.

4.4 Extension to Aggregation

In this following, we show that our algorithm (Algorithm 3) can be similarly extended to evaluate acyclic FAQs. We obtain the following result.

Theorem 4.3. *Given an acyclic* FAQ *query $\varphi(\mathbf{x}_F)$ over a semiring σ, database $\mathcal{D}$, we can compute the output $\varphi(\mathcal{D})$ in time $O(|\mathcal{D}| + |\mathit{OUT}| + |\mathcal{D}| \cdot |\mathit{OUT}|^{1-1/\mathsf{pw}(Q)})$.*

The algorithm requires only minor modifications to Algorithm 3: specifically, replacing natural joins (e.g., Line 22) with $\otimes$, and replacing unions (at Line 16 and Line 20) with $\oplus$, to aggregate the query results of $\varphi(\mathbf{x}_F)$. The correctness of this algorithm stems from the disjoint partitions of relations corresponding to each leaf on line 13 of Algorithm 3. In other words, it holds that $T^{\mathcal{D}}_{\chi(s)} = T^{\mathcal{D},H}_{\chi(s)} \oplus T^{\mathcal{D},L}_{\chi(s)}$ and by distributivity, we have

$$\varphi(\mathcal{D}) = \varphi\left((\mathcal{D} \setminus T^{\mathcal{D}}_{\chi(s)}) \cup T^{\mathcal{D},H}_{\chi(s)}\right) \oplus \varphi\left((\mathcal{D} \setminus T^{\mathcal{D}}_{\chi(s)}) \cup T^{\mathcal{D},L}_{\chi(s)}\right),$$

where the first sub-query is evaluated upfront at line 15. For the latter sub-query, if the leaf s is the last leaf of its parent being processed, the sub-query is directly evaluated in the next for-loop iteration at its parent level (i.e. the else branch at line 21). Otherwise, the relations in the database $(\mathcal{D} \setminus T^{\mathcal{D}}_{\chi(s)}) \cup T^{\mathcal{D},L}_{\chi(s)}$ will be further partitioned by the next sibling of s in the post-order traversal. The runtime argument follows exactly from the proof of Theorem 4.2.

4.5 Applications

In this section, we we apply our framework to recover state-of-the-art results, as well as obtain new results, for queries of practical interest.

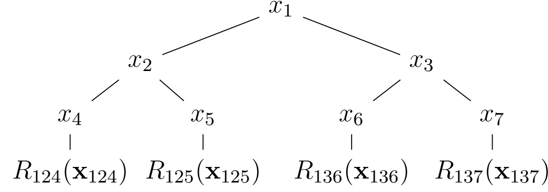


Figure 4.5: Relations formed by variables arranged as a complete binary tree. Every root-to-leaf path forms a relation (labeled).

4.5.1 Path Queries

We will first study the *path queries*, a class of queries that has immense practical significance. The projection width of a path query $P_k(x_1, x_{k+1})$ is k. Applying our main result, we get:

Theorem 4.4. *Given a path query $P_k(x_1, x_{k+1})$ and a database $\mathcal{D}$, there exists an algorithm that can evaluate the path query in time $O(|\mathcal{D}| + |\mathcal{D}| \cdot |\mathit{OUT}|^{1-1/k})$.*

For $k = 2$, our result matches the bound shown in [AP09]. Comparing our result to Hu, the bound obtained in Theorem 4.4 strictly improves the results obtained by Hu for $3 \leq k \leq 5$ and matches for $k = 6$ even if we assume $\omega = 2$. This result suggests there is room for further improvement in the use of fast matrix multiplication to obtain tighter bounds. For $k = 7$, our running time is better than the one obtained by Hu assuming the current best known value of $\omega = 2.371552$ [WXXZ24].

4.5.2 Hierarchical Queries

We show here the application of our result to *hierarchical queries*. A CQ is *hierarchical* if for any two of its variables, either their sets of atoms are disjoint or one is contained in the other. All hierarchical queries are acyclic, and all star queries $Q^\star_\ell$ (defined in Example 4.2) are hierarchical queries. They have $\mathtt{pw}(Q^\star_\ell) = \ell$, thus:

Theorem 4.5. *Given a star query $Q^\star_\ell(x_\ell)$ and a database $\mathcal{D}$, the star query can be evaluated in time $O(|\mathcal{D}| + |\mathcal{D}| \cdot |\mathit{OUT}|^{1-1/\ell})$.*

Theorem 4.5 recovers the bound from [AP09] for star queries and thus, provides an alternate proof of the combinatorial result in [AP09] for star queries. We note that for star queries the results merely improve the *analysis* of the Yannakakis algorithm. In other words, Yannakakis algorithm also achieves the time bound as specified by Theorem 4.5 but the routinely used upper bound of $O(|\mathcal{D}| \cdot |\mathtt{OUT}|)$ does not reflect that.

As another example, consider the query $Q(\mathbf{x}_{4567}) \leftarrow R_{124}(\mathbf{x}_{124}) \wedge R_{125}(\mathbf{x}_{125}) \wedge R_{136}(\mathbf{x}_{136}) \wedge R_{137}(\mathbf{x}_{137})$ formed by the relations shown in Figure 4.5. For this query, the projection width is

four and thus, we obtain an evaluation time of $O(|\mathcal{D}| + |\mathcal{D}| \cdot |\texttt{OUT}|^{3/4})$. [KNOZ20] proposed an algorithm for enumerating the results of any hierarchical query (not necessarily full) with delay[5] guarantees after preprocessing the input. In particular, they showed that after preprocessing time $T_p = O(|\mathcal{D}|^{1+(\mathsf{w}-1)\cdot\epsilon})$, it is possible to enumerate the query result with delay $\delta = O(|\mathcal{D}|^{1-\epsilon})$, where w is the *static width* (a width parameter introduced by [KNOZ20]) of a hierarchical query, for any $0 \leq \epsilon \leq 1$. Note that an algorithm with preprocessing T_p and delay guarantee δ directly leads to a join evaluation algorithm that takes time $O(T_p + \delta \cdot |\texttt{OUT}|)$. For the example query $Q(\mathbf{x}_{4567})$, it turns out that $\mathsf{w} = 4$, and thus, the running time can be minimized for a suitable choice of threshold ϵ to also obtain the same running time that our algorithm achieves. A deeper exploration of this intriguing connection is a topic left for future research.

4.5.3 General Queries

Abo Khamis et al. [KNS17] showed that given any CQ Q (possibly cyclic) and an input database $\mathcal{D}$, the PANDA algorithm can decompose the query and database instance into a constant number of pairs $(Q_i, \mathcal{D}_i)$ such that $Q(D) \leftarrow \bigcup_i Q_i(\mathcal{D}_i)$. Further, it is also guaranteed that each Q_i is acyclic, $|\mathcal{D}_i| = O(|\mathcal{D}|^{\mathsf{subw}(Q)})$ (each $\mathcal{D}_i$ is computed in time and of size $\tilde{O}(|\mathcal{D}|^{\mathsf{subw}(Q)})$), and $|Q_i(\mathcal{D}_i)| \leq |Q(\mathcal{D})|$. Since each Q_i is acyclic, we can apply our main result and obtain the following theorem.

Theorem 4.6. *Given a* CQ Q *and database* $\mathcal{D}$*, there exists an algorithm to evaluate* $Q(\mathcal{D})$ *in time* $\tilde{O}(|\mathcal{D}|^{\mathsf{subw}(Q)} + |\mathcal{D}|^{\mathsf{subw}(Q)} \cdot |\mathit{OUT}|^{1-1/\max_i \mathtt{pw}(Q_i)})$*, where* $(Q_i, \mathcal{D}_i)$ *is the set of decomposed queries generated by* PANDA.

Example 4.6. Consider the 4-cycle query $Q^\diamond(\mathbf{x}_{123}) \leftarrow R_{12}(\mathbf{x}_{12}) \wedge R_{23}(\mathbf{x}_{23}) \wedge R_{34}(\mathbf{x}_{34}) \wedge R_{14}(\mathbf{x}_{14})$. For this query , $\mathsf{subw}(Q^\diamond(\mathbf{x}_{123})) = 3/2$ and PANDA partitions $Q^\diamond(\mathbf{x}_{123})$ into two queries, $Q^\diamond_1(\mathbf{x}_{123}) \leftarrow S_{123}(\mathbf{x}_{123}) \wedge S_{134}(\mathbf{x}_{134})$ and $Q^\diamond_2(\mathbf{x}_{123}) \leftarrow S_{124}(\mathbf{x}_{124}) \wedge S_{234}(\mathbf{x}_{234})$. It is easy to see that $\mathtt{pw}(Q^\diamond_1) = 1$ and $\mathtt{pw}(Q^\diamond_2) = 2$. Thus, the query can be evaluated in time $\tilde{O}(|\mathcal{D}|^{3/2} \cdot |\texttt{OUT}|^{1/2})$. We note that [Hu24a] requires $\tilde{O}(|\mathcal{D}|^{3/2} \cdot |\texttt{OUT}|^{5/6})$ time for the 4-cycle query $Q^\diamond(\mathbf{x}_{123})$.

4.6 Lower Bounds

In this section, we demonstrate several lower bounds that show optimality for a subclass of CQs. First, we define the k-clique problem that will be central to our lower bounds. Given an undirected graph $G = (V, E)$ and an integer $k \leq |V|$, the k-clique problem consists of deciding if the graph G contains k vertices such that each of the k vertices are connected to each other via an edge in

[5] The delay of enumerating query results refers to the upper bound on the time between outputting any two consecutive output tuples (including from start of the algorithm to the first tuple, and the last tuple to the end of the algorithm).

E. We also define the *minimum-weight* k-clique problem: suppose the graph G is equipped with an edge-weight function that maps edges to weights in $[0, M]$ for integer $M > 0$, find the k-clique where the edge sum is minimized. We will use the well-established conjectures from fine-grained complexity for the two k-clique problems to prove our lower bounds.

Definition 4.2 (Boolean k-Clique Conjecture)**.** There is no real $\epsilon > 0$ such that computing the k-clique problem (with $k \geq 3$) over the Boolean semiring in an (undirected) n-node graph requires time $O(n^{k-\epsilon})$ using a combinatorial algorithm.

Definition 4.3 (Min-Weight k-Clique Conjecture)**.** There is no real $\epsilon > 0$ such that computing the k-clique problem (with $k \geq 3$) over the tropical semiring in an (undirected) n-node graph with integer edge weights can be done in time $O(n^{k-\epsilon})$.

Our first lower bound tells us that the dependence in the bound of Theorem 4.6 on both the submodular width and projection width is somewhat necessary.

Theorem 4.7. *Take any integer $\ell \geq 2$ and any rational $w \geq 1$ such that $\ell \cdot w$ is an integer. Then, there exists a query Q with projection width (i.e ℓ free variables) and submodular width w such that no combinatorial algorithm can compute it over input $\mathcal{D}$ in time $O(|\mathcal{D}|^{w} \cdot |\mathtt{OUT}|^{1-1/\ell-\epsilon})$ for any real $\epsilon > 0$, assuming the Boolean k-Clique Conjecture.*

Proof. Consider the graph $G = (V, E)$ for which we need to decide whether there exists a clique of size $k = (\ell - 1) + \ell \cdot w$. Let $n = |V|$. It will be helpful to distinguish the k variables of the clique into ℓ variables $x_1, \ldots, x_\ell$ and the remaining $k - \ell$ variables $y_1, \ldots, y_{k-\ell}$. Note that $k - \ell = \ell \cdot w - 1$ is an integer ≥ 1; hence, there exists at least one y-variable.

We will first compute all the cliques of size ℓ in G and store them in a relation $C(v_1, \ldots, v_\ell)$; this forms the input database $\mathcal{D}$. Note that this step needs $O(n^\ell)$ time, and moreover the size of C is n^ℓ. Now, consider the following query:

$$Q(x_1, \ldots, x_\ell) \leftarrow U_1[x_1, y_1, \ldots, y_{k-\ell}] \wedge U_2[x_2, y_1, \ldots, y_{k-\ell}] \wedge \cdots \wedge U_\ell[x_\ell, y_1, \ldots, y_{k-\ell}]$$

where $U_i[x_i, y_1, \ldots, y_{k-\ell}]$ stands for the join of following atoms:

$$\{C(v_1, \ldots, v_\ell) \mid v_1, \ldots, v_\ell \text{ are all possible sets of size } \ell \text{ from } \{x_i, y_1, \ldots, y_{k-\ell}\}\}$$

This is a well-defined expression, since $k - \ell = \ell \cdot w - 1$ and $w \geq 1$. The submodular width of this query is w, as proved in [ZDK+24a]. It is also easy to see that the query has ℓ free variables.

Suppose now we can compute Q in time $O(|\mathcal{D}|^{\mathsf{subw}} \cdot |\mathtt{OUT}|^{1-1/\ell-\epsilon})$ for some $\epsilon > 0$. Note that $|\mathcal{D}| \leq n^\ell$ and $|\mathtt{OUT}| \leq n^\ell$. Hence, this implies that we can compute Q on the above input in time $O(n^{w \cdot \ell + \ell - 1 - \ell\epsilon}) = O(n^{k-\epsilon'})$ where $\epsilon' = \ell\epsilon$.

Once the output has been evaluated, for each tuple $t \in Q(x_1, \ldots, x_\ell)$, we can verify in constant time that any two vertices $t(i)$ and $t(j)$ ($1 \le i, j \le \ell$) are connected via an edge. Thus, in $O(n^\ell)$ time, we can verify whether the vertices form an ℓ-clique. Further, by our join query construction, each of the ℓ vertices is guaranteed to connect with a common vertex v' (which corresponds to a valuation of the remaining variables $y_1, \ldots, y_{k-\ell}$).

All together, this obtains a (combinatorial) algorithm that computes whether a k-clique exists in time $O(n^{k-\epsilon'} + n^\ell) = O(n^{k-\epsilon'})$, contradicting the lower bound conjecture. □

We next prove a general result that gives output-sensitive lower bounds for arbitrary CQs. W use the notion of *clique embedding* [FKZ23]. We say that two sets of vertices $X, Y \subseteq V(H)$ *touch* in $\mathcal{H}$ if either $X \cap Y \neq \emptyset$ or there is a hyperedge $e \in E(\mathcal{H})$ that intersects both X and Y.

Definition 4.4 (Clique Embedding)**.** Let $k \ge 3$ and $\mathcal{H}$ be a hypergraph. A *k-clique embedding*, denoted as $C_k \mapsto \mathcal{H}$, is a mapping ψ that maps every $v \in \{1, \ldots, k\}$ to a non-empty subset $\psi(v) \subseteq V(\mathcal{H})$ such that the following hold:

1. $\psi(v)$ induces a connected subgraph;
2. for any two $u \neq v \in \{1, \ldots, k\}$ then $\psi(u), \psi(v)$ *touch* in $\mathcal{H}$.

It is often convenient to describe a clique embedding ψ by the reverse mapping $\psi^{-1}(x) = \{i \mid x \in \psi(i)\}$, for $x \in V(\mathcal{H})$. For a hyperedge $K \in E(\mathcal{H})$, its *weak edge depth* is $d_\psi(J) := |\{v \mid \psi(v) \cap J \neq \emptyset\}|$, i.e., the number of vertices from $V(\mathcal{H})$ that map to some variable in J. We also define the *edge depth* of $K \in E(\mathcal{H})$ as $d^+_\psi(K) := \sum_{v \in K} d_\psi(v)$, i.e. weak edge depth but counting multiplicity. For an embedding ψ and a hyperedge $F \in E(\mathcal{H})$, the *F-weak edge depth of* ψ is defined as $\mathsf{wed}^F(\psi) := \max\limits_{K \in E(\mathcal{H}) \setminus F} d_\psi(J)$, i.e. the maximum of weak edge depths excluding F.

Theorem 4.8. *Given any* CQ $Q(\mathbf{x}_F) \leftarrow \bigwedge_{K \in \mathcal{E}} R_K(\mathbf{x}_K)$ *and database* $\mathcal{D}$*, let* ψ *be a* k*-clique embedding of the hypergraph* $\mathcal{H}' = ([n], E(\mathcal{H}) \cup \{F\})$ *such that* $x \cdot \mathsf{wed}^F(\psi) + y \cdot d^+_\psi(F) \le k$ *for positive* $x, y > 0$*. Then, there is no combinatorial algorithm with running time* $O(|\mathcal{D}|^{x-\epsilon} \cdot |\mathit{OUT}|^{y-\epsilon'})$ *for evaluating* $Q(\mathcal{D})$ *assuming the Boolean* k*-Clique Conjecture, where* ϵ, ϵ' *are non-negative with* $\epsilon + \epsilon' > 0$*.*

Proof. Construct the instance as Theorem 11 in [FKZ23] and observe that there exists a k-clique if and only if there exists a tuple in the output that is consistent with the domains in variables F. By a consistent tuple, we mean a tuple whose values associated to the same partitions of the input graph for finding k-clique are the same. The time and size to construct such instance $\mathcal{D}$ is bounded by $O(n^{\mathsf{wed}^F(\psi)})$, and observe the size of $|\mathtt{OUT}|$ is bounded by $O(n^{d^+_\psi(F)})$. Thus, such output-sensitive combinatorial algorithm will yield a combinatorial algorithm for k-clique that runs in time $O(n^{x \cdot (\mathsf{wed}^F(\psi) - \epsilon)} \cdot n^{y \cdot (d^+_\psi(F) - \epsilon')}) = O(n^{k - \epsilon''})$ for some $\epsilon'' > 0$, contradicting the Boolean k-Clique Conjecture. □

We can show an analogous result for FAQ queries over the tropical semiring.

Theorem 4.9. *Given any* FAQ $\varphi(\mathbf{x}_F)$ *as* (2.10) *and database* $\mathcal{D}$*, let* ψ *be a* k*-clique embedding of the hypergraph* $\mathcal{H}' = ([n], E(\mathcal{H}) \cup \{F\})$ *such that* $x \cdot \mathsf{wed}^F(\psi) + y \cdot d^+_\psi(F) \leq k$ *for positive* $x, y > 0$*. Then, there exists no algorithm that evaluates* $\varphi(\mathcal{D})$ *over the tropical semiring in time* $O(|\mathcal{D}|^{x-\epsilon} \cdot |\mathit{OUT}|^{y-\epsilon'})$ *assuming the Min-Weight* k*-Clique Conjecture, where* ϵ, ϵ' *are non-negative with* $\epsilon + \epsilon' > 0$.

Proof. Construct the instance as Theorem 11 in [FKZ23]. Given the output tuples, we simply check for the consistency of domains in variables $\mathbf{x}_F$ and return the smallest annotation among consistent tuples after sorting. Observe that this procedure correctly solves the Min-Weight k-clique problem. The time analysis is then identical as the proof of Theorem 4.8. □

Hu defined the notion of freew of any acyclic query and proved that any "semiring algorithm" requires $\Omega(|\mathcal{D}| \cdot |\mathtt{OUT}|^{1-\frac{1}{\mathsf{freew}(Q)}} + |\mathtt{OUT}|)$ [Hu24a]. Our lower bound is complementary to Hu's lower bound. Specifically, Hu showed a stronger result that the lower bound applies for every value of $|\mathcal{D}|$ and $|\mathtt{OUT}|$, whereas our result shows the existence of a hard database instance. However, our lower bound also applies to cyclic CQs whereas Hu's $\mathsf{freew}(Q)$ is not defined for cyclic queries. Therefore, the two results are incomparable. For the special case of $x = 1$, we match Hu's lower bound (see Proposition 4.5) by providing an alternate proof that there exists a clique embedding for Q' such that $y = 1 - \frac{1}{\mathsf{freew}(Q)}$. We now apply Theorem 4.8 and Theorem 4.9 to get tight lower bounds for star queries and can be extended to hierarchical queries (note that star queries are also hierarchical queries).

Proposition 4.5. *Given any* CQ $Q(\mathbf{x}_F) \leftarrow \bigwedge_{K \in \mathcal{E}} R_K(\mathbf{x}_K)$*, define* $Q'(\mathbf{x}_F) \leftarrow \bigwedge_{K \in \mathcal{E}} R_K(\mathbf{x}_K) \wedge R(\mathbf{x}_F)$*. Then, there exists a* k*-clique embedding* ψ *such that* $\frac{k - \mathsf{wed}^F(\psi)}{d^+_\psi(F)} = 1 - \frac{1}{\mathsf{freew}(Q)}$.

Proof. It suffices to construct a clique embedding on the connected components that achieves $\mathsf{freew}(Q)$ after Hu's cleanse and decomposition steps [Hu24a]. It is straightforward to see that one can assign each element in S an isolated output variable [Hu24a]. Consider the $(\mathsf{freew}(Q)+1)$-clique embedding ψ where ψ maps $\mathsf{freew}(Q)$ distinct colors to each one of those distinguished isolated output variables, and map an additional fresh color to all other variables (including all join variables and possibly some isolated output variables that are not assigned to S). Clearly, ψ is a $(\mathsf{freew}(Q)+1)$-clique embedding for Q'. Moreover, $\mathsf{wed}^F(\psi) = 2$ and $d^+_\psi(F) = \mathsf{freew}(Q)$, and thus $\frac{k - \mathsf{wed}^F(\psi)}{d^+_\psi(F)} = \frac{\mathsf{freew}(Q)+1-2}{\mathsf{freew}(Q)} = 1 - \frac{1}{\mathsf{freew}(Q)}$, completing the proof. □

Theorem 4.10. *For the star query* $Q^\star_\ell$*, there exists a database* $\mathcal{D}$ *such that no algorithm can have runtime of* $O(|\mathcal{D}| \cdot |Q^\star_\ell(\mathcal{D})|^{1-1/\ell-\epsilon})$ *for any real* $\epsilon > 0$ *subject to the Boolean* k*-clique conjecture.*

Proof. The star query $Q^\star_\ell$ is defined as $Q^\star_\ell(x_1, \dots, x_\ell) \leftarrow R_1(x_1, y) \wedge \cdots \wedge R_\ell(x_\ell, y)$. Consider the query $(Q^\star_\ell)'(x_1, \dots, x_\ell) \leftarrow R_1(x_1, y) \wedge \cdots \wedge R_\ell(x_\ell, y) \wedge R(x_1, x_2, \dots, x_\ell)$. Observe that $\psi(i) = x_i$ for

$1 \leq i \leq \ell$ and $\psi(\ell+1) = y$ is a $(\ell+1)$-clique embedding of $(Q^\star_\ell)'$. This has $\mathsf{wed}^F(\psi) = 2$ and $d^+_\psi(F) = \ell$. For $x = 1$ and $y = 1 - 1/\ell$, we have $x \cdot \mathsf{wed}^F(\psi) + y \cdot d^+_\psi(F) \leq \ell + 1$. Now apply Theorem 4.8. □

Theorem 4.11. *For the star query $Q^\star_\ell$, there exists a database $\mathcal{D}$ such that no algorithm can have runtime of $O(|\mathcal{D}| \cdot |Q^\star_\ell(\mathcal{D})|^{1-1/\ell-\epsilon})$ for any real $\epsilon > 0$ subject to the Min-Weight k-Clique Conjecture.*

4.7 Related Work

Prior works have investigated problems closely related to output-sensitive evaluation. Deng et al. [DLT23a] presented a dynamic index structure for output-sensitive join sampling. Riko and Stöckel [JS15] studied output-sensitive matrix multiplication (i.e. the two path query), a result that built upon the join-project query evaluation results from [AP09]. Deng et al. [DTL18] presented output-sensitive evaluation algorithms for set similarity, an important practical class of queries used routinely in recommender systems, graph analytics, etc. An alternate way to evaluate join queries is to use delay based algorithms. As we previously saw in Section 4.4, our result is able to match the join running time obtained via [KNOZ20] for star queries. Recent work [AHSY24] has also studied the problem of reporting t patterns in graphs when the input is temporal, i.e., the input is changing over time. Output-sensitive algorithms has also been developed for the problem of maximal clique enumeration [CYQ13, MU04, CGMV16]. A series of interesting results have been known for listing k-cliques and k-cycles in output-sensitive way. [BPWZ14] designed output-sensitive algorithms for listing t triangles, the simplest k-clique, using fast matrix multiplication. The work of [JX23] and [AKLS22] show algorithms for list t 4-cycles, and [JWZ24] shows how to list t 6-cycles. The upper bounds were further shown to be tight under the 3SUM hypothesis.

4.8 Conclusion

In this chapter, we presented a novel generalization of Yannakakis algorithm that is provably more efficient for a large class of CQs. We show several applications of the generalized algorithm (which is combinatorial in nature) to recover state-of-the-art results known in existing literature, as well as new results for popular queries such as star queries and path queries. Surprisingly, our results show that for some queries, our combinatorial algorithm is better than best known results that use fast matrix multiplication. We complement our upper bounds with a matching lower bound for a subclass of cyclic and acyclic CQs, for both Boolean semirings (a.k.a joins) as well as aggregations.

Chapter 5

Join with Negations

This chapter focuses on the query evaluation problem for Conjunctive Queries with negation ($\mathsf{CQ}^\neg$), a fundamental class of relational queries.

5.1 Introduction

We will consider a $\mathsf{CQ}^\neg$ as having the following form:

$$Q(\mathbf{x}_F) \leftarrow \bigwedge_{K \in \mathcal{E}^+} R_K(\mathbf{x}_K) \wedge \bigwedge_{K \in \mathcal{E}^-} \neg R_K(\mathbf{x}_K) \tag{5.1}$$

where the variables are of the form x_i, $i \in [n] = \{1, 2, \ldots, n\}$, $\mathcal{E}^+, \mathcal{E}^-$ are two sets of hyperedges that are subsets of $[n]$, and $F \subseteq [n]$ are the free variables. We will call the triple $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ the *signed hypergraph* of Q. We will consider only *safe queries*, where $\bigcup_{K \in \mathcal{E}^+} K = [n]$. When $\mathcal{E}^- = \emptyset$, then Q is a Conjunctive Query (CQ) with the associated hypergraph $([n], \mathcal{E}^+)$.

The complexity of query evaluation for CQs is well-understood. Yannakakis [Yan81] first showed that a Boolean CQ (i.e., $F = \emptyset$) can be evaluated on a database $\mathcal{D}$ of size $|\mathcal{D}|$ in time $O(|Q| \cdot |\mathcal{D}|)$ if the hypergraph $([n], \mathcal{E}^+)$ is α-acyclic ($|Q|$ denotes the size of the query). Further work [BDG07, BGS20] generalized this result to show that if Q is *free-connex α-acyclic* then the tuples in $Q(\mathcal{D})$ can be enumerated with constant delay $O(|Q|)$ after a linear-time preprocessing step. Free-connex α-acyclicity means that both $([n], \mathcal{E}^+)$ and $([n], \mathcal{E}^+ \cup \{F\})$ are α-acyclic hypergraphs. It was also shown [BDG07] that this tractability result is tight under widely believed lower-bound conjectures.

5.1.1 CQs with Negation

Our first goal in this chapter is to generalize the above classic result to the case where $\mathcal{E}^- \neq \emptyset$. Prior work has looked into this problem, but without achieving a complete answer.

To explain the current progress, let us first consider the case of a Boolean $\mathsf{CQ}^\neg$ and attempt to solve the (seemingly harder) problem of counting the number of valuations that satisfy the body of Q

with input a database $\mathcal{D}$, which we will denote as $\#Q(\mathcal{D})$. We should note here that $\#Q$ is solvable in $O(|\mathcal{D}|)$ time if Q is an α-acyclic CQ. Brault-Baron [Bra13] had the insight that we can compute $\#Q$ using the inclusion-exclusion principle. Indeed, let Q_S be the Boolean CQ with hypergraph $([n], \mathcal{E}^+ \cup S)$ for any $S \subseteq \mathcal{E}^-$. Then we can write:

$$\#Q(\mathcal{D}) = \sum_{S \subseteq \mathcal{E}^-} (-1)^{|S|} \#Q_S(\mathcal{D}) \tag{5.2}$$

Hence, if every Q_S is α-acyclic, then $\#Q$ (and thus Q) can be computed with data complexity $O(|\mathcal{D}|)$. This naturally leads to the notion of *signed acyclicity*, introduced in [Bra13]: a Boolean $\mathsf{CQ}^\neg$ is signed-acyclic if the hypergraph $([n], \mathcal{E}^+ \cup S)$ is α-acyclic for every $S \subseteq \mathcal{E}^-$. Thus, a Boolean $\mathsf{CQ}^\neg$ can be computed in linear time (data complexity) if it is signed-acyclic. Interestingly, if $\mathcal{E}^+$ consists only of singleton hyperedges, then signed acyclicity is equivalent to β-acyclicity of the hypergraph $([n], \mathcal{E}^-)$. However, there are two issues with applying the inclusion-exclusion approach. First, it has an exponential dependency on $|Q|$ and thus does not give a polynomial-time algorithm in combined complexity. Second, it cannot be used to provide any delay guarantees for the enumeration problem in non-Boolean queries.

The second issue was partially addressed by Brault-Baron [Bra12, Bra13], who proposed an enumeration algorithm for any free-connex signed-acyclic $\mathsf{CQ}^\neg$. However, this algorithm either achieves constant delay with a $O(|\mathcal{D}| \log^{|Q|} |\mathcal{D}|)$ preprocessing time, or achieves logarithmic delay with linear preprocessing time. The logarithmic factor is a consequence of the technique used, which translates a database instance to an instance over the Boolean domain.

Our first main result shows that the translation to the Boolean domain is not necessary and in fact we can achieve both constant delay and linear time preprocessing for free-connex signed-acyclic queries. Moreover, our algorithm has only a polynomial dependence on the size of the query.

Theorem 5.1. *Let Q be a free-connex signed-acyclic $\mathsf{CQ}^\neg$. Then there is an algorithm that can enumerate the results of $Q(\mathcal{D})$ with $O(|Q|^3 + |Q| \cdot |\mathcal{D}|)$ preprocessing time and $O(|Q|)$ delay.*

5.1.2 FAQ with Negation

Our second goal is to study the evaluation of $\mathsf{CQ}^\neg$ in the presence of aggregation. We do this by studying a more general problem, that of computing a $\mathsf{CQ}^\neg$ under a general semiring, following the approach of FAQs [AKNR16]. More precisely, given a commutative semiring $\sigma = (\boldsymbol{D}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, we define an $\mathsf{FAQ}^\neg$ as an expression of the form:

$$\varphi(\mathbf{x}_F) = \bigoplus_{\mathbf{x}_{[n] \setminus F}} \bigotimes_{K \in \mathcal{E}^+} R_K(\mathbf{x}_K) \otimes \bigotimes_{N \in \mathcal{E}^-} \overline{R}_N(\mathbf{x}_N), \tag{5.3}$$

Here, a *positive factor* R_K can be viewed as a table of entries of the form $\langle \mathbf{x}_K, R_K(\mathbf{x}_K)\rangle$ (where the weight of tuple $\mathbf{x}_K$ is the value $R_K(\mathbf{x}_K) \in \boldsymbol{D}$), and for entries not in the table, the weight is implicitly $\mathbf{0}$. On the other hand, a *negative factor* $\overline{R}_N$ is a table of entries of the form $\langle \mathbf{x}_K, R_N(\mathbf{x}_N)\rangle$, and for entries not in the table, the weight is a default constant value $\mathbf{c}_N \neq \mathbf{0}$. To recover the $\mathsf{CQ}^{\neg}$ setting, we choose the Boolean semiring, and encode the values of the negative factor such that it is $\mathbf{0}$ if the tuple is in the table, otherwise $\mathbf{1}$. Our semiring formulation is though much more general: the only difference between a negative and a positive factor in our setting is whether the "default" value of a tuple not in the table is $\mathbf{0}$. This observation offers a novel angle to the semantics of negation, and is critical to our algorithms. For $\mathsf{FAQ}^{\neg}$, we can show the following.

Theorem 5.2. *Let φ be a free-connex signed-acyclic* $\mathsf{FAQ}^{\neg}$ *over a commutative semiring σ. Then there is an algorithm that can enumerate φ with $O(|\varphi|^3 + |\varphi| \cdot |\mathcal{D}| \cdot \alpha(14 \cdot |\mathcal{D}|, |\mathcal{D}|))$ preprocessing time and $O(|\varphi|)$ delay.*

Here, $|\varphi|$ denotes the size of the query φ and $\alpha(m, n)$ denotes the inverse Ackermann function, which grows extremely slowly as a bi-variate function of m, n. The appearance of this function in the runtime expression is surprising in our opinion. It occurs because the aggregation problem reduces to the well-studied problem of computing interval sums over an arbitrary semiring, called $\mathsf{RangeSum}$ [CR91, Yao82]. If the semiring structure allows for linear preprocessing and constant-time answering for its $\mathsf{RangeSum}$ problem variant, then we can drop the Ackermann factor. Examples of such semirings are the Boolean semiring $(\{\mathsf{true}, \mathsf{false}\}, \vee, \wedge, \mathsf{false}, \mathsf{true})$, tropical semiring $(\mathbb{R}, \min, +, +\infty, 0)$, and semirings with additive inverse (e.g. the counting ring over integers, i.e. $(\mathbb{Z}, +, \times, 0, 1)$ which we use to count solutions).

Theorem 5.3. *Let φ be a free-connex signed-acyclic* $\mathsf{FAQ}^{\neg}$ *over a commutative semiring σ with additive inverse. Then there is an algorithm that can enumerate φ with $O(|\varphi|^3 + |\varphi| \cdot |\mathcal{D}|)$ preprocessing time and $O(|\varphi|)$ delay.*

5.1.3 Lower Bounds

Our third goal is to match our linear-time upper bounds with lower bounds. In this direction, we show that under believable conjectures, any $\mathsf{CQ}^{\neg}$ that is not free-connex signed-acyclic does not admit an algorithm that can emit the first result in linear time (hence matching the upper bound). Our conditional lower bounds are somewhat weaker than the ones obtained for CQs because the presence of negation means that we cannot use the sparse version of some problems (e.g., detecting a triangle, or Boolean matrix multiplication). For $\mathsf{FAQ}^{\neg}$, we show stronger conditional lower bounds over the tropical semiring and counting ring based on weaker lower bound conjectures. Finally, we

provide some evidence that the inverse Ackermann factor in the runtime for general semirings is unavoidable. In particular, we show that the query $\varphi(x) = \bigoplus_y A(x) \otimes B(y) \otimes \overline{R}(x, y)$ corresponds to a variant of the offline $\mathsf{RangeSum}$ problem over a general $\oplus$ operator. Using this observation, we can modify a construction of Chazelle [CR91] to show a superlinear lower bound on the number of $\oplus$ operations necessary to compute φ.

5.1.4 Extensions

In the final part, we demonstrate that our algorithm for $\mathsf{CQ}^\neg$ can be applied to obtain optimal algorithms with linear-time preprocessing and constant delay for the problem of computing the difference between two CQs of the form $Q_1 - Q_2$, which was recently studied by Hu and Wang [HW23]. We also discuss briefly how to reason about the evaluation of $\mathsf{CQ}^\neg$ when the query is not acyclic.

5.2 Preliminaries

We start with some basic definitions and notations that will be used throughout this chapter.

Hypergraphs. A hypergraph is a pair $\mathcal{H} = ([n], \mathcal{E})$ where $[n] = \{1, \dots, n\}$ is the set of vertices of $\mathcal{H}$ and $\mathcal{E}$ is a multiset[1] of hyperedges where each $K \in \mathcal{E}$ is a nonempty subset of $[n]$.

A signed hypergraph is a tuple $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ where $[n]$ is the set of vertices of $\mathcal{H}$, $\mathcal{E}^+$ and $\mathcal{E}^-$ are two multisets of hyperedges where each hyperedge $K \in \mathcal{E}^+$ (resp. $N \in \mathcal{E}^-$) is a subset of $[n]$. We consider only *safe* signed hypergraphs, where every vertex in $[n]$ occurs in some hyperedge $K \in \mathcal{E}^+$.

CQs with Negation. We define a query in $\mathsf{CQ}^\neg$ with signed hypergraph $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ as having the following form:

$$Q(\mathbf{x}_F) \leftarrow \bigwedge_{K \in \mathcal{E}^+} R_K(\mathbf{x}_K) \wedge \bigwedge_{K \in \mathcal{E}^-} \neg R_K(\mathbf{x}_K)$$

For any $i \in [n]$, x_i denotes a variable, and for any subset $K \subseteq [n]$, we define $\mathbf{x}_K = (x_i)_{i \in K}$. Overloading notation, we also refer to $[n]$ as the set of variables. The set $F \subseteq [n]$ are the free variables of Q. We will consider only *safe queries*, where $\bigcup_{K \in \mathcal{E}^+} K = [n]$. When $\mathcal{E}^- = \emptyset$, then Q is a Conjunctive Query (CQ) with the associated hypergraph $([n], \mathcal{E}^+)$.

For any $i \in [n]$, we will use a_i to denote a value in the discrete domain $\mathsf{Dom}(x_i)$ of the variable x_i. For any subset $K \subseteq [n]$, we also define tuples as $\mathbf{a}_K = (a_i)_{i \in K} \in \mathsf{Dom}(\mathbf{x}_K)$, where $\mathsf{Dom}(\mathbf{x}_K) = \prod_{i \in K} \mathsf{Dom}(x_i)$.

Enumeration and Complexity. In this chapter, we study the enumeration problem for $\mathsf{FAQ}^\neg$, $\mathsf{Enum}(\varphi, \mathcal{D})$, which takes as input a $\mathsf{FAQ}^\neg$ φ and a database instance $\mathcal{D}$ and outputs a sequence of

[1] A multiset is a collection of elements each of which can occur multiple times

answers such that every *answer* in $\varphi(\mathcal{D})$ is printed precisely once. An enumeration algorithm for $\mathsf{Enum}(\varphi, \mathcal{D})$ may consist of two phases:

- (preprocessing phase) it constructs efficient data structures from φ and $\mathcal{D}$; and
- (enumeration phase) it may access the data structures built during preprocessing, and emit the answers of $\varphi(\mathcal{D})$ one by one, without repetitions.

We say that an enumeration algorithm enumerates with delay $O(\tau)$ if the time between the emission of any two consecutive answers (and the time to emit the first answer, and the time from the last answer to the end) is bounded by $O(\tau)$. In particular, we say that an enumeration algorithm is *constant-delay* if it enumerates with delay independent of the input database size $|\mathcal{D}|$.

Context-free Grammar (CFG). If u, v, w are strings of terminals and non-terminals, and $\ell ::= w$ is a rule of the CFG, we say that $u\ell v$ *yields* uwv in the CFG (written as $u\ell v \Rightarrow uwv$). We say that u *derives* v ($u \overset{*}{\Rightarrow} v$) in the CFG if $u = v$ or if there is a sequence $u_1, \ldots, u_k$ for $k \geq 0$ such that

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v.$$

A *derivation* of a string for a grammar is a sequence of grammar rule applications that transform the start symbol into the string. A derivation proves that the string belongs to the grammar's language.

Ackermann function. The Ackermann function $A(m, n)$ for integers $m, n \geq 0$ is recursively defined as follows:

1. $A(0, n) = n + 1$;
2. $A(m + 1, 0) = A(m, 1)$; and
3. $A(m + 1, n + 1) = A(m, A(m + 1, n))$.

It is known that the Ackermann function grows faster than any primitive recursive function and therefore is not itself primitive recursive. The inverse Ackermann function $\alpha(m, n)$ is defined as

$$\alpha(m, n) = \min\{i \geq 1 : A(i, \lfloor \frac{m}{n} \rfloor) \geq \log_2(n)\}$$

and $\alpha(m, n)$ grows very slowly. For example, we have $\alpha(n, n) < 5$ for any practical integer n.

5.3 Signed Acyclicity

Before we introduce signed acyclicity, we first go over the notions of α- and β-acyclicity.

A hypergraph $\mathcal{H} = ([n], \mathcal{E})$ is α-acyclic if there is a tree $\mathcal{T} = (V(\mathcal{T}), E(\mathcal{T}))$ and a bijective function $\chi : \mathcal{E} \to V(\mathcal{T})$ such that for every vertex $v \in [n]$, the set of nodes $\{\chi(K) \mid v \in K, K \in \mathcal{E}\}$ induces a connected component in $\mathcal{T}$. A vertex $v \in [n]$ is an α-leaf of $\mathcal{H}$ if the multiset $\{K \in \mathcal{E} \mid v \in K\}$ contains a maximal element K_v with respect to $\subseteq$. It is known that every α-acyclic hypergraph has an α-leaf [BB16].

A hypergraph $\mathcal{H} = ([n], \mathcal{E})$ is β-acyclic if for any subset $\mathcal{E}' \subseteq \mathcal{E}$, $\mathcal{H}' = ([n], \mathcal{E}')$ is α-acyclic. A vertex $v \in [n]$ is a β-leaf of $\mathcal{H}$ if the multiset $\{K \in \mathcal{E} \mid x \in K\}$ can be linearly ordered by $\subseteq$. It is known that every β-acyclic hypergraph has a β-leaf [BB16, BK80].

We can now introduce the notion of signed acyclicity, slightly modified from [Bra13] to take multisets into account.

Definition 5.1 (Signed Acyclicity)**.** A signed hypergraph $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ is *signed-acyclic* if $([n], \mathcal{E}^+ \cup \mathcal{E}')$ is α-acyclic for every multiset $\mathcal{E}' \subseteq \mathcal{E}^-$.

5.3.1 Signed Leaves

Similar to α and β-leaves, we can define signed-leaves.

Definition 5.2 (signed-leaf)**.** Let $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ be a signed hypergraph. We say that $x \in [n]$ is a *signed-leaf* if there exists a hyperedge $U \in \mathcal{E}^+$, called the pivot of x, such that:

1. (α-property) $K \subseteq U$ for every $K \in \mathcal{E}^+$ that contains x; and
2. (β-property) the multiset $\{N \in \mathcal{E}^- \mid x \in N, N \not\subseteq U\} \cup \{U\}$ can be linearly ordered by $\subseteq$ with U being the minimal element.

The notion of a signed-leaf degenerates to an α-leaf when $\mathcal{E}^- = \emptyset$ (since the β-property holds trivially) and reduces to a β-leaf when $\mathcal{E}^+$ contains only singleton hyperedges (since the α-property holds trivially and the pivot is a singleton set). Recall that every α-acyclic hypergraph has an α-leaf and every β-acyclic hypergraph has a β-leaf.

We introduce some useful notations for our proofs. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E}^+, \mathcal{E}^-)$ be a signed hypergraph. We define $[\mathcal{H}]$ as the hypergraph obtained by iteratively removing a hyperedge $K \in \mathcal{E}^+ \cup \mathcal{E}^-$ if there exists some hyperedge $U \in \mathcal{E}^+$ such that $K \subseteq U$. We say a signed hypergraph $\mathcal{H}$ is *reduced* if $\mathcal{H} = [\mathcal{H}]$. For a multiset of hyperedges $\mathcal{E}$, we define $\mathcal{E}[\backslash x] := \{K \setminus \{x\} \mid K \in \mathcal{E}\}$. We define $\mathcal{H}[\backslash x]$ as the hypergraph obtained by removing x from every hyperedge in $\mathcal{H}$.

It is worth noting that in [Bra13], $x \in [n]$ is called a *bicolor-leaf* in $\mathcal{H}$ if x is a β-leaf in $[\mathcal{H}]$. This notion of bicolor-leaf is equivalent to our definition of signed-leaf (Definition 5.2). We establish the equivalence in the following.

Lemma 5.1. *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E}^+, \mathcal{E}^-)$ be a signed-acyclic signed hypergraph. Then, x is a signed-leaf of $\mathcal{H}$ if and only if x is a β-leaf of $[\mathcal{H}]$.*

Proof. Let $[\mathcal{H}] = (\mathcal{V}, \mathcal{E}'_+, \mathcal{E}'_-)$.

$\boxed{\Longrightarrow}$ Assume that x is a signed-leaf in $\mathcal{H}$. Let U be the pivot edge of x in $\mathcal{H}$ and assume that $U \subseteq R_1 \subseteq R_2 \subseteq \cdots \subseteq R_n$ such that $R_i \in \mathcal{E}^-$ for $i \in [n]$. Then $U, R_1, R_2, \ldots, R_n \in \mathcal{E}'_-$. Let K be an arbitrary hyperedge in $\mathcal{E}^+ \cup \mathcal{E}^- \setminus \{U, R_1, R_2, \ldots, R_n\}$ that contains x. Then by definition of $[\mathcal{H}]$, we must have that $K \subseteq U$, and therefore K is not present in $[\mathcal{H}]$. Therefore, $U, R_1, R_2, \ldots, R_n$ are all hyperedges in $[\mathcal{H}]$ that contain x. Hence x is a β-leaf of $[\mathcal{H}]$.

$\boxed{\Longleftarrow}$ Assume that x is a β-leaf in $[\mathcal{H}]$. Let $R_1, R_2, \ldots, R_n$ be all hyperedges in $[\mathcal{H}]$ that contain x, and assume that and $R_i \subseteq R_{i+1}$ for $i \in [n-1]$.

We argue that $\{R_1\} = \mathcal{E}^+ \cap \mathcal{E}'_+$, i.e. $R_1 \in \mathcal{E}'_+$ and for every $i \in \{2, 3, \ldots, n-1\}$, $R_i \in \mathcal{E}'_-$. Let i be the smallest index such that $R_i \in \mathcal{E}'_+$. Such an i must exist, since otherwise otherwise we have $x \in \bigcup_{K \in \mathcal{E}'_-} K$ but $x \notin \bigcup_{K \in \mathcal{E}'_+} K$, a contradiction to $\bigcup_{K \in \mathcal{E}'_-} K \subseteq \bigcup_{K \in \mathcal{E}'_+} K$. If $i > 1$, since $R_1 \subseteq R_i$, then R_1 should have been removed from $[\mathcal{H}]$, a contradiction. Finally, if there is any $j \in \{2, 3, \ldots, n-1\}$ such that $R_j \in \mathcal{E}'_+$, since $R_1 \subseteq R_j$, R_1 would have been removed from $[\mathcal{H}]$, a contradiction.

Let $U = R_1$. We show that U is a pivot edge in $\mathcal{H}$.

For (1), let R be an arbitrary hyperedge in $\mathcal{E}^+$ that contains x. Consider the maximal sequence of hyperedges $\{R^{(i)}\}_{i \geq 0}$ such that $R^{(0)} = R$, and for each $i \geq 0$, if $R^{(i)} \notin \mathcal{E}'_+$, then by construction of $[\mathcal{H}]$, let $R^{(i+1)}$ be the hyperedge in $\mathcal{E}^+$ such that $R^{(i)} \subset R^{(i+1)}$. Since the hypergraph is finite, this sequence must terminate and assume that its last hyperedge is $R^{(t)}$. We thus have that $R^{(t)} \in \mathcal{E}'_+$, and by previous argument, $R^{(t)} = R_1$. Therefore, $R = R^{(0)} \subset R^{(1)} \subset \cdots \subset R^{(t)} = R_1 = U$, as desired.

Consider (2). let N be an arbitrary hyperedge in $\mathcal{E}^-$ such that $x \in N$ and $N \not\subseteq U$. We show that $N \in \mathcal{E}'_-$. Indeed, if not, we must have that there exists some hyperedge $K \in \mathcal{E}^+$ such that $N \subseteq K$. Since $K \subseteq U$, we have $N \subseteq U$, a contradiction. Therefore, we have $\{N \in \mathcal{E}^- \mid x \in N, N \not\subseteq R_x\} \cup \{U\} = \{R_1, R_2, \ldots, R_n\}$, which is linearly ordered by $\subseteq$ by construction and $U = R_1$ is the minimal element, as desired. $\square$

The next proposition should not be surprising.

Proposition 5.1. *Every signed-acyclic signed hypergraph has a signed-leaf.*

Our proof for Proposition 5.1 is inspired from the proof of Theorem 7 in [BB16]. Some additional definitions are required. Two vertices x and y are said to be *non-neighbors* in a signed hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E}^+, \mathcal{E}^-)$ if there is no hyperedge $R \in \mathcal{E}^+ \cup \mathcal{E}^-$ such that $R \subset \mathcal{V}$ and $x, y \in R$. We need some additional structural properties on signed-acyclicity, Lemma 5.2 through Lemma 5.4.

Lemma 5.2. *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be an α-acyclic hypergraph. Then we have*

1. *the hypergraph $\mathcal{H}[\backslash x]$ is α-acyclic for every vertex $x \in \mathcal{V}$; and*
2. *the hypergraph $\mathcal{H}' = (\mathcal{V}, \mathcal{E} \setminus \{R\})$ is α-acyclic for every two distinct hyperedges R and S in $\mathcal{H}$ such that $R \subseteq S$.*

Proof. Let $\mathcal{T}$ be the join tree of $\mathcal{H}$ witnessed by the bijection mapping $\chi : \mathcal{E} \to V(\mathcal{T})$. Consider two items.

(1) We have that $\mathcal{H}[\backslash x] = (\mathcal{V} \setminus \{x\}, \mathcal{E}[\backslash x])$. Consider the new mapping $\chi' : \mathcal{E}[\backslash x] \to V(\mathcal{T})$ such that $\chi'(K \setminus \{x\}) = \chi(K)$ for every $K \in \mathcal{E}$. It is easy to verify that for every vertex $v \in \mathcal{V} \setminus \{x\}$, the set of nodes $\{\chi'(K \setminus \{x\}) \mid K \setminus \{x\} \in \mathcal{E}[\backslash x]\}$ induces a connected component in $\mathcal{T}$ since the set of nodes $\{\chi(K) \mid K \in \mathcal{E}\}$ does.

(2) Let R and S be two distinct hyperedges in $\mathcal{H}$ such that $R \subseteq S$. Let $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$ be the connected components of $\mathcal{T} - \chi(R)$, the tree obtained by removing the node $\chi(R)$ and all edges incident to $\chi(R)$ from $\mathcal{T}$. Assume that $\chi(R)$ is adjacent to $\chi(A_i)$ in $\mathcal{T}_i$ for each $i \in [k]$ and without loss of generality that $\chi(S)$ is contained in $\mathcal{T}_k$. Consider the tree $\mathcal{T}'$ obtained by removing $\chi(R)$ from $\mathcal{T}$ and adding an edge between every $\chi(A_i)$ to $\chi(S)$ for each $i \in [k-1]$.

We argue that $\mathcal{T}'$ is a join tree of $\mathcal{H}'$. Consider any vertex x in $\mathcal{H}'$ and let $K_x = \{\chi(K) \mid x \in K, K \in \mathcal{E} \setminus \{R\}\}$. The goal is to show that K_x induces a connected component in $\mathcal{T}'$.

Consider $I = \{i \in [k] \mid V(\mathcal{T}_i) \cap K_x \neq \emptyset\}$, the indices of all subtrees $\mathcal{T}_i$ that contains some hyperedge containing x. If $I = \{i\}$ for some index $i \in [k]$, then K_x is properly contained in $\mathcal{T}_i$, and since $\mathcal{T}_i$ is a join tree, $\mathcal{T}_x$ induces a connected component in $\mathcal{T}_i$ and thus $\mathcal{T}'$. Otherwise, $|I| \geq 2$. In this case, we must have $A_i \in K_x$ for every $i \in I$ and thus $x \in R \subseteq S$. Therefore, $K_x \setminus V(\mathcal{T}_k)$ induces a connected component in $\mathcal{T}'$. By construction, every $\chi(A_i)$ is connected to $\chi(S)$ in $\mathcal{T}'$ and $\chi(S) \in K_x$. Note that $K_x \cap V(\mathcal{T}_k)$ also induces a connected component in $\mathcal{T}_k$ and $\chi(S) \in K_x \cap V(\mathcal{T}_k)$, K_x induces a connected component in $\mathcal{T}'$. □

Lemma 5.3. *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E}^+, \mathcal{E}^-)$ be a reduced signed-acyclic signed hypergraph with at least two vertices. Then $\mathcal{H}$ contains two non-neighbor β-leaves (and therefore signed-leaves).*

Proof. We define the size of a signed hypergraph as the sum of the size of each hyperedge in it. We use an induction on the size k of the reduced signed-acyclic signed hypergraph $\mathcal{H}$.

Basis $k = 2$**.** In this case, we can only have $\mathcal{H} = (\{x, y\}, \{\{x\}, \{y\}\}, \emptyset)$, $\mathcal{H} = (\{x, y\}, \{\{x\}, \{y\}\}, \{\{x, y\}\})$ or $\mathcal{H} = (\{x, y\}, \{\{x, y\}\}, \emptyset)$ and the claim follows.

Inductive step. Assume that the claim holds for any reduced signed-acyclic signed hypergraph with size between 2 and $k - 1$. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E}^+, \mathcal{E}^-)$ be a reduced signed-acyclic signed hypergraph with size k.

Assume first that there is some hyperedge $R \in \mathcal{E}^+ \cup \mathcal{E}^-$ in $\mathcal{H}$ such that $R = \mathcal{V}$. If $R \in \mathcal{E}^+$, since $\mathcal{H}$ is reduced, we must have $\mathcal{H} = (\mathcal{V}, \{R\}, \emptyset)$, and the lemma follows since pair of vertices in $\mathcal{H}$ are non-neighbor β-leaves. If $R \in \mathcal{E}^-$, consider $\mathcal{H}' = (\mathcal{V}, \mathcal{E}^+, \mathcal{E}^- \setminus \{R\})$. Note that $\mathcal{H}'$ is also signed-acyclic, reduced and of smaller size. Thus by the inductive hypothesis, $\mathcal{H}'$ contains two non-neighbor β-leaves x and y. Then x and y are also β-leaves in $\mathcal{H}$, since $K \subseteq \mathcal{V} = R$ for every hyperedge K in $\mathcal{H}$. The vertices x and y are non-neighbors, because if not, there exists some hyperedge K in $\mathcal{H}$ such that $x, y \in K$ and $K \subset R$, and thus K is in $\mathcal{H}'$, a contradiction to that x and y are non-neighbors in $\mathcal{H}'$

In what follows, we assume that $\mathcal{H}$ contains no hyperedge R with $R = \mathcal{V}$ and has at least 3 vertices (by the inductive step).

Claim 5.1. If x is a α-leaf in a reduced signed hypergraph $\mathcal{H}$, then there exists a β-leaf y in $\mathcal{H}$ such that x and y are non-neighbors in $\mathcal{H}$.

Proof. Assume that x is an α-leaf in $\mathcal{H}$. Then there exists a hyperedge $R_x \in \mathcal{E}^+ \cup \mathcal{E}^-$ such that for any hyperedge $S \in \mathcal{E}^+ \cup \mathcal{E}^-$ that contains x, $S \subseteq R_x$. Note that $R_x \neq \mathcal{V}$.

Consider the signed hypergraph $\mathcal{H}' = [\mathcal{H}[\backslash x]]$, i.e., $\mathcal{H}'$ is obtained by first removing x from $\mathcal{H}$ and then taking its reduced hypergraph.

By Proposition 5.3, $\mathcal{H}'$ is signed-acyclic with size less than k. By the inductive hypothesis, $\mathcal{H}'$ contains two non-neighbor β-leaves y and z. Since y and x are non-neighbors and $R_x \neq \mathcal{V}$, either y or z is not contained in $R_x \setminus \{x\}$ and we assume that $y \notin R_x \setminus \{x\}$.

We argue that there is no hyperedge S in $\mathcal{H}$ that can contain both vertices x and y. Indeed, if not, assume that S contains both x and y in $\mathcal{H}$. By definition of R_x, we have that $S \subseteq R_x$, and we would have $y \in R_x$, a contradiction to our choice of y.

Let $R_1, R_2, \ldots, R_n$ be all hyperedges in $\mathcal{H}$ such that $y \in R_i$ for each $i \in [n]$. We argue that every R_i also appear in $\mathcal{H}' = [\mathcal{H}[\backslash x]]$. First, every R_i appears in $\mathcal{H}' = \mathcal{H}[\backslash x]$ since we argued that there is no hyperedge in $\mathcal{H}$ that contains both x and y. Suppose for contradiction that some hyperedge R_i is not contained in $[\mathcal{H}[\backslash x]]$. Then there exists some hyperedge $K \in \mathcal{E}^+$ such that $R_i \subseteq K \setminus \{x\} \subseteq K$, a contradiction that $\mathcal{H}$ is reduced.

By the inductive hypothesis, y is a β-leaf in $\mathcal{H}' = [\mathcal{H}[\backslash x]]$, and thus y is a β-leaf in $\mathcal{H}$ since the chain $R_1, R_2, \ldots, R_n$ in $\mathcal{H}'$ remains in $\mathcal{H}$. $\square$

Since $\mathcal{H}$ is signed-acyclic, $\mathcal{H}$ is α-acyclic and thus contains an α-leaf x. By Claim 5.1, there is a β-leaf y in $\mathcal{H}$ that is non-neighbor with x. Then y is also an α-leaf in $\mathcal{H}$, and by Claim 5.1 again, there is a β-leaf z in $\mathcal{H}$ that is non-neighbor with y. Hence y and z are non-neighbor β-leaves in $\mathcal{H}$, as desired.

This concludes the proof. □

Lemma 5.4. *Every signed-acyclic signed hypergraph with at least two-vertices have two non-neighbor signed-leaves.*

Proof. When $\mathcal{H}$ contains at least two vertices, by Lemma 5.2 and 5.3, $[\mathcal{H}]$ is reduced, signed-acyclic and contains two non-neighbor β-leaves x and y. By Lemma 5.1, x and y are both signed-leaves of $\mathcal{H}$. □

Finally we come back to the proof of Proposition 5.1.

Proof of Proposition 5.1. Let $\mathcal{H}$ be a signed-acyclic signed hypergraph. If $\mathcal{H}$ contains only one vertex x, then it is only possible that $\mathcal{H} = (\{x\}, \{\{x\}\}, \mathcal{E}^-)$ where $\mathcal{E}^- \subseteq \{\{x\}\}$, and in all cases x is a signed-leaf of $\mathcal{H}$. If $\mathcal{H}$ contains at least two vertices, the claim follows by Lemma 5.4. □

5.3.2 Elimination Sequences

Next, we generalize the notion of an elimination sequence [AKNR16] to signed hypergraphs. Given $\mathcal{H}$ and a signed-leaf v with a pivot U, we define $\langle \mathcal{H}, v \rangle$ to be the hypergraph that (i) removes from $\mathcal{H}$ any (positive or negative) hyperedge V that contains v such that $V \subsetneq U$, and (ii) removes v from any hyperedge that contains it.

Definition 5.3 (signed-elimination sequence)**.** Let $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ be a signed hypergraph. A vertex ordering $\sigma = (v_1, v_2, \ldots, v_n)$ of all vertices in $[n]$ is a signed-elimination sequence of $\mathcal{H}$ if v_i is a signed-leaf of $\mathcal{H}_i$ for every $i \in [n]$, where $\mathcal{H}_n = \mathcal{H}$, and $\mathcal{H}_j = \langle \mathcal{H}_{j+1}, v_{j+1} \rangle$ for $j = n-1, n-2, \ldots, 1$.

For a vertex ordering $\sigma = (v_1, v_2, \ldots, v_{n-1}, v_n)$, we often denote $\sigma = \sigma' \cdot v_n$ where $\sigma' = (v_1, v_2, \ldots, v_{n-1})$. By definition, if $\sigma = \sigma' \cdot v$ is a signed-elimination sequence of a signed hypergraph $\mathcal{H}$, then σ' is a signed-elimination sequence of $\langle \mathcal{H}, v \rangle$.

Proposition 5.2. *Let $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ be a signed hypergraph. If $\mathcal{H}$ is signed-acyclic, then*

1. *$\mathcal{H}$ has a signed-elimination sequence; and*
2. *for every hyperedge $\{u_1, u_2, \ldots, u_k\} \in \mathcal{E}^-$, $\mathcal{H}$ has a signed-elimination sequence of the form $(u_1, u_2, \ldots, u_k, v_{k+1}, \ldots, v_n)$.*

We state the following before proving Proposition 5.2.

Corollary 5.1. *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E}^+, \mathcal{E}^-)$ be a signed-acyclic signed hypergraph. Then we have*

1. *the hypergraph $\mathcal{H}[\backslash x]$ is signed-acyclic, for every vertex $x \in \mathcal{V}$; and*
2. *let $R \in \mathcal{E}^+ \cup \mathcal{E}^-$ and $S \in \mathcal{E}^+$ be two distinct hyperedges with $R \subseteq S$, and we have $\mathcal{H}' = (\mathcal{V}, \mathcal{E}^+ \setminus \{R\}, \mathcal{E}^- \setminus \{R\})$ is signed-acyclic.*

Proof. We again consider two items.

(1) Let $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$ and we have that $\mathcal{H}[\backslash x] = (\mathcal{V} \setminus \{x\}, \mathcal{E}^+[\backslash x], \mathcal{E}^-[\backslash x])$. Let $\mathcal{E}'_x \subseteq \mathcal{E}^-[\backslash x]$ and consider the hypergraph $\mathcal{H}'_x = (\mathcal{V} \setminus \{x\}, \mathcal{E}^+[\backslash x] \cup \mathcal{E}'_x)$. By construction, we must have some $\mathcal{E}' \subseteq \mathcal{E}^-$ such that $\mathcal{E}'_x = \mathcal{E}^-[\backslash x]$. Since $\mathcal{H}$ is signed-acyclic, we have that the hypergraph $\mathcal{H}' = (\mathcal{V}, \mathcal{E}^+ \cup \mathcal{E}')$ is α-acyclic. Then $\mathcal{H}'[\backslash x] = (\mathcal{V} \setminus \{x\}, \mathcal{E}^+[\backslash x] \cup \mathcal{E}'[\backslash x]) = (\mathcal{V} \setminus \{x\}, \mathcal{E}^+[\backslash x] \cup \mathcal{E}'_x) = \mathcal{H}'_x$ is also α-acyclic by Lemma 5.2, as desired.

(2) The claim is straightforward if $R \in \mathcal{E}^-$. Assume that $R \in \mathcal{E}^+$. Consider any hypergraph $\mathcal{H}' = (\mathcal{V}, (\mathcal{E}^+ \cup \mathcal{E}') \setminus \{R\})$ where $\mathcal{E}' \subseteq \mathcal{E}^-$. Note that the hypergraph $\mathcal{H}'' = (\mathcal{V}, \mathcal{E}^+ \cup \mathcal{E}')$ is α-acyclic since $\mathcal{H}$ is signed-acyclic. Since $\mathcal{H}''$ is α-acyclic, $R \subseteq S$ and $S \in \mathcal{E}^+$, by Lemma 5.2, $\mathcal{H}'$ is α-acyclic, as desired. □

Proposition 5.3. *If $\mathcal{H}$ is a signed-acyclic signed hypergraph and x is a signed-leaf of $\mathcal{H}$, then $\langle \mathcal{H}, x \rangle$ is signed-acyclic.*

Proof. Immediate from Corollary 5.1. □

Now we are ready to prove Proposition 5.2.

Proof of Proposition 5.2. Let $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ be a signed-acyclic signed hypergraph. By Proposition 5.3 and 5.1, $\mathcal{H}$ always has a signed-elimination sequence, and (1) follows.

For (2), let $R = \{u_1, u_2, \ldots, u_k\} \in \mathcal{E}^-$. The claim follows if $R = [n]$. Assume that $\mathcal{H}$ contains at least two vertices and $R \subset [n]$.

Construct the following sequence of hypergraphs $\mathcal{H}_n$, $\mathcal{H}_{n-1}$, $\ldots$, $\mathcal{H}_k$, where

- $\mathcal{H}_n = \mathcal{H}$;
- for every $j = n, n-1, \ldots, k+1$, let v_j be a signed-leaf in $\mathcal{H}_j$ such that $v_j \notin R$, and let $\mathcal{H}_{j-1} = \langle \mathcal{H}_j, v_j \rangle$.

We argue that in the second step, such a v_j must exist. For $j = n$, since $R \subset [n]$ and $\mathcal{H}_n$ is signed-acyclic, by Lemma 5.4, $\mathcal{H}_n$ contains two non-neighbor signed-leaves, and there must be a

signed-leaf v_n of $\mathcal{H}_n$ such that $v_n \notin R$. Hence, $R \subset [n] \setminus \{v_n\}$ and R remain in $\langle \mathcal{H}_n, v_n \rangle$ by definition. This argument can thus continue inductively.

Note that R would contain every vertex in $\mathcal{H}_k$ since $\mathcal{H}_k$ has exactly k vertices. Then $\mathcal{H}_k$ would admit a signed-elimination sequence $(u_1, u_2, \ldots, u_k)$, and thus $\mathcal{H} = \mathcal{H}_n$ admits a signed elimination sequence $(u_1, u_2, \ldots, u_k, v_{k+1}, \ldots, v_n)$ as desired. □

Example 5.1. Consider the following $\mathsf{CQ}^{\neg}$, which will function as our running example.

$$Q(x_1, x_2, x_3, x_4) \leftarrow A(x_1, x_2, x_3) \wedge U(x_3, x_4) \wedge \neg V(x_4) \wedge \neg R(x_2, x_3, x_4) \wedge \neg S(x_1, x_2, x_3, x_4)$$

The vertex 4 is a signed-leaf for the hypergraph $\mathcal{H}$ of Q with pivot the hyperedge $\{3, 4\}$ that corresponds to the atom U. The query corresponding to the resulting hypergraph $\langle \mathcal{H}, 4 \rangle$ is:

$$Q'(x_1, x_2, x_3) = A(x_1, x_2, x_3) \wedge U(x_3) \wedge \neg R(x_2, x_3) \wedge \neg S(x_1, x_2, x_3)$$

In fact, Q is signed-acyclic with the signed-elimination sequence $\sigma = (1, 2, 3, 4)$.

5.4 Enumeration of Full $\mathsf{CQ}^{\neg}$

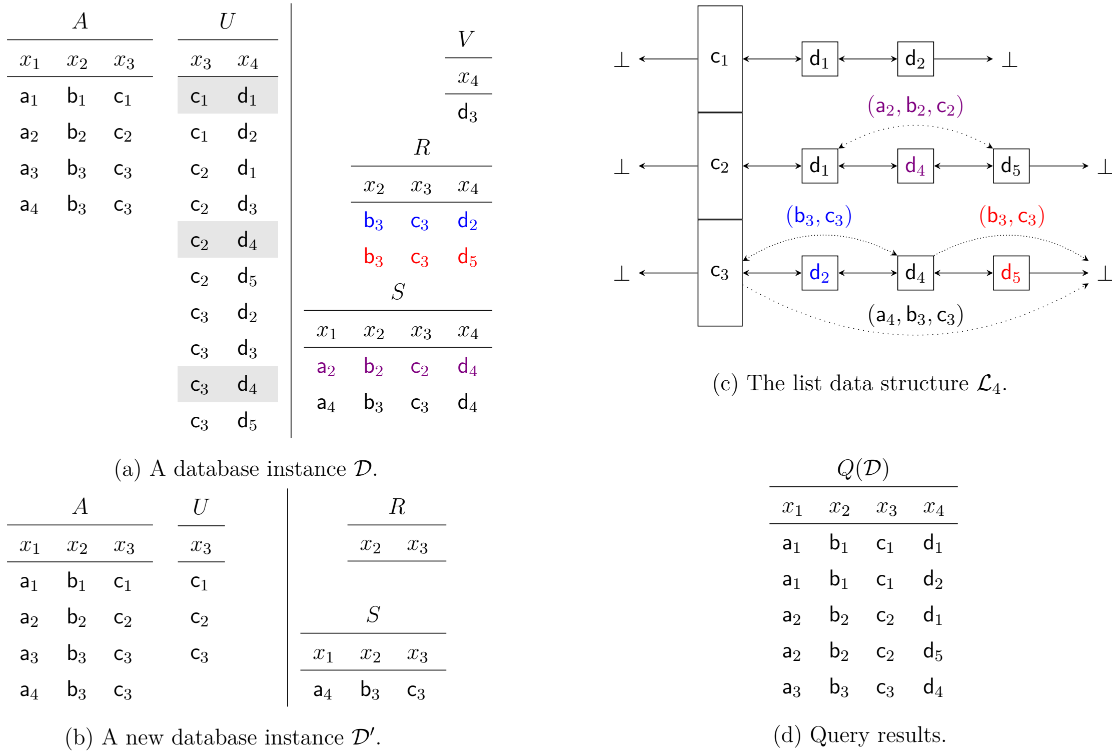
In this section, we present an algorithm that can enumerate the answers of a signed-acyclic full $\mathsf{CQ}^{\neg}$ (where $F = [n]$) with constant delay after linear preprocessing time.

Let Q be a full signed-acyclic $\mathsf{CQ}^{\neg}$ with a signed-acyclic signed hypergraph $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ and $\mathcal{D}$ a database instance. Let $\sigma = \sigma' \cdot v$ be a signed-elimination sequence of $\mathcal{H}$.

Our preprocessing phase recursively eliminates a variable v in the signed-elimination sequence. When eliminating v, the key idea is to construct in linear time a database $\mathcal{D}'$, and reduce the problem of computing the answers of $Q(\mathcal{D})$ to computing the answers of $Q'(\mathcal{D}')$, where the signed hypergraph of Q' is $\mathcal{H}' = \langle \mathcal{H}, v \rangle$ and has a signed-elimination sequence σ'. The reduction needs to ensure that $\Pi_{[n]\setminus\{v\}} Q(\mathcal{D}) = Q'(\mathcal{D}')$.[2] To achieve constant delay enumeration, we construct (also in linear time) a data structure over the domain of the variable x_v, that can, given an answer to $Q'(\mathcal{D}')$, extend it to an answer to $Q(\mathcal{D})$ with constant delay.

Example 5.2. We continue the query Q in Example 5.1, the database $\mathcal{D}$ in Figure 5.1(a) and a signed-elimination sequence $\sigma = (1, 2, 3, 4)$. The result $Q(\mathcal{D})$ is depicted in Figure 5.1(d). After eliminating x_4, we obtain Q' and the instance $\mathcal{D}'$ in Figure 5.1(b). We have $Q'(\mathcal{D}') = \{(\mathsf{a}_1, \mathsf{b}_1, \mathsf{c}_1), (\mathsf{a}_2, \mathsf{b}_2, \mathsf{c}_2), (\mathsf{a}_3, \mathsf{b}_3, \mathsf{c}_3)\}$. The preprocessing phase reduces computing $Q(\mathcal{D})$ to $Q'(\mathcal{D}')$ and produces a data structure as in Figure 5.1(c), such that given any tuple $\mathbf{a}' \in Q'(\mathcal{D}')$, we may use the data structure to enumerate all answers of the form $(\mathbf{a}', a_4)$ in $Q(\mathcal{D})$, a_4 being a value of x_4.

[2] For a tuple $\mathbf{a}_N$ and $X \subseteq N$, we denote $\Pi_X \mathbf{a}_N$ as the projection of $\mathbf{a}_N$ on X.



A

x_1	x_2	x_3
a$_1$	b$_1$	c$_1$
a$_2$	b$_2$	c$_2$
a$_3$	b$_3$	c$_3$
a$_4$	b$_3$	c$_3$

U

x_3	x_4
c$_1$	d$_1$
c$_1$	d$_2$
c$_2$	d$_1$
c$_2$	d$_3$
c$_2$	d$_4$
c$_2$	d$_5$
c$_3$	d$_2$
c$_3$	d$_3$
c$_3$	d$_4$
c$_3$	d$_5$

V

x_4
d$_3$

R

x_2	x_3	x_4
b$_3$	c$_3$	d$_2$
b$_3$	c$_3$	d$_5$

S

x_1	x_2	x_3	x_4
a$_2$	b$_2$	c$_2$	d$_4$
a$_4$	b$_3$	c$_3$	d$_4$

(a) A database instance $\mathcal{D}$.

(c) The list data structure $\mathcal{L}_4$.

A

x_1	x_2	x_3
a$_1$	b$_1$	c$_1$
a$_2$	b$_2$	c$_2$
a$_3$	b$_3$	c$_3$
a$_4$	b$_3$	c$_3$

U

x_3
c$_1$
c$_2$
c$_3$

R

x_2	x_3

S

x_1	x_2	x_3
a$_4$	b$_3$	c$_3$

(b) A new database instance $\mathcal{D}'$.

$Q(\mathcal{D})$

x_1	x_2	x_3	x_4
a$_1$	b$_1$	c$_1$	d$_1$
a$_1$	b$_1$	c$_1$	d$_2$
a$_2$	b$_2$	c$_2$	d$_1$
a$_2$	b$_2$	c$_2$	d$_5$
a$_3$	b$_3$	c$_3$	d$_4$

(d) Query results.

Figure 5.1: Given the query Q in Example 5.2, the database instance in (a) and the signed-elimination sequence $\sigma = (1, 2, 3, 4)$ as inputs, Algorithm 4 first produces the data structure $\mathcal{L}_4$ in (c), and then constructs a new query Q' in Example 5.2, a new database as in (b), and the sequence $\sigma' = (1, 2, 3)$ as inputs to the recursive call.

5.4.1 Preprocessing phase

Algorithm 4 describes the preprocessing phase that takes as input the signed hypergraph $\mathcal{H}$, a database instance $\mathcal{D}$, and a signed elimination sequence σ of $\mathcal{H}$. Let U be a pivot hyperedge of the signed-leaf v in $\mathcal{H}$ and $N_0 = \{v\} \subseteq U \subseteq N_1 \subseteq \cdots \subseteq N_m$ the linear order of the negative hyperedges that contain v. The preprocessing phase outputs a data structure $\mathcal{L}_v$ that maps tuples of the form $\Pi_{U\setminus\{v\}}\mathbf{a}_U$ to a doubly linked list of elements from the domain of x_v, with some extra tuple-labeled skipping links that we will introduce later. We explain how the algorithm eliminates the variable x_v in two steps: α-step and β-step.

Algorithm 4: PreprocessFullCQ($\mathcal{H}, \mathcal{D}, \sigma$)

Input: signed hypergraph $\mathcal{H}$, instance $\mathcal{D}$, signed-elimination sequence $\sigma = \sigma' \cdot v$

Construct: $\mathcal{L}_v$ for every vertex v in σ

```
1  if σ is empty then
2      terminate
3  U ← a pivot hyperedge for the signed-leaf v in H
4  foreach K ∈ E⁺ ∪ E⁻ such that v ∈ K ⊆ U do                       ▷ α-step
5      if K ∈ E⁺ then
6          R_U ← R_U ⋉ R_K
7      else
8          R_U ← R_U \ (R_U ⋉ R_K)
9      remove relation R_K from D
10 L_v ← BuildList(v, U)
11 replace R_U with R_{U\{v}} ← Π_{U\{v}} R_U
12 Let N_1, ..., N_m ∈ E⁻ s.t. U ⊆ N_1 ⊆ ··· ⊆ N_m                 ▷ β-step
13 foreach i = 1, 2, ..., m do
14     ExtendList(L_v, N_i)
15     replace R_{N_i} with R_{N_i\{v}} that contains all a_{N_i\{v}} ∈ Π_{N_i\{v}} R_{N_i} such that
           Π_{U\{v}} a_{N_i\{v}} ∈ L_v
16         and L_v[Π_{U\{v}} a_{N_i\{v}}].nextM(a_{N_i\{v}}) = ⊥
17 PreprocessFullCQ(⟨H, v⟩, D, σ')
```

α-step.

The algorithm first performs an α-step (that mimics Yannakakis' algorithm) in which we simply remove tuples from R_U that will not contribute to a query answer of $Q(\mathcal{D})$ by filtering R_U with any positive (or negative) atom R_K with $v \in K$ and $K \subseteq U$.

Then, Algorithm 5 builds a hash table $\mathcal{L}_v$ that maps each tuple $\mathbf{a}_{U\setminus\{v\}} \in \Pi_{U\setminus\{v\}} R_U$ to a doubly linked list of the set $\{a_v \mid (\mathbf{a}_{U\setminus\{v\}}, a_v) \in R_U\}$, with a slight modification that every pointer is now parameterized/labeled with $\emptyset$.

Algorithm 5: BuildList(v, U)

Input: $U \in \mathcal{E}^+, v \in U$

Output: a data structure $\mathcal{L}_v$

```
1  L_v ← an empty hashtable (returns ⊥ always)
2  foreach a_U ∈ R_U do
3      if L_v[Π_{U\{v}} a_U] = ⊥ then
4          newHead.prev[∅] ← ⊥
5          newHead.next[∅] ← ⊥
6          L_v[Π_{U\{v}} a_U] ← newHead
7      head ← L_v[Π_{U\{v}} a_U]
8      Node.val ← Π_{{v}} a_U
9      Node.prev[∅] ← head
10     Node.next[∅] ← head.next[∅]
11     (head.next[∅]).prev[∅] ← Node
12 return L_v
```

The extra label on the pointer allows us to add pointers with different labels in later steps. If Q contains no negative atoms, this data structure is sufficient to achieve constant delay enumeration by traversing the list from the head to the end. Finally, R_U is replaced with its projection $\Pi_{U\setminus\{v\}} R_U$.

Example 5.3. For our running example $U(x_3, x_4)$ is chosen as the pivot, and the tuples highlighted in gray in Figure 5.1(a) denotes the tuples in U that survive the semijoin step with the negative atom $\neg V(x_4)$. The linked lists of the hash table correspond to the data structure in Figure 5.1(c) if we ignore the dotted edges.

Algorithm 6: $\mathsf{ExtendList}(\mathcal{L}_v, N_i)$

Input: a data structure $\mathcal{L}_v$, $N_i \in \mathcal{E}^-$

Global variables: $U \in \mathcal{E}^+$, $N_1, N_2 \dots, N_i \in \mathcal{E}^-$ s.t. $v \in U$, $U \subseteq N_1 \subseteq N_2 \subseteq \cdots \subseteq N_i$

1 **foreach** $\mathbf{a}_{N_i} \in R_{N_i}$ **do**
2   **if** $\Pi_U \mathbf{a}_{N_i} \notin R_U$ *or* $\exists j \in \{1, 2, \dots, i-1\}$ *such that* $\Pi_{N_j} \mathbf{a}_{N_i} \in R_{N_j}$ **then**
3   **continue**
4     $\mathsf{currNode} \leftarrow v \in \mathcal{L}_v[\Pi_{U \setminus \{v\}} \mathbf{a}_{N_i}]$ with $v.\mathsf{val} = \Pi_{\{v\}} \mathbf{a}_{N_i}$
5     $\mathsf{prevNode} \leftarrow \mathsf{currNode.prevM}(\mathbf{a}_{N_i \setminus \{v\}})$
6     $\mathsf{nextNode} \leftarrow \mathsf{currNode.nextM}(\mathbf{a}_{N_i \setminus \{v\}})$
7     $\mathsf{prevNode.next}[\mathbf{a}_{N_i \setminus \{v\}}] \leftarrow \mathsf{nextNode}$
8   **if** $\mathsf{nextNode} \neq \perp$ **then**
9         $\mathsf{nextNode.prev}[\mathbf{a}_{N_i \setminus \{v\}}] \leftarrow \mathsf{prevNode}$

β-step.

Assume $\mathbf{a} \in Q^+(\mathcal{D})$, where $Q^+(\mathbf{x}_{[n]}) \leftarrow \bigwedge_{K \in \mathcal{E}^+} R_K(\mathbf{x}_K)$. When negative atoms are present, for $\mathbf{a}$ to be an answer to $Q(\mathcal{D})$, we also need that $\Pi_{N_i}\mathbf{a} \notin R_{N_i}$ for every $i \in [m]$. Therefore, we need to augment $\mathcal{L}_v$ such that we can skip the enumeration of any value in the linked list that does not contribute to the answer. The linear order on the schemas of negative atoms allows us to construct this data structure in a dynamic fashion. If only one negative atom $\neg R_{N_1}$ is present (i.e., $m = 1$), for each tuple $\mathbf{a}_{N_1} \in R_{N_1}$ we can add a *skipping link* labeled with $\Pi_{N_1 \setminus \{v\}} \mathbf{a}_{N_1}$ that bypasses the node with value $\Pi_{\{v\}}\mathbf{a}_{N_1}$ in the doubly linked list $\mathcal{L}_v[\Pi_{U \setminus \{v\}} \mathbf{a}_{N_1}]$. If $m = 2$, then there is another negative atom $\neg R_{N_2}$ with $N_1 \subseteq N_2$. For each tuple $\mathbf{a}_{N_2} \in R_{N_2}$, we only need to add a skipping link labeled with $\Pi_{N_2 \setminus \{v\}} \mathbf{a}_{N_2}$ that bypasses the node with value $\Pi_{\{v\}}\mathbf{a}_{N_2}$ in the doubly linked list $\mathcal{L}_v[\Pi_{U \setminus \{v\}} \mathbf{a}_{N_2}]$ if that node has not been bypassed by a skipping link with label $\Pi_{N_1 \setminus \{v\}} \mathbf{a}_{N_2}$ yet (i.e., $\Pi_{N_1}\mathbf{a}_{N_2} \notin R_{N_1}$). In general, for every tuple $\mathbf{a}_{N_i} \in R_{N_i}$ such that $\Pi_{\{v\}}\mathbf{a}_{N_i}$ has not been bypassed by any skipping link labeled with $\Pi_{N_j \setminus \{v\}} \mathbf{a}_{N_i}$ (i.e., for every $1 \leq j < i \leq m$, $\mathbf{a}_{N_j} \notin R_{N_j}$), we add a skipping link labeled with $\Pi_{N_i \setminus \{v\}} \mathbf{a}_{N_i}$ that bypasses $\Pi_{\{v\}}\mathbf{a}_{N_i}$, as well as every skipping links labeled with $\Pi_{N_j \setminus \{v\}} \mathbf{a}_{N_i}$ with $1 \leq j < i$ starting in $\Pi_{\{v\}}\mathbf{a}_{N_i}$.

Algorithm 6 implements this idea using the parameterized/labeled pointers to support the skipping links. For any node in the doubly linked list, its next node given a tuple $\mathbf{a}_W$ with $W \supseteq U \setminus \{v\}$ is fetched by following $\mathsf{next}[\Pi_{N_i \setminus \{v\}} \mathbf{a}_W]$ for the largest possible $i \in \{0, 1, \dots, m\}$ in that node. Recall that $N_0 = \{v\}$ and thus pointers $\mathsf{next}[\emptyset]$ and $\mathsf{prev}[\emptyset]$ are always present for each node. This operation

Algorithm 7: $\textsf{currNode.nextM}(\mathbf{a}_W)$

Input: a list node $\textsf{currNode}$ from $\mathcal{L}_v$, a tuple $\mathbf{a}_W$ with $W \supseteq U \setminus \{v\}$

Global variables of $\mathcal{L}_v$: $U \in \mathcal{E}^+$, $\{v\} = N_0 \subseteq N_1, N_2 \ldots, N_i \in \mathcal{E}^-$ s.t. $v \in U$, $U \subseteq N_1 \subseteq N_2 \subseteq \cdots \subseteq N_i$

Output: the next node of $\textsf{currNode}$ following $\mathbf{a}_W$

1 $\ell \leftarrow$ the largest in $\{0, 1, \ldots, m\}$ such that $\textsf{currNode.next}[\Pi_{N_i \setminus \{v\}} \mathbf{a}_W]$ is defined

2 **return** $\textsf{currNode.next}[\Pi_{N_\ell \setminus \{v\}} \mathbf{a}_W]$

Algorithm 8: $\textsf{currNode.prevM}(\mathbf{a}_W)$

Input: a list node $\textsf{currNode}$ from $\mathcal{L}_v$, a tuple $\mathbf{a}_W$ with $W \supseteq U \setminus \{v\}$

Global variables of $\mathcal{L}_v$: $U \in \mathcal{E}^+$, $\{v\} = N_0 \subseteq N_1, N_2 \ldots, N_i \in \mathcal{E}^-$ s.t. $v \in U$, $U \subseteq N_1 \subseteq N_2 \subseteq \cdots \subseteq N_i$

Output: the previous node of $\textsf{currNode}$ following $\mathbf{a}_W$

1 $\ell \leftarrow$ the largest in $\{0, 1, \ldots, m\}$ such that $\textsf{currNode.prev}[\Pi_{N_i \setminus \{v\}} \mathbf{a}_W]$ is defined

2 **return** $\textsf{currNode.prev}[\Pi_{N_\ell \setminus \{v\}} \mathbf{a}_W]$

is denoted as $\textsf{nextM}(\mathbf{a}_W)$, and we define $\textsf{prevM}(\mathbf{a}_W)$ similarly. The formal definitions for the operations $\textsf{currNode.nextM}(\mathbf{a}_W)$ and $\textsf{currNode.prevM}(\mathbf{a}_W)$ are defined in Algorithm 7 and 8, respectively.

To traverse $\mathcal{L}_v$, we implement an iterator (Algorithm 9) that takes a tuple $\mathbf{a}_W$ with $W \supseteq U \setminus \{v\}$, and then traverses the linked list $\mathcal{L}_v[\Pi_{U \setminus \{v\}} \mathbf{a}_W]$ using $\textsf{nextM}(\mathbf{a}_W)$.

Example 5.4. Algorithm 6 takes as input the basic linked list data structure and all tuples in the negative atoms $\neg R(x_2, x_3, x_4)$ and $\neg S(x_1, x_2, x_3, x_4)$ in Figure 5.1(a) to produce the data structure in Figure 5.1(c). Note that the skipping link bypassing d_4 with label $(\mathsf{a}_4, \mathsf{b}_3, \mathsf{c}_4)$ also needs to bypass both skipping links labeled by $(\mathsf{b}_3, \mathsf{c}_3)$ and $(\mathsf{b}_3, \mathsf{c}_3)$ that bypass d_2 and d_5 respectively.

Finally, for each negative atom $\neg R_{N_i}$, we need to keep precisely the subset of $\Pi_{N_i \setminus \{v\}} R_{N_i}$ so that we can *avoid* emitting an answer to $Q'(\mathcal{D}')$ that cannot be extended to an answer to $Q(\mathcal{D})$. This is done by keeping the tuple $\mathbf{a}_{N_i \setminus \{v\}} \in \Pi_{N_i \setminus \{v\}} R_{N_i}$ only if $\Pi_{U \setminus \{v\}} \mathbf{a}_{N_i \setminus \{v\}} \in \mathcal{L}_v$, but the tuple cannot be extended to any answer to $Q(\mathcal{D})$, i.e., the traversal on $\mathcal{L}_v$ using $\mathbf{a}_{N_i \setminus \{v\}}$ does not lead to any value of x_v.

Example 5.5. After removing x_4 from relations $A(x_1, x_2, x_3, x_4)$ and $U(x_3, x_4)$, we end up with relations $A(x_1, x_2, x_3)$ and $U(x_3)$ in Figure 5.1(b). The query containing only the positive atoms of Q' would return $\{(\mathsf{a}_1, \mathsf{b}_1, \mathsf{c}_1), (\mathsf{a}_2, \mathsf{b}_2, \mathsf{c}_2), (\mathsf{a}_3, \mathsf{b}_3, \mathsf{c}_3), (\mathsf{a}_4, \mathsf{b}_3, \mathsf{c}_3)\}$, but we wish to skip enumerating $(\mathsf{a}_4, \mathsf{b}_3, \mathsf{c}_3)$, since it cannot be extended to an answer to $Q(\mathcal{D})$. Indeed, the traversal on the linked list

Algorithm 9: $\mathcal{L}_v.\mathsf{Iterate}(\mathbf{a}_W)$

Input: a tuple $\mathbf{a}_W$ such that $(U \setminus \{v\}) \subseteq W$

1 $\mathsf{currNode} \leftarrow \mathcal{L}_v[\Pi_{U\setminus\{v\}}\mathbf{a}_W]$
2 **while** $\mathsf{currNode.nextM}(\Pi_{W\setminus\{v\}}\mathbf{a}_W) \neq \perp$ **do**
3 $\quad$ $\mathsf{currNode} \leftarrow \mathsf{currNode.nextM}(\Pi_{W\setminus\{v\}}\mathbf{a}_W)$
4 $\quad$ **emit** $\mathsf{currNode.val}$
5 **emit** $\perp$

$\mathcal{L}_4[\mathsf{c}_3]$ will simply follow the black skipping link $(\mathsf{a}_4, \mathsf{b}_3, \mathsf{c}_3)$ to reach the end of the list. We achieve this by adding $(\mathsf{a}_4, \mathsf{b}_3, \mathsf{c}_3)$ to $S(x_1, x_2, x_3)$ so that this tuple is not returned in the recursive call.

5.4.2 Enumeration phase

The enumeration phase is shown in Algorithm 10. It follows the reverse order of $\sigma = \sigma' \cdot v$ to first recursively enumerate a tuple $\mathbf{a}'_{\sigma'} \in Q'(\mathcal{D}')$, and then (with a slight abuse of notation), use every $a_v \in \mathcal{L}_v.\mathsf{Iterate}(\mathbf{a}'_{\sigma'})$ to obtain an answer $(\mathbf{a}'_{\sigma'}, a_v)$ to $Q(\mathcal{D})$. The preprocessing phase guarantees that the iterator is nonempty for every answer $\mathbf{a}'_{\sigma'}$.

5.4.3 A Comprehensive Example

This section demonstrates Algorithm 4 (pre-processing phase) and Algorithm 10 (enumeration phase) end-to-end on our running example.

Example 5.6. We illustrate on the running example query $Q(= Q_4)$ in Example 5.1, the database $\mathcal{D}(= \mathcal{D}_4)$ in Figure 5.2(a) and the signed-elimination sequence $\sigma = (1, 2, 3, 4)$.

$$Q_4(x_1, x_2, x_3, x_4) \leftarrow A(x_1, x_2, x_3) \wedge U(x_3, x_4) \wedge \neg V(x_4) \wedge \neg R(x_2, x_3, x_4) \wedge \neg S(x_1, x_2, x_3, x_4)$$

Preprocessing phase. We dive into the recursive steps of Algorithm 4 on input $(\mathcal{H}_4, \mathcal{D}_4, (1, 2, 3, 4))$:

1. First, we remove the signed leaf x_4 from $\mathcal{H}_4$, which yields the following query $Q_3(x_1, x_2, x_3)$ whose hypergraph corresponds to the hypergraph $\mathcal{H}_3 = \langle \mathcal{H}_4, x_4 \rangle$ and emits a skipping list data structure $\mathcal{L}_4$ as shown in Figure 5.2(e):

   $$Q_3(x_1, x_2, x_3) \leftarrow A(x_1, x_2, x_3) \wedge U(x_3) \wedge \neg R(x_2, x_3) \wedge \neg S(x_1, x_2, x_3).$$

   The database is changed to $\mathcal{D}_3$ as in Figure 5.2(b). This is a high-level summary of the previous examples where $\mathcal{D}_3 = \mathcal{D}'$ (Example 5.2 through Example 5.5). Next, we keep following the signed-elimination sequence $(1, 2, 3)$, i.e. on input $(\mathcal{H}_3, \mathcal{D}_3, (1, 2, 3))$.

Algorithm 10: $\mathsf{EnumerationFullCQ}(\sigma)$

Input: a signed-elimination sequence $\sigma = (v_1, v_2, \ldots, v_n)$

Output: an enumeration of the query answers $Q(\mathcal{D})$

```
1 foreach a_{v_1} ∈ L_{v_1}.Iterate(∅) do
2   foreach a_{v_2} ∈ L_{v_2}.Iterate((a_{v_1})) do
3     foreach a_{v_3} ∈ L_{v_3}.Iterate((a_{v_1}, a_{v_2})) do
4       ...
5       foreach a_{v_n} ∈ L_{v_n}.Iterate((a_{v_1}, ..., a_{v_{n-1}})) do
6         emit (a_{v_1}, a_{v_2}, ..., a_{v_n})
```

2. We remove the signed leaf x_3 from $\mathcal{H}_3$, yielding $Q_2(x_1, x_2)$ whose hypergraph corresponds to the hypergraph $\mathcal{H}_2 = \langle \mathcal{H}_3, x_3 \rangle$ and emits $\mathcal{L}_3$ as shown in Figure 5.2(f):

$$Q_2(x_1, x_2) \leftarrow A(x_1, x_2).$$

   Note that in this step, the positive atom $U(x_3)$ and all negated atoms $\neg R(x_2, x_3)$ and $\neg S(x_1, x_2, x_3)$ are removed from Q_3 since their corresponding negative hyperedges are contained by the positive hyperedge corresponding to $A(x_1, x_2, x_3)$. Further, the skipping list $\mathcal{L}_3$ does not contain any skipping edge. The database is changed to $\mathcal{D}_2$ as in Figure 5.2(c).

3. We further remove the signed leaf x_2 from $\mathcal{H}_2$, finally yielding $Q_1(x_1)$ with a hypergraph corresponding to $\mathcal{H}_1 = \langle \mathcal{H}_2, x_2 \rangle$ and emits $\mathcal{L}_2$ shown in Figure 5.2(g):

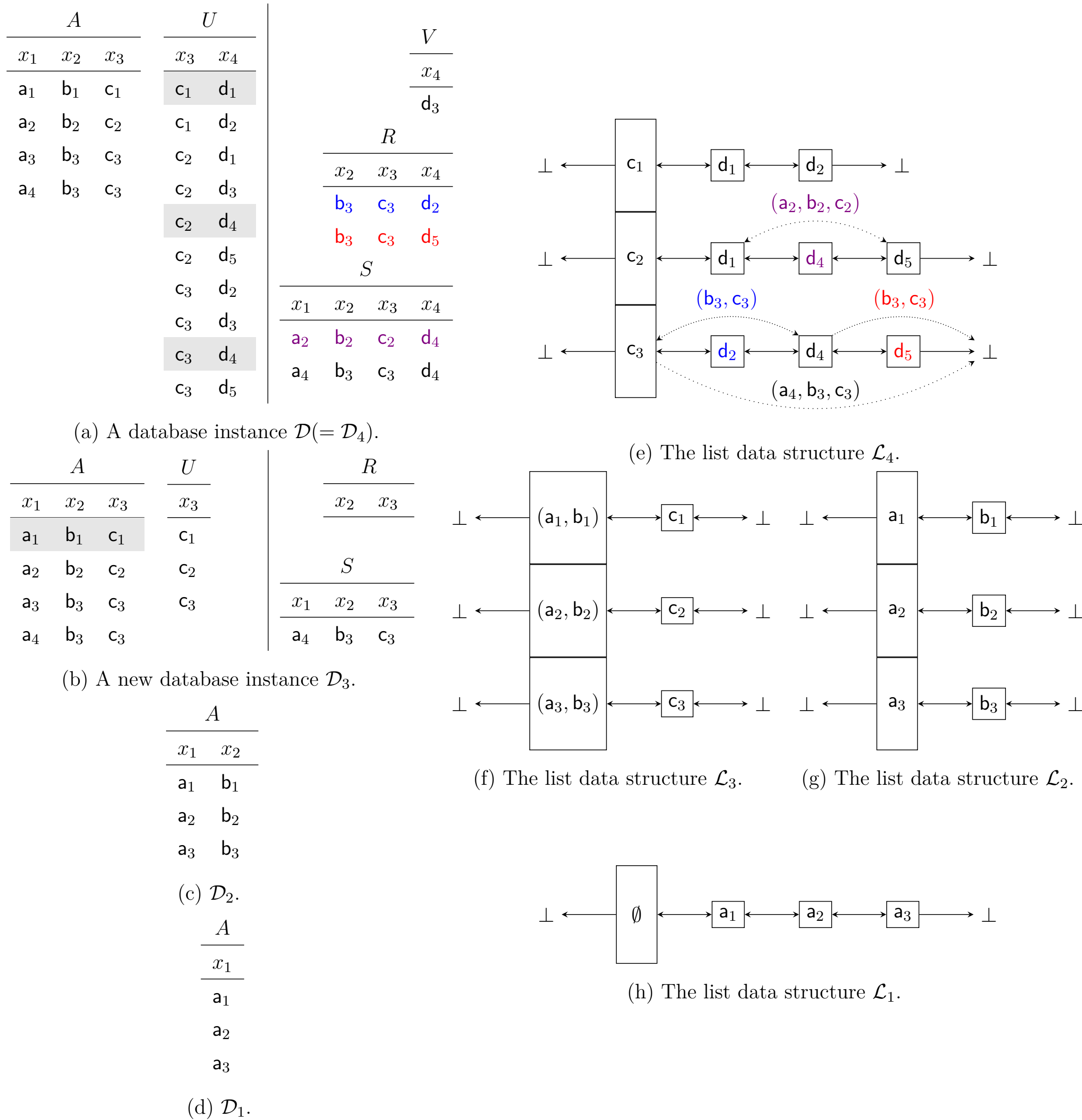
$$Q_1(x_1) \leftarrow A(x_1).$$

   Note that $\mathcal{L}_2$ also does not contain any skipping edge. The database is changed to $\mathcal{D}_1$ as in Figure 5.2(d).

4. Finally, we remove the signed leaf x_1 from $\mathcal{H}_1$, and this step essentially creates a linked-list $\mathcal{L}_1$ (shown in Figure 5.2(h)) on the remaining elements in the relation A in Figure 5.2(d).

Enumeration phase. For the enumeration step, we first enumerate every element in $\mathcal{L}_1$ (yielding $\mathsf{a}_1, \mathsf{a}_2$ and a_3). Then we use that element enumerated in $\mathcal{L}_1$ as a probing tuple in the enumeration process of every element in $\mathcal{L}_2$. For example, $\mathcal{L}_2.\mathsf{nextM}(\mathsf{a}_1)$ leads to b_1, and $\mathcal{L}_2.\mathsf{nextM}(\mathsf{a}_2)$ leads to b_2. We continue this step using the combined tuple enumerated from $\mathcal{L}_1$ and $\mathcal{L}_2$ (say, $(\mathsf{a}_1, \mathsf{b}_1)$), to enumerate the elements in $\mathcal{L}_3$. For example, $\mathcal{L}_3.\mathsf{nextM}((\mathsf{a}_1, \mathsf{b}_1))$ gives c_1 and $\mathcal{L}_3.\mathsf{nextM}((\mathsf{a}_2, \mathsf{b}_2))$ gives c_2.

Finally, we use the combined tuple enumerated from $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$ (say, $(\mathsf{a}_2, \mathsf{b}_2, \mathsf{c}_2)$), to enumerate the elements in $\mathcal{L}_4$. This step uses the skipping links: for example, $\mathcal{L}_4.\mathsf{nextM}((\mathsf{a}_2, \mathsf{b}_2, \mathsf{c}_2))$ would locate the doubly linked-list stored at the key c_2, and then we traverse that doubly linked-list using the tuple $(\mathsf{a}_2, \mathsf{b}_2, \mathsf{c}_2)$. We first yield d_1, but since the skipping links leaving d_1 contain $(\mathsf{a}_2, \mathsf{b}_2, \mathsf{c}_2)$, we follow that skipping-link and reach d_5, bypassing d_4 and correctly enumerating $(\mathsf{a}_2, \mathsf{b}_2, \mathsf{c}_2, \mathsf{d}_1)$ and $(\mathsf{a}_2, \mathsf{b}_2, \mathsf{c}_2, \mathsf{d}_5)$ as answers.



A

x_1	x_2	x_3
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b3	c3

U

x_3	x_4
c1	d1
c1	d2
c2	d1
c2	d3
c2	d4
c2	d5
c3	d2
c3	d3
c3	d4
c3	d5

V

x_4
d3

R

x_2	x_3	x_4
b3	c3	d2
b3	c3	d5

S

x_1	x_2	x_3	x_4
a2	b2	c2	d4
a4	b3	c3	d4

(a) A database instance $\mathcal{D}(= \mathcal{D}_4)$.

A

x_1	x_2	x_3
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b3	c3

U

x_3
c1
c2
c3

R

x_2	x_3

S

x_1	x_2	x_3
a4	b3	c3

(b) A new database instance $\mathcal{D}_3$.

A

x_1	x_2
a1	b1
a2	b2
a3	b3

(c) $\mathcal{D}_2$.

A

x_1
a1
a2
a3

(d) $\mathcal{D}_1$.

(e) The list data structure $\mathcal{L}_4$.

(f) The list data structure $\mathcal{L}_3$.

(g) The list data structure $\mathcal{L}_2$.

(h) The list data structure $\mathcal{L}_1$.

Figure 5.2: Intermediate databases and list data structures produced for Example 5.6.

5.4.4 Correctness Proof of Full $\mathsf{CQ}^{\neg}$

Lemma 5.5 establishes the correctness of our preprocessing and enumeration algorithm.

Lemma 5.5. *Let Q be a signed-acyclic full $\mathsf{CQ}^{\neg}$ and $\mathcal{D}$ be a database instance. Let $\sigma = \sigma' \cdot v$ be a signed-elimination sequence of the signed hypergraph of Q. Let Q', $\mathcal{D}'$, σ' be the input to the recursive call of Algorithm 4. Then the following statements hold:*

1. *if $\mathbf{a}_\sigma \in Q(\mathcal{D})$, then $\mathbf{a}_{\sigma'} \in Q'(\mathcal{D}')$; and*
2. *if $\mathbf{a}_{\sigma'} \in Q'(\mathcal{D}')$, then $\mathcal{L}_v.\mathsf{Iterate}(\mathbf{a}_{\sigma'})$ is nonempty and for every a_v emitted by $\mathcal{L}_v.\mathsf{Iterate}(\mathbf{a}_{\sigma'})$, we have $(\mathbf{a}_{\sigma'}, a_v) \in Q(\mathcal{D})$.*

Proof. Consider two items.

(1) Assume that $\mathbf{a}_\sigma \in Q(\mathcal{D})$. Let $\mathbf{a}_{\sigma'} = \Pi_{\sigma'}\mathbf{a}_\sigma$. Hence $\mathbf{a}_\sigma$ satisfies all positive atoms of Q, and therefore $\mathbf{a}_{\sigma'}$ must satisfy all positive atoms of Q' by the α-step.

Suppose for contradiction that $\Pi_{N_i \setminus \{v\}}\mathbf{a}_{\sigma'} \in R_{N_i \setminus \{v\}}$ for some $i \in [m]$. Therefore, we have that $\Pi_{U \setminus \{v\}}\mathbf{a}_{\sigma'} \in \Pi_{U \setminus \{v\}} R_U$, and $\mathcal{L}_v.\mathsf{nextM}(\Pi_{N_i \setminus \{v\}}\mathbf{a}_{\sigma'})$ is empty. This means that for every a_v such that $(\Pi_{U \setminus \{v\}}\mathbf{a}_{\sigma'}, a_v) \in R_U$ in $\mathcal{D}$, there is some $j \in [i]$ such that $(\Pi_{N_j \setminus \{v\}}\mathbf{a}_{\sigma'}, a_v) \in R_{N_j}$. Then in particular, there is some $j \in [i]$ such that $(\Pi_{N_j \setminus \{v\}}\mathbf{a}_{\sigma'}, \Pi_{\{v\}}\mathbf{a}_\sigma) = \Pi_{N_j}\mathbf{a}_\sigma \in R_{N_j}$. However, since $\mathbf{a}_\sigma \in Q(\mathcal{D})$, $\Pi_{N_j}\mathbf{a}_\sigma \notin R_{N_j}$, a contradiction. Therefore, $\Pi_{N_i \setminus \{v\}}\mathbf{a}_{\sigma'} \notin \Pi_{N_i \setminus \{v\}} R_{N_i}$ in $\mathcal{D}'$ for every $i \in [m]$, and $\mathbf{a}_{\sigma'} \in Q'(\mathcal{D}')$, as desired.

(2) Assume that $\mathbf{a}_{\sigma'} \in Q'(\mathcal{D}')$. Then, we have $\Pi_{U \setminus \{v\}}\mathbf{a}_{\sigma'} \in R_{U \setminus \{v\}}$, and thus $\Pi_{U \setminus \{v\}}\mathbf{a}_{\sigma'} \in \mathcal{L}_v$. First, we argue that $\mathcal{L}_v.\mathsf{Iterate}(\mathbf{a}_{\sigma'})$ is nonempty. Indeed, if $\mathcal{L}_v.\mathsf{Iterate}(\mathbf{a}_{\sigma'})$ is empty, then $\mathcal{L}_v.\mathsf{nextM}(\mathbf{a}_{N_m})$ must also be empty since N_m is the largest in the chain $U \subseteq N_1 \subseteq N_2 \subseteq \cdots \subseteq N_m$. Therefore, there exists some $i \in [m]$ such that $\Pi_{N_i \setminus \{v\}}\mathbf{a}_{\sigma'} \in \Pi_{N_i \setminus \{v\}} R_{N_i}$ in $\mathcal{D}'$. However, since $\mathbf{a}_{\sigma'} \in Q'(\mathcal{D}')$, $\Pi_{N_i \setminus \{v\}}\mathbf{a}_{\sigma'} \notin \Pi_{N_i \setminus \{v\}} R_{N_i}$ in $\mathcal{D}'$, a contradiction.

Let a_v be an arbitrary constant iterated by $\mathcal{L}_v.\mathsf{Iterate}(\mathbf{a}_{\sigma'})$. We show that $(\mathbf{a}_{\sigma'}, a_v) \in Q(\mathcal{D})$. Since $\mathbf{a}_{\sigma'} \in Q'(\mathcal{D}')$, $\Pi_{U \setminus \{v\}}\mathbf{a}_{\sigma'} \in \Pi_{U \setminus \{v\}} R_U$ in $\mathcal{D}'$, and by construction, $(\Pi_{U \setminus \{v\}}\mathbf{a}_{\sigma'}, a_v) \in R_U$ in $\mathcal{D}$ and thus satisfies every positive atom in Q by the α-step. It remains to show that $(\Pi_{N_i \setminus \{v\}}\mathbf{a}_{\sigma'}, a_v) \notin R_{N_i}$ for every $i \in [m]$. Indeed, if $(\Pi_{N_i \setminus \{v\}}\mathbf{a}_{\sigma'}, a_v) \in R_{N_i}$ for some $i \in [m]$, then a_v would have been skipped by a skipping link labeled $\Pi_{N_i \setminus \{v\}}\mathbf{a}_{\sigma'}$, and thus will not be iterated by $\mathcal{L}_v.\mathsf{Iterate}(\mathbf{a}_{\sigma'})$, a contradiction. We conclude the proof. ◻

Now we present our proof for Theorem 5.1.

Proof of Theorem 5.1. We first prove the theorem for full queries. Given a signed hypergraph $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$, its signed-elimination sequence can be found in time $O(|Q|^3)$ using brute force: we may find a signed leaf in $\mathcal{H}$ in time $O(|Q|^2)$ by first iterating over every vertex and then checking whether

it is a signed leaf by definition. Then we remove this signed leaf from $\mathcal{H}$, and iteratively apply the previous process until the graph is empty within $O(|Q|)$ iterations. We remark that this step can potentially be improved.

Let $\mathcal{H}$ be the hypergraph of Q before the preprocessing step. For every vertex v in $\mathcal{H}$, we denote $d(v)$ as the number of positive and negative hyperedges in $\mathcal{H}$ that contains v. A key observation is that: for any data structure $\mathcal{L}_v$, the procedures prevM, nextM runs in time $O(d(v))$, since there are at most $d(v)$ projections to check. For implementation, we also need to add create a book keeping hash table from each tuple $\mathbf{a}_U$ to the exact node in the hash table $\mathcal{L}_v$ that contains the value $\Pi_{\{v\}}\mathbf{a}_U$ for later use. Therefore, the α-step runs in $O(d(v) \cdot |R_U|)$ time, since for each tuple in R_U, we need to probe at most $d(v)$ relations to process it. For the β-step, the data structure $\mathcal{L}_v$ can be initialized in $O(|R_U|)$ time for line 11. Note that line 4 of $\mathsf{ExtendList}(\mathcal{L}_v, N_i)$ only takes constant time using the book keeping hash table created in the α-step. For line 15–16 and each $i \in \{1, 2, \ldots, m\}$, the running time is $O(d(v)\cdot|R_{N_i}|)$, since for each tuple in R_{N_i}, nextM and prevM are called, both requiring $O(d(v))$ time.

Hence one recursive step of the preprocessing phase runs in $O(d(v) \cdot (|R_U| + |R_{N_1}| + |R_{N_2}| + \ldots |R_{N_m}|)) = O(d(v) \cdot |\mathcal{D}|)$ time. Summing over all possible vertex v, we have that the preprocessing phase runs in time $\sum_{v\in\mathcal{V}} O(d(v) \cdot |\mathcal{D}|) = O(|Q| \cdot |\mathcal{D}|)$.

For the enumeration phase, let $(c_1, c_2, \ldots, c_n)$ and $(c'_1, c'_2, \ldots, c'_n)$ be two consecutive answers enumerated by Algorithm 10. Let i be the largest such that $c_j = c'_j$ for every $1 \le j \le i$. For every $i + 1 \le j \le n$, $\mathcal{L}_{v_j}.\mathsf{nextM}((c_1, c_2, \ldots, c_j))$ is called and returns $\perp$, incurring a running time of $O(d(v_j))$. Hence the overall delay is $\sum_{i+1\le j\le n} O(d(v_j)) = O(|Q|)$.

Next, we consider the case where Q is free-connex signed-acyclic with signed hypergraph $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ and free variables $F = [f]$ for an integer $0 \le f < n$. Then the signed hypergraph $\mathcal{H}' = ([n], \mathcal{E}^+, \mathcal{E}^- \cup \{F\})$ is signed-acyclic. By Proposition 5.2, there exists a signed-elimination sequence

$$\sigma = (1, 2, \ldots, f) \cdot \sigma'.$$

To enumerate $Q(\mathcal{D})$, the crux is to apply Algorithm 4 on inputs $(\mathcal{H}', \mathcal{D}, \sigma)$ in the preprocessing phase, *as if* Q were a full query. Subsequently in the enumeration step, we only uses the list data structures $\mathcal{L}_1$, $\mathcal{L}_2$, $\ldots$, $\mathcal{L}_f$, ignoring every $\mathcal{L}_v$ where v appears in the sequence σ'. This is correct, because every answer to $Q(\mathcal{D})$ must also participate in the first f positions of some full query answers, which can be recovered exactly by traversing only the first f list data structures.

$\square$

5.5 Enumeration of $\mathsf{NestFAQ}^{\neg}$

This section presents an algorithm for enumeration of $\mathsf{NestFAQ}^{\neg}$ (a superset of $\mathsf{FAQ}^{\neg}$) queries. We first introduce $\mathsf{NestFAQ}^{\neg}$ queries, and then we present the enumeration algorithm. We will use as a running example the following $\mathsf{FAQ}^{\neg}$ query, where we will also motivate the need to use the more expressive class of $\mathsf{NestFAQ}^{\neg}$ queries.

Example 5.7. Consider the following signed-acyclic $\mathsf{FAQ}^{\neg}$ $\varphi(x_1, x_2)$, where 3 is a signed-leaf of its associated hypergraph with a pivot hyperedge $\{3\}$ (corresponding to the positive factor R_3):

$$\varphi(x_1, x_2) = \bigoplus_{x_3} R_1(x_1) \otimes R_2(x_2) \otimes R_3(x_3) \otimes \overline{R}_{123}(x_1, x_2, x_3) \otimes \overline{R}_{23}(x_2, x_3)$$

Our strategy for $\mathsf{FAQ}^{\neg}$ queries is to follow the elimination sequence as in $\mathsf{CQ}^{\neg}$. However, the introduction of aggregation complicates the signed-elimination step. Consider for example the signed-elimination step for the signed-leaf 3. A naive attempt may aim for a reduction to

$$\varphi'(x_1, x_2) = R_1(x_1) \otimes R_2(x_2) \otimes \overline{R'}_{12}(x_1, x_2) \otimes \overline{R'}_2(x_2),$$

asking for $\overline{R'}_{12}(x_1, x_2) \otimes \overline{R'}_2(x_2) = \bigoplus_{x_3} R_3(x_3) \otimes \overline{R}_{123}(x_1, x_2, x_3) \otimes \overline{R}_{23}(x_2, x_3)$. However, this is not attainable by the semantics of $\mathsf{FAQ}^{\neg}$ over a general semiring. Indeed, if we examine the weight $\overline{R'}_2(a_2)$, for some $x_2 = a_2$ such that $\Pi_{x_3}\overline{R}_{23}(a_2, x_3) \neq \emptyset$: if $x_1 = a_1$ with $\Pi_{x_1,x_2}\overline{R}_{123}(a_1, a_2, x_3) = \emptyset$, then it should encode $\bigoplus_{x_3} R_3(x_3) \otimes \mathbf{c}_{123} \otimes \overline{R}_{23}(a_2, x_3)$, where $\mathbf{c}_{123}$ is the constant value for entries out of the table of $\overline{R}_{123}$; otherwise, we want $\overline{R'}_2(a_2) = \mathbf{1}$ and $\overline{R'}_{12}(a_1, a_2)$ to store $\bigoplus_{x_3} R_3(x_3) \otimes \overline{R}_{123}(a_1, a_2, x_3) \otimes \overline{R}_{23}(a_2, x_3)$. The two weights may not coincide over an arbitrary semiring. This insufficiency calls for lifting $\mathsf{FAQ}^{\neg}$ to a more expressive class of queries $\mathsf{NestFAQ}^{\neg}$.

5.5.1 $\mathsf{NestFAQ}^{\neg}$ Queries

We now formally introduce *nested functional aggregate queries with negation* ($\mathsf{NestFAQ}^{\neg}$) over a single semiring $\sigma = (\Sigma, \oplus, \otimes, \mathbf{0}, \mathbf{1})$.

Let $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ be a signed hypergraph. To each $K \in \mathcal{E}^+$ (and $K \in \mathcal{E}^-$), we associate a distinct function $R_K : \mathsf{Dom}(\mathbf{x}_K) \to \Sigma$, called a *factor*. We assume that all factors are represented via the *listing representation*: each factor R_K is a table of all tuples of the form $\langle \mathbf{a}_K, R_K(\mathbf{a}_K)\rangle$, where $R_K(\mathbf{a}_K) \in \Sigma$ is the *weight* of the tuple $\mathbf{a}_K$. For entries not in the table, the factor implicitly encodes their weights as $\mathbf{0}$. Under set semantics, we also use R_K to denote the set of tuples of schema $\mathbf{x}_K$ explicitly stored in the factor table and $\neg R_K = \mathsf{Dom}(\mathbf{x}_K) \setminus R_K$. Our definition of listing representation is slightly more general than [AKNR16, KCM+20] in that we allow $R_K(\mathbf{x}_K) = \mathbf{0}$ for some $\mathbf{x}_K$ in the table (i.e. $\mathbf{x}_K \in R_K$). A factor R_K where $K \in \mathcal{E}^+$ (resp. R_N where $N \in \mathcal{E}^-$) is called a *positive* (resp. *negative*).

A $\mathsf{NestFAQ}^{\neg}$ expression $\psi(\mathbf{x}_{[n]})$ associated with $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ is a function $\mathsf{Dom}(\mathbf{x}_{[n]}) \to \Sigma$ which is syntactically recognized by the following context-free grammar $\mathsf{CFG}_{\mathcal{H}}$:

$$\begin{aligned} &\forall S \subseteq [n], \\ &\quad \ell(\mathbf{x}_\emptyset) ::= e && \text{where } S = \emptyset \text{ and } e \in \Sigma \setminus \{\mathbf{0}\} \\ &\quad \ell(\mathbf{x}_S) ::= R_K(\mathbf{x}_K) \otimes \ell(\mathbf{x}_{S^-}) && \text{where } S = K \cup S^-, K \in \mathcal{E}^+ \\ &\qquad\qquad \mid\ (R_N(\mathbf{x}_N) \vdash \ell(\mathbf{x}_{N^-})) \otimes \ell(\mathbf{x}_{S^-}) && \text{where } S = N \cup S^-, N^- \subseteq N \in \mathcal{E}^- \end{aligned} \tag{5.4}$$

where

Terminals: $e, R_K(\mathbf{x}_K)$ and $R_N(\mathbf{x}_N)$, where $e \in \Sigma \setminus \{\mathbf{0}\}, K \in \mathcal{E}^+$ and $N \in \mathcal{E}^-$, the semiring operators $\otimes$ and $\vdash$, and the parentheses are the terminal symbols of the CFG. On its semantics, $R_N(\mathbf{x}_N) \vdash \ell(\mathbf{x}_{N^-})$ is a shorthand for $R_N(\mathbf{x}_N) \oplus (\mathbb{1}_{\neg R_N}(\mathbf{x}_N) \otimes \ell(\mathbf{x}_{N^-}))$, where $N^- \subseteq N$ and $\mathbb{1}_{R_N}$ is an indicator factor defined as the following (also defined in [KCM+20]):

$$\mathbb{1}_{R_N}(\mathbf{x}_N) = \begin{cases} \mathbf{1} & \text{if } \mathbf{x}_N \in R_N \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

Thus, $\vdash$ is a non-commutative operator that takes a negative factor as its left operand and a $\mathsf{NestFAQ}^{\neg}$ subexpression as its right operand.

Non-terminals: $\ell(\mathbf{x}_S)$, where $S \subseteq [n]$, are the non-terminals of the CFG. A $\mathsf{NestFAQ}^{\neg}$ subexpression is an expression $\psi(\mathbf{x}_S)$ derived from the $\ell(\mathbf{x}_S)$ non-terminal, following a sequence of grammar rule applications as defined in (5.4).

Start symbol: $\ell(\mathbf{x}_{[n]})$ is the start symbol of the CFG.

Without loss of generality, we assume that each factor R_K or R_N, where $K \in \mathcal{E}^+, N \in \mathcal{E}^-$, shows up precisely once in $\psi(\mathbf{x}_{[n]})$, i.e., there exists a one-to-one mapping from the hyperedges of $\mathcal{H}$ to factors in $\psi(\mathbf{x}_{[n]})$. Moreover, we assume that a $\mathsf{NestFAQ}^{\neg}$ expression $\psi(\mathbf{x}_{[n]})$ is *safe*, that is, every $i \in [n]$ appears in some $K \in \mathcal{E}^+$ (i.e. $\bigcup_{K \in \mathcal{E}^+} K = [n]$). The *size* of a $\mathsf{NestFAQ}^{\neg}$ subexpression $\psi(\mathbf{x}_{[n]})$ is defined as $|\psi| = \sum_{i \in [n]} d(i)$, where $d(i)$ is the number of hyperedges in $\mathcal{H}$ that contain i.

A $\mathsf{NestFAQ}^{\neg}$ query φ (with free variables $F \subseteq [n]$) associated with a signed hypergraph $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ is a function defined as:

$$\varphi(\mathbf{x}_F) = \bigoplus_{\mathbf{x}_{[n] \setminus F} \in \mathsf{Dom}(\mathbf{x}_{[n] \setminus F})} \psi(\mathbf{x}_{[n]}) \tag{5.5}$$

where

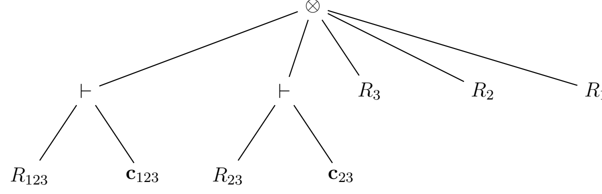


Figure 5.3: The tree representation of the $\mathsf{NestFAQ}^{\neg}$ associated with the query in Example 5.7.

- $\psi(\mathbf{x}_{[n]})$ is a safe $\mathsf{NestFAQ}^{\neg}$ expression recognized by the CFG in (5.4). Sometimes we call $\psi(\mathbf{x}_{[n]})$ a *full* $\mathsf{NestFAQ}^{\neg}$ query (i.e. $F = [n]$).

- $F = [f] \subseteq [n]$ is the set of *free variables* for some integer $0 \leq f \leq n$. We will often write the summation as simply $\bigoplus_{\mathbf{x}_{[n]\setminus F}}$.

An *answer* (or *output*) of φ is a tuple $\langle \mathbf{a}_F, \varphi(\mathbf{a}_F)\rangle$ such that $\mathbf{a}_F \in \mathsf{Dom}(\mathbf{x}_F)$ and $\varphi(\mathbf{a}_F) \neq \mathbf{0}$. The *query size* of a $\mathsf{NestFAQ}^{\neg}$ query φ is $|\varphi| = |\psi| = \sum_{i\in[n]} d(i)$, while the *input size* $\mathcal{D}_\varphi$ is defined as the total size of the list representations for every factor in ψ.

A fragment of $\mathsf{NestFAQ}^{\neg}$ queries is the $\mathsf{FAQ}^{\neg}$ queries, where the right child of the $\vdash$ nodes are $\mathbf{c}_N \neq \mathbf{0}$, that is, the default value of R_N. Indeed, the query can then be compactly written as

$$\varphi(\mathbf{x}_F) = \bigoplus_{\mathbf{x}_{[n]\setminus F}} \bigotimes_{K\in\mathcal{E}^+} R_K(\mathbf{x}_K) \otimes \bigotimes_{N\in\mathcal{E}^-} (R_N(\mathbf{x}_N) \vdash \mathbf{c}_N)$$

This coincides with the definition of $\mathsf{FAQ}^{\neg}$ queries in (5.3) by letting $\overline{R}_N(\mathbf{x}_N) = (R_N(\mathbf{x}_N) \vdash \mathbf{c}_N)$. As an example, Figure 5.3 shows pictorially (in the form of a tree) the $\mathsf{NestFAQ}^{\neg}$ associated with the query in Example 5.7. In this representation,

$$\overline{R}_{123} = R_{123} \vdash \mathbf{c}_{123} = R_{123} \oplus (\mathbb{1}_{\neg R_{123}} \otimes \mathbf{c}_{123}), \qquad \overline{R}_{23} = R_{23} \vdash \mathbf{c}_{23} = R_{23} \oplus (\mathbb{1}_{\neg R_{23}} \otimes \mathbf{c}_{23})$$

Definition 5.4 (Free-connex signed-acyclicity)**.** A $\mathsf{NestFAQ}^{\neg}$ query φ is *free-connex signed-acyclic* if the signed hypergraph $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^- \cup \{F\})$ is signed-acyclic.

In the following sections, we will show the following main theorem for free-connex signed-acyclic $\mathsf{NestFAQ}^{\neg}$ queries, which subsumes Theorem 5.2 as a special case.

Theorem 5.4. *Let φ be a free-connex signed-acyclic $\mathsf{NestFAQ}^{\neg}$ query with free variables F over a commutative semiring σ. Then, there is an algorithm that can enumerate the answers of φ in $O(|\varphi|)$ delay, after a $O(|\varphi|^3 + |\varphi| \cdot \mathcal{D}_\varphi \cdot \alpha(14 \cdot \mathcal{D}_\varphi, \mathcal{D}_\varphi))$ preprocessing time.*

5.5.2 Enumeration Algorithm: An Overview

In the following sections, we let φ be a free-connex signed-acyclic $\mathsf{NestFAQ}^{\neg}$ query (5.5) with free variables $F = [f]$ and $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ be its associated signed hypergraph. We let $\psi(\mathbf{x}_{[n]})$ be the $\mathsf{NestFAQ}^{\neg}$ expression of φ, recognized by the CFG defined in (5.4). As $\mathcal{H}$ is signed-acyclic, w.l.o.g., it accepts a signed-elimination sequence $(1, 2, \dots, f, \dots, n)$.

At a high level, our preprocessing algorithm takes as input the $\mathsf{NestFAQ}^{\neg}$ expression $\psi(\mathbf{x}_{[n]})$, and for each $i = n-1, n-2, \dots, f$ (following the signed-elimination order), it executes a **signed-elimination step on** i, that constructs an intermediate $\mathsf{NestFAQ}^{\neg}$ expression $\psi_i(\mathbf{x}_{[i]})$, where

- $\psi_i(\mathbf{x}_{[i]})$ is associated with the signed hypergraph $\mathcal{H}_i = \langle \mathcal{H}_{i+1}, i+1 \rangle$, i.e. the signed hypergraph after eliminating $n, n-1, \dots, i+1$ from $\mathcal{H}$, (if $i = n-1$, then $\mathcal{H}_{n-1} = \langle \mathcal{H}, n \rangle$);
- for any $\mathbf{a}_{[i]} \in \mathsf{Dom}(\mathbf{x}_{[i]})$, we have $\psi_i(\mathbf{a}_{[i]}) = \bigoplus_{x_{i+1} \in \mathsf{Dom}(x_{i+1})} \psi_{i+1}(x_{i+1}, \mathbf{a}_{[i]})$.

In other words, a signed-elimination step on i aggregates out x_{i+1} from the $\mathsf{NestFAQ}^{\neg}$ expression $\psi_i(\mathbf{x}_{[i]})$. We now break down the signed-elimination step. W.l.o.g., let n be the given signed-leaf of $\mathcal{H}$ and appoint U to be a pivot hyperedge, breaking ties arbitrarily. We call the factor $R_U(\mathbf{x}_U)$ corresponding to U the *pivot factor*. A signed-elimination step runs the following 3 algorithms:

Refactoring (Section 5.5.3) The refactoring algorithm takes the given $\mathsf{NestFAQ}^{\neg}$ expression $\psi(\mathbf{x}_{[n]})$ as input and returns an equivalent $\mathsf{NestFAQ}^{\neg}$ expression $\psi_\dagger(\mathbf{x}_{[n]})$, recognized by the following grammar ${\mathsf{CFG}^\dagger}_{\mathcal{H}_\dagger}$ associated with the signed hypergraph $\mathcal{H}_\dagger = ([n], \mathcal{E}^+_\dagger, \mathcal{E}^-_\dagger)$, where $\mathcal{E}^+_\dagger = \{K \in \mathcal{E}^+ \mid n \notin K\} \cup \{U\} \subseteq \mathcal{E}^+$ and $\mathcal{E}^-_\dagger = \{N \in \mathcal{E}^- \mid n \notin N \vee U \subset N\} \subseteq \mathcal{E}^-$,

$$\forall S \subseteq [n], \tag{5.6}$$

$$\ell_\dagger(\mathbf{x}_\emptyset) ::= e \qquad \text{where } S = \emptyset \text{ and } e \in \Sigma \setminus \{\mathbf{0}\} \tag{5.7}$$

$$\ell_\dagger(\mathbf{x}_S) ::= R_U(\mathbf{x}_U) \qquad \text{where } S = U \tag{5.8}$$

$$\mid\ R_K(\mathbf{x}_K) \otimes \ell_\dagger(\mathbf{x}_{S^-}) \qquad \text{w. } S = K \cup S^- \text{ and } K \in \{K \in \mathcal{E}^+ \mid n \notin K\} \tag{5.9}$$

$$\mid\ (R_N(\mathbf{x}_N) \vdash \ell_\dagger(\mathbf{x}_{N^-})) \qquad \text{where } S = N \text{ and } N^- \in \{N \in \mathcal{E}^- \mid U \subset N\} \tag{5.10}$$

$$\mid\ (R_N(\mathbf{x}_N) \vdash \ell_\dagger(\mathbf{x}_{N^-})) \otimes \ell_\dagger(\mathbf{x}_{S^-}) \qquad \text{w. } S = N \cup S^- \text{ and } N^- \in \{N \in \mathcal{E}^- \mid n \notin N\} \tag{5.11}$$

The new grammar is more restrictive than $\mathsf{CFG}_{\mathcal{H}}$ (5.4) in that it has strictly less productions for the non-terminals. The refactoring algorithm uses the pivot factor R_U to "absorb" the factors R_K and R_N where $n \in K \subseteq U$, $n \in N \subseteq U$. By the properties of a signed-leaf n,

the only hyperedges containing n left in $\mathcal{H}_\dagger$ are U and $N_1, \ldots, N_k \in \mathcal{E}^-$ for some k such that $U \subseteq N_1 \subseteq \cdots \subseteq N_k$.

Oracle Construction (Section 5.5.5) The oracle-construction algorithm takes the $\mathsf{NestFAQ}^\neg$ expression $\psi_\dagger(\mathbf{x}_{[n]})$ from the refactoring algorithm and constructs an oracle data structure for $\mathsf{NestFAQ}^\neg$ subexpressions that show up in $\psi_\dagger(\mathbf{x}_{[n]})$. The oracle-construction algorithm is analogous to the β-step in the enumeration of full $\mathsf{CQ}^\neg$ queries, but instead of linked lists, it builds arrays and RangeSum data structures that support fast aggregation.

Aggregation (Section 5.5.4) The aggregation algorithm takes $\psi_\dagger(\mathbf{x}_{[n]})$ and uses the constructed oracles from the previous step to construct the desired $\psi_{n-1}(\mathbf{x}_{n-1})$, where

- $\psi_{n-1}(\mathbf{x}_{n-1})$ is a new $\mathsf{NestFAQ}^\neg$ expression recognized by the grammar $\mathsf{CFG}_{\mathcal{H}_{n-1}}$ associated with $\mathcal{H}_{n-1} = \langle \mathcal{H}, n \rangle = \mathcal{H}_{n-1} = ([n-1], \mathcal{E}^+_{n-1}, \mathcal{E}^-_{n-1})$, where $\mathcal{E}^+_{n-1} = \{K \setminus \{n\} \mid K \in \mathcal{E}^+_n\}$ and $\mathcal{E}^-_{n-1} = \{N \setminus \{n\} \mid N \in \mathcal{E}^-_n\}$.
- $\psi_{n-1}(\mathbf{x}_{n-1}) = \bigoplus_{x_n} \psi_n(\mathbf{x}_{[n]})$.

We can keep applying the signed-elimination step on the next signed-leaf $n-1$ and so on, untill all variables in $[n] \setminus F$ have been removed. The last $\mathsf{NestFAQ}^\neg$ expression $\psi_f(\mathbf{x}_{[f]})$, after the sequence of signed-elimination steps on $n-1, n-2, \ldots, f$, becomes a full $\mathsf{NestFAQ}^\neg$ query whose associated signed hypergraph $\mathcal{H}_f$ accepts a signed-elimination sequence $(1, 2, \ldots, f)$ and

$$\varphi(\mathbf{x}_F) = \bigoplus_{\mathbf{x}_{[n]\setminus F}} \psi(\mathbf{x}_{[n]}) = \bigoplus_{\mathbf{x}_{[n-1]\setminus F}} \psi_{n-1}(\mathbf{x}_{[n-1]}) = \cdots = \psi_f(\mathbf{x}_{[f]}).$$

Thus, we arrive to the case where φ is a full signed-acyclic $\mathsf{NestFAQ}^\neg$ query. At this point, we can reduce $\mathsf{Enum}(\psi_f(\mathbf{x}_{[f]}))$ into $\mathsf{Enum}(Q^*_f(\mathbf{x}_{[f]}), \mathcal{D}^*_f)$, where Q^*_f is a full signed-acyclic $\mathsf{CQ}^\neg$ (Section 5.5.6) and apply the preprocessing algorithm for a full $\mathsf{CQ}^\neg$. At the enumeration phase, we emit each answer $\mathbf{a}_F$ of $Q^*_f(\mathcal{D}_f)$ (applying the enumeration algorithm for full $\mathsf{CQ}^\neg$) and plug the emitted tuple $\mathbf{a}_F$ into ψ_f to recover its weight $\psi_f(\mathbf{a}_F) = \varphi(\mathbf{a}_F)$.

5.5.3 The Refactoring Algorithm

In this section, we present the algorithm that refactors $\psi(\mathbf{x}_{[n]})$ into $\psi_\dagger(\mathbf{x}_{[n]})$, a $\mathsf{NestFAQ}^\neg$ expression associated with $\mathcal{H}_\dagger$. We start from $\ell(\mathbf{x}_{[n]}) \overset{*}{\Rightarrow} \psi(\mathbf{x}_{[n]})$, the derivation in CFG (5.4), and refactor it step-by-step into a derivation $\ell_\dagger(\mathbf{x}_{[n]}) \overset{*}{\Rightarrow} \psi_\dagger(\mathbf{x}_{[n]})$.

Lemma 5.6. *Let $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ be a signed hypergraph, n be a signed-leaf of $\mathcal{H}$ and U be a pivot hyperedge. Let φ be a free-connex signed-acyclic* $\mathsf{NestFAQ}^\neg$ *query* (5.5) *associated with $\mathcal{H}$. There is an algorithm (*Refactor*) that takes as input*

1. *a derivation of a* NestFAQ$^{\neg}$ *subexpression* $\mu(\mathbf{x}_M)$*, i.e.* $\ell(\mathbf{x}_M) \overset{*}{\Rightarrow} \mu(\mathbf{x}_M)$*, for some* $M \subseteq [n]$ *such that if* $n \notin M$*, then* $M = \emptyset$ *and* $\mu(\mathbf{x}_M) = \mathbf{1}$*;*
2. *a derivation of a* NestFAQ$^{\neg}$ *subexpression* $\psi(\mathbf{x}_S)$*, i.e.* $\ell(\mathbf{x}_S) \overset{*}{\Rightarrow} \psi(\mathbf{x}_S)$*, for some* $S \subseteq [n]$*;*

and returns as ouput

1. *an expression in CFG* (5.6) *equivalent to* $\mu(\mathbf{x}_M) \otimes \psi(\mathbf{x}_S)$ *, s.t. the length of the derivation* $\ell_{\dagger}(\mathbf{x}_{M\cup S}) \overset{*}{\Rightarrow} \mu(\mathbf{x}_M) \otimes \psi(\mathbf{x}_S)$ *is at most* $|\eta|$*, where* $|\eta|$ *is the length of the derivation* $\ell(\mathbf{x}_S) \overset{*}{\Rightarrow} \psi(\mathbf{x}_S)$ *plus the length of the derivation* $\ell(\mathbf{x}_M) \overset{*}{\Rightarrow} \mu(\mathbf{x}_M)$*.*
2. *in time* $O(|\eta|^2 + d(n) \cdot |\mathcal{D}_{\psi}|)$*, where* $d(n)$ *is the number of hyperedges in* $\mathcal{H}$ *that contain* n*.*

In particular, we let

$$\psi_{\dagger}(\mathbf{x}_{[n]}) = \mathsf{Refactor}\left(\ell(\mathbf{x}_{\emptyset}) \Rightarrow \mathbf{1}, \ell(\mathbf{x}_{[n]}) \overset{*}{\Rightarrow} \psi(\mathbf{x}_{[n]})\right),$$

where $\mathsf{Refactor}\left(\ell(\mathbf{x}_M) \overset{*}{\Rightarrow} \mu(\mathbf{x}_M), \ell(\mathbf{x}_S) \overset{*}{\Rightarrow} \psi(\mathbf{x}_S)\right)$, for some $M, S \subseteq [n]$, is an invokation of a recursive algorithm Refactor that takes the following inputs:

1. a stashed derivation of a NestFAQ$^{\neg}$ subexpression $\mu(\mathbf{x}_M)$, i.e. $\ell(\mathbf{x}_M) \overset{*}{\Rightarrow} \mu(\mathbf{x}_M)$, such that its first yield step will be maintained to be one of the followings:
   - if $n \notin M$, then it follows that $M = \emptyset$ and $\mu(\mathbf{x}_M) \Rightarrow \mathbf{1}$; or
   - otherwise $n \in M$, then it applies the production of CFG that uses the largest hyperedge among all productions that can be used as the next yield step, if there is any (an arbitrary choice otherwise) – this is possible because $\otimes$ is commutative. The reader can consider this as re-arranging the children of the root $\otimes$ in the AST (Figure 5.4) to put the largest hyperedge containing n in the leftmost subtree.
2. a derivation of a NestFAQ$^{\neg}$ subexpression $\psi(\mathbf{x}_S)$, i.e. $\ell(\mathbf{x}_S) \overset{*}{\Rightarrow} \psi(\mathbf{x}_S)$, for some $S \subseteq [n]$,

and returns a NestFAQ$^{\neg}$ subexpression $\psi_{\dagger}(\mathbf{x}_{M\cup S})$ recognized by CFG (5.6) such that $\ell_{\dagger}(\mathbf{x}_{M\cup S}) \overset{*}{\Rightarrow} \psi_{\dagger}(\mathbf{x}_{M\cup S}) = \mu(\mathbf{x}_M) \otimes \psi(\mathbf{x}_S)$. The refactoring algorithm Refactor is illustrated in Algorithm 11.

Proof. We prove the lemma by induction on $|\eta| \geq 2$ for every if-else branch in Refactor. We first examine the case where the next yield step of $\ell(\mathbf{x}_S)$ is $\ell(\mathbf{x}_{\emptyset}) \Rightarrow \psi(\mathbf{x}_{\emptyset}) = \mathbf{1}$, so $S = \emptyset$. We divide into 2 main cases (case (1) and (2)), one for each next yield step of $\ell(\mathbf{x}_M)$:

1. (line 2) the base case is when $M = \emptyset$ and $\ell(\mathbf{x}_M) \Rightarrow \mu(\mathbf{x}_{\emptyset}) = e$, in which case ($M = S = \emptyset, |\mu| = |\psi| = 1$) we simply return e and the lemma is trivially true. Indeed, it uses the production $\ell_{\dagger}(\mathbf{x}_{\emptyset}) ::= e$ in CFG (5.6).

Algorithm 11: The refactoring algorithm $\mathsf{Refactor}(\ell(\mathbf{x}_M) \stackrel{*}{\Rightarrow} \mu(\mathbf{x}_M), \ell(\mathbf{x}_S) \stackrel{*}{\Rightarrow} \psi(\mathbf{x}_S))$

Input: two derivations in $\mathsf{CFG}_{\mathcal{H}}$, $\ell(\mathbf{x}_M) \stackrel{*}{\Rightarrow} \mu(\mathbf{x}_M)$ and $\ell(\mathbf{x}_S) \stackrel{*}{\Rightarrow} \psi(\mathbf{x}_S)$, for some $M, S \subseteq [n]$

Output: an expression $\psi_\dagger(\mathbf{x}_{M\cup S})$ recognized by $\mathsf{CFG}^\dagger_{\mathcal{H}}$ such that
$\ell_\dagger(\mathbf{x}_{M\cup S}) \stackrel{*}{\Rightarrow} \psi_\dagger(\mathbf{x}_{M\cup S}) = \mu(\mathbf{x}_M) \otimes \psi(\mathbf{x}_S)$.

```
1  if ℓ(x_S) ⇒ e then
2      if ℓ(x_M) ⇒ 1 then                                              ▷ base case
3          return e ;
4      else if ℓ(x_M) ⇒ R_U(x_U) ⊗ ℓ(x_{M⁻}) then                        ▷ (2.2)
5          R†_U(x_U) ← e ⊗ R_U(x_U) ⊗ μ(x_{M⁻})          ▷ update the factor R_U ;
6          return R†_U(x_U) ;
7      else if ℓ(x_M) ⇒ (R_N(x_N) ⊢ ℓ(x_{N⁻})) ⊗ ℓ(x_{M⁻}) then          ▷ (2.1)
8          R†_N(x_N) ← e ⊗ R_N(x_N) ⊗ μ(x_{M⁻})          ▷ update the factor R_N ;
9          return (R†_N(x_N) ⊢ Refactor(ℓ(x_{M⁻}) ⇒* μ(x_{M⁻}), ℓ(x_{N⁻}) ⇒* ψ(x_{N⁻}))) ;
10 else if ℓ(x_S) ⇒ R_K(x_K) ⊗ ℓ(x_{S⁻}) then                    ▷ K ∈ ℰ⁺, S = K ∪ S⁻
11     if n ∉ K then                                                     ▷ (3.1)
12         return R_K(x_K) ⊗ Refactor(ℓ(x_M) ⇒* μ(x_M), ℓ(x_{S⁻}) ⇒* ψ(x_{S⁻})) ;
13     else                                                              ▷ (4.1)
14         augment ℓ(x_M) ⇒* μ(x_M) into ℓ(x_{M∪K}) ⇒ R_K(x_K) ⊗ ℓ(x_M) ⇒* R_K(x_K) ⊗ μ(x_M) ;
15         return Refactor(ℓ(x_{M∪K}) ⇒* R_K(x_K) ⊗ μ(x_M), ℓ(x_{S⁻}) ⇒* ψ(x_{S⁻})) ;
16 else if ℓ(x_S) ⇒ (R_N(x_N) ⊢ ℓ(x_{N⁻})) ⊗ ℓ(x_{S⁻}) then   ▷ N ∈ ℰ⁻, N⁻ ⊆ N, S = N ∪ S⁻
17     if n ∉ N then                                                     ▷ (3.2)
18         return (R_N(x_N) ⊢ μ(x_{N⁻})) ⊗ Refactor(ℓ(x_M) ⇒* μ(x_M), ℓ(x_{S⁻}) ⇒* ψ(x_{S⁻})) ;
19     else                                                              ▷ (4.2)
20         augment ℓ(x_M) ⇒* μ(x_M) into
             ℓ(x_{M∪N}) ⇒ (R_N(x_N) ⊢ ℓ(x_{N⁻})) ⊗ ℓ(x_M) ⇒* μ(x_{M∪N}) ;
21         return Refactor(ℓ(x_{M∪N}) ⇒* μ(x_{M∪N}), ℓ(x_{S⁻}) ⇒* ψ(x_{S⁻})) ;
```

2. otherwise, ($n \in M$), recall $S = \emptyset$ now, so there must be one largest hyperedge in $\mu(\mathbf{x}_M)$, by the β-property of the signed-leaf n and we have the following cases:

   (2.1) (line 7) let the next yield step of $\ell(\mathbf{x}_M)$ be

$$\ell(\mathbf{x}_M) \Rightarrow (R_N(\mathbf{x}_N) \vdash \ell(\mathbf{x}_{N^-})) \otimes \ell(\mathbf{x}_{M^-}) \stackrel{*}{\Rightarrow} (R_N(\mathbf{x}_N) \vdash \mu(\mathbf{x}_{N^-})) \otimes \mu(\mathbf{x}_{M^-})$$

where $N \in \mathcal{E}^-$, $N^- \subseteq N$ and $M^- \subseteq M$ such that $M = N \cup M^-$. As N is the largest hyperedge used among the next yield steps of $\ell(\mathbf{x}_M)$, we have $M^- \subseteq N$. Then we have

$$\begin{aligned}\ell(\mathbf{x}_M) \otimes \ell(\mathbf{x}_S) &\overset{*}{\Rightarrow} (R_N(\mathbf{x}_N) \vdash \ell(\mathbf{x}_{N^-})) \otimes \ell(\mathbf{x}_{M^-}) \otimes e \\ &\overset{*}{\Rightarrow} (R_N(\mathbf{x}_N) \vdash \mu(\mathbf{x}_{N^-})) \otimes \mu(\mathbf{x}_{M^-}) \otimes e \\ &= \ R_N^\dagger(\mathbf{x}_N) \vdash (\mu(\mathbf{x}_{N^-}) \otimes \mu(\mathbf{x}_{M^-}))\end{aligned}$$

where $R_N^\dagger(\mathbf{x}_N) = R_N(\mathbf{x}_N) \otimes \mu(\mathbf{x}_{M^-})$ is an updated negative factor that can be computed in $O(|R_N|)$ time. Recall that we return in this case

$$\left(R_N^\dagger(\mathbf{x}_N) \vdash \mathsf{Refactor}\left(\ell(\mathbf{x}_{M^-}) \overset{*}{\Rightarrow} \mu(\mathbf{x}_{M^-}), \ell(\mathbf{x}_{N^-}) \overset{*}{\Rightarrow} \mu(\mathbf{x}_{N^-})\right)\right).$$

Now, we observe that the first yield step in the stashed derivation $\ell(\mathbf{x}_M) \Rightarrow (R_N(\mathbf{x}_N) \vdash \ell(\mathbf{x}_{N^-})) \otimes \ell(\mathbf{x}_{M^-}) \overset{*}{\Rightarrow} \psi(\mathbf{x}_M)$ is trimmed off in this inner recursive call. That is, $|\eta|$ decrements by 1 in the recursion. By the induction hypothesis, $\ell_\dagger(\mathbf{x}_{M^- \cup N^-}) \overset{*}{\Rightarrow}$

$$\psi_\dagger(\mathbf{x}_{M^- \cup N^-}) = \mu(\mathbf{x}_{M^-}) \otimes \mu(\mathbf{x}_{N^-}) = \mathsf{Refactor}\left(\ell(\mathbf{x}_{M^-}) \overset{*}{\Rightarrow} \mu(\mathbf{x}_{M^-}), \ell(\mathbf{x}_{N^-}) \overset{*}{\Rightarrow} \mu(\mathbf{x}_{N^-})\right).$$

Thus, we get

$$\begin{aligned}\mu(\mathbf{x}_M) \otimes \psi(\mathbf{x}_S) &= (R_N(\mathbf{x}_N) \vdash \mu(\mathbf{x}_{N^-})) \otimes \mu(\mathbf{x}_{M^-}) \otimes e \\ &= R_N^\dagger(\mathbf{x}_N) \vdash (\mu(\mathbf{x}_{M^-}) \otimes \mu(\mathbf{x}_{N^-})) \\ &= \left(R_N^\dagger(\mathbf{x}_N) \vdash \mathsf{Refactor}\left(\ell(\mathbf{x}_{M^-}) \overset{*}{\Rightarrow} \mu(\mathbf{x}_{M^-}), \ell(\mathbf{x}_{N^-}) \overset{*}{\Rightarrow} \mu(\mathbf{x}_{N^-})\right)\right).\end{aligned}$$

and $\ell_\dagger(\mathbf{x}_{M \cup S})$ indeed derives this subexpression in CFG (5.6) because (now $S = \emptyset$ and $M = N$, so $M \cup S = N$):

$$\begin{aligned}\ell_\dagger(\mathbf{x}_{M \cup S}) &\Rightarrow \widehat{R_N}(\mathbf{x}_N) \vdash \ell_\dagger(\mathbf{x}_{M^- \cup N^-}) \\ &\overset{*}{\Rightarrow} R_N^\dagger(\mathbf{x}_N) \vdash \psi_\dagger(\mathbf{x}_{M^- \cup N^-}) && \text{(induction hypothesis)} \\ &= R_N^\dagger(\mathbf{x}_N) \vdash (\mu(\mathbf{x}_{M^-}) \otimes \mu(\mathbf{x}_{N^-})) \\ &= \mu(\mathbf{x}_M) \otimes \psi(\mathbf{x}_S)\end{aligned}$$

where the first yield step here uses the production $\ell_\dagger(\mathbf{x}_N) ::= R_N(\mathbf{x}_N) \vdash \ell_\dagger(\mathbf{x}_{N^-})$, where $S = N$ and $N^- \subseteq N \in \{N \in \mathcal{E}^- \mid n \in U \subset N\}$.

(2.2) (line 4) if there is no $N \supseteq U$ (only $N \subset U$ lower), then in the children of the root, there must be R_U, because R_U can no longer hide lower in any subtrees of the AST, thus let the next yield step of $\ell(\mathbf{x}_M)$ be

$$\ell(\mathbf{x}_M) \Rightarrow R_U(\mathbf{x}_U) \otimes \ell(\mathbf{x}_{M^-}) \overset{*}{\Rightarrow} R_U(\mathbf{x}_U) \otimes \mu(\mathbf{x}_{M^-})$$

where $M^- \subseteq M$. As U is the largest hyperedge used among the next yield steps of $\ell(\mathbf{x}_M)$, we have $M^- \subseteq U$. Then we have

$$\begin{aligned}\ell(\mathbf{x}_M) \otimes \ell(\mathbf{x}_S) &\overset{*}{\Rightarrow} R_U(\mathbf{x}_U) \otimes \ell(\mathbf{x}_{M^-}) \otimes e \\ &\overset{*}{\Rightarrow} R_U(\mathbf{x}_U) \otimes \mu(\mathbf{x}_{M^-}) \otimes e := R_U^{\dagger}(\mathbf{x}_U)\end{aligned}$$

where $R_U^{\dagger}(\mathbf{x}_U) = R_U(\mathbf{x}_U) \otimes \mu(\mathbf{x}_{M^-})$ is an updated positive factor that can be computed in $O(|R_U|)$ time.Recall that we simply return in this case $R_U^{\dagger}(\mathbf{x}_U)$ and the lemma is trivially true. Indeed, the next production of CFG (5.6) to be used is $\ell_{\dagger}(\mathbf{x}_U) ::= R_U(\mathbf{x}_U)$.

Next, we examine the case where the next yield step of $\ell(\mathbf{x}_S)$ is not $\ell(\mathbf{x}_S) \Rightarrow e$. $\mathsf{Refactor}$ prioritizes processing the next yield steps of $\ell(\mathbf{x}_S)$ over those of $\ell(\mathbf{x}_M)$. We distinguish two cases: one for n not being contained in the hyperedge used in the next yield step, and the other for the opposite.

(3) **n is not contained in the hyperedge in the next yield step of $\ell(\mathbf{x}_S)$.** There are two subcases, one for each next yield step of $\ell(\mathbf{x}_S)$:

(3.1) (line 10) $\ell(\mathbf{x}_S) \Rightarrow R_K(\mathbf{x}_K) \otimes \ell(\mathbf{x}_{S^-}) \overset{*}{\Rightarrow} R_K(\mathbf{x}_K) \otimes \psi(\mathbf{x}_{S^-})$, for some $K \in \mathcal{E}^+, S^- \subseteq [n]$ such that $S = K \cup S^-$. If $n \notin K$, then by the induction hypothesis,

$$\ell_{\dagger}(\mathbf{x}_{M \cup S^-}) \overset{*}{\Rightarrow} \psi_{\dagger}(\mathbf{x}_{M \cup S^-}) = \mathsf{Refactor}(\ell(\mathbf{x}_M) \overset{*}{\Rightarrow} \mu(\mathbf{x}_M), \ell(\mathbf{x}_{S^-}) \overset{*}{\Rightarrow} \psi(\mathbf{x}_{S^-}))$$

since $|\eta|$ decrements by 1 (the first yield step $\ell(\mathbf{x}_S) \Rightarrow R_K(\mathbf{x}_K) \otimes \ell(\mathbf{x}_{S^-})$ is trimmed off in the recursive call). In the CFG, we have

$$\begin{aligned}\ell(\mathbf{x}_M) \otimes \ell(\mathbf{x}_S) \Rightarrow\ & \ell(\mathbf{x}_M) \otimes (R_K(\mathbf{x}_K) \otimes \ell(\mathbf{x}_{S^-})) \\ \overset{*}{\Rightarrow}\ & R_K(\mathbf{x}_K) \otimes (\mu(\mathbf{x}_M) \otimes \psi(\mathbf{x}_{S^-})) \\ =\ & R_K(\mathbf{x}_K) \otimes \mathsf{Refactor}(\ell(\mathbf{x}_M) \overset{*}{\Rightarrow} \mu(\mathbf{x}_M), \ell(\mathbf{x}_{S^-}) \overset{*}{\Rightarrow} \psi(\mathbf{x}_{S^-}))\end{aligned}$$

(recall that we return $R_K(\mathbf{x}_K) \otimes \mathsf{Refactor}(\ell(\mathbf{x}_M) \overset{*}{\Rightarrow} \mu(\mathbf{x}_M), \ell(\mathbf{x}_{S^-}) \overset{*}{\Rightarrow} \psi(\mathbf{x}_{S^-})))$. Moreover, $\ell_{\dagger}(\mathbf{x}_{M \cup S})$ derives this expression in the new CFG because

$$\ell_{\dagger}(\mathbf{x}_{M \cup S}) \Rightarrow R_K(\mathbf{x}_K) \otimes \ell_{\dagger}(\mathbf{x}_{M \cup S^-}) \overset{*}{\Rightarrow} R_K(\mathbf{x}_K) \otimes \psi_{\dagger}(\mathbf{x}_{M \cup S^-})$$

where the first yield step here uses the production $\ell_{\dagger}(\mathbf{x}_S) ::= R_K(\mathbf{x}_K) \otimes \ell_{\dagger}(\mathbf{x}_{S^-})$, where $S = K \cup S^-$ and $K \in \{K \in \mathcal{E}^+ \mid n \notin K\}$.

(3.2) (line 17) $\ell(\mathbf{x}_S) \Rightarrow (R_N(\mathbf{x}_N) \vdash \ell(\mathbf{x}_{N^-})) \otimes \ell(\mathbf{x}_{S^-}) \overset{*}{\Rightarrow} (R_N(\mathbf{x}_N) \vdash \mu(\mathbf{x}_{N^-})) \otimes \psi(\mathbf{x}_{S^-})$, for some $N \in \mathcal{E}^-, N^- \subseteq N$, and $S^- \subseteq [n]$ such that $S = N \cup S^-$. If $n \notin N$, then by the induction hypothesis,

$$\ell_{\dagger}(\mathbf{x}_{M \cup S^-}) \overset{*}{\Rightarrow} \psi_{\dagger}(\mathbf{x}_{M \cup S^-}) = \mathsf{Refactor}(\ell(\mathbf{x}_M) \overset{*}{\Rightarrow} \mu(\mathbf{x}_M), \ell(\mathbf{x}_{S^-}) \overset{*}{\Rightarrow} \psi(\mathbf{x}_{S^-}))$$

since $|\eta|$ decrements by 1 (the first yield step $\ell(\mathbf{x}_S) \Rightarrow (R_N(\mathbf{x}_N) \vdash \ell(\mathbf{x}_{N^-})) \otimes \ell(\mathbf{x}_{S^-})$ is trimmed off). Indeed, in the CFG, we have

$$\begin{aligned}\ell(\mathbf{x}_M) \otimes \ell(\mathbf{x}_S) \Rightarrow\ & \ell(\mathbf{x}_M) \otimes (R_N(\mathbf{x}_N) \vdash \ell(\mathbf{x}_{N^-})) \otimes \ell(\mathbf{x}_{S^-}) \\ =\ & (R_N(\mathbf{x}_N) \vdash \ell(\mathbf{x}_{N^-})) \otimes \ell(\mathbf{x}_M) \otimes \ell(\mathbf{x}_{S^-}) \\ \overset{*}{\Rightarrow}\ & (R_N(\mathbf{x}_N) \vdash \mu(\mathbf{x}_{N^-})) \otimes (\mu(\mathbf{x}_M) \otimes \psi(\mathbf{x}_{S^-})) \\ =\ & (R_N(\mathbf{x}_N) \vdash \mu(\mathbf{x}_{N^-})) \otimes \mathsf{Refactor}(\ell(\mathbf{x}_M) \overset{*}{\Rightarrow} \mu(\mathbf{x}_M), \ell(\mathbf{x}_{S^-}) \overset{*}{\Rightarrow} \psi(\mathbf{x}_{S^-}))\end{aligned}$$

(recall that we return $(R_N(\mathbf{x}_N) \vdash \mu(\mathbf{x}_{N^-})) \otimes \mathsf{Refactor}(\ell(\mathbf{x}_M) \overset{*}{\Rightarrow} \mu(\mathbf{x}_M), \ell(\mathbf{x}_{S^-}) \overset{*}{\Rightarrow} \psi(\mathbf{x}_{S^-})))$. Moreover, $\ell_\dagger(\mathbf{x}_{M\cup S})$ derives this expression in the new CFG because

$$\begin{aligned}\ell_\dagger(\mathbf{x}_{M\cup S}) &\Rightarrow (R_N(\mathbf{x}_N) \vdash \ell_\dagger(\mathbf{x}_{N^-})) \otimes \ell_\dagger(\mathbf{x}_{M\cup S^-}) \overset{*}{\Rightarrow} (R_N(\mathbf{x}_N) \vdash \mu_\dagger(\mathbf{x}_{N^-})) \otimes \ell_\dagger(\mathbf{x}_{M\cup S^-}) \\ &\overset{*}{\Rightarrow} (R_N(\mathbf{x}_N) \vdash \mu_\dagger(\mathbf{x}_{N^-})) \otimes \psi_\dagger(\mathbf{x}_{M\cup S^-}).\end{aligned}$$

where (i) the first yield step here uses the production $\ell_\dagger(\mathbf{x}_{M\cup S}) ::= (R_N(\mathbf{x}_N) \vdash \ell_\dagger(\mathbf{x}_{N^-})) \otimes \ell_n(\mathbf{x}_{M\cup S^-})$, where $S = N \cup S^-$ and $N^- \subseteq N \in \{N \in \mathcal{E}^- \mid n \notin N\}$, (ii) the second step is valid because $\ell_\dagger(\mathbf{x}_{N^-}) \overset{*}{\Rightarrow} \mu_n(\mathbf{x}_{N^-})$ in the new CFG as $n \notin N^-$ and (iii) the last derivation follows from the induction hypothesis.

(4) otherwise, n is contained in the hyperedge in the next yield step of $\ell(\mathbf{x}_S)$. In this case, we apply a 'stash' step that temporarily 'stashes' this yield step into $\ell(\mathbf{x}_M)$ and then keep processing the next yield step of $\ell(\mathbf{x}_S)$.

(4.1) (line 13) $\ell(\mathbf{x}_S) \Rightarrow R_K(\mathbf{x}_K) \otimes \ell(\mathbf{x}_{S^-}) \overset{*}{\Rightarrow} R_K(\mathbf{x}_K) \otimes \psi(\mathbf{x}_{S^-})$, for some $K \in \mathcal{E}^+, S^- \subseteq [n]$ such that $S = K \cup S^-$. Now $n \in K$. Let $\mu(\mathbf{x}_{M\cup K}) = R_K(\mathbf{x}_K) \otimes \mu(\mathbf{x}_M)$, we have

$$\begin{aligned}\ell(\mathbf{x}_M) \otimes \ell(\mathbf{x}_S) \Rightarrow\ & \ell(\mathbf{x}_M) \otimes R_K(\mathbf{x}_K) \otimes \ell(\mathbf{x}_{S^-}) \\ \overset{*}{\Rightarrow}\ & \mu(\mathbf{x}_M) \otimes R_K(\mathbf{x}_K) \otimes \psi(\mathbf{x}_{S^-}) \\ =\ & \mu(\mathbf{x}_{M\cup K}) \otimes \psi(\mathbf{x}_{S^-})\end{aligned}$$

We insert a new yield step for R_K into the stashed derivation $\ell(\mathbf{x}_M) \overset{*}{\Rightarrow} \mu(\mathbf{x}_M)$ and keep track of the largest hyperedge to be used as the next yield step of $\ell(\mathbf{x}_M)$ in the augmented derivation. W.L.O.G, the agumented derivation becomes

$$\ell(\mathbf{x}_{M\cup K}) \Rightarrow R_K(\mathbf{x}_K) \otimes \ell(\mathbf{x}_M) \overset{*}{\Rightarrow} R_K(\mathbf{x}_K) \otimes \mu(\mathbf{x}_M) = \mu(\mathbf{x}_{M\cup K})$$

and then we return $\mathsf{Refactor}\Big(\ell(\mathbf{x}_{M\cup K}) \overset{*}{\Rightarrow} \mu(\mathbf{x}_{M\cup K}), \ell(\mathbf{x}_{S^-}) \overset{*}{\Rightarrow} \psi(\mathbf{x}_{S^-})\Big)$.

(4.2) (line 19) $\ell(\mathbf{x}_S) \Rightarrow (R_N(\mathbf{x}_N) \vdash \ell(\mathbf{x}_{N^-})) \otimes \ell(\mathbf{x}_{S^-}) \overset{*}{\Rightarrow} (R_N(\mathbf{x}_N) \vdash \mu(\mathbf{x}_{N^-})) \otimes \psi(\mathbf{x}_{S^-})$, for some $N \in \mathcal{E}^-, N^- \subseteq N$, and $S^- \subseteq [n]$ such that $S = N \cup S^-$. Now $n \in N$. Let $\mu(\mathbf{x}_{M\cup N}) = (R_N(\mathbf{x}_N) \vdash \mu(\mathbf{x}_{N^-})) \otimes \mu(\mathbf{x}_M)$, we have

$$\begin{aligned}\ell(\mathbf{x}_M) \otimes \ell(\mathbf{x}_S) \Rightarrow\ & \ell(\mathbf{x}_M) \otimes (R_N(\mathbf{x}_N) \vdash \ell(\mathbf{x}_{N^-})) \otimes \ell(\mathbf{x}_{S^-}) \\ \overset{*}{\Rightarrow}\ & \mu(\mathbf{x}_M) \otimes (R_N(\mathbf{x}_N) \vdash \mu(\mathbf{x}_{N^-})) \otimes \psi(\mathbf{x}_{S^-}) \\ =\ & [(R_N(\mathbf{x}_N) \vdash \mu(\mathbf{x}_{N^-})) \otimes \mu(\mathbf{x}_M)] \otimes \psi(\mathbf{x}_{S^-}) \\ =\ & \mu(\mathbf{x}_{M\cup N}) \otimes \mu(\mathbf{x}_{S^-})\end{aligned}$$

$$\ell(\mathbf{x}_{M\cup N}) \Rightarrow (R_N(\mathbf{x}_N) \vdash \ell(\mathbf{x}_{N^-})) \otimes \ell(\mathbf{x}_M) \overset{*}{\Rightarrow} (R_N(\mathbf{x}_N) \vdash \mu(\mathbf{x}_{N^-})) \otimes \mu(\mathbf{x}_M) = \mu(\mathbf{x}_{M\cup N})$$

and then we return $\mathsf{Refactor}\Big(\ell(\mathbf{x}_{M\cup N}) \overset{*}{\Rightarrow} \mu(\mathbf{x}_{M\cup N}), \ell(\mathbf{x}_{S^-}) \overset{*}{\Rightarrow} \psi(\mathbf{x}_{S^-})\Big)$.

Case (4) is the only if-else branch where $|\eta|$ stays the same in the recursive call, because this step essentially 'stashes' the next yield step of $\ell(\mathbf{x}_S)$ into the derivation for $\ell(\mathbf{x}_M)$. Therefore, the length of the derivation for $\psi(\mathbf{x}_S)$ decrements by 1 but that of $\mu(\mathbf{x}_M)$ increments by 1 in the recursive call. However, in the worst-case, this if-else branch can be visited at most $O(|\eta|)$ times, each visit incurs an $O(|\eta|)$ time overhead to place the yield step of R_N into the stashed derivation $\ell(\mathbf{x}_{M\cup N}) \overset{*}{\Rightarrow} \mu(\mathbf{x}_{M\cup N})$ so that the largest hyperedge can be directly accessed upon request. Eventually, Refactor will recurse back to one of the previous cases and $|\eta|$ will decrement by 1 in that subsequent recursive call. As a result, the correctness of the lemma in this case simply follows through from the correctness of $\mathsf{Refactor}\Big(\ell(\mathbf{x}_{M\cup N}) \overset{*}{\Rightarrow} \mu(\mathbf{x}_{M\cup N}), \ell(\mathbf{x}_{S^-}) \overset{*}{\Rightarrow} \psi(\mathbf{x}_{S^-})\Big)$.

Lastly, we justify that Refactor runs in the desired time. The Refactor algorithm recursively visits each yield step in both derivations $\ell(\mathbf{x}_M) \overset{*}{\Rightarrow} \mu(\mathbf{x}_M)$ and $\ell(\mathbf{x}_S) \overset{*}{\Rightarrow} \psi(\mathbf{x}_S)$ at most twice (accounting for "stashes"). For each visit, if it does not fall into the last stash case (4), $|\eta|$ decrements by 1, and Refactor takes $O(|\mathcal{D}_\psi|)$ time to update the factors. Furthermore, every such update leverages the α or β properties of the signed-leaf n to guarantee that $|\mathcal{D}_{\psi_\dagger}| \leq |\mathcal{D}_\psi|$ and the update can only happen once for each hyperedge containing n (since that corresponding factor does not appear in the subsequent recursive call). Thus, all updates leading to $\mathcal{D}_{\psi_\dagger}$ take $O(d(n) \cdot |\mathcal{D}_\psi|)$ time in total.

For the last case (4), as discussed in paragraph of case (4), Refactor can fall into it for at most $O(|\eta|)$ times, each time with a $O(|\eta|)$ time overhead before the next recursive call (for the 'stashes'). Therefore, the total time complexity of Refactor is $O(|\eta|^2 + d(n) \cdot |\mathcal{D}_\psi|)$. □

The following theorem is a direct consequence of Lemma 5.6.

Theorem 5.5. *There is an algorithm that takes as input an* NestFAQ$^\neg$ *expression* $\psi(\mathbf{x}_{[n]})$ *associated with* $\mathcal{H}$*, thus recognized by CFG* (5.4)*, and returns a* NestFAQ$^\neg$ *expression* $\psi_\dagger(\mathbf{x}_{[n]})$ *associated with* $\mathcal{H}_n$*,*

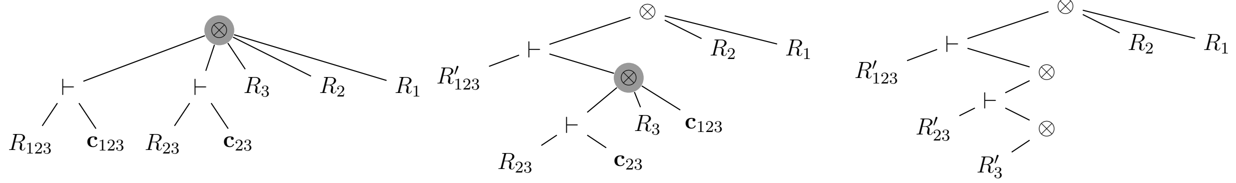


Figure 5.4: The refactor steps on φ for the signed-leaf 3, recursively from left to right. The shaded $\otimes$ nodes indicate the subtrees to be recursively refactored. The left is the original expression representation of φ, and the right is its refactored expression.

thus recognized by CFG (5.6)*, such that* $\psi(\mathbf{x}_{[n]}) = \psi_\dagger(\mathbf{x}_{[n]})$*,* $|\psi_\dagger| \leq |\psi|$ *and* $|\mathcal{D}_{\psi_\dagger}| \leq |\mathcal{D}_\psi|$*. Moreover, the algorithm runs in time* $O(|\psi|^2 + d(n) \cdot |\mathcal{D}_\psi|)$*, where* $d(n)$ *is the number of hyperedges containing* n *in* $\mathcal{H}$*.*

Proof. We call $\mathsf{Refactor}\left(\ell(\mathbf{x}_\emptyset) \Rightarrow \mathbf{1}, \ell(\mathbf{x}_{[n]}) \overset{*}{\Rightarrow} \psi(\mathbf{x}_{[n]})\right)$ and trace the recursive call steps of Refactor to construct the derivation $\ell_\dagger(\mathbf{x}_{[n]}) \overset{*}{\Rightarrow} \psi_\dagger(\mathbf{x}_{[n]})$ using the production rules of CFG (5.6). We complete the proof by invoking Lemma 5.6. □

Example 5.8. We illustrate the refactor step in our running example: it reorganizes φ to make x_3 present only in one subtree rooted at a child of the root $\otimes$ node. First, it replaces R_{123} with a new factor $R'_{123} := R_{123} \otimes (R_{23} \vdash \mathbf{c}_{23}) \otimes R_3$, the entry table of which is exactly that of R_{123}, except for multiplying the extra weight $(R_{23} \vdash \mathbf{c}_{23}) \otimes R_3$ to each table entry. Then, we get a new expression in Figure 5.4 (middle) since

$$(R_{123} \vdash \mathbf{c}_{123}) \otimes (R_{23} \vdash \mathbf{c}_{23}) \otimes R_3 = R'_{123} \vdash ((R_{23} \vdash \mathbf{c}_{23}) \otimes R_3 \otimes \mathbf{c}_{123})\,.$$

We keep recursing on the subtree for $(R_{23} \vdash \mathbf{c}_{23}) \otimes R_3 \otimes \mathbf{c}_{123}$, leading to the desired expression in Figure 5.4 (right) with new factors $R'_{23} := R_{23} \otimes (R_3 \vdash \mathbf{c}_3)$ and $R'_3 := R_3 \otimes \mathbf{c}_{123} \otimes \mathbf{c}_{23}$ computed. Indeed,

$$(R_{23} \vdash \mathbf{c}_{23}) \otimes R_3 \otimes \mathbf{c}_{123} = R'_{23} \vdash (R_3 \otimes \mathbf{c}_{123} \otimes \mathbf{c}_{23}) = R'_{23} \vdash R'_3.$$

As we go, R_3 sinks down and the linear inclusion $[3] \supseteq \{2,3\} \supseteq \{3\}$ surfaces along the subtrees that contain x_3. In the end, R_3 absorbs the constants $\mathbf{c}_{123}$ and $\mathbf{c}_{23}$. The refactor step runs in linear time with no query (or database) size blow-up.

5.5.4 The Aggregation Algorithm

We assume in this section that we have obtained $\psi_\dagger(\mathbf{x}_{[n]})$ from the call of Refactor as in Theorem 5.5. We now shift our focus directly to the aggregation algorithm that eliminates x_n, where

n is the signed-leaf of the signed hypergraph $\mathcal{H}$. We defer the oracle-construction algorithm to Section 5.5.5 for now because the aggregation algorithm will provide an intuitive motivation that necessitates the oracles to be constructed. The aggregation algorithm takes as input $\psi_\dagger(\mathbf{x}_{[n]})$ and returns a new $\mathsf{NestFAQ}^\neg$ expression $\psi_{n-1}(\mathbf{x}_{[n-1]})$ recognized by $\mathsf{CFG}_{\mathcal{H}_{n-1}}$ such that $\bigoplus_{x_n} \psi_n(\mathbf{x}_{[n]}) = \psi_{n-1}(\mathbf{x}_{[n-1]})$.

The main benefit of turning the $\mathsf{NestFAQ}^\neg$ expression $\psi(\mathbf{x}_{[n]})$ into $\psi_\dagger(\mathbf{x}_{[n]})$ is that the structure of $\psi_\dagger(\mathbf{x}_{[n]})$ respects the β-property of the signed-leaf n. Indeed, w.l.o.g assume by β-property that $R_{N_k}, R_{N_{k-1}}, \cdots, R_{N_1}, R_U$ are the only factors of $\psi_\dagger(\mathbf{x}_{[n]})$ that contain x_n and $N_k \supseteq N_{k-1} \supseteq \cdots \supseteq N_1 \supseteq U$ (we let $N_0 = U$ for convenience). Then, these factors occur along a root-to-leaf path in the tree representation of $\psi_\dagger(\mathbf{x}_{[n]})$ (see Figure 5.4 (right) for R'_{123}, R'_{23} and R'_3 that contain x_3). Using this observation, a convenient way to derive $\psi_\dagger(\mathbf{x}_{[n]})$ is as follows (let $N_i \supseteq N_i^- = S_{i-1} \cup N_{i-1}$ for $i \in [k]$): $\quad \ell_\dagger(\mathbf{x}_{[n]}) \overset{*}{\Rightarrow} \psi_\dagger(\mathbf{x}_{S_k}) \otimes \ell_\dagger(\mathbf{x}_{N_k})$

$$
\begin{aligned}
&\Rightarrow \psi_\dagger(\mathbf{x}_{S_k}) \otimes \Big(R_{N_k}(\mathbf{x}_{N_k}) \vdash \ell_\dagger(\mathbf{x}_{N_k^-})\Big) \\
&\overset{*}{\Rightarrow} \psi_\dagger(\mathbf{x}_{S_k}) \otimes (R_{N_k}(\mathbf{x}_{N_k}) \vdash \psi_\dagger(\mathbf{x}_{S_{k-1}}) \otimes \ell_\dagger(\mathbf{x}_{N_{k-1}})) \\
&\Rightarrow \psi_\dagger(\mathbf{x}_{S_k}) \otimes \Big(R_{N_k}(\mathbf{x}_{N_k}) \vdash \psi_\dagger(\mathbf{x}_{S_{k-1}}) \otimes \Big(R_{N_{k-1}}(\mathbf{x}_{N_{k-1}}) \vdash \ell_\dagger(\mathbf{x}_{N_{k-1}^-})\Big)\Big) \\
&\cdots \\
&\Rightarrow \psi_\dagger(\mathbf{x}_{S_k}) \otimes \Big(R_{N_k}(\mathbf{x}_{N_k}) \vdash \psi_\dagger(\mathbf{x}_{S_{k-1}}) \otimes \Big(R_{N_{k-1}}(\mathbf{x}_{N_{k-1}}) \vdash \cdots \Big(R_{N_1}(\mathbf{x}_{N_1}) \vdash \ell_\dagger(\mathbf{x}_{N_1^-})\Big) \cdots \Big)\Big) \\
&\Rightarrow \psi_\dagger(\mathbf{x}_{S_k}) \otimes (R_{N_k}(\mathbf{x}_{N_k}) \vdash \psi_\dagger(\mathbf{x}_{S_{k-1}}) \otimes (R_{N_{k-1}}(\mathbf{x}_{N_{k-1}}) \vdash \cdots (R_{N_1}(\mathbf{x}_{N_1}) \vdash \psi_\dagger(\mathbf{x}_{S_0}) \otimes \ell_\dagger(\mathbf{x}_U)) \cdots)) \\
&\Rightarrow \psi_\dagger(\mathbf{x}_{S_k}) \otimes (R_{N_k}(\mathbf{x}_{N_k}) \vdash \psi_\dagger(\mathbf{x}_{S_{k-1}}) \otimes (R_{N_{k-1}}(\mathbf{x}_{N_{k-1}}) \vdash \cdots (R_{N_1}(\mathbf{x}_{N_1}) \vdash \psi_\dagger(\mathbf{x}_{S_0}) \otimes R_U(\mathbf{x}_U)) \cdots))
\end{aligned} \tag{5.12}
$$

Here, every $\overset{*}{\Rightarrow}$ (derive step) in (5.12) applies the productions (5.7), (5.9) or (5.11) to factor out all the terms ($e, R_K(\mathbf{x}_K)$ or $R_N(\mathbf{x}_N) \vdash \ell_n(\mathbf{x}_{N^-})$) that do not contain x_n and wrap them in some $\psi_\dagger(\mathbf{x}_{S_i})$ such that $\ell_\dagger(\mathbf{x}_{S_i}) \overset{*}{\Rightarrow} \psi_\dagger(\mathbf{x}_{S_i})$, where $i \in [k]$ and $n \notin S_i$. Then it is followed by a yield step that applies the production (5.10) that produces the negative factors $R_{N_k}, R_{N_{k-1}}, \cdots, R_{N_1}$ from $\ell_\dagger(\mathbf{x}_{N_k}), \ell_\dagger(\mathbf{x}_{N_{k-1}}), \cdots, \ell_\dagger(\mathbf{x}_{N_1})$, except for the last yield step where we use the production (5.8) to produce the pivot factor $R_U(\mathbf{x}_U)$ from $\ell_\dagger(\mathbf{x}_U)$.

At a high level, our aggregation algorithm will follow the derivation in (5.12) line-by-line and "push-in" the aggregation operator $\bigoplus_{x_n}$ into the more and more nested subexpressions. This "push-in" step for every derive step is immediate: we can simply push the aggregation operator $\bigoplus_{x_n}$ by factoring out the term $\psi_\dagger(\mathbf{x}_{S_i})$ that does not contain x_n. That is, for $i = 0, 1, \ldots, k$,

$$
\bigoplus_{x_n} \psi_\dagger(\mathbf{x}_{S_i}) \otimes \Big(R_{N_i}(\mathbf{x}_{N_i}) \vdash \psi_\dagger(\mathbf{x}_{N_i^-})\Big) = \psi_\dagger(\mathbf{x}_{S_i}) \otimes \bigoplus_{x_n} \Big(R_{N_i}(\mathbf{x}_{N_i}) \vdash \psi_\dagger(\mathbf{x}_{N_i^-})\Big)
$$

$$
\bigoplus_{x_n} \psi_n(\mathbf{x}_{S_0}) \otimes R_U(\mathbf{x}_U) = \psi_\dagger(\mathbf{x}_{S_0}) \otimes \bigoplus_{x_n} R_U(\mathbf{x}_U) = \psi_\dagger(\mathbf{x}_{S_0}) \otimes R_{U \setminus \{n\}}(\mathbf{x}_{U \setminus \{n\}})
$$

In the last yield step (applying the production (5.8)), computing $\bigoplus_{x_n} R_U(\mathbf{x}_U)$ is straightforward: we scan the list representation of the pivot factor R_U once and get a new factor $R_{U\setminus\{n\}}(\mathbf{x}_{U\setminus\{n\}})$ in time $O(|R_U|)$ by summing the appropriate rows.

Now we turn to the tricky yield steps in the derivation (5.12) that apply the production (5.10), that is, we want to aggregate $\bigoplus_{x_n} R_N(\mathbf{x}_N) \vdash \psi_\dagger(\mathbf{x}_{N^-})$. Let us define a new factor $R_{N\setminus\{n\}}$ as follows (its list representation is obviously of size $O(|R_N|)$):

$$R_{N\setminus\{n\}}(\mathbf{a}_{N\setminus\{n\}}) = \begin{cases} \bigoplus_{x_n} R_N(x_n, \mathbf{a}_{N\setminus\{n\}}) \vdash \psi_\dagger(x_n, \mathbf{a}_{N^-\setminus\{n\}}) & \text{if } \mathbf{a}_{N\setminus\{n\}} \in \Pi_{N\setminus\{n\}} R_N \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

Now, we can write:

$$\begin{aligned} \bigoplus_{x_n} \psi_\dagger(\mathbf{x}_N) &= \bigoplus_{x_n} (R_N(\mathbf{x}_N) \vdash \psi_\dagger(\mathbf{x}_{N^-})) \\ &= \bigoplus_{x_n} (R_N(\mathbf{x}_N) \oplus \mathbb{1}_{\neg R_N}(\mathbf{x}_N) \otimes \psi_\dagger(\mathbf{x}_{N^-})) \\ &= R_{N\setminus\{n\}}(\mathbf{x}_{N\setminus\{n\}}) \oplus \mathbb{1}_{\neg R_{N\setminus\{n\}}}(\mathbf{x}_{N\setminus\{n\}}) \otimes \bigoplus_{x_n} \psi_\dagger(\mathbf{x}_{N^-}) \\ &= R_{N\setminus\{n\}}(\mathbf{x}_{N\setminus\{n\}}) \vdash \bigoplus_{x_n} \psi_\dagger(\mathbf{x}_{N^-}) \end{aligned}$$

Indeed, if $\mathbf{a}_{N\setminus\{n\}} \notin \Pi_{N\setminus\{n\}} R_N$, then the aggregation becomes $\bigoplus_{x_n} \psi_\dagger(\mathbf{x}_{N^-})$ as $R_N(x_n, \mathbf{a}_{N\setminus\{n\}}) = \mathbf{0}$ and $\mathbb{1}_{\neg R_N}(x_n, \mathbf{a}_{N\setminus\{n\}}) = \mathbf{1}$, for all $x_n \in \mathsf{Dom}(x_n)$. Otherwise, we have $\mathbf{a}_{N\setminus\{n\}} \in \Pi_{N\setminus\{n\}} R_N$, which implies that the indicator function $\mathbb{1}_{\neg R_{N\setminus\{n\}}}(\mathbf{a}_{N\setminus\{n\}})$ is $\mathbf{0}$. Therefore, if we can obtain (the list representation of) this new factor $R_{N\setminus\{n\}}$ efficiently, we can "push-in" the aggregation operator one

line at a time following the derivation (5.12), as we show next: $\bigoplus_{x_n} \psi_n(\mathbf{x}_{[n]}) =$

$$
\begin{aligned}
&\psi_\dagger(\mathbf{x}_{S_k}) \otimes \bigoplus_{x_n} \psi_\dagger(\mathbf{x}_{N_k}) \\
&= \psi_\dagger(\mathbf{x}_{S_k}) \otimes \left(R_{N_k \setminus \{n\}}(\mathbf{x}_{N_k \setminus \{n\}}) \vdash \bigoplus_{x_n} \psi_\dagger(\mathbf{x}_{N_k^-}) \right) \\
&= \psi_\dagger(\mathbf{x}_{S_k}) \otimes \left(R_{N_k \setminus \{n\}}(\mathbf{x}_{N_k \setminus \{n\}}) \vdash \psi_\dagger(\mathbf{x}_{S_{k-1}}) \otimes \bigoplus_{x_n} \psi_\dagger(\mathbf{x}_{N_{k-1}}) \right) \\
&= \psi_\dagger(\mathbf{x}_{S_k}) \otimes \left(R_{N_k \setminus \{n\}}(\mathbf{x}_{N_k \setminus \{n\}}) \vdash \psi_\dagger(\mathbf{x}_{S_{k-1}}) \otimes \left(R_{N_{k-1} \setminus \{n\}}(\mathbf{x}_{N_{k-1} \setminus \{n\}}) \vdash \bigoplus_{x_n} \psi_\dagger(\mathbf{x}_{N_{k-1}^-}) \right) \right) \\
&\cdots \\
&= \psi_\dagger(\mathbf{x}_{S_k}) \otimes \left(R_{N_k \setminus \{n\}}(\mathbf{x}_{N_k \setminus \{n\}}) \vdash \psi_\dagger(\mathbf{x}_{S_{k-1}}) \otimes \cdots \left(R_{N_1}(\mathbf{x}_{N_1}) \vdash \bigoplus_{x_n} \psi_\dagger(\mathbf{x}_{N_1^-}) \right) \cdots \right) \\
&= \psi_\dagger(\mathbf{x}_{S_k}) \otimes \left(R_{N_k \setminus \{n\}}(\mathbf{x}_{N_k \setminus \{n\}}) \vdash \psi_\dagger(\mathbf{x}_{S_{k-1}}) \otimes \cdots \left(R_{N_1}(\mathbf{x}_{N_1}) \vdash \psi_n(\mathbf{x}_{S_0}) \otimes \bigoplus_{x_n} \psi_\dagger(\mathbf{x}_U) \right) \cdots \right) \\
&= \psi_\dagger(\mathbf{x}_{S_k}) \otimes \left(R_{N_k \setminus \{n\}}(\mathbf{x}_{N_k \setminus \{n\}}) \vdash \psi_\dagger(\mathbf{x}_{S_{k-1}}) \otimes \cdots \left(R_{N_1}(\mathbf{x}_{N_1}) \vdash \psi_n(\mathbf{x}_{S_0}) \otimes \bigoplus_{x_n} R_U(\mathbf{x}_U) \right) \cdots \right) \\
&=: \psi_{n-1}(\mathbf{x}_{[n-1]})
\end{aligned}
\tag{5.13}
$$

As one can see from the comparison with (5.12), the derivation of $\psi_{n-1}(\mathbf{x}_{[n-1]})$ exactly follows the derivation of $\psi_n(\mathbf{x}_{[n]})$ except that x_n has been peeled off from the set of variables and every $\mathsf{NestFAQ}^\neg$ subexpression show up in $\psi_n(\mathbf{x}_{[n]})$. So, $\psi_{n-1}(\mathbf{x}_{[n-1]})$ is a $\mathsf{NestFAQ}^\neg$ expression recognized by $\mathsf{CFG}_{\mathcal{H}_{n-1}}$. Indeed, recall that $\mathsf{CFG}_{\mathcal{H}_{n-1}}$ is associated with the signed hypergraph $\mathcal{H}_{n-1} = \langle \mathcal{H}, n \rangle = ([n-1], \mathcal{E}^+_{n-1}, \mathcal{E}^-_{n-1})$. Then, $U \setminus \{n\} \in \mathcal{E}^+_{n-1}$, $\{K \in \mathcal{E}^+ \mid n \notin K\} \subseteq \mathcal{E}^+_{n-1}$ and $\{N \setminus \{n\} \mid N \in \mathcal{E}^-, n \notin N \vee n \in U \subset N\} \subseteq \mathcal{E}^-_{n-1}$. Therefore, the $\mathsf{NestFAQ}^\neg$ expression $\psi_{n-1}(\mathbf{x}_{[n-1]})$ is also recognized by $\mathsf{CFG}_{\mathcal{H}_{n-1}}$ at the end of the elimination step for the signed-leaf n.

Example 5.9. We continue our running example, and go over the aggregation step that effectively peels off x_3 from the refactored expression. To that end, we push the aggregation operator $\bigoplus_{x_3}$ downward the refactored tree from root to $R'_3(x_3)$. Start with the root $\otimes$ node, we will construct new factors $R''_{12}(x_1, x_2)$ and $R''_2(x_2)$ such that:

$$
\bigoplus_{x_3} R'_{123} \vdash (R'_{23} \vdash R'_3) = R''_{12} \vdash \bigoplus_{x_3} (R'_{23} \vdash R'_3) = R''_{12} \vdash (R''_2 \vdash \bigoplus_{x_3} R'_3) = R''_{12} \vdash (R''_2 \vdash (\bigoplus_{x_3} R'_3)).
$$

Indeed, the first equality holds because for all values of (x_1, x_2) not encoded in the table of R''_{12}, we always have $R'_{123} = \mathbf{0}$ and $\mathbb{1}_{\neg R'_{123}} = \mathbf{1}$, therefore for those (x_1, x_2)s, $R'_{123} \vdash (R'_{23} \vdash R'_3) = \mathbf{0} \oplus \mathbf{1} \otimes (R'_{23} \vdash R'_3) = R'_{23} \vdash R'_3$. The second equality holds because for all values of x_2 not encoded in the table of

R_2'', we have $R_{23}' = \mathbf{0}$ and $\mathbb{1}_{\neg R_{23}'} = \mathbf{1}$. Thus, $R_{23}' \vdash R_3' = \mathbf{0} \oplus \mathbf{1} \otimes R_3' = R_3'$. The resulting $\textsf{NestFAQ}^{\neg}$ query φ'' after the signed-elimination on 3 is

$$\varphi''(x_1, x_2) = R_1(x_1) \otimes R_2(x_2) \otimes (R_{12}''(x_1, x_2) \vdash (R_2''(x_2) \vdash c))$$

where c is a constant value such that $c = \bigoplus_{x_3} R_3'(x_3)$. The tree representation of φ'' is exactly the refactored expression in Figure 5.4 (right), except that x_3 is being eliminated. Now the signed-elimination step is complete.

The only missing piece is the computation of the list representation of the new factor $R_{N\setminus\{n\}}$. We will next introduce an efficient algorithm for this task using RangeSum data structures.

5.5.5 The Oracle-construction Algorithm

Chazelle and Rosenberg [CR89] studied the range query problem RangeSum in the semigroup model defined as follows. Preprocess a array A of w elements from a semigroup $\sigma = (\Sigma, \oplus)$, and then support the following query: given a query range $[\omega^-, \omega^+]$, return the range sum $\bigoplus_{\omega \in [\omega^-, \omega^+]} A[\omega]$, i.e. $\bigoplus_{\omega^- \leq \omega \leq \omega^+} A[\omega]$. In particular, they proved the following theorem.

Theorem 5.6 ([CR89])**.** *There is a data structure of size $O(w)$ that works in the word RAM, and support a RangeSum query in $O(\alpha(14w, w))$ time, where α is the inverse Ackermann function, after spending $O(w)$ preprocessing time.*

It is the absence of additive inverse for general semirings that results in the hardness of RangeSum, and an unconditional hard instance can be recursively constructed that mimics the definition of the inverse Ackermann function $\alpha(\cdot, \cdot)$ as in [CR89]. RangeSum queries can be answered in $O(1)$ time after $O(w)$ preprocessing if the underlying semigroup σ allows for additive inverse: one candidate algorithm can precompute the partial sums $\bigoplus_{\omega \in [1, \omega^+]} A[\omega]$ and represent each RangeSum query $[\omega^-, \omega^+]$ as the semigroup sum of $\bigoplus_{\omega \in [1, \omega^+]} A[\omega]$ and the additive inverse of $\bigoplus_{\omega \in [1, \omega^- - 1]} A[\omega]$.

The RangeSum problem with minimum as the semigroup operation is studied intensively and typically known as RangeMinimumQuery (RMQ). Gabow, Bentley and Tarjan [GBT84] have shown that there is an algorithm that works in the word RAM and supports a RMQ query in $O(1)$ time, after spending $O(w)$ preprocessing time and space. They recognized the Cartesian tree as the instrumental data structure that was introduced by Vuillemin [Vui80] in the context of average time analysis of searching.

In this section, we introduce a key oracle (called RangeSumOrcale) that supports efficient computation of the new negative factors $R_{N\setminus\{n\}}$ in the aggregation algorithm introduced in (5.13) of the last section. In particular, RangeSumOrcale uses the RangeSum data structures as a black-box, thus

inherits the inverse Ackermann factor of Theorem 5.6 under general semirings. Before we introduce the formal construction, we present an example.

Example 5.10. We assume the counting ring and on the refactored tree in Figure 5.4 (right), let R'_3 store $\{\langle i, 1\rangle, i \in [15]\}$, R'_{23} store $\{\langle(\mathsf{a}_1, 3), 2\rangle, \langle(\mathsf{a}_1, 9), 2\rangle, \langle(\mathsf{a}_2, 6), 3\rangle, \langle(\mathsf{a}_2, 11), 3\rangle, \langle(\mathsf{a}_3, 8), 4\rangle\}$ and R'_{123} store $\langle(\mathsf{b}_1, \mathsf{a}_1, 4), 5\rangle, \langle(\mathsf{b}_1, \mathsf{a}_1, 15), 5\rangle, \langle(\mathsf{b}_2, \mathsf{a}_1, 3), 6\rangle$, $\langle(\mathsf{b}_2, \mathsf{a}_2, 12), 6\rangle$.

The oracle-construction step starts from an array representation of the database instance drawn in Figure 5.5, indexed by all possible values of x_3. As an example, the tuple $\langle(\mathsf{b}_1, \mathsf{a}_1, 4), 5\rangle$ of R'_{123} can be accessed as the 4-th element of the array $R'_{123}(\mathsf{b}_1, \mathsf{a}_1, x_3)$. The necessity of RangeSum becomes natural: after factoring out the term $R_1(\mathsf{b}_1) \otimes R_2(\mathsf{a}_1)$ (independent of x_3), the aggregation for $x_1 = \mathsf{b}_1, x_2 = \mathsf{a}_1$, from the lens of the array $R'_{123}(\mathsf{b}_1, \mathsf{a}_1, x_3)$, is

$$\begin{gathered} \bigoplus_{x_3} R'_{123}(\mathsf{b}_1, \mathsf{a}_1, x_3) \vdash (R'_{23}(\mathsf{a}_1, x_3) \vdash R'_3(x_3)) = \sum_{1 \le x_3 \le 2} 1 + \\ \sum_{x_3=3} 2 + \sum_{x_3=4} 5 + \sum_{5 \le x_3 \le 8} 1 + \sum_{x_3=9} 2 + \sum_{10 \le x_3 \le 14} 1 + \sum_{x_3=15} 5 \end{gathered} \tag{5.14}$$

where each term is a sum over a range. To support fast aggregation, we construct a RangeSum oracle on the array of $R'_3(x_3)$. Then, we build a RangeSum oracle on the array of $R'_{23}(\mathsf{a}_1, x_3)$ and $R'_{23}(\mathsf{a}_2, x_3)$. In particular, we query the oracle of $R'_3(x_3)$ to fill in the partial sums (as in the 3rd, 4th row in Figure 5.5) and then construct RangeSum oracles over the partial sums. Take $R'_{23}(\mathsf{a}_1, x_3)$ as the example, $[5, 8] : \mathbf{4}$ denotes querying the oracle $R'_3(x_3)$ with $5 \le x_3 \le 8$ as range and getting back the value $\mathbf{4}$. Then the RangeSum oracle for $R'_{23}(\mathsf{a}_1, x_3)$ is constructed over the array (of partial sums):

$$[1, 2] : \mathbf{2} \quad \mathbf{2} \quad [4, 4] : \mathbf{1} \quad [5, 8] : \mathbf{4} \quad \mathbf{2} \quad [10, 14] : \mathbf{5}. \quad [15, 15] : \mathbf{1}$$

where the ranges above split at $x_3 = 3, 9$ and $x_3 = 4, 15$ in account for $R'_{23}(\mathsf{a}_1, x_3)$ and $R'_{123}(\mathsf{b}_1, \mathsf{a}_1, x_3)$, respectively. The rationale here is to avoid mis-aligned ranges when querying the oracle of $R'_{23}(\mathsf{a}_1, x_3)$ during the oracle-construction of $R'_{123}(\mathsf{b}_1, \mathsf{a}_1, x_3)$. The details are in the 6th row of Figure 5.5. The oracle-construction of $R'_{23}(\mathsf{a}_2, x_3)$ and $R'_{123}(\mathsf{b}_2, \mathsf{a}_2, x_3)$ are similarly depicted in the 4th and 7th row of Figure 5.5. It is worth noting that the splits of ranges should be choreographed for query range alignment, and yet can not be arbitrarily fine-grained due to possible blow-ups in time and space. Indeed, we formally show in Section 5.5.5 that desired oracles can be carefully constructed in linear time and space.

To realize the first equality, we ask the table of $R''_{12}(x_1, x_2)$ to store two entries, $\langle(\mathsf{b}_1, \mathsf{a}_1), 25\rangle$ and $\langle(\mathsf{b}_2, \mathsf{a}_2), 29\rangle$, where 25 is the aggregation for $x_1 = \mathsf{b}_1, x_2 = \mathsf{a}_1$ in (5.14), that can be obtained by a simple range query $[1, 15]$ on the oracle of $R'_{123}(\mathsf{b}_1, \mathsf{a}_1, x_3)$. Similarly for 29 from a $[1, 15]$ query on the oracle of $R'_{123}(\mathsf{b}_2, \mathsf{a}_2, x_3)$.

[1, 15]	x_3	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
15	$R'_3(x_3)$	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
	$R'_{23}(\mathsf{a}_1, x_3)$	[1, 2]:**2**		**2**	[4, 4]:**1**	[5, 8]:**4**				**2**	[10, 14]:**5**					[15, 15]:**1**
	$R'_{23}(\mathsf{a}_2, x_3)$	[1, 2]:**2**		[3, 3]:**1**	[4, 5]:**2**		**3**	[7, 10]:**4**				**3**	[12, 12]:**1**	[13, 15]:**3**		
18	$R'_{23}(\mathsf{a}_3, x_3)$	[1, 7]:**7**							**4**	[9, 15]:**7**						
25	$R'_{123}(\mathsf{b}_1, \mathsf{a}_1, x_3)$	[1, 3]: **4**			**5**	[5, 14]: **11**										**5**
29	$R'_{123}(\mathsf{b}_2, \mathsf{a}_2, x_3)$	[1, 2]:**2**		**6**	[4, 11]:**12**								**6**	[13, 15]:**3**		

Figure 5.5: The oracle-construction steps on the refactored AST. The oracle of $R'_{23}(\mathsf{a}_1, x_3)$, $R'_{23}(\mathsf{a}_2, x_3)$ and $R'_{23}(\mathsf{a}_3, x_3)$ are built atop the oracle of $R'_3(x_3)$; the oracle of $R'_{123}(\mathsf{b}_1, \mathsf{a}_1, x_3)$ and $R'_{123}(\mathsf{b}_2, \mathsf{a}_2, x_3)$ are built atop the oracle of $R'_{23}(\mathsf{a}_1, x_3)$ and $R'_{23}(\mathsf{a}_2, x_3)$, respectively. The leftmost column shows the requested range sums over $[1, 15]$ in the aggregation step.

Now that we have pushed $\bigoplus_{x_3}$ to $(R'_{23} \vdash R'_3)$ (i.e. the $\otimes$ node on the 2nd level of the refactored AST) and we insert a single tuple $\langle(\mathsf{a}_3), 18\rangle$ into the new factor $R''_2(x_2)$, where 18 is yet another $[1, 15]$ query on the oracle of $R'_{23}(\mathsf{a}_3, x_3)$.

Range Oracle. Let A be an an array of size w over a semigroup $\sigma = (\Sigma, \oplus)$, where we implicitly assume that $A[\omega] = \mathbf{0}$ if $\omega \notin [1, w]$. An *array decomposition* of A is an array of pairs as follows $(1 \leq \omega_1 \leq \omega_2 \leq \cdots \leq \omega_r = w)$:

$$\left([1, \omega_1], \bigoplus_{\omega \in [1, \omega_1]} A[\omega]\right), \left([\omega_1 + 1, \omega_2], \bigoplus_{\omega \in [\omega_1+1, \omega_2]} A[\omega]\right), \ldots,$$
$$\left([\omega_r + 1, w], \bigoplus_{\omega \in [\omega_r+1, w]} A[\omega]\right), ([w + 1, \perp], \mathbf{0})$$

All ω_i, for $i \in [p]$, along with $1, \perp$, are the *break points* of the array decomposition. Each pair consists of a range and a semigroup sum over that range. For convenience, we add a dummy pair $([w + 1, \perp], \mathbf{0})$ to mark the end of the array decomposition. A *trivial* array decomposition of A is:

$$([1, 1], A[1]), ([2, 2], A[2]), \ldots, ([w, w], A[w]), ([w + 1, \perp], \mathbf{0}).$$

whose set of break points is $\{1, 2, \ldots, w, \perp\}$. A $\mathsf{RangeOracle}$ of the array A is a $\mathsf{RangeSum}$ data structure built on an array decomposition of A such that: given a query range $[\omega_i + 1, \omega_j]$, where $\omega_i < \omega_j$ are two break points of the array decomposition (so that the query range aligns with the smaller range sums stored in the array decomposition), it returns the sum over the query range

$$\bigoplus_{i \leq k < j} \bigoplus_{\omega \in [\omega_k+1, \omega_{k+1}]} A[\omega] = \bigoplus_{\omega \in [\omega_i+1, \omega_j]} A[\omega]$$

Theorem 5.6 states that there is a data structure of size $O(w)$ that supports a RangeOracle query in $O(\alpha(14w, w))$ time after spending $O(w)$ preprocessing time. For a trivial array decomposition, the RangeOracle is essentially a RangeSum data structure of A.

Now we describe our RangeOracle(s). To better identify the NestFAQ$^\neg$ subexpressions in $\psi_\dagger(\mathbf{x}_{[n]})$, we will use the notation $\ell_\dagger(\mathbf{x}_{N_i}) \overset{*}{\Rightarrow} \psi_\dagger(\mathbf{x}_{N_i})$ for $n \in U = N_0 \subseteq N_1^- \subseteq N_1 \subseteq \cdots \subseteq N_k^- \subseteq N_k$ and $\ell_\dagger(\mathbf{x}_S) \overset{*}{\Rightarrow} \mu_\dagger(\mathbf{x}_S)$ for all $n \notin S \subseteq [n]$. Then, (5.12) can be written as: $\psi_\dagger(\mathbf{x}_{[n]}) =$

$$\begin{aligned}
&= \mu_\dagger(\mathbf{x}_{S_k}) \otimes \psi_\dagger(\mathbf{x}_{N_k}) \\
&= \mu_\dagger(\mathbf{x}_{S_k}) \otimes \left(R_{N_k}(\mathbf{x}_{N_k}) \vdash \psi_\dagger(\mathbf{x}_{N_k^-})\right) \\
&= \mu_\dagger(\mathbf{x}_{S_k}) \otimes (R_{N_k}(\mathbf{x}_{N_k}) \vdash \mu_\dagger(\mathbf{x}_{S_{k-1}}) \otimes \psi_\dagger(\mathbf{x}_{N_{k-1}})) \\
&= \mu_\dagger(\mathbf{x}_{S_k}) \otimes \left(R_{N_k}(\mathbf{x}_{N_k}) \vdash \mu_\dagger(\mathbf{x}_{S_{k-1}}) \otimes \left(R_{N_{k-1}}(\mathbf{x}_{N_{k-1}}) \vdash \psi_\dagger(\mathbf{x}_{N_{k-1}^-})\right)\right) \\
&\cdots \\
&= \mu_\dagger(\mathbf{x}_{S_k}) \otimes \left(R_{N_k}(\mathbf{x}_{N_k}) \vdash \mu_\dagger(\mathbf{x}_{S_{k-1}}) \otimes \left(R_{N_{k-1}}(\mathbf{x}_{N_{k-1}}) \vdash \cdots \left(R_{N_1}(\mathbf{x}_{N_1}) \vdash \psi_\dagger(\mathbf{x}_{N_1^-})\right) \cdots\right)\right) \\
&= \mu_\dagger(\mathbf{x}_{S_k}) \otimes (R_{N_k}(\mathbf{x}_{N_k}) \vdash \mu_\dagger(\mathbf{x}_{S_{k-1}}) \otimes (R_{N_{k-1}}(\mathbf{x}_{N_{k-1}}) \vdash \cdots (R_{N_1}(\mathbf{x}_{N_1}) \vdash \mu_\dagger(\mathbf{x}_{S_0}) \otimes \psi_\dagger(\mathbf{x}_U)) \cdots)) \\
&= \mu_\dagger(\mathbf{x}_{S_k}) \otimes (R_{N_k}(\mathbf{x}_{N_k}) \vdash \mu_\dagger(\mathbf{x}_{S_{k-1}}) \otimes (R_{N_{k-1}}(\mathbf{x}_{N_{k-1}}) \vdash \cdots (R_{N_1}(\mathbf{x}_{N_1}) \vdash \mu_\dagger(\mathbf{x}_{S_0}) \otimes R_U(\mathbf{x}_U)) \cdots))
\end{aligned} \tag{5.15}$$

We will construct a RangeOracle for the NestFAQ$^\neg$ subexpression $\psi_\dagger(\mathbf{x}_U)$ (let $U = N_0$) and then $\psi_\dagger(\mathbf{x}_{N_1}), \psi_\dagger(\mathbf{x}_{N_2}), \ldots, \psi_\dagger(\mathbf{x}_{N_k})$, in an inside-out manner as laid out in (5.15). The BuildOracle algorithm (Algorithm 12) takes as input the NestFAQ$^\neg$ expression $\psi_\dagger(\mathbf{x}_{[n]})$ and returns a RangeOracle for the every NestFAQ$^\neg$ subexpression $\psi_\dagger(\mathbf{x}_{N_i})$, where $i = 0, 1, \ldots, k$. Now we walk through its steps, starting from the inner-most $\psi_\dagger(\mathbf{x}_U) = R_U(\mathbf{x}_U)$ in (5.15), R_U being the pivot factor.

Algorithm 12: $\mathsf{BuildOracle}(\psi_n(\mathbf{x}_{[n]}), \mathcal{D}_n)$

Input: a $\mathsf{NestFAQ}^{\neg}$ expression $\psi_\dagger(\mathbf{x}_{[n]})$

Output: An $\mathsf{RangeOracle}$ $\mathcal{T}_i(\mathbf{x}_{N_i \setminus \{n\}})$ for each $\psi_\dagger(\mathbf{x}_{N_i})$, where $i = 0, 1, \ldots, k$ and $U = N_0 \subseteq N_1 \subseteq \cdots \subseteq N_k$ are the only hyperedges containing n

1 **foreach** $\mathbf{a}_{U\setminus\{n\}} \in \bigcup_{0\le i\le k} \Pi_{U\setminus\{n\}} R_{N_i}$ **do** ▷ base case for $\psi_\dagger(\mathbf{x}_U)$

2   **init** $A_0(\mathbf{a}_{U\setminus\{n\}}) \leftarrow [],\ \mathcal{T}_0(\mathbf{a}_{U\setminus\{n\}}) \leftarrow [],\ \omega_{\mathbf{a}_{U\setminus\{n\}}} \leftarrow 1$

3 **foreach** $\mathbf{a}_U \in \bigcup_{0\le i\le k} \Pi_U R_{N_i}$ **do**

4     $A_0(\mathbf{a}_{U\setminus\{n\}})[\omega_{\mathbf{a}_{U\setminus\{n\}}}] \leftarrow \Pi_{\{n\}} \mathbf{a}_U$

5   **append** $\left([\omega_{\mathbf{a}_{U\setminus\{n\}}}, \omega_{\mathbf{a}_{U\setminus\{n\}}}], R_U(\mathbf{a}_U)\right)$ **to** $\mathcal{T}_0(\mathbf{a}_{U\setminus\{n\}})$

6     $\omega_{\mathbf{a}_{U\setminus\{n\}}} \leftarrow \omega_{\mathbf{a}_{U\setminus\{n\}}} + 1$

7 **foreach** $\mathbf{a}_{U\setminus\{n\}} \in \bigcup_{0\le i\le k} \Pi_{U\setminus\{n\}} R_{N_i}$ **do**

8   **append** $\left([\omega_{\mathbf{a}_{U\setminus\{n\}}} + 1, \perp],\ \mathbf{0}\right)$ **to** $\mathcal{T}_0(\mathbf{a}_{U\setminus\{n\}})$

9   **construct** $\mathsf{RangeSum}$ data structures **on** $\mathcal{T}_0(\mathbf{a}_{U\setminus\{n\}})$

10 **foreach** $i = 1, 2, \ldots k$ *(in order)* **do** ▷ inductive case for $\psi_\dagger(\mathbf{x}_{N_i})$

11   **foreach** $\mathbf{a}_{N_i\setminus\{n\}} \in \bigcup_{i\le s\le k} \Pi_{N_i\setminus\{n\}} R_{N_s}$ **do**

12   **init** $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}}) \leftarrow [],\ \omega^-_{\mathbf{a}_{N_i\setminus\{n\}}} \leftarrow 0$

13   **foreach** $\mathbf{a}_{N_{i-1}\setminus\{n\}} \in \bigcup_{i-1\le s\le k} \Pi_{N_{i-1}\setminus\{n\}} R_{N_s}$ **do**

14         $\omega^+ \leftarrow 1$

15   **while** $\omega^+ \neq \perp$ **do**

16             $a_n \leftarrow A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega^+]$

17   **foreach** $\mathbf{a}_{N_i\setminus\{n\}} \in \bigcup_{i\le s\le k} \Pi_{N_i\setminus\{n\}} R_{N_s}(a_n, \mathbf{a}_{N_{i-1}\setminus\{n\}}, \mathbf{x}_{N_s\setminus N_{i-1}})$ **do**

18                 $M_1 \leftarrow \bigoplus_{\omega \in [\omega^-_{\mathbf{a}_{N_i\setminus\{n\}}}+1, \omega^+-1]} \psi_\dagger\left(A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega], \mathbf{a}_{N_{i-1}\setminus\{n\}}\right)$

19                 $M_2 \leftarrow \psi_\dagger(a_n, \Pi_{N_{i-1}\setminus\{n\}} \mathbf{a}_{N_i\setminus\{n\}})$

20   **append** $\left([\omega^-_{\mathbf{a}_{N_i\setminus\{n\}}} + 1, \omega^+ - 1],\ \mu_n(\Pi_{S_{i-1}\setminus\{n\}} \mathbf{a}_{N_i\setminus\{n\}}) \otimes M_1\right)$ **to** $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$

21   **append** $\left([\omega^+, \omega^+],\ R_{N_i}(a_n, \mathbf{a}_{N_i\setminus\{n\}}) \oplus \mathbb{1}_{\neg R_{N_i}}(a_n, \mathbf{a}_{N_i\setminus\{n\}}) \otimes \mu_n(\Pi_{S_{i-1}\setminus\{n\}} \mathbf{a}_{N_i\setminus\{n\}}) \otimes M_2\right)$ **to** $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$

22                 $\omega^-_{\mathbf{a}_{N_i\setminus\{n\}}} \leftarrow \omega^+ + 1$

23   **move** ω^+ to the next break point of $\mathcal{T}_{i-1}(\mathbf{a}_{N_{i-1}\setminus\{n\}})$

24   **foreach** $\mathbf{a}_{N_i\setminus\{n\}} \in \bigcup_{i\le s\le k} \Pi_{N_i\setminus\{n\}} R_{N_s}$ **do**

25   **append** $([\omega^-_{\mathbf{a}_{N_i\setminus\{n\}}} + 1, \perp],\ \mathbf{0})$ into $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$

26   **construct** $\mathsf{RangeSum}$ data structures **on** $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$

27 **return** $\left(\mathcal{T}_i(\mathbf{x}_{N_i\setminus\{n\}})\right)_{i=0,1,\ldots,k}$

The base case of $\psi_\dagger(\mathbf{x}_U) = R_U(\mathbf{x}_U)$. The base case first sets up the indices for accessing all x_n values for RangeSum data structures constructed later. More precisely, we first build a hashtable $A_0(\mathbf{x}_{U\setminus\{n\}})$ from $\bigcup_{0\leq i\leq k}\Pi_U R_{N_i}$, where $\mathbf{x}_{U\setminus\{n\}}$ is its key and each entry $\mathbf{x}_{U\setminus\{n\}} = \mathbf{a}_{U\setminus\{n\}}$ stores an array of x_n values where $(x_n, \mathbf{a}_{U\setminus\{n\}}) \in \bigcup_{0\leq i\leq k}\Pi_U R_{N_i}$. The array is identified as $A_0(\mathbf{a}_{U\setminus\{n\}})$ and the ordering of x_n values in the array can be posited arbitrarily but fixed afterwards. Thus, $A_0(\mathbf{x}_{U\setminus\{n\}})$ is of size $|\bigcup_{0\leq i\leq k}\Pi_U R_{N_i}| = O(|\mathcal{D}_\psi|)$ and allows for every tuple $\mathbf{a}_{U\setminus\{n\}} \in \bigcup_{0\leq i\leq k}\Pi_{U\setminus\{n\}}R_{N_i}$, a direct (array) access to all possible x_n values that could appear alongside $\mathbf{a}_{U\setminus\{n\}}$. It is easy to see that this construction costs $O(|\mathcal{D}_\psi|)$ time and space.

We start constructing the base-case oracle $\mathcal{T}_0(\mathbf{x}_{U\setminus\{n\}})$ for $\psi_\dagger(\mathbf{x}_U) = R_U(\mathbf{x}_U)$. It follows exactly as $A_0(\mathbf{x}_{U\setminus\{n\}})$ except that each entry $\mathbf{x}_{U\setminus\{n\}} = \mathbf{a}_{U\setminus\{n\}} \in \mathcal{T}_0(\mathbf{x}_{U\setminus\{n\}})$ stores instead a trivial array decomposition of $A_0(\mathbf{a}_{U\setminus\{n\}})$, replacing each value of x_n in $A_0(\mathbf{a}_{U\setminus\{n\}})$ by its weight $R_U(x_n, \mathbf{a}_{U\setminus\{n\}}) \in \Sigma$. So the ω-th element in $A_0(\mathbf{a}_{U\setminus\{n\}})$, say $a_n = A_0(\mathbf{a}_{U\setminus\{n\}})[\omega]$, corresponds to the ω-th entry in the array decomposition $\mathcal{T}_0(\mathbf{a}_{U\setminus\{n\}})$, i.e. $\Big([\omega,\omega], R_U(a_n, \mathbf{a}_{U\setminus\{n\}})\Big) = \Big([\omega,\omega], \psi_\dagger(a_n, \mathbf{a}_{U\setminus\{n\}})\Big)$. Thus, $\mathcal{T}_0(\mathbf{x}_{U\setminus\{n\}})$ is of size $|A_0(\mathbf{x}_{U\setminus\{n\}})| = O(|\mathcal{D}_\psi|)$. We recall that for $(a_n, \mathbf{a}_{U\setminus\{n\}}) \notin R_U$, we have $R_U(a_n, \mathbf{a}_{U\setminus\{n\}}) = \mathbf{0}$. Applying Theorem 5.6, we construct a RangeOracle on each array decomposition $\mathcal{T}_0(\mathbf{a}_{U\setminus\{n\}})$, after spending $O(|\mathcal{D}_\psi|)$ preprocessing time building the RangeSum data structures. Abusing notations, we denote the entire oracle as $\mathcal{T}_0(\mathbf{x}_{U\setminus\{n\}})$ to indicate that for each $\mathbf{a}_{U\setminus\{n\}} \in \bigcup_{0\leq i\leq k}\Pi_{U\setminus\{n\}}R_{N_i}$, it stores an array decomposition equipped with a RangeOracle that supports the following query in $O(\alpha(14|\mathcal{D}_\psi|, |\mathcal{D}_\psi|))$ time:

ORACLE $\mathcal{T}_0(\mathbf{x}_{U\setminus\{n\}})$

INPUT. A tuple $\mathbf{a}_{U\setminus\{n\}} \in \bigcup_{0\leq i\leq k}\Pi_{U\setminus\{n\}}R_{N_i}$ (to identity the array decomposition $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$ to be queried) and a query range $[\omega^- + 1, \omega^+]$, where $0 \leq \omega^- < \omega^+$ are two break points of the array decomposition.

OUTPUT. A range sum $\bigoplus_{\omega\in[\omega^-+1,\omega^+]} \psi_\dagger\Big(A_0(\mathbf{a}_{U\setminus\{n\}})[\omega],\ \mathbf{a}_{U\setminus\{n\}}\Big)$.

The inductive case of $\psi_\dagger(\mathbf{x}_{N_i})$. For $\psi_\dagger(\mathbf{x}_{N_i})$, where $i \in [k]$, following (5.15), we have that $\psi_\dagger(\mathbf{x}_{N_i}) = R_{N_i}(\mathbf{x}_{N_i}) \vdash \psi_\dagger(\mathbf{x}_{N_i^-}) = R_{N_i}(x_n, \mathbf{a}_{N_i\setminus\{n\}}) \vdash \mu_\dagger(\mathbf{x}_{S_{i-1}}) \otimes \psi_\dagger(\mathbf{x}_{N_{i-1}})$. Thus, for a fixing of $\mathbf{x}_{N_i\setminus\{n\}} = \mathbf{a}_{N_i\setminus\{n\}}$, we have

$$\begin{aligned}\psi_\dagger(x_n, \mathbf{a}_{N_i\setminus\{n\}}) &= R_{N_i}(x_n, \mathbf{a}_{N_i\setminus\{n\}}) \vdash \psi_\dagger(x_n, \Pi_{N_i^-\setminus\{n\}}\mathbf{a}_{N_i\setminus\{n\}}) \\ &= R_{N_i}(x_n, \mathbf{a}_{N_i\setminus\{n\}}) \vdash \mu_\dagger(\Pi_{S_{i-1}\setminus\{n\}}\mathbf{a}_{N_i\setminus\{n\}}) \otimes \psi_\dagger(x_n, \Pi_{N_{i-1}\setminus\{n\}}\mathbf{a}_{N_i\setminus\{n\}})\end{aligned} \tag{5.16}$$

Following the inside-out construction, for the i-th ($i > 0$) time entering in the for-loop on line 10, we have already constructed the inner $(i-1)$-th RangeOracle denoted as $\mathcal{T}_{i-1}(\mathbf{x}_{N_{i-1}\setminus\{n\}})$ for $\psi_\dagger(\mathbf{x}_{N_{i-1}})$.

Indeed, $\mathcal{T}_{i-1}(\mathbf{x}_{N_{i-1}\setminus\{n\}})$ is a hashtable of size $O(|\mathcal{D}_\psi|)$ containing array decompositions as entries that supports the following range query in $O(\alpha(14|\mathcal{D}_\psi|, |\mathcal{D}_\psi|))$ time:

ORACLE $\mathcal{T}_{i-1}(\mathbf{x}_{N_{i-1}\setminus\{n\}})$

INPUT. A tuple $\mathbf{a}_{N_{i-1}\setminus\{n\}} \in \bigcup_{i-1\le s\le k} \Pi_{N_{i-1}\setminus\{n\}} R_{N_s}$ (i.e. a query key to identity the array decomposition $\mathcal{T}_{i-1}(\mathbf{a}_{N_{i-1}\setminus\{n\}})$ to be queried) and a query range $[\omega^- + 1, \omega^+]$, where $0 \le \omega^- < \omega^+$ are two break points of the array decomposition $\mathcal{T}_{i-1}(\mathbf{a}_{N_{i-1}\setminus\{n\}})$. In particular, the break points of the array decomposition $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$ are ω and $\omega - 1$, where

$$\omega \in \left\{ \omega \mid \left(A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega], \mathbf{a}_{N_{i-1}\setminus\{n\}} \right) \in \bigcup_{i-1\le s\le k} \Pi_{N_{i-1}} R_{N_s} \right\}$$

OUTPUT. A range sum $\bigoplus_{\omega\in[\omega^-+1,\omega^+]} \psi_\dagger \left(A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega],\ \mathbf{a}_{N_{i-1}\setminus\{n\}} \right)$.

We now start constructing a new RangeOracle $\mathcal{T}_i(\mathbf{x}_{N_i\setminus\{n\}})$ for $\psi_\dagger(\mathbf{x}_{N_i})$, using $\mathcal{T}_{i-1}(\mathbf{x}_{N_{i-1}\setminus\{n\}})$. We initialize $\mathcal{T}_i(\mathbf{x}_{N_i\setminus\{n\}})$ as a hashtable with $\mathbf{x}_{N_i\setminus\{n\}}$ as its key, where each entry $\mathbf{x}_{N_i\setminus\{n\}} = \mathbf{a}_{N_i\setminus\{n\}} \in \bigcup_{i\le s\le k} \Pi_{N_t\setminus\{n\}} R_{N_s}$ is intialized as an empty array decomposition, denoted as $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$. Furthermore, we initialize a set of scanning iterators $\omega^-_{\mathbf{a}_{N_i\setminus\{n\}}} = 0$, one for each $\mathbf{a}_{N_i\setminus\{n\}} \in \bigcup_{i\le s\le k} \Pi_{N_t\setminus\{n\}} R_{N_s}$ that pinpoints the last constructed break point throughout its ongoing construction process.

Next, we will make one scan over every entry (array decomposition) stored in $\mathcal{T}_{i-1}(\mathbf{x}_{N_{i-1}\setminus\{n\}})$ by iterating over its break points, and for each array decomposition $\mathcal{T}_{i-1}(\mathbf{a}_{N_i\setminus\{n\}})$, we construct the array decompositions $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$ simultaneously for all $\mathbf{a}_{N_i\setminus\{n\}} \in \bigcup_{i\le s\le k} \Pi_{N_t\setminus\{n\}} R_{N_s}$ such that $\Pi_{N_{i-1}\setminus\{n\}} \mathbf{a}_{N_i\setminus\{n\}} = \mathbf{a}_{N_{i-1}\setminus\{n\}}$. Let us describe the exact steps when scanning an array decomposition $\mathcal{T}_{i-1}(\mathbf{a}_{N_{i-1}\setminus\{n\}})$ in the oracle $\mathcal{T}_{i-1}(\mathbf{x}_{N_{i-1}\setminus\{n\}})$. First, we initialize a scanning iterator $\omega^+ = 1$ to scan the array decomposition $\mathcal{T}_{i-1}(\mathbf{a}_{N_{i-1}\setminus\{n\}})$ by moving ω^+ forward to the next break point.

As we are scanning at the break point ω^+ of the array decomposition $\mathcal{T}_{i-1}(\mathbf{x}_{N_{i-1}\setminus\{n\}})$, we retrieve its corresponding x_n value via an array look-up $A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega^+] = a_n$. For every ongoing construction of $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$ (i.e. $\Pi_{N_{i-1}\setminus\{n\}} \mathbf{a}_{N_i\setminus\{n\}} = \mathbf{a}_{N_{i-1}\setminus\{n\}}$), if $(a_n, \mathbf{a}_{N_i\setminus\{n\}}) \in \bigcup_{i\le s\le k} \Pi_{N_i} R_{N_s}$, we set both $\omega^+ - 1$ and ω^+ as it new break points by appending the following two pairs/ranges to the array decomposition $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$:

1. the first pair to be appended is

$$\left([\omega^-_{\mathbf{a}_{N_i\setminus\{n\}}} + 1, \omega^+ - 1],\ \mu_n(\Pi_{S_{i-1}\setminus\{n\}} \mathbf{a}_{N_i\setminus\{n\}}) \otimes M_1 \right)$$

   where M_1 is the result of a call to the RangeOracle $\mathcal{T}_{i-1}(\mathbf{a}_{N_{i-1}\setminus\{n\}})$ with the input parameters $\mathbf{a}_{N_{i-1}\setminus\{n\}} = \Pi_{N_{i-1}\setminus\{n\}} \mathbf{a}_{N_i\setminus\{n\}}$ (the query key) and $[\omega^-_{\mathbf{a}_{N_i\setminus\{n\}}} + 1, \omega^+ - 1]$ (the query range).

Therefore, by induction hypothesis,

$$M_1 = \bigoplus_{\omega \in [\omega^-_{\mathbf{a}_{N_i \setminus \{n\}}}+1, \omega^+ - 1]} \psi_\dagger \left(A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega], \mathbf{a}_{N_{i-1}\setminus\{n\}} \right)$$

and M_1 is returned by $\mathcal{T}_{i-1}(\mathbf{x}_{N_{i-1}\setminus\{n\}})$ in time $O(\alpha(14|\mathcal{D}_\psi|, |\mathcal{D}_\psi|))$.

2. the second pair to be appended is

$$\left([\omega^+, \omega^+],\ R_{N_i}(a_n, \mathbf{a}_{N_i\setminus\{n\}}) \oplus \mathbb{1}_{\neg R_{N_i}}(a_n, \mathbf{a}_{N_i\setminus\{n\}}) \otimes \mu_\dagger(\Pi_{S_{i-1}\setminus\{n\}} \mathbf{a}_{N_i\setminus\{n\}}) \otimes M_2\right)$$

where M_2 is a call to the RangeOracle $\mathcal{T}_{i-1}(\mathbf{a}_{N_{i-1}\setminus\{n\}})$ with the input parameters $\mathbf{a}_{N_{i-1}\setminus\{n\}} = \Pi_{N_{i-1}\setminus\{n\}} \mathbf{a}_{N_i\setminus\{n\}}$ (the query key) and $[\omega^+, \omega^+]$ (the query range). Therefore, by induction hypothesis, $M_2 =$

$$\bigoplus_{\omega \in [\omega^+, \omega^+]} \psi_\dagger \left(A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega], \mathbf{a}_{N_{i-1}\setminus\{n\}} \right) = \psi_\dagger(a_n, \mathbf{a}_{N_{i-1}\setminus\{n\}}) = \psi_\dagger(a_n, \Pi_{N_{i-1}\setminus\{n\}} \mathbf{a}_{N_i\setminus\{n\}})$$

and M_2 is returned by $\mathcal{T}_{i-1}(\mathbf{x}_{N_{i-1}\setminus\{n\}})$ in time $O(\alpha(14|\mathcal{D}_\psi|, |\mathcal{D}_\psi|))$.

The correctness of the second pair follows from (5.16), fixing x_n as $a_n = A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega]$. We now reason the correctness of the first pair as follows: by construction of break points, $\omega^-_{\mathbf{a}_{N_i\setminus\{n\}}}$ and $\omega^+ - 1$ are two consecutive break points for $\mathcal{T}_i(\mathbf{x}_{N_i\setminus\{n\}})$, thus $(A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega], \mathbf{a}_{N_i\setminus\{n\}}) \notin R_{N_i}$, for all $\omega \in [\omega^-_{\mathbf{a}_{N_i\setminus\{n\}}} + 1, \omega^+ - 1]$. So for all $\omega \in [\omega^-_{\mathbf{a}_{N_i\setminus\{n\}}} + 1, \omega^+ - 1]$, $R_{N_i}(A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega]) = \mathbf{0}$ and $\mathbb{1}_{R_{N_i}}(A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega]) = \mathbf{1}$. Still following (5.16), we have that

$$\begin{aligned}
\psi_\dagger(A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega], \mathbf{a}_{N_i\setminus\{n\}}) &= R_{N_i}(A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega]) \oplus \\
&\quad \mathbb{1}_{R_{N_i}}(A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega], \mathbf{a}_{N_i\setminus\{n\}}) \otimes \psi_\dagger(A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega], \Pi_{N_i^-\setminus\{n\}} \mathbf{a}_{N_i\setminus\{n\}}) \\
&= \mathbf{0} \oplus \mathbf{1} \otimes \psi_\dagger(A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega],\ \Pi_{N_i^-\setminus\{n\}} \mathbf{a}_{N_i\setminus\{n\}}) \\
&= \psi_\dagger(A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega],\ \Pi_{N_i^-\setminus\{n\}} \mathbf{a}_{N_i\setminus\{n\}}) \\
&= \mu_\dagger(\Pi_{S_{i-1}\setminus\{n\}} \mathbf{a}_{N_i\setminus\{n\}}) \otimes \psi_\dagger(A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega],\ \Pi_{N_{i-1}\setminus\{n\}} \mathbf{a}_{N_i\setminus\{n\}})
\end{aligned}$$

Hence, the correctness of the first pair follows because

$$\bigoplus_{\omega \in [\omega^-_{\mathbf{a}_{N_i\setminus\{n\}}}+1, \omega^+-1]} \psi_\dagger(A_0(\Pi_{U\setminus\{n\}} \mathbf{a}_{N_{i-1}\setminus\{n\}})[\omega], \mathbf{a}_{N_i\setminus\{n\}}) = \mu_\dagger(\Pi_{S_{i-1}\setminus\{n\}} \mathbf{a}_{N_i\setminus\{n\}}) \otimes M_1.$$

As a summary, one scan over the break points of every array decomposition stored in $\mathcal{T}_{i-1}(\mathbf{x}_{N_{i-1}\setminus\{n\}})$ costs time $O(|\mathcal{D}_\psi|)$. For a fixed $\mathbf{a}_{N_{i-1}\setminus\{n\}} \in \bigcup_{i-1\le s\le k} \Pi_{N_{i-1}\setminus\{n\}} R_{N_s}$, we make break points across

all the (constructing) array decompositions of $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$ where $\mathbf{a}_{N_i\setminus\{n\}} \in \bigcup_{i\le s\le k} \Pi_{N_i} R_{N_s}$ and $\Pi_{N_{i-1}\setminus\{n\}}\mathbf{a}_{N_i\setminus\{n\}} = \mathbf{a}_{N_{i-1}\setminus\{n\}}$ and by construction, the break points are ω and $\omega - 1$ where

$$\omega \in \left\{\omega \mid \left(A_0(\Pi_{U\setminus\{n\}}\mathbf{a}_{N_i\setminus\{n\}})[\omega], \mathbf{a}_{N_i\setminus\{n\}}\right) \in \bigcup_{i\le s\le k} \Pi_{N_i} R_{N_s}\right\}$$

Observe that the break points constructed in $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$ is a subset of those of $\mathcal{T}_{i-1}(\Pi_{N_{i-1}\setminus\{n\}}\mathbf{a}_{N_i\setminus\{n\}})$, thus we are free from the risk of asking the oracle $\mathcal{T}_{i-1}(\Pi_{N_{i-1}\setminus\{n\}}\mathbf{a}_{N_i\setminus\{n\}})$ with mis-aligned intervals, when constructing $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$. The number of break points to be constructed in the array decomposition $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$ is at most $\sum_{i\le s\le k} \mathsf{deg}_{R_{N_s}}(\mathbf{a}_{N_i\setminus\{n\}})$, where $\mathsf{deg}_{R_{N_s}}(\mathbf{a}_{N_i\setminus\{n\}})$ denotes the degree of $\mathbf{a}_{N_i\setminus\{n\}}$ in R_{N_s}, i.e., the number of tuples in R_{N_s} that coincide with $\mathbf{a}_{N_i\setminus\{n\}}$ on the variables $\mathbf{x}_{N_i\setminus\{n\}}$. Therefore, the total number of break points in the oracle $\mathcal{T}_i(\mathbf{x}_{N_i\setminus\{n\}})$ is at most

$$\sum_{\mathbf{a}_{N_i\setminus\{n\}}\in\bigcup_{i\le s\le k}\Pi_{N_i\setminus\{n\}}R_{N_s}} \sum_{i\le s\le k} \mathsf{deg}_{R_{N_s}}(\mathbf{a}_{N_i\setminus\{n\}}) \le \sum_{i\le s\le k} |R_{N_s}| = O(|\mathcal{D}_\psi).$$

The new-built oracle $\mathcal{T}_i(\mathbf{x}_{N_i\setminus\{n\}})$ is thus of size $O(|\mathcal{D}_\psi|)$. Furthermore, for each break point, it costs $O(\alpha(14|\mathcal{D}_\psi|, |\mathcal{D}_\psi|))$ to get the two range sums M_1, M_2 from the oracle $\mathcal{T}_{i-1}(\mathbf{x}_{N_{i-1}\setminus\{n\}})$, thus the total time of the construction of $\mathcal{T}_i(\mathbf{x}_{N_i\setminus\{n\}})$ is $O(|\mathcal{D}_\psi| \cdot \alpha(14|\mathcal{D}_\psi|, |\mathcal{D}_\psi|))$. The RangeOracle $\mathcal{T}_i(\mathbf{x}_{N_i\setminus\{n\}})$ supports the following range query in time $O(\alpha(14|\mathcal{D}_\psi|, |\mathcal{D}_\psi|))$:

ORACLE $\mathcal{T}_i(\mathbf{x}_{N_i\setminus\{n\}})$

INPUT. A tuple $\mathbf{a}_{N_i\setminus\{n\}} \in \bigcup_{i\le s\le k} \Pi_{N_i\setminus\{n\}} R_{N_s}$ (i.e. a query key to identity the array decomposition $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$ to be queried) and a query range $[\omega^- + 1, \omega^+]$, where $0 \le \omega^- < \omega^+$ are two break points of the array decomposition $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$. In particular, the break points of the array decomposition $\mathcal{T}_i(\mathbf{a}_{N_i\setminus\{n\}})$ are ω and $\omega - 1$, where

$$\omega \in \left\{\omega \mid \left(A_0(\Pi_{U\setminus\{n\}}\mathbf{a}_{N_i\setminus\{n\}})[\omega], \mathbf{a}_{N_i\setminus\{n\}}\right) \in \bigcup_{i\le s\le k} \Pi_{N_i} R_{N_s}\right\}$$

OUTPUT. A range sum $\bigoplus_{\omega\in[\omega^-+1,\omega^+]} \psi_\dagger \left(A_0(\Pi_{U\setminus\{n\}}\mathbf{a}_{N_i\setminus\{n\}})[\omega],\ \mathbf{a}_{N_i\setminus\{n\}}\right)$.

To conclude this section, we present the following theorem, whose proof is immediate from the prior discussion in this section. Here, $k = d(n)$, where we recall that $d(n)$ is number of hyperedges containing the signed-leaf n in $\mathcal{H}_\dagger$.

Theorem 5.7. *Let $\mathcal{H}_\dagger$ be a signed hypergraph with a signed-leaf n. Let $\psi_\dagger(\mathbf{x}_{[n]})$ be a* NestFAQ$^\neg$ *expression* (5.12) *associated with $\mathcal{H}_\dagger$ (thus recognized by CFG* (5.6)*). Suppose that $U = N_0 \subseteq N_1 \subseteq$*

$\cdots \subseteq N_k$ are the only hyperedges containing the signed-leaf n in $\mathcal{H}_n$. Then, there is an algorithm that takes $\psi_\dagger(\mathbf{x}_{[n]})$ as input and constructs in $O(|\psi_\dagger| + k \cdot |\mathcal{D}_{\psi_\dagger}|)$ time a RangeOracle *$\mathcal{T}_i(\mathbf{x}_{N_i \setminus \{n\}})$, for every $i = 0, 1, \dots, k$, and the oracle $\mathcal{T}_i(\mathbf{x}_{N_i \setminus \{n\}})$ is of size $O(|\mathcal{D}_{\psi_\dagger}|)$ and supports the following range query in $O(\alpha(14|\mathcal{D}_{\psi_\dagger}|, |\mathcal{D}_{\psi_\dagger}|))$ time:*

oracle $\mathcal{T}_i(\mathbf{x}_{N_i \setminus \{n\}})$

input *a query key $\mathbf{a}_{N_i \setminus \{n\}} \in \bigcup_{i \le s \le k} \Pi_{N_i \setminus \{n\}} R_{N_s}$ to identity the array decomposition $\mathcal{T}_i(\mathbf{a}_{N_i \setminus \{n\}})$ to be queried, and a query range $[\omega^- + 1, \omega^+]$, where $0 \le \omega^- < \omega^+$ are two break points of the array decomposition $\mathcal{T}_i(\mathbf{a}_{N_i \setminus \{n\}})$. In particular, the break points of the array decomposition $\mathcal{T}_i(\mathbf{a}_{N_i \setminus \{n\}})$ are ω and $\omega - 1$, where*

$$\omega \in \left\{ \omega \mid \left(A_0(\Pi_{U \setminus \{n\}} \mathbf{a}_{N_i \setminus \{n\}})[\omega], \mathbf{a}_{N_i \setminus \{n\}} \right) \in \bigcup_{i \le s \le k} \Pi_{N_i} R_{N_s} \right\}$$

output *a range sum $\bigoplus_{\omega \in [\omega^- + 1, \omega^+]} \psi_\dagger \left(A_0(\Pi_{U \setminus \{n\}} \mathbf{a}_{N_i \setminus \{n\}})[\omega],\ \mathbf{a}_{N_i \setminus \{n\}} \right)$.*

5.5.6 Enumeration of Full NestFAQ$^\neg$

In this part, we study the enumeration problem Enum(φ) for signed-acyclic *full* NestFAQ$^\neg$ queries. First, we recall the definitions (5.5), (5.4): a NestFAQ$^\neg$ query associated with a signed hypergraph $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ is full if $F = [n]$, that is, $\varphi(\mathbf{x}_{[n]}) = \psi(\mathbf{x}_{[n]})$, where $\psi(\mathbf{x}_{[n]})$ is a safe NestFAQ$^\neg$ expression associated with $\mathcal{H}$. The enumeration algorithm can be summarized as:

1. **Pre-processing phase:** we first reduce the Enum(φ) problem into a Enum($Q^*, \mathcal{D}^*$) problem, where Q^* is a signed-acyclic *full* CQ$^\neg$ query that evaluates on a new database instance $\mathcal{D}^*$; then, we follow the pre-processing algorithm for signed-acyclic *full* CQ$^\neg$ queries presented in Section 5.4.

2. **Enumeration phase:** we enumerate the output tuples of Q^* via the enumeration algorithm for signed-acyclic *full* CQ$^\neg$ queries presented in Section 5.4; then we plug each tuple into φ to get its corresponding weight.

Now we present the construction of Q^* and $\mathcal{D}^*$. First, the desired Q^* is the following full CQ$^\neg$ query associated with $\mathcal{H}$:

$$Q^*(\mathbf{x}_{[n]}) = \bigwedge_{K \in \mathcal{E}^+} R^*_K(\mathbf{x}_K) \wedge \bigwedge_{N \in \mathcal{E}^-} \neg R^*_N(\mathbf{x}_N), \tag{5.17}$$

where for every $K \in \mathcal{E}^+$, we add a positive atom $R^*_K(\mathbf{x}_K)$, and for every $N \in \mathcal{E}^-$, we add a negative atom $R^*_N(\mathbf{x}_N)$ (and place a $\neg$ symbol in front) into the body of Q^*. It is easy to see that (5.17) is a

full signed-acyclic *full* $\mathsf{CQ}^{\neg}$ query associated with $\mathcal{H}$. Next, we construct the corresponding database instance $\mathcal{D}^*$ for every atom in Q^* defined as follows:

$$\begin{aligned} R_K^* &= \{\mathbf{a}_K \in R_K \mid R_K(\mathbf{a}_K) \neq \mathbf{0}\}, \quad \text{for every } K \in \mathcal{E}^+, \\ R_N^* &= \{\mathbf{a}_N \in R_N \mid R_N(\mathbf{a}_N) = \mathbf{0}\}, \quad \text{for every } N \in \mathcal{E}^-. \end{aligned} \tag{5.18}$$

Thus, $\mathcal{D}^*$ is of size $O(\mathcal{D}_\varphi)$ and can be constructed in $O(\mathcal{D}_\varphi)$ time by scanning over the list representation of every factor in φ. We reason about the construction of $\mathsf{Enum}(Q^*, \mathcal{D}^*)$ through the following theorem:

Theorem 5.8. *Let φ be a signed-acyclic full $\mathsf{NestFAQ}^{\neg}$ query associated with a signed hypergraph $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ over a commutative semiring σ. Then, a tuple $\mathbf{a}_{[n]}$ is an answer of φ (i.e. $\varphi(\mathbf{a}_{[n]}) \neq \mathbf{0}$) if and only if $\mathbf{a}_{[n]}$ is an answer of $Q^*(\mathcal{D}^*)$ (i.e. $\mathbf{a}_{[n]} \in Q^*(\mathcal{D}^*)$, or $Q^*(\mathbf{a}_{[n]}) = \mathsf{true}$), where Q^* and $\mathcal{D}^*$ are defined in* (5.17) *and* (5.18)*, respectively.*

Proof. We first prove the "if" direction. Suppose $\mathbf{a} = \mathbf{a}_{[n]} \in Q^*(\mathcal{D}^*)$. Then, we have $\Pi_K\mathbf{a} \in R_K^*$, for every $K \in \mathcal{E}^+$, and $\Pi_N\mathbf{a} \notin R_N^*$, for every $N \in \mathcal{E}^-$. This implies that $R_K(\Pi_K\mathbf{a}) \neq \mathbf{0}$, for every $K \in \mathcal{E}^+$. However, for every $N \in \mathcal{E}^-$, either $\Pi_N\mathbf{a} \notin R_N$, or $\Pi_N\mathbf{a} \in R_N \wedge R_N(\Pi_N\mathbf{a}) \neq \mathbf{0}$. We prove that $\varphi(\mathbf{a}) = \psi(\mathbf{a}) \neq \mathbf{0}$ by induction on the $\mathsf{NestFAQ}^{\neg}$ subexpression $\psi(\mathbf{x}_S)$ following the rules of the CFG (5.4) in a bottom-up fashion. The base case $S = \emptyset$ is trivial, because $\psi(\Pi_\emptyset\mathbf{a}) = e$, where $e \in \Sigma \setminus \{\mathbf{0}\}$. We argue the inductive step. If $\ell(\mathbf{x}_S) \overset{*}{\Rightarrow} \psi(\mathbf{x}_S)$ uses the production $\ell(\mathbf{x}_S) ::= \; R_K(\mathbf{x}_K) \otimes \ell(\mathbf{x}_{S^-})$ next, and by induction hypothesis $\psi(\Pi_{S^-}\mathbf{a}) \neq \mathbf{0}$, then $\psi(\Pi_S\mathbf{a}) = R_K(\Pi_K\mathbf{a}) \otimes \psi(\Pi_{S^-}\mathbf{a}) \neq \mathbf{0}$. Otherwise, $\ell(\mathbf{x}_S) \overset{*}{\Rightarrow} \psi(\mathbf{x}_S)$ uses the production $(R_N(\mathbf{x}_N) \vdash \ell(\mathbf{x}_{N^-})) \otimes \ell(\mathbf{x}_{S^-})$ as the next yield step, and by induction hypothesis $\psi(\Pi_{S^-}\mathbf{a}) \neq \mathbf{0}$ and $\psi(\Pi_{N^-}\mathbf{a}) \neq \mathbf{0}$. We distinguish the following two cases:

- If $\Pi_N\mathbf{a} \notin R_N$, then $R_N(\Pi_N\mathbf{a}) \vdash \psi(\Pi_{N^-}\mathbf{a}) =$
$$R_N(\Pi_N\mathbf{a}) \oplus \mathbb{1}_{\neg R_N}(\Pi_N\mathbf{a}) \otimes \psi(\Pi_{N^-}\mathbf{a}) = \mathbf{0} \oplus \mathbf{1} \otimes \psi(\Pi_{N^-}\mathbf{a}) = \psi(\Pi_{N^-}\mathbf{a}) \neq \mathbf{0}$$
- If $\Pi_N\mathbf{a} \in R_N$ and $R_N(\Pi_N\mathbf{a}) \neq \mathbf{0}$, then
$$R_N(\Pi_N\mathbf{a}) \vdash \psi(\Pi_{N^-}\mathbf{a}) = R_N(\Pi_N\mathbf{a}) \oplus \mathbf{0} \otimes \psi(\Pi_{N^-}\mathbf{a}) = R_N(\Pi_N\mathbf{a}) \neq \mathbf{0}$$

In both cases, we have $\psi(\Pi_S\mathbf{a}) = (R_N(\Pi_N\mathbf{a}) \vdash \psi(\mathbf{x}_{N^-})) \otimes \psi(\Pi_{S^-}\mathbf{a}) \neq \mathbf{0}$. Therefore, $\varphi(\mathbf{a}) \neq \mathbf{0}$.

We then prove the "only if" direction. Suppose $\varphi(\mathbf{a}) \neq \mathbf{0}$. We prove that $\mathbf{a} \in Q^*(\mathcal{D}^*)$, or in other words, $\Pi_K\mathbf{a} \in R_K^*$, for every $K \in \mathcal{E}^+$, and $\Pi_N\mathbf{a} \notin R_N^*$, for every $N \in \mathcal{E}^-$. We prove by induction on the $\mathsf{NestFAQ}^{\neg}$ subexpression $\psi(\mathbf{x}_S)$ following the rules of the CFG (5.4) in a top-down fashion, i.e. tracing the derivation from the start symbol $\ell(\mathbf{x}_{[n]})$ of the CFG (5.4). The base case is simply $\varphi(\mathbf{a}) = \psi(\mathbf{a}) \neq \mathbf{0}$. We now argue the inductive step and suppose $\ell(\mathbf{x}_S)$ $(S \neq \emptyset)$ is an intermediate non-terminal along the derivation of $\psi(\mathbf{x}_{[n]})$. By inductive hypothesis, we have that $\psi(\Pi_S\mathbf{a}) \neq \mathbf{0}$. We then distinguish the following two cases:

- If $\ell(\mathbf{x}_S) \overset{*}{\Rightarrow} \psi(\mathbf{x}_S)$ uses the production $\ell(\mathbf{x}_S) ::= R_K(\mathbf{x}_K) \otimes \ell(\mathbf{x}_{S^-})$ next, then $\psi(\mathbf{a}_{S^-}) \neq \mathbf{0}$ and $R_K(\Pi_K \mathbf{a}) \neq \mathbf{0}$. Therefore, $\Pi_K \mathbf{a} \in R_K^*$.
- If $\ell(\mathbf{x}_S) \overset{*}{\Rightarrow} \psi(\mathbf{x}_S)$ uses the production $\ell(\mathbf{x}_S) ::= (R_N(\mathbf{x}_N) \vdash \ell(\mathbf{x}_{N^-})) \otimes \ell(\mathbf{x}_{S^-})$ next, then $\psi(\mathbf{a}_{S^-}) \neq \mathbf{0}$ and $(R_N(\Pi_N \mathbf{a}) \vdash \psi(\Pi_{N^-} \mathbf{a})) \neq \mathbf{0}$. The latter further implies that $\Pi_N \mathbf{a} \notin R_N^*$ because if not, then

  $$R_N(\Pi_N \mathbf{a}) \vdash \psi(\Pi_{N^-} \mathbf{a}) = \mathbf{0} \oplus \mathbf{0} \otimes \psi(\Pi_{N^-} \mathbf{a}) = \mathbf{0}$$

  and we get a contradiction.

Thus, we have proven that $\mathbf{a} \in Q^*(\mathcal{D}^*)$ and completed the proof. □

Example 5.11. Continuing our running example, we cast the enumeration of the (full) $\mathsf{NestFAQ}^\neg$ query φ'' into the enumeration of the full signed-acyclic $\mathsf{CQ}^\neg$ Q^*, by directly extracting factors from φ'' as relations:

$$Q^*(x_1, x_2) \leftarrow R_1^*(x_1) \wedge R_2^*(x_2) \wedge \neg R_{12}^{**}(x_1, x_2) \wedge \neg R_2^{**}(x_2)$$

where R_1^* stores $a_1 \in R_1$ with $R_1(a_1) \neq \mathbf{0}$ (similarly for R_2^*), and R_{12}^{**} stores $(a_1, a_2) \in R_{12}''$ with $R_{12}''(a_1, a_2) = \mathbf{0}$ (similarly for R_2^{**}). Intuitively, Q^* forbids exactly the tuples (a_1, a_2) such that $\varphi''(x_1, x_2) = \mathbf{0}$ to be emitted. Thus, we can safely enumerate answers of Q^* using techniques in Section 5.4 and use φ'' to recover the non-$\mathbf{0}$ weights.

An immediate corollary from Theorem 5.8 is the following, as a special case of Theorem 5.4.

Corollary 5.2. *Let φ be a* full *signed-acyclic* $\mathsf{NestFAQ}^\neg$ *query over a commutative semiring σ. Then, there is an algorithm that can enumerate the answers of φ in $O(|\varphi|)$ delay, after $O(|\varphi|^3 + |\varphi| \cdot \mathcal{D}_\varphi)$ preprocessing time.*

5.5.7 Putting Everything Together

Finally, we put together Section 5.5.3, Section 5.5.5 and Section 5.5.4 to culminate in the main theorem for the signed-elimination sequence. First, we prove the following statement.

Theorem 5.9. *Let $\mathcal{H}_n$ be a signed hypergraph with a signed-leaf n. Let $\psi_\dagger(\mathbf{x}_{[n]})$ be a* $\mathsf{NestFAQ}^\neg$ *expression recognized by $\mathsf{CFG}^\dagger{}_{\mathcal{H}_{n,\dagger}}$. There is an (aggregation) algorithm that takes $\psi_\dagger(\mathbf{x}_{[n]})$ as input and outputs $\psi_{n-1}(\mathbf{x}_{[n-1]})$ such that*

1. *$\psi_{n-1}(\mathbf{x}_{[n-1]})$ is the* $\mathsf{NestFAQ}^\neg$ *expression associated with $\mathcal{H}_{n-1} = \langle \mathcal{H}, n \rangle$ obtained by directly peeling off in x_n from $\psi_\dagger(\mathbf{x}_{[n]})$, thus it is recognized by $\mathsf{CFG}_{\mathcal{H}_{n-1}}$ and $|\psi_{n-1}| = O(|\psi_n|)$,*

2. $\bigoplus_{x_n} \psi_n(\mathbf{x}_{[n]}) = \psi_{n-1}(\mathbf{x}_{[n-1]})$ *and* $|\mathcal{D}_{\psi_{n-1}}| = O(|\mathcal{D}_{\psi_\dagger}|)$.

Moreover, the algorithm runs in $O(|\psi_n| + d(n) \cdot |\mathcal{D}_{\psi_\dagger}| \cdot \alpha(14|\mathcal{D}_{\psi_\dagger}|, |\mathcal{D}_{\psi_\dagger}|))$ *time, where* $d(n)$ *is the number of hyperedges containing* n *in* $\mathcal{H}_n$*.*

The inverse Ackermann factor $\alpha(14|\mathcal{D}_{\psi_\dagger}|, |\mathcal{D}_{\psi_\dagger}|)$ in the aggregation algorithm is inherited from the algorithm supporting RangeSum queries. Thus, the guarantee of the RangeSum problem carries over to our aggregation algorithm: for semigroups that accepts additive inverse, or for the semigroup with minimum (or maximum) as the operation (RMQ as in [GBT84]), the aggregation algorithm runs in $O(|\psi_\dagger| \cdot \mathcal{D}_{\psi_\dagger})$ time, without the additional inverse Ackermann overhead.

Proof. The aggregation algorithm follows (5.13) line-by-line and "pushes" the aggregation operator $\bigoplus_{x_n}$ into the inner-most parenthesis, therefore costing time $O(|\psi_n|)$ to scan over the derivation of $\psi_n(\mathbf{x}_{[n]})$. Factoring out the term $\psi_n(\mathbf{x}_{S_i})$, where $i = 0, \ldots, k = d(n)$ and $n \notin S_i$ is a pure syntactic step. The last step aggregates out x_n from the pivot factor and gets a new positive factor $R_{U\setminus\{n\}}(\mathbf{x}_{U\setminus\{n\}})$ in $O(|R_U|)$ time and space.

The only tricky step in (5.13) when pushing the aggregation operator $\bigoplus_{x_n}$ into the inner-most parenthesis is to efficiently compute the list representation of $R_{N_i\setminus\{n\}}(\mathbf{x}_{N_i\setminus\{n\}})$, where $i = 1, \ldots, k = d(n)$, and by definition,

$$R_{N_i\setminus\{n\}}(\mathbf{a}_{N_i\setminus\{n\}}) = \begin{cases} \bigoplus_{x_n} \psi_n(x_n, \mathbf{a}_{N_i\setminus\{n\}}) & \text{if } \mathbf{a}_{N\setminus\{n\}} \in \Pi_{N\setminus\{n\}} R_N \\ \mathbf{0} & \text{otherwise} \end{cases}$$

where $\psi_n(x_n, \mathbf{a}_{N_i\setminus\{n\}}) = R_{N_i}(x_n, \mathbf{a}_{N\setminus\{n\}}) \oplus \mathbb{1}_{\neg R_{N_i}}(x_n, \mathbf{a}_{N_i\setminus\{n\}}) \otimes \psi_n(x_n, \mathbf{a}_{N_i^-\setminus\{n\}})$. Now we apply Theorem 5.7 to get the oracles $\mathcal{T}_i(\mathbf{x}_{N_i\setminus\{n\}})$ for each $i = 1, \ldots, k$. Then, the weight $R_{N_i\setminus\{n\}}(\mathbf{a}_{N_i\setminus\{n\}})$ can be obtained in $O(\alpha(14|\mathcal{D}_{\psi_\dagger}|, |\mathcal{D}_{\psi_\dagger}|))$ time, for each $\mathbf{a}_{N_i\setminus\{n\}} \in \Pi_{N_i\setminus\{n\}} R_{N_i}$, by calling the oracle $\mathcal{T}_i(\mathbf{x}_{N_i\setminus\{n\}})$ with inputs $\left(\mathbf{a}_{N_i\setminus\{n\}}, [1, \perp]\right)$. Indeed, the oracle runs in time $O(\alpha(14|\mathcal{D}_{\psi_\dagger}|, |\mathcal{D}_{\psi_\dagger}|))$ and returns the range sum

$$\bigoplus_{\omega\in[1,\perp]} \psi_n\left(A_0(\Pi_{U\setminus\{n\}}\mathbf{a}_{N_i\setminus\{n\}})[\omega],\ \mathbf{a}_{N_i\setminus\{n\}}\right) = \bigoplus_{x_n\in\mathsf{Dom}(x_n)} \psi_n(x_n, \mathbf{a}_{N_i\setminus\{n\}}) =: R_{N_i\setminus\{n\}}(\mathbf{a}_{N_i\setminus\{n\}})$$

Therefore, the list representation of the new factor $R_{N_i\setminus\{n\}}$ is computed in $O(|R_{N_i}| \cdot \alpha(14|\mathcal{D}_{\psi_\dagger}|, |\mathcal{D}_{\psi_\dagger}|))$ time and is of size $O(|R_{N_i}|)$. The overall time complexity of the aggregation algorithm is $O(d(n) \cdot |\mathcal{D}_{\psi_\dagger}| \cdot \alpha(14|\mathcal{D}_{\psi_\dagger}|, |\mathcal{D}_{\psi_\dagger}|))$ since there are $d(n)$ steps in the derivation of $\psi_n(\mathbf{x}_{[n]})$ that requires the construction of a new factor $R_{N_i\setminus\{n\}}$ and each construction takes at most $O(|\mathcal{D}_{\psi_\dagger}| \cdot \alpha(14|\mathcal{D}_{\psi_\dagger}|, |\mathcal{D}_{\psi_\dagger}|))$ time (to further push the aggregation operator $\bigoplus_{x_n}$ into the nested subexpression). After we aggregate out the variable x_n from $\psi_n(\mathbf{x}_{[n]})$, we automatically get a new NestFAQ$^\neg$ expression $\psi_{n-1}(\mathbf{x}_{[n-1]})$. Since

we keep the derivation of $\psi_n(\mathbf{x}_{[n]})$ intact, we have $|\psi_{n-1}| = O(|\psi_n|)$ and $\psi_{n-1}(\mathbf{x}_{[n-1]})$ is indeed a $\mathsf{NestFAQ}^\neg$ expression recognized by CFG_{n-1}. Moreover, as all the new factors $R_{N_i\setminus\{n\}}$ are still of size $O(|R_{N_i}|)$, we have $|\mathcal{D}_{\psi_{n-1}}| = O(|\mathcal{D}_{\psi_n}|)$. □

We finally prove the main theorem (Theorem 5.4) for free-connex signed-acyclic $\mathsf{NestFAQ}^\neg$.

Proof of Theorem 5.4. Recall that for a signed-leaf n, we run the following algorithms in order:

1. First, by Theorem 5.5, we apply the Refactor algorithm in time $O(|\psi|^2 + d(n) \cdot |\mathcal{D}_\psi|)$, where $d(n)$ is the number of hyperedges in $\mathcal{H}$ that contain n. It transforms $\psi(\mathbf{x}_{[n]})$ into a $\mathsf{NestFAQ}^\neg$ expression $\psi_\dagger(\mathbf{x}_{[n]})$ such that $\psi_\dagger(\mathbf{x}_{[n]}) = \psi(\mathbf{x}_{[n]})$. Moreover, we have that $|\psi_\dagger| \leq |\psi|$ and $|\mathcal{D}_{\psi_\dagger}| \leq |\mathcal{D}_\psi|$.
2. Next, we run the oracle-construction algorithm in Section 5.5.5 for $\psi_\dagger(\mathbf{x}_{[n]})$. By Theorem 5.7, the construction of oracles takes $O(|\psi_\dagger| + |\mathcal{D}_{\psi_\dagger}| \cdot \alpha(14|\mathcal{D}_{\psi_\dagger}|, |\mathcal{D}_{\psi_\dagger}|))$ time.
3. Finally, we apply the aggregation algorithm on $\psi_\dagger(\mathbf{x}_{[n]})$ and by Theorem 5.9, we obtain in $O(|\psi_\dagger| + d(n) \cdot |\mathcal{D}_{\psi_\dagger}| \cdot \alpha(14|\mathcal{D}_{\psi_\dagger}|, |\mathcal{D}_{\psi_\dagger}|))$ time $\psi_{n-1}(\mathbf{x}_{[n-1]})$, where $\psi_{n-1}(\mathbf{x}_{[n-1]})$ is a $\mathsf{NestFAQ}^\neg$ expression associated with $\mathcal{H}_{n-1} = \langle \mathcal{H}_n, n-1 \rangle$ such that $\psi_{n-1}(\mathbf{x}_{[n-1]}) = \bigoplus_{\mathbf{x}_{[n]\setminus F} \in} \psi_n(\mathbf{x}_{[n]})$. By Theorem 5.9, we have that $|\psi_{n-1}| \leq |\psi|$ and $|\mathcal{D}_{\psi_{n-1}}| \leq |\mathcal{D}_\psi|$.

Thus, the signed-elimination step on n takes $O(|\psi|^2 + d(n) \cdot |\mathcal{D}_\psi| \cdot \alpha(14|\mathcal{D}_\psi|, |\mathcal{D}_\psi|))$ time in total.

Now we perform the next signed-elimination step for $n-2$ with signed-leaf $n-1$. We repeat the signed-elimination step for $i = n-2, \dots, f$ on every intermediate $\mathsf{NestFAQ}^\neg$ expression $\psi_i(\mathbf{x}_{[i]})$. At any point, the formula is bounded by $O(|\psi|)$ and the input size by $O(|\mathcal{D}_\psi|)$. In the end, we get a full $\mathsf{NestFAQ}^\neg$ expression $\psi_f(\mathbf{x}_{[f]})$ such that

$$\varphi(\mathbf{x}_F) = \bigoplus_{\mathbf{x}_{[n]\setminus F} \in \mathsf{Dom}(\mathbf{x}_{[n]\setminus F})} \psi(\mathbf{x}_{[n]}) = \bigoplus_{\mathbf{x}_{[n-1]\setminus F} \in \mathsf{Dom}(\mathbf{x}_{[n-1]\setminus F})} \psi_{n-1}(\mathbf{x}_{[n-1]}) = \dots = \psi_f(\mathbf{x}_{[f]}).$$

Therefore, the total time for the signed-elimination sequence (for $i = n-1, \dots, f$) is bounded by (by definition, $|\varphi| = |\psi|$)

$$\sum_{i=f}^{n-1} O(|\psi|^2 + d(i) \cdot |\mathcal{D}_\psi| \cdot \alpha(14|\mathcal{D}_\psi|, |\mathcal{D}_\psi|)) = O(|\psi|^3 + |\psi| \cdot |\mathcal{D}_\psi| \cdot \alpha(14|\mathcal{D}_\psi|, |\mathcal{D}_\psi|)).$$

Finally, we follow the enumeration algorithm for full $\mathsf{NestFAQ}^\neg$ queries as proposed in Section 5.5.6 and Section 5.2. □

5.6 Lower Bounds

In this section, we present conditional and unconditional lower bounds which complement our upper bounds. All lower bounds here refer to *self-join-free* queries, which means that each relation name appears at most once in the body of the query. Technical proofs of this section can be found in our PODS 2024 paper [ZFOK23].

5.6.1 Lower Bounds for $\mathsf{CQ}^\neg$

In this section, we show lower bound results for any $\mathsf{CQ}^\neg$ that is not free-connex signed-acyclic. We split this result into two theorems that use different conditional lower bounds.

Problem $(k+1,k)$-Hyperclique

Input a k-uniform hypergraph[3] (for $k>2$)

Output does it contain a hyperclique of size $k+1$, i.e. a set of $k+1$ vertices where every subset of size k forms a hyperedge

Conjecture 5.1 ($(k+1,k)$-Hyperclique)**.** *There is no algorithm that solves $(k+1,k)$-Hyperclique in $O(m)$ time, where m is the number of edges in the input hypergraph.*

Readers are referred to [BGS20] for evidence why Conjecture 5.1 is believable. When $k=2$, the (3,2)-Hyperclique problem is the problem of finding a triangle in a graph.

Conjecture 5.2 (Triangle)**.** *There is no algorithm that decides whether a graph with n nodes contains a triangle in $O(n^2)$ time.*

The following is a combination of results from [Bra13, BGS20].

Theorem 5.10. *Let Q be a $\mathsf{CQ}^\neg$ that is not signed-acyclic. Assuming Conjecture 5.1 and Conjecture 5.2, then there is no algorithm for Q that has linear preprocessing time and $O(1)$ delay.*

To show a lower bound for queries that are not free-connex (but are signed-acyclic), we will use a weaker lower bound conjecture that implies Triangle.

Conjecture 5.3 (BMM)**.** *There is no algorithm that computes the product $A\times B$ of two $n\times n$ Boolean matrices A and B in $O(n^2)$ time.*

[3] A hypergraph is said to be *k-uniform* if every hyperedge contains exactly k vertices.

Evidence for Conjecture 5.3 can be found in [Raz03]. Bagan et al. [BDG07] reduce the BMM problem to the non-free-connex acyclic query $Q(x, y) \leftarrow \bigvee_z A(x, z) \wedge B(z, y)$ and apply Conjecture 5.3 to obtain a conditional lower bound. The *matrix multiplication exponent* ω is the smallest number such that for any $\epsilon > 0$, there is an algorithm that multiplies two n-by-n matrices with at most $O(n^{\omega+\epsilon})$ operations (assuming RAM model). The best bound known so-far on ω is (roughly) $\omega < 2.373$ in [Wil12, Gal14]. We note that Conjecture 5.3 does not violate the common belief that $\omega = 2$, since that only implies that BMM can be computed in time $n^{2+o(1)}$. For non-free-connex CQs, a weaker lower bound conjecture was used, sparse BMM (the matrices have m non-zero entries and no $O(m)$ algorithm exists). However, this conjecture cannot be applied in our case because we need to take the complement of the matrix to populate a negated atom, and that means that a sparse matrix becomes dense.

Theorem 5.11. *Let Q be a $\mathsf{CQ}^\neg$ that is signed-acyclic and not free-connex. Assuming Conjecture 5.3, there is no algorithm with linear preprocessing time and $O(1)$ delay.*

5.6.2 Lower Bounds for $\mathsf{FAQ}^\neg$

We present next lower bounds for $\mathsf{FAQ}^\neg$ when restricted to queries with head $\varphi()$, which we denote as $\mathsf{SumProd}^\neg$. To show these bounds, we will use weaker conjectures than the ones used in the previous section.

Conjecture 5.4 (Minimum-Weight k-Clique)**.** *There is no algorithm that computes the minimum weight of a k-clique in a edge-weighted graph with n nodes in $O(n^k)$ time.*

Our reduction from Minimum-Weight k-Clique [ABDN18] is an application of the clique embedding power technique introduced in [FKZ23].

Theorem 5.12. *Assuming Conjecture 5.4, a $\mathsf{SumProd}^\neg$ φ over the tropical semiring can be solved in linear time if and only if φ is signed-acyclic.*

Over the counting ring, we can instead show that any lower bound for counting Boolean CQs transfers immediately to $\mathsf{SumProd}^\neg$.

Theorem 5.13. *Suppose that no linear-time algorithm can count the solutions of a non α-acyclic Boolean CQ. Then, there is no linear-time algorithm for a $\mathsf{SumProd}^\neg$ query over the counting ring that is not signed-acyclic.*

5.6.3 An Unconditional Lower Bound for $\mathsf{FAQ}^\neg$

Finally, we show an unconditional lower bound that provides some evidence on the necessity of the inverse Ackermann factor in the runtime of Theorem 5.2.

The lower bound is based on the additive structure of the underlying semiring for $\mathsf{FAQ}^\neg$, i.e. a commutative semigroup with operator $\oplus$. It uses the *arithmetic model of computation* [Yao85, CR91], which charges one unit of computation for every $\oplus$ operation performed, while all other computation is free. Essentially, the computation can be viewed as a sequence of instructions of the form $z_i = az_j \oplus bz_k$, where $\{z_i\}_i$ form an unbounded set of variables. Moreover, this sequence should be agnostic to the actual values of the semiring. The only thing we need is that the semigroup is *faithful* [Yao85, CR91], meaning that for every $T_1, T_2 \subseteq \{1, 2, \dots, n\}$, and integers $\delta_i, \delta'_j > 0$, $\bigoplus_{i \in T_1} \delta_i \cdot a_i = \bigoplus_{j \in T_2} \delta'_j \cdot a_j$ cannot be an identity for all $a_1, a_2, \dots, a_n \in S$ unless $T_1 = T_2$. This is essentially saying that there is no "magical shortcut" to compute the sums.

Theorem 5.14. *Under the arithmetic model of computation, any constant delay enumeration algorithm (in data complexity) for*

$$\varphi(x) = \bigoplus_y A(x) \otimes B(y) \otimes \mathbb{1}_{\neg R}(x, y)$$

on factors A, B each of size n and factor R of size $2m$ must require $\Omega(m \cdot \alpha(m, m))$ preprocessing time for every $m \leq n$.

5.7 Difference of CQs

As an application of our results, we consider the class of queries of the form $Q_1 - Q_2$, where Q_1, Q_2 are full CQs with the same set of free variables. It is shown in a recent paper [HW23] that $Q_1 - Q_2$ can be computed in time $O(|\mathcal{D}| + \mathsf{OUT})$, where OUT is the output size of $Q_1 - Q_2$, if Q_1 is α-acyclic and $Q_1 \wedge R_e$ is α-acyclic for every R_e in Q_2. We use our main theorem to strengthen this result by providing a constant-delay enumeration guarantee.

Theorem 5.15. *Let $Q = Q_1 - Q_2$, where Q_1, Q_2 are full CQs over the same set of free variables. If Q_1 is α-acyclic and $Q_1 \wedge R_e$ is α-acyclic for every R_e in Q_2, then the result Q can be enumerated with constant delay after $O(|\mathcal{D}|)$ preprocessing time.*

Proof of Theorem 5.15. Following [HW23], we can write Q as follows:

$$Q = \bigcup_{R_e \in Q_2} \big(\bigwedge_{S_e \in Q_1} S_e \wedge \neg R_e \big)$$

Hence, Q can be viewed as a union of $\mathsf{CQ}^\neg$, each $\mathsf{CQ}^\neg$ of the form $Q_1 \wedge \neg R_e$. Since Q_1 is α-acyclic and $Q_1 \wedge R_e$ is α-acyclic, $Q_1 \wedge \neg R_e$ is signed-acyclic and free-connex (trivially, since it is full) and thus by Theorem 5.1 can be enumerated with linear-time preprocessing and constant delay. We can now apply the Cheater's Lemma [CK21] to obtain linear-time preprocessing and constant-delay enumeration for Q. $\square$

As a corollary, we obtain the following generalization to differences of non-full CQs.

Corollary 5.3. *Let $Q = Q_1 - Q_2$, where Q_1, Q_2 are CQs with the same set of free variables. If Q is difference-linear (see [HW23] Definition 2.3), then the output of Q can be enumerated with constant delay after $O(|\mathcal{D}|)$ preprocessing time.*

Proof of Corollary 5.3. From [HW23], the property of being difference-linear implies a linear-time reduction from Q to a difference of CQs $Q'_1 - Q'_2$ (i.e. $Q = Q'_1 - Q'_2$) such that Q'_1, Q'_2 are full CQs, and satisfy the properties of Theorem 5.15. We then simply apply Theorem 5.15. □

5.8 $\mathsf{CQ}^\neg$ Beyond Linear Time

Now, what can we reason for evaluating a $\mathsf{CQ}^\neg$ if it is not signed-acyclic? Lanzinger [Lan21] defined a width measure of a $\mathsf{CQ}^\neg$ called *nest-set width* (nsw) and showed that a Boolean $\mathsf{CQ}^\neg$ with bounded nsw can be evaluated in polynomial time (combined complexity). One consequence of (5.2) is that we can obtain in a straightforward way a much tighter upper bound in terms of data complexity.

Definition 5.5. Let Q be a $\mathsf{CQ}^\neg$ with signed hypergraph $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$. Then, $\beta\text{-}\#\mathsf{subw}(Q) = \max_{S \subseteq \mathcal{E}^-} \#\mathsf{subw}(Q_S)$, where Q_S is the CQ associated with hypergraph $([n], \mathcal{E}^+ \cup S)$ and $\#\mathsf{subw}(Q)$ is the *counting* version of the submodular width defined by Abo Khamis et al. [KCM+20].

As shown in [KNS17, KCM+20], for any Boolean CQ Q, we can evaluate $\#Q$ in time $\widetilde{O}(|\mathcal{D}|^{\#\mathsf{subw}(Q)})$, where $\widetilde{O}$ hides a polylogarithmic factor in $|\mathcal{D}|$. Combining this with (5.2), we have:

Theorem 5.16. *Let Q be a Boolean $\mathsf{CQ}^\neg$. Then, we can evaluate $\#Q$ (and thus Q) in time $\widetilde{O}(|\mathcal{D}|^{\beta\text{-}\#\mathsf{subw}(Q)})$.*

From Lanzinger [Lan21], we know that $\beta\text{-}\#\mathsf{subw}(Q)$ is always at most its nest-set width (see [Lan21] Theorem 4.1); moreover, there are queries were nest-set width is unbounded but $\beta\text{-}\#\mathsf{subw}$ is bounded (see [Lan21] Lemma 4.3).

Example 5.12. We now revisit one of the motivating examples of this dissertation: the open triangle query (Figure 1.2, right). The query is a $\mathsf{CQ}^\neg$ of the following

$$Q() \leftarrow R(x_1, x_2) \wedge S(x_2, x_3) \wedge \neg T(x_3, x_1).$$

Since the query is not signed-acyclic, a linear-time evaluation is out of reach. Nonetheless, we present an algorithm that evaluates it in time $O(|\mathcal{D}|^{3/2})$ (no log factors), by applying heavy-light

partitioning on the variable x_2! Indeed, we partition $S(x_2, x_3)$ based on the degree of x_2—that is, the number of x_3 values associated with each x_2 in S. If the degree is at most $\sqrt{|\mathcal{D}|}$, we assign the tuple to the light part S_L; otherwise, it goes to the heavy part S_H.

For the light part, we join R with S_L and then antijoin with T. Since each x_2 in S_L has low degree, the join and antijoin together cost $O(|\mathcal{D}|^{3/2})$ time. For the heavy part, observe that there are at most $|\mathcal{D}|^{1/2}$ values of x_2 in S_H. For each such x_2 value (say a_2), we evaluate the subquery

$$Q_{a_2}() \leftarrow R(x_1, a_2) \wedge S(a_2, x_3) \wedge \neg T(x_3, x_1).$$

which is signed-acyclic. As shown earlier in this chapter, signed-acyclic queries can be evaluated in linear time. Since there are at most $|\mathcal{D}|^{1/2}$ such subqueries, each costing $O(|\mathcal{D}|)$, the total cost for the heavy part is also $O(|\mathcal{D}|^{3/2})$.

5.9 Conclusion

$\mathsf{CQ}^{\neg}$ is a fundamental class of relational queries and yet recent literature [Bra13, Lan21, HW23, CI24] has demonstrated that its complexity is far from being well-understood. This chapter has made an initial foray into a novel way of interpreting $\mathsf{CQ}^{\neg}$ from the perspective of semiring and FAQs [AKNR16]. We presented a constant-delay enumeration algorithm for the class of free-connex signed-acyclic $\mathsf{FAQ}^{\neg}$, after linear preprocessing (modulo a surprising inverse Ackermann factor), and showed conditional and unconditional lower bounds for $\mathsf{FAQ}^{\neg}$ out of this class. As for semirings of special structures, the algorithm needs no extra inverse Ackermann overhead and such semirings include the Boolean (i.e. $\mathsf{CQ}^{\neg}$), tropical and counting semiring. We leave as an intriguing open question the parameterized complexity of general $\mathsf{CQ}^{\neg}$ and $\mathsf{FAQ}^{\neg}$.

Chapter 6

Join with Access Patterns

In this chapter, we study a class of problems that splits an algorithmic task into two phases: the *preprocessing phase*, which constructs a space-efficient data structure from the input, and the *online phase*, which uses the data structure to answer requests of a specific form over the input as fast as possible. Many fundamental algorithmic tasks such as set intersection problems [CP10a, GKLP17], reachability in directed graphs [AGHP11, Aga14, CP10b], histogram indexing [CL15, KRR13], and problems related to document retrieval [AN16, LMNT15] can be expressed in this way. The core algorithmic question related to these problems is to find *the tradeoff between the space S necessary for storing the data structures and the time T for answering a request.*

Let us look at one of the simplest tasks in this setup. Consider the 2-Set Disjointness problem: given a universe of elements U and a collection of m sets $S_1, \dots, S_m \subseteq U$, 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 $S_i \cap S_j$ is empty or not. It is well-known that the space-time tradeoff for 2-Set Disjointness is captured by the equation $S \cdot T^2 = O(N^2)$, where N is the total size of all sets [CP10a, GKLP17]. Similar tradeoffs have also been established for other data structure problems. In the k-Reachability problem [GKLP17, CP10a], we are given as 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. The data structure obtained was conjectured to be optimal by [GKLP17], and the conjectured optimality was used to develop conditional lower bounds for other problems, such as approximate distance oracles [AGHP11, Aga14] where no progress has been made in improving the upper bounds in the last decade. In the edge triangle detection problem [GKLP17], we are given as input a graph $G = (V, E)$, and the goal is to develop a data structure that can answer whether a given edge $e \in E$ participates in a triangle or not. Each of these problems has been studied in isolation and therefore, the algorithmic insights are not readily generalizable into a comprehensive framework.

In this chapter, we cast many of the above problems into answering *Conjunctive Queries with Access Patterns (CQAPs)* over a relational database. 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 CQAP: $\varphi(\mid y_1, y_2) \leftarrow R(x, y_1) \wedge R(x, y_2)$. The expression $\varphi(\mid y_1, y_2)$ can be interpreted as follows: given values for y_1, y_2, compute whether the query returns true or not. Different access patterns capture different ways of accessing the result of the CQ and result in different tradeoffs.

Tradeoffs for enumerating Conjunctive Query results under static and dynamic settings have been a subject of previous literature [OS16, GS13, DK18, KNOZ23a, KNOZ20, KNN+19]. However, previous work either focuses on the tradeoff between preprocessing time and answering time [KNOZ23a, KNOZ20, KNN+19], or the tradeoff between space and delay in enumeration [OS16, DK18]. In this chapter, we focus explicitly on the tradeoff between space and answering time, without optimizing for preprocessing time. Most closely related to our setting is the problem of answering Boolean CQs [DHK23]. In that work, the authors slightly improve upon the data structure proposed in [DK18] and adapt it for Boolean CQ answering. Further, [DHK23] identified that the conjectured tradeoff for the k-reachability problem is suboptimal by showing slightly improved tradeoffs for all $k \geq 3$. The techniques used in this chapter are quite different and a vast generalization of the techniques used in [DHK23]. The proposed improvements in [DHK23] for k-reachability are already captured in this work and in many cases, we surpass the ones from [DHK23].

Our Contribution Our key contribution is a general algorithmic framework for obtaining space-time tradeoffs for any CQAP. Our framework builds upon the PANDA algorithm [KNS17], heavy-light partitioning and tree decomposition techniques [GGS14, Mar13]. Given any CQAP, it computes a tradeoff that can find the best possible time under a given space budget. To achieve this goal, we have made two key technical contributions:

First, we introduce the novel notion of *partially-materialized tree decompositions* (PMTDs, in short) that allow us to capture different possible materialization strategies on a given tree decomposition (Section 6.2). At a high level, a PMTD augments a tree decomposition with information on which bags should be materialized and which should be computed online. In other words, it acts conceptually as a “two-phase query plan”. To use a PMTD, we propose a variant of the Yannakakis algorithm (Section 6.2.1) such that during the online phase, we incur only the cost of visiting the non-materialized bags.

The second key ingredient is an extension of the PANDA algorithm [KNS17] that evaluates a disjunctive rule in two phases. The evaluation of a disjunctive rule allows placing an answer to any of the targets in the head of the rule. A key technical component in the PANDA algorithm is the notion of a *Shannon-flow inequality.* For any Shannon-flow inequality, one can construct a proof sequence that has a direct correspondence with relational operators. Consequently, a proof sequence can be transformed into a join algorithm. The disjunctive rules we consider are computed in two phases: in the first phase (preprocessing), we can place an answer only to targets that will

be materialized during the preprocessing phase. In the second phase (online), we place an answer to the remaining targets. We call these rules *2-phase disjunctive rules* (Section 6.3.1). To achieve this 2-phase computation, we introduce a type of Shannon-flow inequalities, called *joint Shannon-flow inequalities* (Section 6.4), such that each inequality gives rise to a space-time tradeoff. The joint Shannon-flow inequality generates two parallel proof sequences, one proof sequence for the preprocessing phase and another proof sequence for the answering phase. This transformation allows us to use the PANDA algorithm as a blackbox on each of the proof sequences independently and is instrumental in achieving space-time tradeoffs.

We demonstrate the versatility of our framework by recovering state-of-the-art space-time tradeoffs for 2-Set Disjointness as well as its generalization k-Set Disjointness and k-Reachability (Section 6.7). For Boolean CQAPs, we show that our techniques subsume and improve the prior results obtained by [DHK23]. The most notable finding is that we can obtain complex tradeoffs for k-Reachability that exhibit different behavior for different regimes of S. For the 3-Reachability problem, we show an improvement of the tradeoff over a significant part of the spectrum. For the 4-Reachability problem, we are able to show (via a rather involved analysis) that the space-time tradeoff can be strictly improved *everywhere* when compared to the conjectured optimal! These results falsify the proposed optimal tradeoff of $S \cdot T^{2/(k-1)} = \widetilde{O}(|E|^2)$ for k-Reachability for regimes that are even larger than what was shown in [DHK23].

Organization We introduce the basic terminology and problem definition in Section 6.1. In Section 6.2, we describe the augmented tree decompositions that are necessary for our framework. Section 6.3 introduces the general framework while Section 6.4 presents the algorithms used in our framework. Section 6.5 and Section 6.6 are dedicated to the formal proofs of the main results. We present the applications of the framework in Section 6.7. The related work is described in Section 6.8 and we conclude with a list of open problems in Section 6.9.

6.1 Background

We define the basic notations and concepts used in this chapter. Some of these differ from the ones used in the previous chapters in order to allow convenience and maintain consistency with the literature on CQAPs.

Conjunctive Query Recall that a Conjunctive Query (CQ) φ can be represented using a hypergraph $\mathcal{H} = ([n], \mathcal{E})$, where $[n] = \{1, \dots, n\}$ is the set of nodes and $\mathcal{E} \subseteq 2^{[n]}$ are the hyperedges. The body of the query has atoms R_F, where $F \in \mathcal{E}$. To each node $i \in [n]$, we associate a variable x_i. The

CQ is then

$$\varphi(\mathbf{x}_H) \leftarrow \bigwedge_{F \in \mathcal{E}} R_F(\mathbf{x}_F),$$

where $\mathbf{x}_I$ denotes the tuple $(x_i)_{i \in I}$ for any $I \subseteq [n]$. The variables in $\mathbf{x}_H$ are called the *head variables* of the CQ. The CQ is *full* if $H = [n]$ and *Boolean* if $H = \emptyset$. We use φ to denote the output of the CQ φ. We assume in this chapter that (query) hypergraphs are connected.

Degree Constraints A *degree constraint* [KNS17] is a triple $(X, Y, N_{Y|X})$ where $X \subset Y \subseteq [n]$ and $N_{Y|X}$ is a natural number. A relation R_F is said to *guard* the degree constraint $(X, Y, N_{Y|X})$ if $X \subset Y \subseteq F$ and for every tuple $\mathbf{t}_X$ (over X), $\max_{\mathbf{t}_X} \deg_F(Y|\mathbf{t}_X) \leq N_{Y|X}$, where $\deg_F(Y|\mathbf{t}_X) = |\Pi_Y(\sigma_{X=\mathbf{t}_X} R_F)|$. We use DC to denote a set of degree constraints and say that DC is guarded by a database instance $\mathcal{D}$ if every $(X, Y, N_{Y|X}) \in \mathsf{DC}$ is guarded by some relation in $\mathcal{D}$. A degree constraint $(X, Y, N_{Y|X})$ is a *cardinality constraint* if $X = \emptyset$. This chapter makes the following assumptions henceforth on DC guarded by a database instance $\mathcal{D}$:

- *(best constraints assumption)* w.l.o.g, for any $X \subset Y \subseteq [n]$, there is at most one $(X, Y, N_{Y|X}) \in \mathsf{DC}$. This assumption can be maintained by only keeping the minimum $N_{Y|X}$ if there is more than one.

- for every relation $R_F \in \mathcal{D}$, there is a *cardinality constraint* $(\emptyset, F, |R_F| := N_{F|\emptyset}) \in \mathsf{DC}$. The *size* of the database $\mathcal{D}$ is denoted as $|\mathcal{D}| := \max_{R_F \in \mathcal{D}} |R_F|$.

In this chapter, we use degree constraints to measure data complexity [Var82], i.e. the query size (the total number of relations and the variables) is a constant. All logs are in base 2, unless otherwise stated.

6.1.1 CQs with Access Patterns

We define CQs with access patterns following the definition from [KNOZ23a]:

Definition 6.1 (CQ with access patterns)**.** A Conjunctive Query with *Access Patterns* (CQAP) is an expression of the form

$$\varphi(\mathbf{x}_H \mid \mathbf{x}_A) \leftarrow \bigwedge_{F \in \mathcal{E}} R_F(\mathbf{x}_F),$$

where $A \subseteq [n]$ is called the *access pattern* of the query.

The access pattern tells us how a user accesses the result of the CQ. In particular, the user will provide an instance of a relation $Q_A(\mathbf{x}_A)$, which we call an *access request*. The task is then to return the result of the following CQ, denoted as φ, where

$$\varphi(\mathbf{x}_H) \leftarrow Q_A(\mathbf{x}_A) \wedge \bigwedge_{F \in \mathcal{E}} R_F(\mathbf{x}_F).$$

We call φ the *access CQ*. The most natural access request is one where $|Q_A| = 1$; in other words, the user provides only one fixed value for every variable $x_i, i \in A$. This can be thought of as using the CQ result as an index with search key $\mathbf{x}_A$. By allowing the access request Q_A to consist of more tuples, we can capture other scenarios. For example, one can take a stream of access requests of size 1 and batch them together to obtain a (possibly faster) answer for all of them at once. Prior work [DK18, KNOZ23a] has only considered the case where $|Q_A| = 1$, i,e. single-tuple access requests.

6.1.2 Problem Statement

Let $\varphi(\mathbf{x}_H \mid \mathbf{x}_A)$ be a CQAP under degree constraints DC guarded by the input relations. In addition, we denote by AC another set of degree constraints known in prior, guarded by any access request Q_A. Similar to DC, we assume that there is a cardinality constraint $(\emptyset, A, |Q_A| = N_{A|\emptyset}) \in \mathsf{AC}$ guarded by Q_A. For example, the case where $|Q_A| = 1$ can be interpreted as a cardinality constraint $(\emptyset, A, 1) \in \mathsf{AC}$. Given a database instance $\mathcal{D}$ guarding DC, our goal is to construct a data structure, such that we can answer any access request as fast as possible. More formally, we split query processing into two phases (see Figure 1.3 for a pictorial illustration of the two-phase setup):

Preprocessing phase: it constructs a data structure in space $\widetilde{O}(S)$[1]. The overall space cost takes the form $\widetilde{O}(S + |\mathcal{D}|)$, where S is called the *intrinsic space cost* of the data structure and $|\mathcal{D}|$ is the (unavoidable) space cost for storing the database.

Online phase: given an access request Q_A (guarding AC), it returns the results of the access CQ φ using the data structure built in the preprocessing phase. The (worst-case) answering time is then $\widetilde{O}(T + |Q_A|) + O(|\varphi|)$, where T is called the *intrinsic time cost* and $|Q_A| + |\varphi|$ is the (unavoidable) time cost of reading the access request Q_A and enumerating the output. For the Boolean case and when $|Q_A| = 1$, the answering time simply becomes $\widetilde{O}(T)$.

In this chapter, we study the tradeoffs between the two intrinsic quantities, S and T, which we will call an *intrinsic tradeoff*. At one extreme, the algorithm stores nothing, thus $S = O(1)$, and we answer each access request from scratch. At the other extreme, the algorithm stores the results of the CQ $\varphi_M(\mathbf{x}_{H\cup A}) \leftarrow \bigwedge_{F\in\mathcal{E}} R_F(\mathbf{x}_F)$ as a hash table with index key $\mathbf{x}_A$. For any access request Q_A, we simply evaluate the query $\varphi(\mathbf{x}_H) \leftarrow Q_A \wedge \varphi_M$ in the online phase by probing each tuple of Q_A in the hash table. If $H \supseteq A$, then any access request can be answered in (instance-optimal) time $O(|Q_A| + |\varphi|)$, in which case $T = O(1)$.

Example 6.1 (k-$\mathsf{Set\ Disjointness}$)**.** In this problem, we are given sets $S_1, \dots, S_m$ with elements drawn from the same universe U. Each access request asks whether the intersection between k sets is empty

[1]The notation $\widetilde{O}$ hides a polylogarithmic factor in $|\mathcal{D}|$.

or not. By encoding the family of sets as a binary relation $R(y, x)$ such that element y belongs to set x, we can express the problem as the following CQAP:

$$\varphi(\mid \mathbf{x}_{[k]}) \leftarrow \bigwedge_{i \in [k]} R(y, x_i). \tag{6.1}$$

If we also want to enumerate the elements in their intersection, we would instead use the non-Boolean version:

$$\varphi(y \mid \mathbf{x}_{[k]}) \leftarrow \bigwedge_{i \in [k]} R(y, x_i). \tag{6.2}$$

Example 6.2 (k-Reachability)**.** Given a directed graph G , the k-reachability problem asks, given a pair of vertices (u, v), to check whether they are connected by a path of length k. Representing the graph as a binary relation $R(x, y)$, we can model this problem through the following CQAP (the k-path query):

$$\phi_k(\mid x_1, x_{k+1}) \leftarrow \bigwedge_{i \in [k]} R(x_i, x_{i+1}).$$

We can also check whether there is a path of length at most k by combining the results of k such queries (one for each $1, \ldots, k$).

In this chapter, we focus on the CQAP such that $H \supseteq A$. If given a CQAP where $H \not\supseteq A$, we replace the head of the CQAP with $\varphi(\mathbf{x}_{H \cup A} \mid \mathbf{x}_A)$ and apply projection on the desired results at the end.

6.2 Partially Materialized Tree Decompositions

In this section, we introduce a type of tree decomposition that augments a decomposition with information about what bags we want to materialize. The reader may revisit the definition of tree decomposition in Section 2.2.2.

Let $\varphi(\mathbf{x}_H \mid \mathbf{x}_A)$ be a CQAP such that $A \subseteq H$. Recall that H is the set of head variables. Let $\mathcal{H}$ be the hypergraph associated with $\varphi(\mathbf{x}_H)$, the access CQ. Take a tree decomposition $(\mathcal{T}, \chi)$ of $\mathcal{H}$ and a node $r \in V(\mathcal{T})$. We define $\mathsf{TOP}_r(x)$ as the node closest to r in $\mathcal{T}$ containing x in its bag if we root the tree at r. We now say that $(\mathcal{T}, \chi)$ is *free-connex w.r.t. r* if for any $x \in H$ and $y \in [n] \setminus H$, $\mathsf{TOP}_r(y)$ is not an ancestor of $\mathsf{TOP}_r(x)$ [WY21]. We say that $(\mathcal{T}, \chi)$ is free-connex if it is free-connex w.r.t. some $r \in V(\mathcal{T})$. This is an equivalent definition of free-connexity as the one given in Section 2.2.2 that is more convenient for our purposes.

We can now introduce our key concept of a partially materialized tree decomposition, tailored for CQAPs.

Definition 6.2 (PMTD)**.** A *Partially Materialized Tree Decomposition (PMTD)* of the CQAP $\varphi(\mathbf{x}_H \mid \mathbf{x}_A)$ with $H \supseteq A$ is a tuple $(\mathcal{T}, \chi, M, r)$ such that the following properties hold:

1. $(\mathcal{T}, \chi)$ is a free-connex tree decomposition of $\mathcal{H}$ w.r.t. node r, called the root;
2. $A \subseteq \chi(r)$; and
3. $M \subseteq V(\mathcal{T})$ such that whenever $t \in M$, then all the nodes of its subtree (w.r.t. orienting the tree away from r) are in M.

Given a PMTD $(\mathcal{T}, \chi, M, r)$, we call M the *materialization set*. We also associate with each node $t \in V(\mathcal{T})$ a *view* with variables $\mathbf{x}_{\nu(t)}$, where the mapping $\nu : V(\mathcal{T}) \to 2^{[n]}$ is defined as follows. If the node $t \notin M$, then $\nu(t) := \chi(t)$ and the view is of the form $T_{\nu(t)}(\mathbf{x}_{\nu(t)})$, called a T-*view*. Otherwise, $t \in M$. If $t = r \in M$, define $\nu(r) := \chi(t) \cap H$. Let p be the parent node of a non-root node $t \in M$ and define

$$\nu(t) := \begin{cases} \chi(t) \cap (H \cup \chi(p)) & \text{if } p \notin M \\ \chi(t) \cap H & \text{if } p \in M \text{ and } \chi(t) \cap H \not\subseteq \chi(p) \cap H \\ \emptyset & \text{if } p \in M \text{ and } \chi(t) \cap H \subseteq \chi(p) \cap H. \end{cases}$$

The view (for each $t \in M$) then is of the form $S_{\nu(t)}(\mathbf{x}_{\nu(t)})$, called the S-*view*. This definition of S-views will correspond to running a bottom-up semijoin-reduce pass of the Yannakakis algorithm in the materialization set M of the free-connex tree decomposition $(\mathcal{T}, \chi)$. Indeed, any variables in $\chi(t) \setminus \nu(t)$ are safely projected out after the semijoin-reduce.

On a high level, M specifies the type of views associated with each bag (S-view or T-view), and $\nu(\cdot)$ pinpoints the schema of that view (possibly empty). A PMTD appoints its S-views to be materialized in the preprocessing phase and its T-views to be computed in the online phase. In the case where $M = \emptyset$, every view in the decomposition is obtained in the online phase. When $H = A$ or $H = [n]$, the free-connex property does not put any additional restrictions on the tree decompositions for a PMTD.

Example 6.3. We use the CQAP for 3-Reachability as a running example (to be more general, we use R_{12}, R_{23}, and R_{34} as the relations instead of one relation R):

$$\phi_3(x_1, x_4 \mid x_1, x_4) \leftarrow R_{12}(x_1, x_2) \wedge R_{23}(x_2, x_3) \wedge R_{34}(x_3, x_4).$$

Here, (x_1, x_4) is the access pattern. Figure 6.1 shows three PMTDs for the above query, along with the associated views of each bag in each PMTD. The leftmost PMTD has an empty materialization set. The middle PMTD materializes the bag $\{x_1, x_2, x_3\}$ but the associated view S_{13} projects out x_2. The rightmost PMTD materializes the only bag $\{x_1, x_2, x_3, x_4\}$ but the view S_{14} keeps only the variables x_1, x_4.

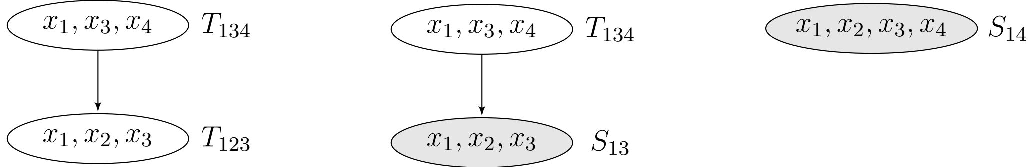


Figure 6.1: Three PMTDs for the 3-Reachability CQAP. The materialized nodes are shaded and labeled as S-views.

Redundancy & Domination We say that a tree decomposition is *non-redundant* if no bag is a subset of another bag. We say that a tree decomposition $(\mathcal{T}_1, \chi_1)$ is *dominated* by another tree decomposition $(\mathcal{T}_2, \chi_2)$ if every bag of $(\mathcal{T}_1, \chi_1)$ is a subset of some bag of $(\mathcal{T}_2, \chi_2)$. Both notions are introduced in [KNS17] for tree decompositions. Here, we will generalize them to PMTDs.

Definition 6.3 (PMTD Redundancy)**.** A PMTD $(\mathcal{T}, \chi, M, r)$ is *non-redundant* if (1) for any $t \in M$, no $\nu(t)$ is a subset of another; and (2) for $t \notin M$, no $\nu(t)$ is a subset of another.

Definition 6.4 (PMTD Domination)**.** A PMTD $(\mathcal{T}_1, \chi_1, M_1, r_1)$ is dominated by another PMTD $(\mathcal{T}_2, \chi_2, M_2, r_2)$ if (1) for every node $t_1 \in M_1$, there is some node $t_2 \in M_2$ such that $\nu(t_1) \subseteq \nu(t_2)$, and (2) for every node $t_1 \in V(\mathcal{T}_1) \setminus M_1$, there is some node $t_2 \in V(\mathcal{T}_2) \setminus M_2$ such that $\nu(t_1) \subseteq \nu(t_2)$.

For PMTDs, both redundancy and domination are defined using the materialization set and views instead of the bags. For PMTDs with $M = \emptyset$, both PMTD redundancy and domination become equivalent to the standard definition.

Example 6.4. Continuing Example 6.3, suppose we consider a PMTD that takes the same tree decomposition as the left PMTD, but with both bags in the materialization set. The S-view associated with the root bag is S_{14}, and $S_\emptyset$ for the child bag; thus, this PMTD is redundant. Moreover, suppose we consider a PMTD with one bag $\{x_1, x_2, x_3, x_4\}$ which is the root, but is not in M. The T-view associated with this bag is T_{1234}; thus, this PMTD dominates the left PMTD in Figure 6.1. On the other hand, all PMTDs in Figure 6.1 are non-redundant and non-dominant to each other.

As we later suggest in our general framework, we mostly focus on sets of non-redundant and non-dominant PMTDs. If $\nu(t) = \emptyset$ for some $t \in M$, then the associated S-view is a scalar of logical `true` and can be safely ignored.

6.2.1 Online Yannakakis for PMTDs

We introduce an adaptation of the Yannakakis algorithm [Yan81] for a non-redundant PMTD (so no empty views), called Online Yannakakis. Recall that for a non-redundant PMTD, the S-views,

one for each $t \in M$, are stored in the preprocessing phase, while the T-views, one for each $t \in \mathcal{T} \setminus M$, and the access request Q_A are accessible only in the online phase.

Theorem 6.1. *Consider a PMTD $(\mathcal{T}, \chi, M, r)$ and its view $\nu(\cdot)$. Given S-views as input tables, we can preprocess them using space linear in their size such that we can compute the free-connex acyclic CQ*

$$\psi(\mathbf{x}_H) \leftarrow Q_A \wedge \bigwedge_{t \in M} S_{\nu(t)} \wedge \bigwedge_{t \in V(\mathcal{T}) \setminus M} T_{\nu(t)} \tag{6.3}$$

for any input tables of T-view and Q_A in time $O(\max_{t \in V(\mathcal{T}) \setminus M} |T_{\nu(t)}| + |Q_A| + |\psi|)$, where $|\psi|$ is the output size of (6.3)*.*

The runtime of Theorem 6.1 has no dependence on the size of S-views, because throughout Online Yannakakis, S-views will be only used for hash probing in semijoin operations.

First, we specify the necessary preprocessing for the given S-views stated in Lemma 6.1. We group every edge of the PMTD by whether the edge connects two S-views, two T-views or one S-view and one T-view. We name them the SS-edges, TT-edges and ST-edges, respectively. A key observation of a PMTD is that each leaf-to-root path follows the pattern: first SS-edges (possibly none), then at most one ST-edges followed by TT-edges (possibly none). Next, we run a bottom-up semijoin pass on the SS-edges. In other words, we run the bottom-up semijoin pass of the Yannakakis algorithm and stop when visiting ST-edges. Then, for each S-view, we create a hash index with search key the common variables with its (unique) parent. The hash indices enables a linear-time (linear only w.r.t. the size of the parent view) semijoins of a parent view with a child S-view on a ST-edge.

Now we illustrate the Online Yannakakis algorithm. In particular, it has two passes: a bottom-up semijoin-reduce pass, followed by a top-down join pass.

Bottom-up Semijoin-Reduce Pass We first apply a bottom-up semijoin-reduce pass to remove all non-head variables in the tree by semijoins and projections. There are three scenarios as we go upwards, depending on the type of the edge we visit. Let $(t, p) \in E(\mathcal{T})$ be the edge we are visiting, where t is the child node and p is the parent of t. We distinguish the following cases:

1. (t, p) is an SS-edge: we simply skip the edge (because we have preprocessed this edge during the bottom-up semijoin-reduce pass in M).

2. (t, p) is an ST-edge: we update the view $T_{\nu(p)} \leftarrow T_{\nu(p)} \ltimes S_{\nu(t)}$. If every head variable in $\nu(t)$ is also in $\nu(p)$ and t is a leaf node, we remove $S_{\nu(t)}$ from the tree.

3. (t, p) is a TT-edge: we update $T_{\nu(p)} \leftarrow T_{\nu(p)} \ltimes T_{\nu(t)}$. If every head variable in $\nu(t)$ is also in $\nu(p)$ and t is a leaf node, we remove $T_{\nu(t)}$ from the tree; otherwise, we update $\nu(t) \leftarrow \nu(t) \cap H$ and $T_{\nu(t)} \leftarrow \Pi_{\nu(t) \cap H}(T_{\nu(t)})$.

As the last step of the bottom-up pass, for the root node r, if $r \in M$, we update $Q_A \leftarrow Q_A \ltimes S_{\nu(r)}$; or if $r \notin M$, we update $\nu(r) \leftarrow \nu(r) \cap H$, $T_{\nu(r)} \leftarrow \Pi_{\nu(t) \cap H}(T_{\nu(r)})$ and $Q_A \leftarrow Q_A \ltimes T_{\nu(r)}$. Now, a bottom-up semi-join reducer is accomplished on the reduced tree. We prove that this reduced tree contains only head variables in the next lemma.

Lemma 6.1. *The bottom-up semijoin-reduce pass results in a reduced tree where every view in the tree is a subset of H.*

Proof. Obviously, T-views in the reduced tree contain only head variables. We only need to show the property for every S-view. This is also straightforward for a root S-view or a non-root S-view that has a parent S-view by definition of $\nu(\cdot)$. We are left to show for an ST-edge (t, p) where either (i) there is a head variable $y \in \nu(t) \setminus \nu(p)$, or (ii) t is not a leaf node.

- (i) Observe that t is the top-most bag to contain y. Suppose the S-view $S_{\nu(t)}$ contains some $z \notin H$, then z must be in $\nu(p) = \chi(p)$ by definition of $\nu(\cdot)$. This contradicts the free-connex property since $\mathsf{TOP}_r(z)$ is an ancestor of $\mathsf{TOP}_r(y) = t$, i.e. a non-head variable $z \in \nu(p)$ is above the head variable $y \in \nu(t)$.
- (ii) t is a non-leaf node implies that there is a head variable $y \notin \nu(t)$ where t is an ancestor of $\mathsf{TOP}_r(y)$, otherwise, the subtree rooted at t violates PMTD redundancy. Thus, there must be no non-head variable in $\nu(t)$ by the free-connex property.

□

Top-down Join Pass If $r \in M$, we compute $Q_A \bowtie S_{\nu(r)}$, or if $r \notin M$, we compute $Q_A \bowtie T_{\nu(r)}$. From here, we apply the exact top-down full-join pass of Yannakakis on the reduced tree (from r, use the parent view to probe the child views) to get the output ψ. The following example walks through the Online Yannakakis algorithm.

Example 6.5. We use the non-redundant PMTD shown in Figure 6.2 of a CQAP $\varphi(x_1, x_2, x_3, x_4, x_7, x_8 \mid x_1, x_2)$, where (x_1, x_2) is the access pattern. We use the following free-connex acyclic CQ to demonstrate Online Yannakakis:

$$\psi(\mathbf{x}_H) \leftarrow Q_{12}(x_1, x_2) \wedge T_{12}(x_1, x_2) \wedge T_{13}(x_1, x_3) \wedge T_{345}(x_3, x_4, x_5) \wedge S_{45}(x_4, x_5) \wedge S_{37}(x_3, x_7) \wedge S_{78}(x_7, x_8),$$

where $\mathbf{x}_H = (x_1, x_2, x_3, x_4, x_7, x_8)$. The following is the sequence of semijoin-reduces in the bottom-up semijoin-reduce pass (the SS-edge (S_{37}, S_{78}) is skipped)

$$TS\text{-edge } (T_{345}, S_{45}) : \quad T_{345}^{(1)} \leftarrow T_{345} \ltimes S_{45}, \quad \text{remove } S_{45}$$
$$TS\text{-edge } (T_{13}, S_{37}) : \quad T_{13}^{(1)} \leftarrow T_{13} \ltimes S_{37}$$

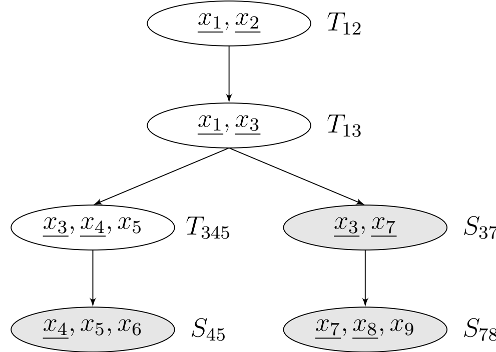


Figure 6.2: A non-redundant PMTD as an example for Online Yannakakis. The materialization set is shaded (and marked as S-views) and the head variables are underlined.

$$\begin{aligned}
TT\text{-edge } (T_{13}, T_{345}) :& \quad T_{13}^{(2)} \leftarrow T_{13} \ltimes T_{345}^{(1)}, \quad T_{34}^{(1)} \leftarrow \Pi_{34}(T_{345}^{(1)}) \\
TT\text{-edge } (T_{12}, T_{13}) :& \quad T_{12}^{(1)} \leftarrow T_{12} \ltimes T_{13}^{(2)} \\
\text{root} \ :& \quad Q_A^{(1)} \leftarrow Q_A \ltimes T_{12}^{(1)}
\end{aligned}$$

In the top-down pass, to get the result of ψ, Online Yannakakis evaluates the following joins from the root to the leaves, starting from $Q_A^{(1)}$,

$$Q_A^{(1)} \bowtie T_{12}^{(1)} \bowtie T_{13}^{(2)} \bowtie T_{34}^{(1)} \bowtie S_{37} \bowtie S_{78}.$$

We now formally show that the Online Yannakakis algorithm attests Theorem 6.1.

Proof of Theorem 6.1. Let $T = \max_{t \in V(\mathcal{T}) \setminus M} |T_{\nu(t)}|$. The bottom-up pass costs time $O(T + |Q_A|)$ as only T-views and Q_A are semijoin-reduced (S-views, by the index construction, are bottom-up semijoin-reduced at a preprocessing stage). Moreover, by Lemma 6.1, the reduced tree has only head variables and hence no further projections are needed. Finally, joining top-down from the root circumvents any intermediate dangling tuples and costs time $O(|\psi|)$. The overall time cost is $O(T + |Q_A| + |\psi|)$. □

6.3 General Framework

Consider a CQAP $\varphi(\mathbf{x}_H \mid \mathbf{x}_A) \leftarrow \bigwedge_{F \in \mathcal{E}} R_F(\mathbf{x}_F)$ with $H \supseteq A$. Recall that our goal is to find the best space-time tradeoffs under degree constraints DC (guarded by input relations) and AC (guarded by any access requests Q_A), as specified in Section 6.1.2. Our main algorithm is parameterized by:

- $\mathcal{P} = \{P_i\}_{i \in I}$, a (finite) indexed set of non-redundant and non-dominant PMTDs such that $P_i = (\mathcal{T}_i, \chi_i, M_i, r_i)$ for every $i \in I$. Including all such PMTDs in $\mathcal{P}$ (which are finite) will

result in the best possible tradeoff within this framework. However, as we will see later, it is also meaningful to consider smaller sets of PMTDs that result in more interpretable space-time tradeoffs.

- S, the space budget.

6.3.1 2-Phase Disjunctive Rules

In this section, we define a specific type of disjunctive rule that will be necessary to acquire the S-views and T-views for PMTDs. We start by recalling the notion of a disjunctive rule. A disjunctive rule has the exact body of a CQ, while the head is a disjunction of output relations $T_B(\mathbf{x}_B)$, which we call *targets*. Let $\mathsf{B}_T \subseteq 2^{[n]}$ be a non-empty set where every node of B_T is contained in some hyperedge, then a *disjunctive rule* ρ takes the form:

$$\rho: \quad \bigvee_{B \in \mathsf{B}_T} T_B(\mathbf{x}_B) \leftarrow \bigwedge_{F \in \mathcal{E}} R_F(\mathbf{x}_F). \tag{6.4}$$

Given a database instance $\mathcal{D}$, a *model* of ρ is a tuple $(T_B)_{B \in \mathsf{B}_T}$ of relations, one for each target, such that the logical implication indicated by (2.7) holds. More precisely, for any tuple $\mathbf{a}$ that satisfies the body, there is a target $T_B \in (T_B)_{B \in \mathsf{B}}$ such that $\Pi_B(\mathbf{a}) \in T_B$. The *size* of a model is defined as the maximum size of its output relations and the *output size* of a disjunctive rule ρ, denoted as $|\rho|$, is defined as the minimum size over all models.

For our purposes, we define a type of disjunctive rules, called *2-phase disjunctive rules*.

Definition 6.5 (2-phase Disjunctive Rules)**.** A *2-phase disjunctive rule* ρ defined by a CQAP $\varphi(\mathbf{x}_H \mid \mathbf{x}_A)$ is a single disjunctive rule that takes the body of the access CQ φ, while the head has two sets of output relations. In other words, ρ takes the form

$$\rho: \quad \bigvee_{B \in \mathsf{B}_S} S_B(\mathbf{x}_B) \vee \bigvee_{B \in \mathsf{B}_T} T_B(\mathbf{x}_B) \leftarrow Q_A(\mathbf{x}_A) \wedge \bigwedge_{F \in \mathcal{E}} R_F(\mathbf{x}_F), \tag{6.5}$$

where $\mathsf{B}_S, \mathsf{B}_T \subseteq 2^{[n]}$ and at most one can be empty. A *model* of ρ thus consists of two sets of output relations, i.e. the *S-targets* $(S_B)_{B \in \mathsf{B}_S}$ and the *T-targets* $(T_B)_{B \in \mathsf{B}_T}$.

As the name suggests, a model of a 2-phase disjunctive rule ρ is computed in two phases, the preprocessing and online phase:

***Preprocessing phase*:** we obtain the S-targets $(S_B)_{B \in \mathsf{B}_S}$ using a *preprocessing disjunctive rule*

$$\rho_S: \quad \bigvee_{B \in \mathsf{B}_S} S_B(\mathbf{x}_B) \leftarrow \bigwedge_{F \in \mathcal{E}} R_F(\mathbf{x}_F), \tag{6.6}$$

The space cost for storing the S-targets is $\widetilde{O}(S_\rho)$, and the overall space cost is $\widetilde{O}(S_\rho + |\mathcal{D}|)$. The preprocessing phase has no knowledge of Q_A except for the degree constraints AC, so as to explicitly force the S-targets to be universal for any instance of access request.

***Online phase*:** given an access request Q_A (under AC), we obtain the T-targets $(T_B)_{B \in \mathsf{B}_T}$ using an *online disjunctive rule*

$$\rho_T : \quad \bigvee_{B \in \mathsf{B}_T} T_B(\mathbf{x}_B) \leftarrow Q_A(\mathbf{x}_A) \wedge \bigwedge_{F \in \mathcal{E}} R_F(\mathbf{x}_F) \tag{6.7}$$

in time and space $\widetilde{O}(T_\rho)$. The overall time is $\widetilde{O}(T_\rho + |Q_A|)$.

If $\mathsf{B}_S = \emptyset$, then the model is computed from scratch in the online phase (and vice versa). As in Section 6.1.2, our focus is on analyzing the space-time tradeoffs between the two intrinsic quantities, S_ρ and T_ρ.

For the next part, assume that we have a 2-phase algorithm (called 2PP) that has a preprocessing procedure 2PP-PREPROC that constructs data structures of space $S_\rho = \widetilde{O}(S)$, S being a given space budget, and an online procedure 2PP-ONLINE that runs in time (and space) T_ρ. We will defer the dicussions for 2PP in the next section and for now, we focus on how 2PP fits into our general algorithmic framework.

6.3.2 Preprocessing Phase

As an initial step, we list out from $\mathcal{P}$ a set of 2-phase disjunctive rules as follows. Let ν_i be the mapping for associated views of P_i. Let us define the cartesian product $\mathbf{P} = \times_{i \in I}\{V(\mathcal{T}_i)\}$ and let $K = |I|$. Informally, every element $\mathbf{p} = (p_1, p_2, \ldots, p_K) \in \mathbf{P}$ picks one node from every PMTD in the indexed set. For every $\mathbf{p} \in \mathbf{P}$, we construct the following 2-phase disjunctive rule (recall that M_i is the materialization set of PMTD $(\mathcal{T}_i, \chi_i, M_i, r_i)$):

$$\bigvee_{p_i \in M_i} S_{\nu_i(p_i)}(\mathbf{x}_{\nu_i(p_i)}) \vee \bigvee_{p_i \notin M_i} T_{\nu_i(p_i)}(\mathbf{x}_{\nu_i(p_i)}) \leftarrow Q_A(\mathbf{x}_A) \wedge \bigwedge_{F \in \mathcal{E}} R_F(\mathbf{x}_F)$$

The body of the rule is identical independent of $\mathbf{p} \in \mathbf{P}$. The head of the rule introduces an S-target whenever the corresponding node is in the materialization set of the PMTD; otherwise, it introduces a T-target. The schema of a target is the view of the picked node in its PMTD. There are exactly K 2-phase disjunctive rules constructed from the given set of PMTDs, where K is independent of the database instance (and considered as a constant under data complexity).

Example 6.6. Continuing Example 6.3, consider three PMTDs in Figure 6.1. These result in 4 ($= K$) 2-phase disjunctive rules (after removing redundant T-targets and S-targets):

$$T_{134}(x_1, x_3, x_4) \vee S_{14}(x_1, x_4) \leftarrow Q_{14}(x_1, x_4) \wedge R_{12}(x_1, x_2) \wedge R_{23}(x_2, x_3) \wedge R_{34}(x_3, x_4)$$

$$T_{134}(x_1,x_3,x_4) \vee S_{13}(x_1,x_3) \vee S_{14}(x_1,x_4) \leftarrow Q_{14}(x_1,x_4) \wedge R_{12}(x_1,x_2) \wedge R_{23}(x_2,x_3) \wedge R_{34}(x_3,x_4)$$
$$T_{123}(x_1,x_2,x_3) \vee T_{134}(x_1,x_3,x_4) \vee S_{14}(x_1,x_4) \leftarrow Q_{14}(x_1,x_4) \wedge R_{12}(x_1,x_2) \wedge R_{23}(x_2,x_3) \wedge R_{34}(x_3,x_4)$$
$$T_{123}(x_1,x_2,x_3) \vee S_{13}(x_1,x_3) \vee S_{14}(x_1,x_4) \leftarrow Q_{14}(x_1,x_4) \wedge R_{12}(x_1,x_2) \wedge R_{23}(x_2,x_3) \wedge R_{34}(x_3,x_4)$$

Another interpretation of these 2-phase disjunctive rules is that we can view the set of PMTDs as a disjunctive normal form (DNF) formula, where each PMTD is a clause. The 2-phase disjunctive rules are then the equivalent conjunctive normal form (CNF) formula (by distributing the disjunction over the conjunction). In this case, we have

$$(T_{134} \wedge T_{123}) \vee (T_{134} \wedge S_{13}) \vee S_{14} =$$
$$(T_{134} \vee S_{14}) \wedge (T_{134} \vee S_{13} \vee S_{14}) \wedge (T_{123} \vee T_{134} \vee S_{14}) \wedge (T_{123} \vee S_{13} \vee S_{14}).$$

We will construct S-views next. For each 2-phase disjunctive rule ρ_k, where $k \in [K]$, we run 2PP-PREPROC, setting the space budget as S. As a result, 2PP-PREPROC populates the S-targets for each ρ_k of size $\widetilde{O}(S)$. We collect each S-view of a PMTD P_i by unioning all S-targets with the same schema as the S-view (possibly from S-targets of multiple disjunctive rules). Then, we semijoin every S-view with the full join $\Join_{F\in\mathcal{E}} R_F$. This semijoin step can be accomplished by tentatively storing $\Join_{F\in\mathcal{E}} R_F$ as an intermediate truth table and discard it after the semijoins of all S-views. This step guarantees that any tuple in an S-view participates in $\Join_{F\in\mathcal{E}} R_F$. Finally, we follow the preprocessing (i.e. the bottom-up semijoin-reduce pass) for the S-views as described in Section 6.2.1 and store them in space $\widetilde{O}(S)$.

Example 6.7. Continuing Example 6.6, 2PP-PREPROC outputs the S-targets for each rule, e.g. $S_{14}^{(1)}$ for the first rule, $S_{13}^{(2)}$ and $S_{14}^{(2)}$ for the second rule, and so on, of size $\widetilde{O}(S)$. Then for each PMTD, we collect the S-views from the unions over S-targets followed by semijoins with the full join, i.e.

$$S_{14}(x_1,x_4) = \bigcup_{k\in[4]} S_{14}^{(k)} \ltimes (R_{12} \Join R_{23} \Join R_{34}),$$
$$S_{13}(x_1,x_3) = \bigcup_{k\in\{2,4\}} S_{13}^{(k)} \ltimes (R_{12} \Join R_{23} \Join R_{34})$$

Following Section 6.2.1, because there are no neighboring S-views, no additional preprocessing is needed for both S-views. We hence store them in space $\widetilde{O}(S)$.

6.3.3 Online Phase

Recall that upon receiving an instance of access request Q_A, we need to return the results of the CQ, $\varphi(\mathbf{x}_H)$. We obtain $\varphi(\mathbf{x}_H)$ as follows. First, we apply 2PP-ONLINE for every ρ_k to get its T-targets (of size $\widetilde{O}(T_{\rho_k})$) in time $\widetilde{O}(T_{\rho_k} + |Q_A|)$. Let $T_{\mathsf{max}} = \max_{k\in[K]} T_{\rho_k}$.

Next, we compute each T-view of a PMTD P_i by unioning all T-targets with the same schema as the T-view (possibly from outputs of multiple disjunctive rules). We semijoin-reduce every T-view (of size $\widetilde{O}(T_{\mathsf{max}})$) with every input relation and Q_A. Then, for every PMTD in $\{P_i\}_{i\in I}$, we evaluate the free-connex acyclic CQ

$$\psi_i(\mathbf{x}_H) \leftarrow Q_A \wedge \bigwedge_{t\in M_i} S_{\nu_i(t)} \wedge \bigwedge_{t\in V_i(\mathcal{T}_i)\setminus M_i} T_{\nu_i(t)} \tag{6.8}$$

by applying Online Yannakakis as described in Section 6.2.1 in time $\widetilde{O}(T_{\mathsf{max}}) + O(|Q_A| + |\psi_i \cap \varphi|)$. We obtain the final result by unioning the outputs across all PMTDs in our set, $\varphi = \bigcup_{i\in I} \psi_i$. We formally reason the correctness of the general framework in Section 6.3.4. Overall, we answer the access request Q_A in time $\widetilde{O}(T_{\mathsf{max}} + |Q_A|) + O(|\varphi|)$ by Theorem 6.1.

Example 6.8. Continuing Example 6.7, let Q_{14} be an access request received in the online phase. We react by calling 2PP-ONLINE for each 2-phase disjunctive rule in Example 6.6, and the outputs are T-targets $T^{(1)}_{134}$ for the first rule, $T^{(2)}_{134}$ and $T^{(2)}_{123}$ for the second rule, and so on. Then for each PMTD, we collect the T-views from the unions over T-targets, followed by semijoins with the input relations and Q_{14}, i.e.

$$\begin{aligned} T_{134}(x_1, x_3, x_4) &= \bigcup_{k\in\{1,2,3\}} T^{(k)}_{134} \ltimes Q_{14} \ltimes R_{12} \ltimes R_{23} \ltimes R_{34}, \\ T_{123}(x_1, x_2, x_3) &= \bigcup_{k\in\{3,4\}} T^{(k)}_{134} \ltimes Q_{14} \ltimes R_{12} \ltimes R_{23} \ltimes R_{34}. \end{aligned}$$

Lastly, we evaluate one free-connex acyclic CQ per PMTD (there are three of them) from Figure 6.1:

$$\psi_1(x_1, x_4) \leftarrow T_{134} \wedge T_{123}, \qquad \psi_2(x_1, x_4) \leftarrow T_{134} \wedge S_{13}, \qquad \psi_3(x_1, x_4) \leftarrow S_{14}.$$

Recall that S_{14} and S_{13} are the S-views from Example 6.7. We obtain the final result by unioning the outputs across the three PMTDs, i.e. $\varphi(x_1, x_4) = \psi_1 \cup \psi_2 \cup \psi_3$ (each ψ_i contains partial results of φ).

6.3.4 Correctness of the General Framework

In this section, we show that the general framework yields the correct result φ, i.e. $\varphi = \bigcup_{i\in I} \psi_i$, for any access request Q_A. As a benefit from semijoin-reduces of every view in a PMTD, it is obvious that $\bigcup_{i\in I} \psi_i \subseteq \varphi$. For the opposite inclusion, we prove the following two claims, following the proof structure of Corollary 7.13 in [KNS17].

Claim 1 Let us pick out one target (either an S-target or a T-target), denote the target as U_k, from the head of every rule ρ_k, where $k \in [K]$. We call the tuple $(U_k)_{k\in[K]}$ that consists of one picked

target per rule a *full-choice* (e.g. each clause in the CNF formula in Example 6.6 is a full-choice). Then, for any *full choice* $(U_k)_{k\in[K]}$, there is a PMTD $P \in \{P_i\}_{i\in I}$ such that for every tree node $t \in M$, the S-target $S_{\nu(t)}$ is in the full-choice and for every tree node $t \notin M$, the T-target $T_{\nu(t)}$ is in the full-choice. Breaking ties arbitrarily, we call this PMTD *the* PMTD associated with the full choice $(U_k)_{k\in[K]}$.

Proof of Claim 1. Fix a full-choice $(U_k)_{k\in[K]}$. Suppose to the contrary that for every PMTD $P_i \in \mathcal{P}$, there is a tree node such that its corresponding target (either a T-target or an S-target), denoted as U_i^*, is not in the full-choice. Then we examine the 2-phase disjunctive rule ρ^* that takes $\bigvee_{i\in I} U_i^*$ as its head. By definition of a full-choice, one target must be picked from ρ^*. However, this is a contradiction because every head U_i^* in ρ^* does not show up in the fixed full-choice $(U_k)_{k\in[K]}$. □

Claim 2 For any full-choice $\boldsymbol{U} := (U_k)_{k\in[K]}$ with its associated PMTD $(\mathcal{T}_i, \chi_i, M_i, r_i)$, we define a CQ

$$\varphi_{\boldsymbol{U}}(\mathbf{x}_H) \leftarrow Q_A \wedge \bigwedge_{k\in[K]} U_k$$

Let $\mathcal{U}$ denote the set of all full-choices. Then,

$$\varphi \subseteq \bigcup_{\boldsymbol{U}\in\mathcal{U}} \varphi_{\boldsymbol{U}} \subseteq \bigcup_{i\in I} \psi_i \tag{6.9}$$

Proof of Claim 2. Take any output tuple $\mathbf{a}_H \in \varphi$. Then, there is some full tuple $\mathbf{a}$ satisfying the body of φ such that $\mathbf{a}_H = \Pi_H(\mathbf{a})$. For each rule ρ_k, where $k \in [K]$, let U_k^* be the target (either S-target or T-target) associated with the view ν_k^* such that $\Pi_{\nu_k^*}(\mathbf{a}) \in U_k^*$. Therefore, $\boldsymbol{U}^* = (U_k^*)_{k\in[K]}$ is a full-choice and $\mathbf{a}$ satisfies the body of $\varphi_{\boldsymbol{U}^*}$. Hence, $\mathbf{a}_H \in \varphi_{\boldsymbol{U}^*}$ and we have shown the first inclusion in (6.9). The second inclusion holds because we can drop the atoms in the body of $\varphi_{\boldsymbol{U}^*}$ that is not any view of $\boldsymbol{U}^*$'s associated PMTD. □

6.4 Constructing the Tradeoffs

Let ρ be a 2-phase disjunctive rule (6.5) under degree constraints DC (guarded by input relations) and AC (guarded by Q_A). In Section 6.5 and Section 6.6, we formally discuss our algorithm to obtain a model of ρ in two phases using PANDA, and the resulting space-time tradeoff. This section aims to give a high-level sketch and introduce key ideas through an example. In particular, we use the following rule as our running example, where $|R_{12}| = |R_{23}| = |\mathcal{D}|$:

$$T_{123}(x_1, x_2, x_3) \vee S_{13}(x_1, x_3) \leftarrow Q_{13}(x_1, x_3), R_{12}(x_1, x_2), R_{23}(x_2, x_3).$$

In fact, this is the only rule we get from considering two PMTDs for the 2-Reachability CQAP. To evaluate this disjunctive rule, PANDA starts with a *Shannon-flow inequality*, which is an inequality

over set functions $h : 2^{[n]} \rightarrow \mathbf{R}_+$ that must hold for any set function that is a polymatroid[2]. For our purposes, we need a *joint Shannon-flow inequality*, which holds over two set functions h_S, h_T that must be polymatroids. Intuitively, h_S governs the preprocessing phase, while h_T governs the online phase. The joint Shannon-flow inequality for our example is:

$$\underbrace{h_S(1) + h_T(12|1)}_{R_{12}} + \underbrace{h_S(3) + h_T(23|3)}_{R_{23}} + 2\underbrace{h_T(13)}_{Q_{13}} \geq \underbrace{h_S(13)}_{S_{13}} + 2\underbrace{h_T(123)}_{T_{123}}$$

where $h_S(Y|X) = h_S(Y) - h_S(X)$ and $h_T(Y|X) = h_T(Y) - h_T(X)$ for $X \subseteq Y$. The right-hand side of the inequality includes terms of h_S that correspond to S-targets and terms of h_T that correspond to T-targets. The left-hand side includes a term of $h_T(13)$ that corresponds to the access request Q_{13}, and possibly terms of h_S (h_T) that encode the degree constraints DC ($\mathsf{DC} \cup \mathsf{AC}$). More notably, it contains terms that correlate the two polymatroids by splitting an input relation with attributes Y into two parts, either (i) $h_S(X) + h_T(Y|X)$, or (ii) $h_T(X) + h_S(Y|X)$, where $X \subseteq Y$. Intuitively, the first split materializes the heavy X-values and sends everything else to the online phase, while the second split preprocesses the light X-values and sends the heavy X-values to the online phase. In our example, the relation R_{12} is split into $h_S(1) + h_T(12|1)$, and each part is sent to a different polymatroid. Using the coefficients of the above joint Shannon-flow inequality, we get the following intrinsic space-time tradeoff:

$$S \cdot T^2 \cong |Q_{13}|^2 \cdot |\mathcal{D}|^2.$$

We will use the $\cong$ notation henchforth to mean that $S \cdot T^2 = \widetilde{O}(|Q_{13}|^2 \cdot |\mathcal{D}|^2)$. Generally, we show that (i) a given joint Shannon-flow inequality immediately translates into a space-time tradeoff: *every joint Shannon-flow inequality for a 2-phase disjunctive rule directly implies a space-time tradeoff by reading the coefficients of the inequality*, and (ii) in addition, fixing a space budget S, we can compute via a linear program the inequality that will result in the best possible answering time.

The 2PP algorithm We now present how our main algorithm runs on the running example. We take $|Q_{13}| = 1$, and S as the fixed space budget.

As a first step, 2PP scans the joint Shannon-flow inequality and partitions $R_{12}(x_1, x_2)$ (on x_1) into R_{12}^H and R_{12}^L, where R_{12}^H contains all (x_1, x_2) tuples where $|\sigma_{x_1 = a_1}(R_{12})| \geq |\mathcal{D}|/\sqrt{S}$, and R_{12}^L contains the tuples that satisfy $\deg_{12}(x_2|x_1) < |\mathcal{D}|/\sqrt{S}$. R_{23} is partitioned symmetrically (on x_3) into R_{23}^H and R_{23}^L. This creates four subproblems, $\{R_{12}^H, R_{23}^H\}$, $\{R_{12}^H, R_{23}^L\}$, $\{R_{12}^L, R_{23}^H\}$ and $\{R_{12}^L, R_{23}^L\}$. In general, the algorithm splits relations according to the correlated terms in the joint flow.

The preprocessing phase (2PP-PREPROC) is governed by the Shannon-flow inequality for h_S, i.e. $h_S(1) + h_S(3) \geq h_S(13)$. We now follow PANDA and construct a *proof sequence* for this inequality.

[2]A polymatroid is a set function $h : 2^{[n]} \rightarrow \mathbf{R}_+$ that is non-negative, monotone, and submodular, with $h(\emptyset) = 0$.

Figure 6.3: Two PMTDs for the square CQAP. The materialized nodes are shaded and labeled as S-views.

A proof sequence proves the inequality via a sequence of smaller steps, such that each step can be interpreted as a relational operator. The proof sequence for our case is:

$$\begin{aligned} h_S(1) + h_S(3) &\geq h_S(13|3) + h_S(3) && (\textit{submodularity}) \\ &= h_S(13) && (\textit{composition}) \end{aligned}$$

In this case, PANDA attempts to join the two relations in each subproblem. However, we allow this to happen only if the resulting space is within the budget S. Because R^H_{12} and R^H_{23} have size at most $|\mathcal{D}|/(|\mathcal{D}|/\sqrt{S}) = \sqrt{S}$ values for x_1, x_3 respectively, the result of the subproblem $\{R^H_{12}, R^H_{23}\}$ is stored as $S_{13} = (\Pi_1 R^H_{12}) \bowtie (\Pi_3 R^H_{23})$ in space at most $\sqrt{S} \cdot \sqrt{S} = S$.

The online phase (2PP-ONLINE) takes an access request $Q_{13}(x_1, x_3)$ that contains one tuple. Now, 2PP-ONLINE follows the second proof sequence for the polymatroid h_T:

$$\begin{aligned} h_T(12|1) + h_T(23|3) + 2h_T(13) &\geq 2h_T(123|13) + 2h_T(13) && (\textit{submodularity}) \\ &= 2h_T(123) && (\textit{composition}) \end{aligned}$$

For the other 3 subproblems, 2PP-ONLINE computes the following 3 joins: $Q_{13}(x_1, x_3) \bowtie R^L_{23}(x_2, x_3)$, $Q_{13}(x_1, x_3) \bowtie R^L_{12}(x_1, x_2)$ and $Q_{13}(x_1, x_3) \bowtie R^L_{12}(x_1, x_2)$. In the submodularity step, 2PP-ONLINE identifies that for the first join, $\deg_{23}(x_2|x_3) < |\mathcal{D}|/\sqrt{S}$, so this join takes time $|Q_{13}| \cdot |\mathcal{D}|/\sqrt{S} \leq |\mathcal{D}|/\sqrt{S}$; and since $\deg_{12}(x_2|x_1) < |\mathcal{D}|/\sqrt{S}$, the last two (identical) joins take time $|Q_{13}| \cdot \deg_{12}(x_2|x_1) \leq |\mathcal{D}|/\sqrt{S}$. Therefore, the overall online computing time is $|\mathcal{D}|/\sqrt{S}$.

Example 6.9 (The square query)**.** We now show a comprehensive example of constructing tradeoffs for the following CQAP:

$$\varphi(x_1, x_3 \mid x_1, x_3) \leftarrow R_{12}(x_1, x_2) \wedge R_{23}(x_2, x_3) \wedge R_{34}(x_3, x_4) \wedge R_{41}(x_4, x_1).$$

This captures the following task: given two vertices of a directed graph, decide whether they occur in two opposite corners of a square-shaped loop. We consider two PMTDs. The first PMTD has a root

bag $\{1,3,4\}$ associated with a T-view T_{134}, and a bag $\{1,3,2\}$ associated with a T-view T_{132}. The second PMTD has one bag $\{1,2,3,4\}$ associated with an S-view S_{13}. The two PMTDs are depicted in Figure 6.3. This in turn generates two disjunctive rules:

$$T_{134}(x_1,x_3,x_4) \vee S_{13}(x_1,x_3) \leftarrow \mathsf{body}, \qquad T_{132}(x_1,x_3,x_2) \vee S_{13}(x_1,x_3) \leftarrow \mathsf{body}$$

where $\mathsf{body} = Q_{13}(x_1,x_3) \wedge R_{12}(x_1,x_2) \wedge R_{23}(x_2,x_3) \wedge R_{34}(x_3,x_4) \wedge R_{41}(x_4,x_1)$. We can construct the following joint Shannon-flow inequality (and its proof sequence) for the first rule:

$$\begin{aligned}
\underbrace{h_S(1) + h_T(14|1)}_{R_{41}} + \underbrace{h_S(3) + h_T(34|3)}_{R_{34}} + 2 \cdot \underbrace{h_T(13)}_{Q_{13}} \\
&\geq h_S(13) + h_T(14|1) + h_T(34|3) + 2 \cdot h_T(13) \\
&\geq h_S(13) + h_T(134|13) + h_T(13) + h_T(134|13) + h_T(13) \\
&= \underbrace{h_S(13)}_{S_{13}} + 2 \cdot \underbrace{h_T(134)}_{T_{134}}.
\end{aligned}$$

Similarly on x_2, for the second rule, we construct a proof sequence for $2\log|\mathcal{D}| + 2\log|Q_{13}| \geq h_S(13) + 2 \cdot h_T(123)$. Hence, reading the coefficients of the above joint Shannon-flow inequalities, we obtain the following intrinsic space-time tradeoff $S \cdot T^2 \cong |\mathcal{D}|^2 \cdot |Q_{13}|^2$ for the square CQAP.

6.5 Algorithms for 2-phase Disjunctive Rules

In this section, we formally present algorithms for a 2-phase disjunctive rule ρ (6.5). We assume for ρ degree constraints DC (guarded by input relations) and degree constraints AC (guarded by access request Q_A). First, we layout some necessary background.

6.5.1 Background

Entropic Functions Given a disjunctive rule (2.7), a set function $h : 2^{[n]} \to \mathbf{R}_+$ is *entropic* if there is a joint probability distribution on $[n]$ such that $h(F)$ is the marginal entropy of F for any $F \subseteq [n]$. Let Γ_n^* be the set of all entropic functions and $h(Y|X) := h(Y) - h(X)$. Under a given set of degree constraints DC, any joint distribution on $[n]$ conforms to the constraints $h(Y|X) \leq n_{Y|X}$, where $n_{Y|X} := \log N_{Y|X}$, for each $(X, Y, N_{Y|X}) \in \mathsf{DC}$.

Polymatroid A polymatroid is a set function $h : 2^{[n]} \to \mathbf{R}_+$ that is non-negative, monotone, and submodular, with $h(\emptyset) = 0$. To be precise, monotonicity implies that $h(Y) \geq h(X)$ for any $X \subseteq Y \subseteq [n]$ and let $h(Y|X) := h(Y) - h(X)$, then submodularity implies that $h(I|I \cap J) \geq h(I \cup J|J)$ for any $I, J \subseteq [n]$. Let Γ_n be the set of all polymatroids on $[n]$. As every entropic function is a polymatroid, it holds that $\Gamma_n^* \subseteq \Gamma_n$.

Size Bound for Disjunctive Rules Let ρ be a disjunctive rule (2.7). Let $\mathcal{D}$ be a database instance under a given set of degree constraints DC. The set

$$\mathsf{HDC} := \left\{ h : 2^{[n]} \to \mathbb{R}_+ | \bigwedge_{(X,Y,N_{Y|X}) \in \mathsf{DC}} h(Y|X) \leq \log N_{Y|X} \right\}$$

contains all entropic functions h on $[n]$ satisfying the degree constraints DC. Fix a closed subset $\mathcal{F} \subseteq \mathbf{R}_+^{2^n}$. We define the log-size-bound with respect to $\mathcal{F}$ of a disjunctive rule ρ to be the quantity:

$$\mathsf{LogSizeBound}_{\mathcal{F}}(\rho) := \max_{h \in \mathcal{F}} \min_{B \in \mathsf{B}_T} h(B)$$

Then, for the output size of ρ, we have the following theorem (see Theorem 1.5 in [KNS17]):

Theorem 6.2 ([KNS17])**.** *Let ρ be any disjunctive rule* (2.7) *under degree constraints* DC. *Then for any database instance $\mathcal{D}$ satisfying* DC*, the following holds:*

$$\begin{aligned} \log|\rho| &\leq \mathsf{LogSizeBound}_{\Gamma_n^* \cap \mathsf{HDC}}(\rho) && \textit{entropic bound} \\ &\leq \mathsf{LogSizeBound}_{\Gamma_n \cap \mathsf{HDC}}(\rho) && \textit{polymatroid bound} \end{aligned}$$

The entropic bound is tight under degree constraints in the worst case, but it is often hard to compute. The polymatroid bound is tight if ρ is a CQ (i.e. has a single target) and there are only cardinality constraints, in which case the polymatroid bound degenerates into the AGM bound. However, it is not tight under general degree constraints.

The PANDA Algorithm Given a disjunctive rule ρ (2.7), the PANDA algorithm takes a database instance $\mathcal{D}$, a set of degree constraint DC (guarded by $\mathcal{D}$) as inputs and computes a model in time and space predicated by its polymatroid bound, i.e.

$$\widetilde{O}(2^{\mathsf{LogSizeBound}_{\Gamma_n \cap \mathsf{HDC}}(\rho)}).$$

The reader can refer to Theorem 1.7 in [KNS17] for details. For now, we will treat PANDA algorithm as a blackbox; we will later present it in-depth.

6.5.2 A 2-phase Algorithmic Paradigm

In this section, we introduce a 2-phase algorithmic paradigm that we will follow to design 2-phase algorithms for a 2-phase disjunctive rule (6.5). The reader may refer to Section 6.4 for an intuitive and comprehensive example.

Let $\mathcal{D}$ be a database instance and Q_A be an arbitrary access request. We assume that *hash tables on necessary index keys (of input relations and degrees of tuples in input relations)* can be pre-built

at the start of the preprocessing phase as needed by the paradigm. That is, we assume constant-time accesses of tuples and degrees of tuples in the input relations during both phases. There are at most $O(2^{2n})$ hash tables to be pre-built per input relation R_F (that is, for every (Y, X)-pair where $X \subset Y \subseteq F \in \mathcal{E}$), so the space cost for storing all necessary hash tables is $O(|\mathcal{D}|)$ in data complexity. Recall that we also assume w.l.o.g the following *best constraint assumption*: we only keep at most one $(X, Y, N_{Y|X}) \in \mathsf{DC}$ for each $X \subset Y \subseteq [n]$ (keeping the minimum $N_{Y|X}$ if there are more than one).

Split Steps Let $R \in \mathcal{D}$ be the guard of a cardinality constraint $(\emptyset, Z, N_{Z|\emptyset}) \in \mathsf{DC}$. A *split step* on a (Y, X)-pair, where $\emptyset \neq X \subset Y \subseteq Z$, applies Lemma 6.1 of [KNS17] and partitions $R_Y := \Pi_Y(R)$ into $k = 2\log N_{Z|\emptyset}$ sub-tables, i.e. $R_Y^{(1)}, \ldots, R_Y^{(k)}$, such that $N_{X|\emptyset}^{(j)} \cdot N_{Y|X}^{(j)} \leq N_{Z|\emptyset}$, for all $j \in [k]$, where

$$N_{X|\emptyset}^{(j)} := \left|\Pi_X(R_Y^{(j)})\right|$$
$$N_{Y|X}^{(j)} := \deg_{R_Y^{(j)}}(Y|X).$$

For each of these sub-tables $R_Y^{(j)}$, we create a subproblem with inputs $(\mathcal{D}^{(j)}, \mathsf{DC}^{(j)})$, where $\mathcal{D}^{(j)} := \mathcal{D} \cup \{R_Y^{(j)}\}$ denotes the input tables and

$$\mathsf{DC}^{(j)} := \mathsf{DC} \cup \left\{(\emptyset, X, N_{X|\emptyset}^{(j)}), (X, Y, N_{Y|X}^{(j)})\right\}$$

denotes the degree constraints guarded by $\mathcal{D}^{(j)}$. *A sequence of split steps* on $(Y_1, X_1), (Y_2, X_2), \ldots,$ (Y_ℓ, X_ℓ), applies the first split step on the (Y_1, X_1)-pair, spawning $k_1 = O(\log|\mathcal{D}|)$ subproblems with inputs $(\mathcal{D}^{(j)}, \mathsf{DC}^{(j)})$, where $j \in [k_1]$. Then, for each subproblem, applies the second split step on the (Y_2, X_2)-pair, spawning $O((\log|\mathcal{D}|)^2)$ subproblems. This iterative process goes on until every split step in the sequence is applied, thus it generates $O(poly(\log|\mathcal{D}|))$ subproblems overall.

Now we formally characterize our 2-phase algorithmic paradigm. Let S be the given space budget. We denote the task of obtaining a model for a 2-phase disjunctive rule ρ (6.5) with input relations $\mathcal{D} \cup \{Q_A\}$ satisfying degree constraints $\mathsf{DC} \cup \mathsf{AC}$ as $\rho(\mathcal{D} \cup \{Q_A\}, \mathsf{DC} \cup \mathsf{AC})$. The paradigm starts by applying a sequence of *split steps* to partition $\rho(\mathcal{D}\cup\{Q_A\}, \mathsf{DC}\cup\mathsf{AC})$ into $O(poly(\log|\mathcal{D}|))$ subproblems. Then, the j-th subproblem with input $\mathcal{D}^{(j)}$ and degree constraint $\mathsf{DC}^{(j)} \supseteq \mathsf{DC}$, denoted as $\rho(\mathcal{D}^{(j)} \cup \{Q_A\}, \mathsf{DC}^{(j)} \cup \mathsf{AC})$, either

(1) populates S-targets $(S_B^{(j)})_{B \in \mathsf{B}_S}$ as a model of the preprocessing disjunctive rule ρ_S (6.6) using PANDA, provided that the output size of ρ_S is within $\widetilde{O}(S)$; or

(2) populates T-targets $(T_B^{(j)})_{B \in \mathsf{B}_T}$ as a model of the online disjunctive rule ρ_T (6.7) using PANDA.

In other words, the subproblem $\rho(\mathcal{D}^{(j)} \cup \{Q_A\}, \mathsf{DC}^{(j)} \cup \mathsf{AC})$ is conquered by applying PANDA to obtain either (1) a preprocessing model of ρ_S with input relations $\mathcal{D}^{(j)}$ under degree constraint $\mathsf{DC}^{(j)}$,

denoted as $\rho_S(\mathcal{D}^{(j)}, \mathsf{DC}^{(j)})$, or an online model of ρ_T with input relations $\mathcal{D}^{(j)} \cup \{Q_A\}$ under degree constraint $\mathsf{DC}^{(j)} \cup \mathsf{AC}$, denoted as $\rho_T(\mathcal{D}^{(j)} \cup \{Q_A\}, \mathsf{DC} \cup \mathsf{AC})$. After all subproblems are conquered, the model of ρ is simply the union over S-targets and T-targets populated from all subproblems.

6.5.3 Constructing Tradeoffs as Optimization under Constraints

Next, we analyze the intrinsic space-time tradeoff (as specified in Section 6.3.1, between S_ρ and T_ρ) that can be captured by the 2-phase paradigm introduced above. First, the split steps incur a poly-logarithmic factor on both S_ρ and T_ρ and spawn $O(poly(\log|\mathcal{D}|))$ subproblems. Then, the j-th spawned subproblem $\rho(\mathcal{D}^{(j)} \cup \{Q_A\}, \mathsf{DC}^{(j)} \cup \mathsf{AC})$ is conquered by PANDA in exactly one of the two phases. Note that $\mathsf{DC}^{(j)} \supseteq \mathsf{DC}$ contains extra degree constraints due to the split steps. For ease of analysis, we isolate the extra constraints by defining $\mathsf{SC}^{(j)} := \mathsf{DC}^{(j)} \setminus \mathsf{DC}$. To apply the polymatroid bound, we use two polymatroids, $h_S \in \Gamma_n$ to represent the preprocessing phase and $h_T \in \Gamma_n$ to represent the online phase. We define

$$\mathsf{HDC} := \left\{ h : 2^{[n]} \to \mathbb{R}_+ | \bigwedge_{(X,Y,N_{Y|X}) \in \mathsf{DC}} h(Y|X) \leq \log N_{Y|X} \right\}$$

$$\mathsf{HSC}^{(j)} := \left\{ h : 2^{[n]} \to \mathbb{R}_+ | \bigwedge_{(X,Y,N_{Y|X}) \in \mathsf{SC}^{(j)}} h(Y|X) \leq \log N_{Y|X} \right\}$$

$$\mathsf{HAC} := \left\{ h : 2^{[n]} \to \mathbb{R}_+ | \bigwedge_{(X,Y,N_{Y|X}) \in \mathsf{AC}} h(Y|X) \leq \log N_{Y|X} \right\}$$

where $h(Y|X) = h(Y) - h(X)$, to denote collections of set functions that satisfy DC, $\mathsf{SC}^{(j)}$ and AC, respectively. To recall, DC contains the degree constraints on inputsx, $\mathsf{SC}^{(j)}$ contains the constraints created during the splitting, and AC contains the constraints from the access patterns. The 2-phase paradigm enforces that h_S conforms to $\mathsf{HDC} \cap \mathsf{HSC}^{(j)}$ and h_T conforms to $\mathsf{HDC} \cap \mathsf{HSC}^{(j)} \cap \mathsf{HAC}$. Thus, the j-th subproblem costs space $\widetilde{O}(S_\rho^{(j)})$ in the preprocessing phase, where

$$\log S_\rho^{(j)} := \mathsf{LogSizeBound}_{h_S \in \Gamma_n \cap \mathsf{HDC} \cap \mathsf{HSC}^{(j)}}(\rho_S) \tag{6.10}$$

provided that $S_\rho^{(j)} \leq S$. Otherwise, j-th subproblem costs time (and space) $\widetilde{O}(T_\rho^{(j)})$, where

$$\log T_\rho^{(j)} := \mathsf{LogSizeBound}_{h_T \in \Gamma_n \cap \mathsf{HDC} \cap \mathsf{HSC}^{(j)} \cap \mathsf{HAC}}(\rho_T) \tag{6.11}$$

By conquering all subproblems, we conclude that $S_\rho = \max_j S_\rho^{(j)} \leq S$ and $T_\rho = \max_j T_\rho^{(j)}$.

The next question is: *what is the smallest possible T_ρ for a fixed space budget S?* As we will see, this question can be answered by constructing an optimization problem. We know that for each

$(\emptyset, Z, N_{Z|\emptyset}) \in \mathsf{DC}$, there are at most 2^{2n} (Y, X)-pairs with $\emptyset \neq X \subset Y \subseteq Z$. In other words, there is a constant number of distinct split steps in data complexity. To exploit the full potential of split steps, we assume for now applying a sequence of all distinct split steps. Also, recall that a sequence of split steps spawns $O(poly(\log |\mathcal{D}|))$ subproblems. Intuitively, this partitions $\mathcal{D}$ into its most fine-grained pieces. In particular, let $\rho(\mathcal{D}^{(j)} \cup \{Q_A\}, \mathsf{DC}^{(j)} \cup \mathsf{AC})$ be the j-th subproblem spawned after the sequence of all distinct split steps and $\mathsf{SC}^{(j)} = \mathsf{DC}^{(j)} \setminus \mathsf{DC}$. The following *splitting property* is a direct result of a sequence of all distinct split steps on DC: for any $(\emptyset, Z, N_{Z|\emptyset}) \in \mathsf{DC}$ and (Y, X)-pair with $\emptyset \neq X \subset Y \subseteq Z$, there are some $(\emptyset, X, N^{(j)}_{X|\emptyset}), (X, Y, N^{(j)}_{Y|X}) \in \mathsf{SC}^{(j)}$ such that $N^{(j)}_{X|\emptyset} \cdot N^{(j)}_{Y|X} \leq N_{Z|\emptyset}$. Though each subproblem varies in its own $\mathsf{SC}^{(j)}$, the splitting property holds across all subproblems. To encode the splitting property, we define the following *split constraints*.

Definition 6.6 (Split Constraints)**.** Let DC be a set of degree constraints. A *split constraint* is a triple $(X, Y|X, N_{Z|\emptyset})$ where $\emptyset \neq X \subset Y \subseteq Z, (\emptyset, Z, N_{Z|\emptyset}) \in \mathsf{DC}$. A relation R_F is said to guard a split constraint $(X, Y|X, N_{Z|\emptyset})$ if R_F guards $(\emptyset, Z, N_{Z|\emptyset})$. The set of all split constraints spanned from DC, is denoted as

$$\mathsf{SC} := \left\{(X, Y|X, N_{Z|\emptyset}) \mid \emptyset \neq X \subset Y \subseteq Z, (\emptyset, Z, N_{Z|\emptyset}) \in \mathsf{DC}\right\}.$$

Intuitively, each triple $(X, Y|X, N_{Z|\emptyset}) \in \mathsf{SC}$ encodes the splitting property for the (Y, X)-pair on $(\emptyset, Z, N_{Z|\emptyset}) \in \mathsf{DC}$. Since we assume that every $(\emptyset, Z, N_{Z|\emptyset}) \in \mathsf{DC}$ has at least one guard, every $(X, Y|X, N_{Z|\emptyset}) \in \mathsf{SC}$ is guarded by some $R_F \in \mathcal{D}$. In addition, we define

$$\begin{aligned}\mathsf{HSC} := \Big\{(h_S, h_T) : 2^{[n]} \times 2^{[n]} \to \mathbf{R}^2_+ \;\Big|\; \\ \forall (X, Y|X, N_{Z|\emptyset}) \in \mathsf{SC}, (h_S(X) + h_T(Y|X) \leq \log N_{Z|\emptyset}) \wedge (h_S(Y|X) + h_T(X) \leq \log N_{Z|\emptyset})\Big\},\end{aligned}$$

where $h_S(Y|X) := h_S(Y) - h_S(X), h_T(Y|X) := h_T(Y) - h_T(X)$.

We now state the following theorem for the best possible T_ρ.

Theorem 6.3. *For a 2-phase disjunctive rule* (6.5)*, under space budget S, the 2-phase algorithmic paradigm attains $T_\rho = 2^{\mathsf{OBJ}(S)}$, where*

$$\begin{aligned}\mathsf{OBJ}(S) = \max_{\substack{h_S \in \mathsf{HDC}, h_T \in \mathsf{HDC} \cap \mathsf{HAC} \\ (h_S, h_T) \in (\Gamma_n \times \Gamma_n) \cap \mathsf{HSC}}} \quad & \min_{B \in \mathsf{B}_T} h_T(B) \\ s.t. \quad & h_S(B) > \log S, \quad B \in \mathsf{B}_S.\end{aligned} \tag{6.12}$$

Proof. Merging (6.10) and (6.11), $\log T^{(j)}_\rho$ for the j-th subproblem can be expressed as

$$\begin{aligned}\log T^{(j)}_\rho = \max_{\substack{h_S \in \mathsf{HDC} \cap \mathsf{HSC}^{(j)} \\ h_T \in \mathsf{HDC} \cap \mathsf{HSC}^{(j)} \cap \mathsf{HAC} \\ h_S, h_T \in \Gamma_n}} \quad & \min_{B \in \mathsf{B}_T} h_T(B) \\ s.t. \quad & h_S(B) > \log S, \quad B \in \mathsf{B}_S.\end{aligned} \tag{6.13}$$

The maximin optimization (6.13) is subproblem-dependent, since it is constrained on $\mathsf{HSC}^{(j)}$. To avoid this dependency, we recall that the naïve algorithm dictates the splitting property. Thus, we use the (subproblem-independent) set HSC, a universal collection of set functions pairs satisfying the splitting property and thus, it correlates h_S and h_T. Since every subproblem satisfies the splitting property, it holds that $\mathsf{HSC}^{(j)} \times \mathsf{HSC}^{(j)} \subseteq \mathsf{HSC}$. By relaxing $\mathsf{HSC}^{(j)} \times \mathsf{HSC}^{(j)}$ to HSC, we get the desired upper bound (6.12) for T_ρ. □

By assigning h_T to be always 0, it is easy to see that the feasible region of (6.12) is empty if and only if

$$\mathsf{LogSizeBound}_{h_S \in \Gamma_n \cap \mathsf{HDC}}(\rho_S) := \max_{h_S \in \Gamma_n \cap \mathsf{HDC}} \min_{B \in \mathsf{B}_S} h_S(B) \leq \log S,$$

in which case we can simply store the S-views within space $\widetilde{O}(S)$. Otherwise, the feasibility of (6.12) is guaranteed. However, applying all distinct split steps naïvely has the following drawbacks in terms of practicality: (1) the exhaustive splitting steps can incur a large poly-logarithmic factor; (2) for every subproblem, we need to run PANDA from scratch with a new instance, (3) the space-time tradeoff Therorem 6.3 alluded to is not readily interpretable. In the following sections, we introduce the 2-phase PANDA algorithm, called 2PP, that also attains the intrinsic tradeoff as specified in (6.12), while efficiently addressing these drawbacks with much practical/interpretable intrinsic tradeoffs.

6.6 The 2-phase PANDA Algorithm

The 2-phase PANDA (2PP) algorithm, similar to PANDA, is built on a class of inequalities called the joint Shannon-flow inequalities. In this section, we first present some background on Shannon-flow inequalities, and then our extension to joint inequalities. Finally, we present the 2PP algorithm.

6.6.1 Joint Shannon-flow Inequalities

To motivate our study of joint Shannon-flow inequalities, we first give a characterization of the optimal objective value of the maximin optimization problem (6.12), $\mathsf{OBJ}(S)$, through the following class of LPs

$$\mathbf{L}(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S}, S) := \max_{\substack{h_S \in \mathsf{HDC}, h_T \in \mathsf{HDC} \cap \mathsf{HAC} \\ (h_S, h_T) \in (\Gamma_n \times \Gamma_n) \cap \mathsf{HSC}}} \sum_{B \in \mathsf{B}_T} \lambda_B \cdot h_T(B) + \sum_{B \in \mathsf{B}_S} \theta_B \cdot h_S(B) - (\log S) \cdot \|\boldsymbol{\theta}_{\mathsf{B}_S}\|_1, \quad (6.14)$$

parameterized by two vectors, $\boldsymbol{\lambda}_{\mathsf{B}_T} := (\lambda_B)_{B \in \mathsf{B}_T} \in \mathbf{Q}_+^{\mathsf{B}_T}$ with $\|\boldsymbol{\lambda}_{\mathsf{B}_T}\|_1 = 1$ and $\boldsymbol{\theta}_{\mathsf{B}_S} := (\theta_B)_{B \in \mathsf{B}_S} \in \mathbf{Q}_+^{\mathsf{B}_S}$. Equivalently, any

$$(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S}) \in \left\{ (\mathbf{e}_T, \mathbf{e}_S) \mid \mathbf{e}_T \in \mathbf{Q}_+^{\mathsf{B}_T}, \mathbf{e}_S \in \mathbf{Q}_+^{\mathsf{B}_S}, \|\mathbf{e}_T\|_1 = 1 \right\} \quad (6.15)$$

gives rise to an LP of the form (6.14) with an optimal objective value $\mathbf{L}(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S}, S)$.

Lemma 6.2. *Let S be a fixed quantity. For any $(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S})$ as specified in* (6.15)*, the optimal objective value of* (6.12)*, $\mathsf{OBJ}(S)$ satisfies,*

$$\mathsf{OBJ}(S) \leq \mathbf{L}(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S}, S)$$

Moreover, assuming that $\mathsf{OBJ}(S)$ is positive and finite, there is an optimal $(\boldsymbol{\lambda}^*_{\mathsf{B}_T}, \boldsymbol{\theta}^*_{\mathsf{B}_S})$ satisfying (6.15) *such that*

$$\mathsf{OBJ}(S) = \mathbf{L}(\boldsymbol{\lambda}^*_{\mathsf{B}_T}, \boldsymbol{\theta}^*_{\mathsf{B}_S}, S).$$

Instead of proving Lemma 6.2 directly, we prove a slightly more general lemma (Lemma 6.3) that may be of independent interest.

Lemma 6.3. *Let $\mathbf{A} \in \mathbf{Q}^{\ell \times m}, \boldsymbol{D} \in \mathbf{Q}_+^{m \times q}$ be matrices. Let $\boldsymbol{b} \in \mathbf{R}^{\ell}$ be a vector and $\mathbf{C} \in \mathbf{Q}_+^{m \times p}$ be a matrix with columns $\mathbf{c}_1, \ldots, \mathbf{c}_p$ and polyhedron $P = \{\mathbf{x} \in \mathbf{R}^m \mid \mathbf{A}\mathbf{x} \leq \boldsymbol{b}, \mathbf{x} \geq \mathbf{0}\}$. Let w^* be the optimal objective value of the following optimization problem (assume S as a fixed quantity)*

$$\begin{aligned} \max_{\mathbf{x} \in P} \quad & \min_{k \in [p]} \mathbf{c}_k^\top \mathbf{x} \\ s.t. \quad & \boldsymbol{D}^\top \mathbf{x} \geq \mathbf{1}_q \log S. \end{aligned} \tag{6.16}$$

If w^* is positive and finite, then for any vectors $\mathbf{z} \in \mathbf{Q}_+^p, \mathbf{u} \in \mathbf{Q}_+^q$ with $\|\mathbf{z}\|_1 = 1$, the following linear program:

$$L(\mathbf{z}, \mathbf{u}) := \max_{\mathbf{x} \in P} \quad (\mathbf{C}\mathbf{z})^\top \mathbf{x} + (\boldsymbol{D}^\top \mathbf{x} - \mathbf{1}_q \log S)^\top \mathbf{u} \tag{6.17}$$

satisfies $w^* \leq L(\mathbf{z}, \mathbf{u})$. In particular, there is a pair of vectors $\mathbf{z}^* \in \mathbf{Q}_+^p, \mathbf{u}^* \in \mathbf{Q}_+^q$ with $\|\mathbf{z}^*\|_1 = 1$ such that $w^* = L(\mathbf{z}^*, \mathbf{u}^*)$.

Proof. First, we introduce $\mathbf{u} \in \mathbf{Q}_+^q$ as the Lagrange multiplier for (6.16) and obtain

$$\begin{aligned} w^* &\leq \max_{\mathbf{x} \in P} \min_{\mathbf{u} \in \mathbf{Q}_+^q} \quad \min_{k \in [p]} \left(\mathbf{c}_k^\top \mathbf{x} + (\boldsymbol{D}^\top \mathbf{x} - \mathbf{1}_q \log S)^\top \mathbf{u}\right) \\ &\leq \max_{\mathbf{x} \in P} \min_{\mathbf{u} \in \mathbf{Q}_+^q, \mathbf{z} \in \mathbf{Q}_+^p, \|\mathbf{z}\|_1 = 1} \quad (\mathbf{C}\mathbf{z})^\top \mathbf{x} + (\boldsymbol{D}^\top \mathbf{x} - \mathbf{1}_q \log S)^\top \mathbf{u} \\ &\leq \min_{\mathbf{u} \in \mathbf{Q}_+^q, \mathbf{z} \in \mathbf{Q}_+^p, \|\mathbf{z}\|_1 = 1} \max_{\mathbf{x} \in P} \quad (\mathbf{C}\mathbf{z})^\top \mathbf{x} + (\boldsymbol{D}^\top \mathbf{x} - \mathbf{1}_q \log S)^\top \mathbf{u} \\ &= \min_{\mathbf{u} \in \mathbf{Q}_+^q, \mathbf{z} \in \mathbf{Q}_+^p, \|\mathbf{z}\|_1 = 1} L(\mathbf{z}, \mathbf{u}) \end{aligned}$$

where the third equality is because of the minimax inequality. Therefore, we have shown that for any vectors $\mathbf{u} \in \mathbf{Q}_+^q, \mathbf{z} \in \mathbf{Q}_+^p$ with $\|\mathbf{z}\|_1 = 1$, $L(\mathbf{z}, \mathbf{u}) \geq w^*$. Next, we show that there are vectors

$\mathbf{u}^* \in \mathbf{Q}_+^q, \mathbf{z}^* \in \mathbf{Q}_+^p$ with $\|\mathbf{z}^*\|_1 = 1$ such that $L(\mathbf{z}^*, \mathbf{u}^*) = w^*$. We first re-write (6.16) as the following equivalent linear program:

$$\begin{aligned} \max_{\mathbf{x}, w} \quad & w \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \le \boldsymbol{b} \\ & \boldsymbol{D}^\top \mathbf{x} \ge \mathbf{1}_q \log S \\ & \mathbf{C}^\top \mathbf{x} \ge \mathbf{1}_p w \\ & \mathbf{x} \ge \mathbf{0}, w \ge 0 \end{aligned} \tag{6.18}$$

Let $(w^*, \mathbf{x}^*)$ be an optimal solution for (6.18). Then, the dual of (6.18) can be written as the following linear program:

$$\begin{aligned} \min_{\mathbf{y}, \mathbf{u}, \mathbf{z}} \quad & \boldsymbol{b}^\top \mathbf{y} - (\mathbf{1}_q \log S)^\top \mathbf{u} \\ \text{s.t.} \quad & \mathbf{A}^\top \mathbf{y} - \boldsymbol{D}\mathbf{u} - \mathbf{C}\mathbf{z} \ge \mathbf{0} \\ & \mathbf{1}_p^\top \mathbf{z} \ge 1 \\ & \mathbf{y}, \mathbf{u}, \mathbf{z} \ge \mathbf{0} \end{aligned} \tag{6.19}$$

Let $(\mathbf{y}^*, \mathbf{u}^*, \mathbf{z}^*)$ be an extreme point of the (rational) dual polyhedron that attains the optimal objective value for (6.19), so $\mathbf{z}^* \in \mathbf{Q}_+^p, \mathbf{u}^* \in \mathbf{Q}_+^q$. The complementary slackness conditions of the (6.18) and (6.19) primal-dual pair and the assumption $w^* > 0$ imply that $\mathbf{1}_p^\top \mathbf{z}^* = \|\mathbf{z}^*\|_1 = 1$, $(\mathbf{1}_p w^* - \mathbf{C}^\top \mathbf{x}^*)^\top \mathbf{z}^* = 0$ and $(\boldsymbol{D}^\top \mathbf{x}^* - \mathbf{1}_q \log S)^\top \mathbf{u}^* = 0$. We then show that $L(\mathbf{z}^*, \mathbf{u}^*) = w^*$. First, we note that $\mathbf{x}^*$ is feasible for (6.17) with objective value $(\mathbf{C}\mathbf{z}^*)^\top \mathbf{x}^* + (\boldsymbol{D}^\top \mathbf{x}^* - \mathbf{1}_q \log S)^\top \mathbf{u}^* = (\mathbf{C}\mathbf{z}^*)^\top \mathbf{x}^* = (\mathbf{C}^\top \mathbf{x}^*)^\top \mathbf{z}^* = (\mathbf{1}_p w^*)^\top \mathbf{z}^* = w^*$. Hence, $L(\mathbf{z}^*, \mathbf{u}^*) \ge w^*$. Furthermore, for any feasible $\mathbf{x}$ to (6.17), we have that

$$\begin{aligned} (\mathbf{C}\mathbf{z}^*)^\top \mathbf{x} + (\boldsymbol{D}^\top \mathbf{x} - \mathbf{1}_q \log S)^\top \mathbf{u}^* &= (\mathbf{C}\mathbf{z}^*)^\top \mathbf{x} + (\boldsymbol{D}\mathbf{u}^*)^\top \mathbf{x} - (\mathbf{1}_q \log S)^\top \mathbf{u}^* && \textit{re-arrange} \\ &\le (\mathbf{A}^\top \mathbf{y}^*)^\top \mathbf{x} - (\mathbf{1}_q \log S)^\top \mathbf{u}^* && \textit{dual feasibility}, \mathbf{x} \ge \mathbf{0} \\ &= (\mathbf{A}\mathbf{x})^\top \mathbf{y}^* - (\mathbf{1}_q \log S)^\top \mathbf{u}^* \\ &\le \boldsymbol{b}^\top \mathbf{y}^* - (\mathbf{1}_q \log S)^\top \mathbf{u}^* && \textit{primal feasibility}, \mathbf{y}^* \ge \mathbf{0} \\ &= w^* && \textit{strong duality} \end{aligned}$$

This implies that $L(\mathbf{z}^*, \mathbf{u}^*) \le w^*$. Together, we get $L(\mathbf{z}^*, \mathbf{u}^*) = w^*$. □

Now, Lemma6.2 becomes a direct corollary of Lemma 6.3 by setting $\boldsymbol{\lambda}_{\mathsf{B}_T}$ as $\mathbf{z}^*$, $\boldsymbol{\theta}_{\mathsf{B}_S}$ as $\mathbf{u}^*$.

In the rest of the section, we implicitly assume that S is a fixed quantity. Moreover, we fix a pair $(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S})$ satisfying (6.15). We can now re-write (6.14), listing out all its constraints and ignore the

constant factor $(\log S) \cdot \|\boldsymbol{\theta}_{\mathsf{B}_S}\|_1$:

$$
\begin{aligned}
\ell(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S}) := \max_{h_S, h_T} \quad & \sum_{B \in \mathsf{B}_T} \lambda_B \cdot h_T(B) + \sum_{B \in \mathsf{B}_S} \theta_B \cdot h_S(B) \\
\text{s.t.} \quad & h_S(Y) - h_S(X) \le n_{Y|X}, && (X, Y, N_{Y|X}) \in \mathsf{DC} \\
& h_T(Y) - h_T(X) \le n_{Y|X}, && (X, Y, N_{Y|X}) \in \mathsf{DC} \cup \mathsf{AC} \\
& h_S(I \cup J|J) - h_S(I|I \cap J) \le 0, && I \perp J \\
& h_T(I \cup J|J) - h_T(I|I \cap J) \le 0, && I \perp J \\
& h_S(X) - h_S(Y) \le 0, && \emptyset \ne X \subset Y \subseteq [n] \\
& h_T(X) - h_T(Y) \le 0, && \emptyset \ne X \subset Y \subseteq [n] \\
& h_S(X) + h_T(Y|X) \le n_{Z|\emptyset}, && (X, Y|X, N_{Z|\emptyset}) \in \mathsf{SC} \\
& h_S(Y|X) + h_T(X) \le n_{Z|\emptyset}, && (X, Y|X, N_{Z|\emptyset}) \in \mathsf{SC} \\
& h_S(Z) \ge 0, \quad h_T(Z) \ge 0, && \emptyset \ne Z \subseteq [n]
\end{aligned}
\tag{6.20}
$$

Recall that implicitly we have $h_S(\emptyset) = h_T(\emptyset) = 0$ and that $n_{Y|X} = \log N_{Y|X}$. Then we write down the dual LP for (6.20). We associate a dual variable $(\delta_S)_{Y|X}$ to $h_S(Y) - h_S(X) \le n_{Y|X}$ for each $(X, Y, N_{Y|X}) \in \mathsf{DC}$ and a dual variable $(\delta_T)_{Y|X}$ to $h_T(Y) - h_T(X) \le n_{Y|X}$ for each $(X, Y, N_{Y|X}) \in \mathsf{DC} \cup \mathsf{AC}$. For each $I \perp J$, where $I, J \subseteq [n]$, we associate a dual variable $(\sigma_S)_{I,J}$ to the submodularity constraint of h_S and $(\sigma_T)_{I,J}$ to the submodularity constraint of h_T. For each $\emptyset \ne X \subset Y \subseteq [n]$, we associate a dual variable $(\mu_S)_{X,Y}$ to the monotonicity constraint of h_S and a dual variable $(\mu_T)_{X,Y}$ to the monotonicity constraint of h_T. Lastly, for each $(X, Y|X, N_{Z|\emptyset}) \in \mathsf{SC}$, we associate a dual variable $\gamma_{X,Y|X}$ to $h_S(X) + h_T(Y) - h_T(X) \le N_{Z|\emptyset}$ and a dual variable $\gamma_{Y|X,X}$ to $h_S(Y) - h_S(X) + h_T(X) \le N_{Z|\emptyset}$. Moreover, we extend vectors $(\lambda_B)_{B \in \mathsf{B}_T}$ and $(\theta_B)_{B \in \mathsf{B}_S}$ to every $Z \in 2^{[n]}$ in the obvious way:

$$
\lambda_Z := \begin{cases} \lambda_B & \text{when } Z = B \in \mathsf{B}_T \\ 0 & \text{otherwise} \end{cases} \qquad \theta_Z := \begin{cases} \theta_B & \text{when } Z = B \in \mathsf{B}_S \\ 0 & \text{otherwise} \end{cases}
$$

For succinctness, we write $(X, Y) \in \mathsf{DC}$ whenever $(X, Y, N_{Y|X}) \in \mathsf{DC}$ and $(X, Y|X) \in \mathsf{SC}$ whenever $(X, Y, N_{Z|\emptyset}) \in \mathsf{SC}$. By maintaining the best constraint assumption, we can always recover the only

$N_{Y|X}$ or $N_{Z|\emptyset}$ from a given (Y,X)-pair. The dual of (6.20) can now be written as

$$\begin{aligned}
\min \quad & \sum_{(X,Y)\in \mathsf{DC}} n_{Y|X}\cdot(\delta_S)_{Y|X} + \sum_{(X,Y)\in \mathsf{DC}\cup\mathsf{AC}} n_{Y|X}\cdot(\delta_T)_{Y|X} \\
& \quad + \sum_{(X,Y|X)\in\mathsf{SC}} n_{Z|\emptyset}\cdot(\gamma_{X,Y|X}+\gamma_{Y|X,X}) \\
\text{s.t.} \quad & \mathsf{inflow}_S(Z)\geq\theta_Z && \emptyset\neq Z\subseteq[n] \\
& \mathsf{inflow}_T(Z)\geq\lambda_Z && \emptyset\neq Z\subseteq[n] \\
& (\delta_S)_{Y|X},(\mu_S)_{X,Y},(\sigma_S)_{I,J}\geq 0 \\
& (\delta_T)_{Y|X},(\mu_T)_{X,Y},(\sigma_T)_{I,J}\geq 0 \\
& \gamma_{X,Y|X},\gamma_{Y|X,X}\geq 0
\end{aligned} \tag{6.21}$$

where for each $\emptyset\neq Z\subseteq[n]$,

$$\begin{aligned}
\mathsf{inflow}_S(Z) := & \left(\sum_{X:(X,Z)\in\mathsf{DC}}(\delta_S)_{Z|X} - \sum_{Y:(Z,Y)\in\mathsf{DC}}(\delta_S)_{Y|Z}\right) + \left(-\sum_{X:X\subset Z}(\mu_S)_{X,Z}+\sum_{Y:Z\subset Y}(\mu_S)_{Z,Y}\right) \\
& + \left(\sum_{\substack{I\perp J\\ I\cap J=Z}}(\sigma_S)_{I,J}+\sum_{\substack{I\perp J\\ I\cup J=Z}}(\sigma_S)_{I,J}-\sum_{J:J\perp Z}(\sigma_S)_{Z,J}\right) \\
& + \left(\sum_{\substack{Z:(Z,Y|Z)\in\mathsf{SC}\\ Z\subset Y}}\gamma_{Z,Y|Z} - \sum_{\substack{Z:(Z,Y|Z)\in\mathsf{SC}\\ Z\subset Y}}\gamma_{Y|Z,Z} + \sum_{\substack{Z:(X,Z|X)\in\mathsf{SC}\\ X\subset Z}}\gamma_{Z|X,X}\right)
\end{aligned}$$

$$\begin{aligned}
\mathsf{inflow}_T(Z) := & \left(\sum_{X:(X,Z)\in\mathsf{DC}\cup\mathsf{AC}}(\delta_T)_{Z|X} - \sum_{Y:(Z,Y)\in\mathsf{DC}\cup\mathsf{AC}}(\delta_T)_{Y|Z}\right) + \left(-\sum_{X:X\subset Z}(\mu_T)_{X,Z}+\sum_{Y:Z\subset Y}(\mu_T)_{Z,Y}\right) \\
& + \left(\sum_{\substack{I\perp J\\ I\cap J=Z}}(\sigma_T)_{I,J}+\sum_{\substack{I\perp J\\ I\cup J=Z}}(\sigma_T)_{I,J}-\sum_{J:J\perp Z}(\sigma_T)_{Z,J}\right) \\
& + \left(\sum_{\substack{Z:(Z,Y|Z)\in\mathsf{SC}\\ Z\subset Y}}\gamma_{Y|Z,Z} - \sum_{\substack{Z:(Z,Y|Z)\in\mathsf{SC}\\ Z\subset Y}}\gamma_{Z,Y|Z} + \sum_{\substack{Z:(X,Z|X)\in\mathsf{SC}\\ X\subset Z}}\gamma_{X,Z|X}\right)
\end{aligned}$$

Next, we introduce the *joint Shannon-flow inequalities.*

Definition 6.7 (Joint Shannon-flow Inequality)**.** The inequality

$$\begin{aligned}
& \sum_{(X,Y)\in\mathsf{DC}} h_S(Y|X)\cdot(\delta_S)_{Y|X} + \sum_{(X,Y)\in\mathsf{DC}\cup\mathsf{AC}} h_T(Y|X)\cdot(\delta_T)_{Y|X} + \sum_{(X,Y|X)\in\mathsf{SC}}(h_S(X)+h_T(Y|X))\cdot\gamma_{X,Y|X} \\
& \quad + \sum_{(X,Y|X)\in\mathsf{SC}}(h_S(Y|X)+h_T(X))\cdot\gamma_{Y|X,X} \geq \sum_{B\in\mathsf{B}_S}\theta_B\cdot h_S(B) + \sum_{B\in\mathsf{B}_T}\lambda_B\cdot h_T(B),
\end{aligned} \tag{6.22}$$

is called a *joint Shannon-flow inequality* if it holds for all $(h_S, h_T) \in \Gamma_n \times \Gamma_n$ and all coefficients are non-negative rational numbers.

By assigning either polymatroid to be always 0, the above inequality implies the following two Shannon-flow inequalities, called the *participating Shannon-flow inequalities*.

$$\sum_{(X,Y)\in\mathsf{DC}} h_S(Y|X)\cdot(\delta_S)_{Y|X} + \sum_{(X,Y|X)\in\mathsf{SC}} h_S(X)\cdot\gamma_{X,Y|X} + \sum_{(X,Y|X)\in\mathsf{SC}} h_S(Y|X)\cdot\gamma_{Y|X,X} \geq \sum_{B\in\mathsf{B}_S}\theta_B\cdot h_S(B) \quad (6.23)$$

$$\sum_{(X,Y)\in\mathsf{DC}\cup\mathsf{AC}} h_T(Y|X)\cdot(\delta_T)_{Y|X} + \sum_{(X,Y|X)\in\mathsf{SC}} h_T(Y|X)\cdot\gamma_{X,Y|X} + \sum_{(X,Y|X)\in\mathsf{SC}} h_T(X)\cdot\gamma_{Y|X,X} \geq \sum_{B\in\mathsf{B}_T}\lambda_B\cdot h_T(B) \quad (6.24)$$

It will be convenient to write a joint Shannon-flow inequality as inequalities over conditional polymatroid. The polymatroid h_S defines a conditional polymatroid $\boldsymbol{h}_S$ and the polymatroid h_T defines a conditional polymatroid $\boldsymbol{h}_T$. More precisely, we define the vectors $\boldsymbol{\lambda}, \boldsymbol{\theta} \in \mathbf{Q}_+^{\mathcal{C}}$ (extend to (X, Y) pairs where $\emptyset \subseteq X \subset Y \subseteq [n]$) with coordinate values assigned as the following:

$$\lambda(Y|X) := \begin{cases} \lambda_B & \text{when } Y = B, X = \emptyset \\ 0 & \text{otherwise} \end{cases} \qquad \theta(Y|X) := \begin{cases} \theta_B & \text{when } Y = B, X = \emptyset \\ 0 & \text{otherwise} \end{cases}$$

and similarly, we define the vectors $\boldsymbol{\delta}_S, \boldsymbol{\delta}_T \in \mathbf{Q}_+^{\mathcal{C}}$ with coordinate values:

$$\delta_S(Y|X) := \begin{cases} (\delta_S)_{Y|X} & \text{when } (X,Y) \in \mathsf{DC} \\ 0 & \text{otherwise} \end{cases} \qquad \delta_T(Y|X) := \begin{cases} (\delta_T)_{Y|X} & \text{when } (X,Y) \in \mathsf{DC}\cup\mathsf{AC} \\ 0 & \text{otherwise} \end{cases}$$

Lastly, the coefficients $\left\{\gamma_{X,Y|X} \mid (X,Y|X) \in \mathsf{SC}\right\} \cup \left\{\gamma_{Y|X,X} \mid (X,Y|X) \in \mathsf{SC}\right\}$ contributes to the coefficients of both inequalities (6.23) and (6.24). We define a pair of vectors $\boldsymbol{\gamma}_S, \boldsymbol{\gamma}_T \in \mathbf{Q}_+^{\mathcal{C}}$ as the following:

$$\gamma_S(Y|X) := \begin{cases} \gamma_{U,V|U} & \text{when } Y = U, X = \emptyset, (U,V|U) \in \mathsf{SC} \\ \gamma_{V|U,U} & \text{when } Y = V, X = U, (U,V|U) \in \mathsf{SC} \\ 0 & \text{otherwise} \end{cases}$$

$$\gamma_T(Y|X) := \begin{cases} \gamma_{U,V|U} & \text{when } Y = V, X = U, (U,V|U) \in \mathsf{SC} \\ \gamma_{V|U,U} & \text{when } Y = U, X = \emptyset, (U,V|U) \in \mathsf{SC} \\ 0 & \text{otherwise} \end{cases}$$

for tracking the contributions to $\boldsymbol{h}_S, \boldsymbol{h}_T$, respectively. Let $\boldsymbol{g}_S := \boldsymbol{\delta}_S + \boldsymbol{\gamma}_S$ and $\boldsymbol{g}_T := \boldsymbol{\delta}_T + \boldsymbol{\gamma}_T$, then the two inequalities (6.23) and (6.24) can be re-written using dot-products:

$$\langle \boldsymbol{g}_S, \boldsymbol{h}_S \rangle := \langle \boldsymbol{\delta}_S, \boldsymbol{h}_S \rangle + \langle \boldsymbol{\gamma}_S, \boldsymbol{h}_S \rangle \geq \langle \boldsymbol{\theta}, \boldsymbol{h}_S \rangle \tag{6.25}$$

$$\langle \boldsymbol{g}_T, \boldsymbol{h}_T \rangle := \langle \boldsymbol{\delta}_T, \boldsymbol{h}_T \rangle + \langle \boldsymbol{\gamma}_T, \boldsymbol{h}_T \rangle \geq \langle \boldsymbol{\lambda}, \boldsymbol{h}_T \rangle \tag{6.26}$$

Moreover, the joint Shannon-flow inequality (6.22) can be written as

$$\langle \boldsymbol{g}_S, \boldsymbol{h}_S \rangle + \langle \boldsymbol{g}_T, \boldsymbol{h}_T \rangle \geq \langle \boldsymbol{\theta}, \boldsymbol{h}_S \rangle + \langle \boldsymbol{\lambda}, \boldsymbol{h}_T \rangle \tag{6.27}$$

Theorem 6.4. *The inequality* (6.27) *is a joint Shannon-flow inequality if and only if there are non-negative vectors of rationals $\boldsymbol{\sigma}_S, \boldsymbol{\mu}_S, \boldsymbol{\sigma}_T, \boldsymbol{\mu}_T$ such that all constraints of the dual* (6.21) *are satisfied. In particular, we call $(\boldsymbol{\sigma}_S, \boldsymbol{\mu}_S, \boldsymbol{\sigma}_T, \boldsymbol{\mu}_T)$ a witness for the joint Shannon-flow inequality.*

Proof. Proposition 5.6 in [KNS17] states that: (6.25) is a Shannon-flow inequality if and only if there is a $(\boldsymbol{\sigma}_S, \boldsymbol{\mu}_S) \geq \mathbf{0}$ (called a witness in [KNS17]) such that $(\boldsymbol{g}_S, \boldsymbol{\sigma}_S, \boldsymbol{\mu}_S)$ satisfies the set of constraints:

$$\{\mathsf{inflow}_S(Z) \geq \theta_Z \mid \emptyset \neq Z \subseteq [n]\}\,;$$

similarly, (6.26) is a Shannon-flow inequality if and only if there is a (witness) $(\boldsymbol{\sigma}_T, \boldsymbol{\mu}_T) \geq \mathbf{0}$ such that $(\boldsymbol{g}_T, \boldsymbol{\sigma}_T, \boldsymbol{\mu}_T)$ satisfies the set of constraints:

$$\{\mathsf{inflow}_T(Z) \geq \lambda_Z \mid \emptyset \neq Z \subseteq [n]\}\,.$$

Recall the formulation of (6.21), these two sets of constraints form exactly all the constraints in (6.21). Thus, (6.27) is a joint Shannon-flow inequality if and only if there is a $(\boldsymbol{\sigma}_S, \boldsymbol{\mu}_S, \boldsymbol{\sigma}_T, \boldsymbol{\mu}_T) \geq \mathbf{0}$ such that $(\boldsymbol{g}_S, \boldsymbol{\sigma}_S, \boldsymbol{\mu}_S, \boldsymbol{g}_T, \boldsymbol{\sigma}_T, \boldsymbol{\mu}_T)$ satisfies all the constraints of the dual (6.21). ☐

Theorem 6.4 implies that a feasible solution of the dual (6.21), and in particular the component

$$\left\{(\delta_S)_{Y|X} \mid (X,Y) \in \mathsf{DC}\right\} \cup \left\{(\delta_T)_{Y|X} \mid (X,Y) \in \mathsf{DC}\right\} \cup \left\{\gamma_{X,Y|X} \mid (X,Y|X) \in \mathsf{SC}\right\}$$
$$\cup \left\{\gamma_{Y|X,X} \mid (X,Y|X) \in \mathsf{SC}\right\}$$

in conjunction with $(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S})$, defines a joint Shannon-flow inequality (6.22). The component $(\boldsymbol{\sigma}_S, \boldsymbol{\mu}_S, \boldsymbol{\sigma}_T, \boldsymbol{\mu}_T)$ of the dual, by Theorem 6.4, is a witness for the joint Shannon-flow inequality. Note that the joint Shannon-flow inequality implies that

$$\langle \boldsymbol{\theta}, \boldsymbol{h}_S \rangle + \langle \boldsymbol{\lambda}, \boldsymbol{h}_T \rangle \leq \sum_{(X,Y) \in \mathsf{DC}} n_{Y|X} \cdot (\delta_S)_{Y|X} + \sum_{(X,Y) \in \mathsf{DC} \cup \mathsf{AC}} n_{Y|X} \cdot (\delta_T)_{Y|X}$$

$$+ \sum_{(X,Y|X) \in \mathsf{SC}} n_{Z|\emptyset} \cdot (\gamma_{X,Y|X} + \gamma_{Y|X,X})$$

where the right-hand side is exactly $\ell(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S})$ by taking the optimal solution of the dual (6.21) (by strong duality). We have established that: for an arbitrary $(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S})$ satisfying (6.15), we can construct a joint Shannon-flow inequality,

$$\langle \boldsymbol{g}_S, \boldsymbol{h}_S \rangle + \langle \boldsymbol{g}_T, \boldsymbol{h}_T \rangle \geq \langle \boldsymbol{\theta}, \boldsymbol{h}_S \rangle + \langle \boldsymbol{\lambda}, \boldsymbol{h}_T \rangle,$$

having a witness $(\boldsymbol{\sigma}_S, \boldsymbol{\mu}_S, \boldsymbol{\sigma}_T, \boldsymbol{\mu}_T)$, such that its implied upper bound coincides with $\ell(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S})$.

6.6.2 An Augmentation of the PANDA Algorithm

This section provides a revisit of the PANDA algorithm as introduced by [KNS17]. Let ρ be a disjunctive rule (2.7) under degree constraints DC. At a high level, PANDA runs the following steps (i) to (iv):

(i) find a vector of non-negative rationals $\boldsymbol{\lambda}_{\mathsf{B}_T} = (\lambda_B)_{B \in \mathsf{B}_T}$ with $\|\boldsymbol{\lambda}_{\mathsf{B}_T}\|_1 = 1$, extend it to a vector (over conditional polymatroid) $\boldsymbol{\lambda} \in \mathbb{Q}_+^{\mathcal{C}}$ and $\mathsf{LogSizeBound}_{\Gamma_n \cap \mathsf{HDC}}(\rho) = \max_{h \in \Gamma_n \cap \mathsf{HDC}} \langle \boldsymbol{\lambda}, \boldsymbol{h} \rangle$, where the right-hand side is a linear program;

(ii) find an optimal dual solution $(\boldsymbol{\delta}, \boldsymbol{\sigma}, \boldsymbol{\mu})$ to the linear program, $\max_{h \in \Gamma_n \cap \mathsf{HDC}} \langle \boldsymbol{\lambda}, \boldsymbol{h} \rangle$, so that

$$\mathsf{OBJ} := \sum_{(X,Y) \in \mathsf{DC}} \log N_{Y|X} \cdot \delta_{Y|X} = \mathsf{LogSizeBound}_{\Gamma_n \cap \mathsf{HDC}}(\rho)$$

and $\langle \boldsymbol{\delta}, \boldsymbol{h} \rangle \geq \langle \boldsymbol{\lambda}, \boldsymbol{h} \rangle$ forms a Shannon-flow inequality;

(iii) construct a proof sequence ProofSeq of length $O(poly(2^n))$ for the Shannon-flow inequality $\langle \boldsymbol{\delta}, \boldsymbol{h} \rangle \geq \langle \boldsymbol{\lambda}, \boldsymbol{h} \rangle$ (see Theorem 2.5);

(iv) run a PANDA instance, denoted as PANDA$(\mathcal{D}, \mathsf{DC}, (\boldsymbol{\lambda}, \boldsymbol{\delta}), \mathsf{ProofSeq})$, which interprets each proof step of ProofSeq as a relational operation on the input relation, where each operation is guaranteed to take time $\widetilde{O}(2^{\mathsf{OBJ}})$. Overall, the PANDA instance runs in time $\widetilde{O}(2^{\mathsf{OBJ}})$ and computes a model of size $\widetilde{O}(2^{\mathsf{OBJ}})$.

In particular, every PANDA instance maintains the following 4 invariants throughout on its input $(\mathcal{D}, \mathsf{DC}, (\boldsymbol{\lambda}, \boldsymbol{\delta}), \mathsf{ProofSeq})$:

PANDA invariants

(1) *Degree-support invariant:* For every $\delta_{Y|X} > 0$, there exist $Z \subseteq X$ and $W \subseteq Y$ such that $W - Z = Y - X$ and $(Z, W, N_{W|Z}) \in \mathsf{DC}$. The degree constraint is said to support the positive $\delta_{Y|X}$. If there are more than one $(Z, W, N_{W|Z}) \in \mathsf{DC}$ supporting $\delta_{Y|X}$, we choose the one with minimum $N_{W|Z}$ and call it the supporting constraint of $\delta_{Y|X}$.

(2) $0 < \|\boldsymbol{\lambda}\|_1 \leq 1$.

(3) The Shannon-flow inequality, together with the supporting degree constraints satisfy the inequality $\sum_{(X,Y)} n(\delta_{Y|X}) \leq \|\boldsymbol{\lambda}\|_1 \cdot \mathsf{OBJ}$, where

$$n(\delta_{Y|X}) := \begin{cases} \delta_{Y|X} \cdot n_{W|Z} & \text{if } \delta_{Y|X} > 0 \text{ and } (Z, W, N_{W|Z}) \text{ supports it} \\ 0 & \text{if } \delta_{Y|X} = 0. \end{cases}$$

and we call the quantity $\sum_{(X,Y)} n(\delta_{Y|X})$ the *potential*.

(4) For every $\delta_{Y|\emptyset} > 0$, the supporting degree constraint $(\emptyset, Y, N_{Y|\emptyset})$ satisfies $n_{Y|\emptyset} \leq \mathsf{OBJ}$.

A slight augmentation on PANDA For our purposes, we augment PANDA to take non-optimal proof sequences. More precisely, instead of (i) and (ii), we are given vectors $(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\delta}_{\mathsf{DC}})$ and a witness $(\boldsymbol{\sigma}, \boldsymbol{\mu})$ such that by extending both vectors as $\boldsymbol{\lambda}, \boldsymbol{\delta} \in \mathbf{Q}_+^{\mathcal{C}}$, $\langle \boldsymbol{\delta}, \boldsymbol{h} \rangle \geq \langle \boldsymbol{\lambda}, \boldsymbol{h} \rangle$ forms a (not necessarily optimal) Shannon-flow inequality with witness $(\boldsymbol{\sigma}, \boldsymbol{\mu})$. The implied upper bound OBJ satisfies

$$\mathsf{LogSizeBound}_{\Gamma_n \cap \mathsf{HDC}}(\rho) \leq \mathsf{OBJ} := \sum_{(X,Y) \in \mathsf{DC}} \log N_{Y|X} \cdot \delta_{Y|X},$$

but not necessarily coincides with $\mathsf{LogSizeBound}_{\Gamma_n \cap \mathsf{HDC}}(\rho)$. In such cases, following (iii) and (iv), we show that the PANDA instance (taking the non-optimal proof sequence for the given Shannon-flow inequality as input), runs in time as predicated by the non-optimal Shannon-flow inequality, i.e. $\widetilde{O}(2^{\mathsf{OBJ}})$.

The augmentation is as follows. We observe that in the proof of correctness of PANDA [KNS17], only invariant (4) (at the entry point of a PANDA call) relies on the optimality of proof sequences. However, as argued in Proposition 6.2 in [KNS17], if initially for some $\delta_{Y|\emptyset} > 0$, $n_{Y|\emptyset} > \mathsf{OBJ}$, then we could replace the original Shannon-flow inequality with a new Shannon-flow inequality $\langle \boldsymbol{\delta}', \boldsymbol{h} \rangle \geq \langle \boldsymbol{\lambda}', \boldsymbol{h} \rangle$ along with witness $(\boldsymbol{\sigma}', \boldsymbol{\mu}')$ such that invariants (1)-(3) hold, and the length of the proof sequence decreases by at least 1. We can repeat this replacement step iteratively until invariant (4) is satisfied. If invariant (4) is never satisfied, then we will end up with a proof sequence of length 0, in which case the Shannon-flow inequality becomes a trivial one, $\langle \boldsymbol{\delta}_0, \boldsymbol{h} \rangle \geq \langle \boldsymbol{\lambda}_0, \boldsymbol{h} \rangle$, for some $\boldsymbol{\lambda}_0, \boldsymbol{\delta}_0$ with $0 < \|\boldsymbol{\lambda}_0\|_1 \leq 1$ (by invariant (2)) and $\boldsymbol{\delta}_0 \geq \boldsymbol{\lambda}_0$ (element-wise comparison). This implies that

any input relation R_F where $(\lambda_0)_{F|\emptyset} > 0$ can be a model and there is a $n_{F|\emptyset} \leq \mathsf{OBJ}$ that can be appointed as *the* output model (hence the model size is $\widetilde{O}(2^{\mathsf{OBJ}})$), because otherwise,

$$\sum_{F:(\lambda_0)_{F|\emptyset}>0} \frac{(\lambda_0)_{F|\emptyset}}{\|\boldsymbol{\lambda}_0\|_1} \cdot n_{F|\emptyset} > \sum_{F:(\lambda_0)_{F|\emptyset}>0} \frac{(\lambda_0)_{F|\emptyset}}{\|\boldsymbol{\lambda}_0\|_1} \cdot \mathsf{OBJ} = \mathsf{OBJ},$$

and this directly contradicts invariant (3). For the 2PP algorithm, we implicitly assume that PANDA is equipped with this minor augmentation.

6.6.3 The Algorithm

We formally present the 2PP algorithm in this section. 2PP follows the 2-phase algorithmic paradigm introduced in Section 6.5.2. Further, it is guided by a joint Shannon-flow inequality to execute only the necessary split steps and PANDA instances, which provides more practicality and interpretability. In particular, we will show the following theorem for 2PP.

Theorem 6.5. *Let ρ be a 2-phase disjunctive rule of the form* (6.5) *satisfying degree constraints* DC *(guarded by the input relations) and* AC *(guarded by the access request). Let $(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S}) \in \left\{(\mathbf{e}_T, \mathbf{e}_S) \mid \mathbf{e}_T \in \mathbf{Q}_+^{\mathsf{B}_T}, \mathbf{e}_S \in \mathbf{Q}_+^{\mathsf{B}_S}, \|\mathbf{e}_T\|_1 = 1\right\}$. The 2PP algorithm obtains a model of ρ in two phases and attains the following (smooth) intrinsic trade-off:*

$$S_\rho^{\|\boldsymbol{\theta}_{\mathsf{B}_S}\|_1} \cdot T_\rho \cong 2^{\ell(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S})} \tag{6.28}$$

where $\cong$ hides a poly-logarithmic factor at the right-hand side.

Note that the intrinsic trade-off can be equivalently written as

$$S_\rho^{\|\boldsymbol{\theta}_{\mathsf{B}_S}\|_1} \cdot T_\rho \cong \prod_{(X,Y)\in\mathsf{DC}} N_{Y|X}^{(\delta_S)_{Y|X}} \cdot \prod_{(X,Y)\in\mathsf{DC}\cup\mathsf{AC}} N_{Y|X}^{(\delta_S)_{Y|X}} \cdot \prod_{(X,Y|X)\in\mathsf{SC}} N_{Z|\emptyset}^{\gamma_{X,Y|X}+\gamma_{Y|X,X}} \tag{6.29}$$

In particular, given a fixed S for S_ρ, we can construct from Lemma 6.2 (using complementary slackness) $(\boldsymbol{\lambda}^*_{\mathsf{B}_T}, \boldsymbol{\theta}^*_{\mathsf{B}_S})$ such that $\mathsf{OBJ}(S) = \mathbf{L}(\boldsymbol{\lambda}^*_{\mathsf{B}_T}, \boldsymbol{\theta}^*_{\mathsf{B}_S}, S)$. Since $\ell(\boldsymbol{\lambda}^*_{\mathsf{B}_T}, \boldsymbol{\theta}^*_{\mathsf{B}_S}) - (\log S) \cdot \|\boldsymbol{\theta}^*_{\mathsf{B}_S}\|_1 = \mathbf{L}(\boldsymbol{\lambda}^*_{\mathsf{B}_T}, \boldsymbol{\theta}^*_{\mathsf{B}_S}, S)$, Theorem 6.5 recovers the best possible instrinsic trade-off as specified in (6.12). In the rest of this section, we present the 2PP algorithm, and defer the full proof of Theorem 6.5 to the next section.

Preparation phase Similar to PANDA, 2PP has a preparation phase to construct the necessary inputs for a 2PP instance. First, we construct from the given $(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S})$ a joint Shannon-flow inequality, $\langle \boldsymbol{g}_S, \boldsymbol{h}_S \rangle + \langle \boldsymbol{g}_T, \boldsymbol{h}_T \rangle \geq \langle \boldsymbol{\theta}, \boldsymbol{h}_S \rangle + \langle \boldsymbol{\lambda}, \boldsymbol{h}_T \rangle$ and a witness for it, $(\boldsymbol{\sigma}_S, \boldsymbol{\mu}_S, \boldsymbol{\sigma}_T, \boldsymbol{\mu}_T)$. Recall that the joint Shannon-flow inequality implies an upper bound that coincides with $\ell(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S})$.

Second, we construct a proof sequence for the joint Shannon-flow inequality. The idea is to build two parallel proof sequences for its two participating Shannon-flow inequalities and stitch them together. Recall that the two participating Shannon-flow inequalities are

$$\langle \boldsymbol{g}_S, \boldsymbol{h}_S \rangle \geq \langle \boldsymbol{\theta}, \boldsymbol{h}_S \rangle$$
$$\langle \boldsymbol{g}_T, \boldsymbol{h}_T \rangle \geq \langle \boldsymbol{\lambda}, \boldsymbol{h}_T \rangle$$

From the proof of Theorem 6.4, $(\boldsymbol{\sigma}_S, \boldsymbol{\mu}_S)$ is a witness for $\langle \boldsymbol{g}_S, \boldsymbol{h}_S \rangle \geq \langle \boldsymbol{\theta}, \boldsymbol{h}_S \rangle$ and $(\boldsymbol{\sigma}_T, \boldsymbol{\mu}_T)$ is a witness for $\langle \boldsymbol{g}_T, \boldsymbol{h}_T \rangle \geq \langle \boldsymbol{\lambda}, \boldsymbol{h}_T \rangle$. We normalize the Shannon-flow inequality $\langle \boldsymbol{g}_S, \boldsymbol{h}_S \rangle \geq \langle \boldsymbol{\theta}, \boldsymbol{h}_S \rangle$ into $\langle \widetilde{\boldsymbol{g}}_S, \boldsymbol{h}_S \rangle \geq \langle \widetilde{\boldsymbol{\theta}}, \boldsymbol{h}_S \rangle$, where $\widetilde{\boldsymbol{g}}_S := \boldsymbol{g}_S / \|\boldsymbol{\theta}\|_1$ and $\widetilde{\boldsymbol{\theta}} := \boldsymbol{\theta}/\|\boldsymbol{\theta}\|_1$, with a new witness $(\widetilde{\boldsymbol{\sigma}}_S, \widetilde{\boldsymbol{\mu}}_S) := (\boldsymbol{\sigma}_S/\|\boldsymbol{\theta}\|_1, \boldsymbol{\mu}_S/\|\boldsymbol{\theta}\|_1)$. From Theorem 2.5, we can construct a proof sequence $\mathsf{ProofSeq}(S)$ for the Shannon-flow inequality $\langle \widetilde{\boldsymbol{g}}_S, \boldsymbol{h}_S \rangle \geq \langle \widetilde{\boldsymbol{\theta}}, \boldsymbol{h}_S \rangle$ and a proof sequence $\mathsf{ProofSeq}(T)$ for $\langle \boldsymbol{g}_T, \boldsymbol{h}_T \rangle \geq \langle \boldsymbol{\lambda}, \boldsymbol{h}_T \rangle$, where both proof sequences have length $O(poly(2^n))$, n being the number of variables. We say that $\mathsf{ProofSeq}(S)$ and $\mathsf{ProofSeq}(T)$ are *the participating proof sequences* for the joint Shannon-flow inequality. As a brief summary, in the preparation phase, 2PP:

(1) (see Theorem 6.4) constructs a joint Shannon-flow inequality $\langle \boldsymbol{g}_S, \boldsymbol{h}_S \rangle + \langle \boldsymbol{g}_T, \boldsymbol{h}_T \rangle \geq \langle \boldsymbol{\theta}, \boldsymbol{h}_S \rangle + \langle \boldsymbol{\lambda}, \boldsymbol{h}_T \rangle$ with a witness $(\boldsymbol{\sigma}_S, \boldsymbol{\mu}_S, \boldsymbol{\sigma}_T, \boldsymbol{\mu}_T)$; and

(2) (see Theorem 2.5) constructs a $\mathsf{ProofSeq}(S)$ for the (normalized) participating Shannon-flow inequality $\langle \widetilde{\boldsymbol{g}}_S, \boldsymbol{h}_S \rangle \geq \langle \widetilde{\boldsymbol{\theta}}, \boldsymbol{h}_S \rangle$ and a $\mathsf{ProofSeq}(T)$ for the participating Shannon-flow inequality $\langle \boldsymbol{g}_T, \boldsymbol{h}_T \rangle \geq \langle \boldsymbol{\lambda}, \boldsymbol{h}_T \rangle$.

Now, we walk through both phases of 2PP. In particular, we denote 2PP-PREPROC as the preprocessing phase of 2PP and 2PP-ONLINE as the online phase of 2PP. The summaries of 2PP-PREPROC and 2PP-ONLINE are in the box of Algorithm 13 and Algorithm 14.

Algorithm 13: 2PP-PREPROC($\mathcal{D}$, DC, AC, $(\widetilde{\boldsymbol{\theta}}, \widetilde{\boldsymbol{g}}_S)$, $\mathsf{ProofSeq}(S)$)

Input : a database instance $\mathcal{D}$ and degree constraints DC guarded by $\mathcal{D}$

Input : the degree constraints AC guarded by any possible access request Q_A

Input : the participating Shannon-flow inequality $\langle \widetilde{\boldsymbol{g}}_S, \boldsymbol{h}_S \rangle \geq \langle \widetilde{\boldsymbol{\theta}}, \boldsymbol{h}_S \rangle$ and its proof sequence $\mathsf{ProofSeq}(S)$

1 Let SC be the set of split constraints spanned from DC

2 $\mathsf{SC}^+ \leftarrow \left\{(Y, X) \mid (X, Y|X, N_{Z|\emptyset}) \in \mathsf{SC}, \gamma_{X,Y|X} > 0 \vee \gamma_{Y|X,X} > 0\right\}$

3 Apply a sequence of split steps, one for every $(Y, X) \in \mathsf{SC}^+$ and spawn k subproblems with inputs $(\mathcal{D}^{(j)}, \mathsf{DC}^j)$, $j \in [k]$

// $k = O(poly(\log |\mathcal{D}|))$

4 $\mathcal{J} \leftarrow \emptyset$

5 **forall** $j \in [k]$ **do**

6 **create** a PANDA instance $\mathsf{PANDA}(\mathcal{D}^{(j)}, \mathsf{DC}^{(j)}, (\widetilde{\boldsymbol{\theta}}, \widetilde{\boldsymbol{g}}_S), \mathsf{ProofSeq}(S))$

7 **if** *the potential satisfies* (6.31) **then**

8 $(S_B^{(j)})_{B \in \mathsf{B}_S} \leftarrow \mathsf{PANDA}(\mathcal{D}^{(j)}, \mathsf{DC}^{(j)}, (\widetilde{\boldsymbol{\theta}}, \widetilde{\boldsymbol{g}}_S), \mathsf{ProofSeq}(S))$

9 **else**

10 **abort** the instance $\mathsf{PANDA}(\mathcal{D}^{(j)}, \mathsf{DC}^{(j)}, (\widetilde{\boldsymbol{\theta}}, \widetilde{\boldsymbol{g}}_S), \mathsf{ProofSeq}(S))$

11 **insert** $(\mathcal{D}^{(j)}, \mathsf{DC}^{(j)})$ to $\mathcal{J}$

12 **return** $\left(\mathcal{J}, (\bigcup_j S_B^{(j)})_{B \in \mathsf{B}_S}\right)$

Algorithm 14: 2PP-ONLINE($\mathcal{J}$, $\{Q_A\}$, AC, $(\boldsymbol{\lambda}, \boldsymbol{g}_T)$, $\mathsf{ProofSeq}(T)$)

Input : an index $\mathcal{J}$ containing $O(poly(\log |\mathcal{D}|))$ entries where each entry contains input relations $\mathcal{D}^{(j)}$ and degree constraints $\mathsf{DC}^{(j)}$ guarded by $\mathcal{D}^{(j)}$

Input : an access request Q_A and the degree constraints AC guarded by Q_A

Input : the participating Shannon-flow inequality $\langle \boldsymbol{g}_T, \boldsymbol{h}_T \rangle \geq \langle \boldsymbol{\lambda}, \boldsymbol{h}_T \rangle$ and its proof sequence $\mathsf{ProofSeq}(T)$

1 **forall** $(\mathcal{D}^{(j)}, \mathsf{DC}^{(j)}) \in \mathcal{J}$ **do**

2 **create** a PANDA instance $\mathsf{PANDA}(\mathcal{D}^{(j)} \cup \{Q_A\}, \mathsf{DC}^{(j)} \cup \mathsf{AC}, (\boldsymbol{\lambda}, \boldsymbol{g}_T), \mathsf{ProofSeq}(T))$

3 $(T_B^{(j)})_{B \in \mathsf{B}_T} \leftarrow \mathsf{PANDA}(\mathcal{D}^{(j)} \cup \{Q_A\}, \mathsf{DC}^{(j)} \cup \mathsf{AC}, (\boldsymbol{\lambda}, \boldsymbol{g}_T), \mathsf{ProofSeq}(T))$

4 **return** $(\bigcup_j T_B^{(j)})_{B \in \mathsf{B}_T}$

The preprocessing phase We call this phase 2PP-PREPROC and it is sketched in the box of Algorithm 13. We first scan over the joint Shannon-flow inequality, $\langle \boldsymbol{g}_S, \boldsymbol{h}_S \rangle + \langle \boldsymbol{g}_T, \boldsymbol{h}_T \rangle \geq \langle \boldsymbol{\theta}, \boldsymbol{h}_S \rangle + \langle \boldsymbol{\lambda}, \boldsymbol{h}_T \rangle$ and $(\boldsymbol{\sigma}_S, \boldsymbol{\mu}_S, \boldsymbol{\sigma}_T, \boldsymbol{\mu}_T)$ and apply a sequence of split steps that consists of one (Y, X)-pair for every (Y, X) satisfying $(X, Y|X, N_{Z|\emptyset}) \in \mathsf{SC}$ and either $\gamma_{X,Y|X} > 0$ or $\gamma_{Y|X,X} > 0$. The sequence of

split steps spawns $poly(\log|\mathcal{D}|)$ subproblems. Let $\rho(\mathcal{D}^{(j)} \cup \{Q_A\}, \mathsf{DC}^{(j)} \cup \mathsf{AC})$ be the j-th subproblem after the sequence of split steps, having degree constraints $\mathsf{DC}^{(j)}$ guarded by $\mathcal{D}^{(j)}$. Next, we create for it a PANDA instance $\mathsf{PANDA}(\mathcal{D}^{(j)}, \mathsf{DC}^{(j)}, (\widetilde{\boldsymbol{\theta}}, \widetilde{\boldsymbol{g}}_S), \mathsf{ProofSeq}(S))$ and look at its initial *potential*

$$\sum_{(X,Y)} n^{(j)}_{W|Z} \cdot (\widetilde{g}_S)_{Y|X} := \sum_{(X,Y)\in\mathsf{DC}} n^{(j)}_{W|Z} \cdot \frac{(\delta_S)_{Y|X}}{\|\boldsymbol{\theta}\|_1} + \sum_{(X,Y|X)\in\mathsf{SC}} n^{(j)}_{X|\emptyset} \cdot \frac{\gamma_{X,Y|X}}{\|\boldsymbol{\theta}\|_1} + \sum_{(X,Y|X)\in\mathsf{SC}} n^{(j)}_{W|Z} \cdot \frac{\gamma_{Y|X,X}}{\|\boldsymbol{\theta}\|_1} \tag{6.30}$$

where $n^{(j)}_{W|Z} := \log N^{(j)}_{W|Z}$ and $(Z, W, N^{(j)}_{W|Z}) \in \mathsf{DC}^{(j)}$ is the constraint that supports a positive $(\widetilde{g}_S)_{Y|X}$. If the potential satisfies

$$\sum_{(X,Y)} n^{(j)}_{W|Z} \cdot (\widetilde{g}_S)_{Y|X} \leq \log S, \tag{6.31}$$

then 2PP-PREPROC allows this PANDA instance to run and stores its output $(S^{(j)}_B)_{B\in\mathsf{B}_S}$. Otherwise, 2PP aborts this PANDA instance and keeps track of the input $(\mathcal{D}^{(j)}, \mathsf{DC}^{(j)})$ for the j-th subproblem using an index $\mathcal{J}$. The data structure(s) stored in the preprocessing phase are the index $\mathcal{J}$ that tracks inputs (and its degree constraints) from aborted instances, and a set of tables $S_B = \bigcup_j S^{(j)}_B, B \in \mathsf{B}_S$ from all succeeded PANDA instances.

The online phase We call this phase 2PP-ONLINE and it is sketched in the box of Algorithm 14. The algorithm scans over the index $\mathcal{J}$ built in the preprocessing phase, and for each $(\mathcal{D}^{(j)}, \mathsf{DC}^{(j)}) \in \mathcal{J}$, it creates and runs a PANDA instance $\mathsf{PANDA}(\mathcal{D}^{(j)} \cup \{Q_A\}, \mathsf{DC}^{(j)} \cup \mathsf{AC}, (\boldsymbol{\lambda}, \boldsymbol{g}_T), \mathsf{ProofSeq}(T))$. At the termination of each PANDA instance, 2PP-ONLINE collects outputs $(T^{(j)}_B)_{B\in\mathsf{B}_T}$. The overall output in the online phase is the set of tables $T_B = \bigcup_j T^{(j)}_B, B \in \mathsf{B}_T$ from all PANDA instances created and executed online.

6.6.4 Analysis of the 2-phase PANDA Algorithm

In the rest of the section, we formally prove Theorem 6.5 for the 2PP algorithm.

Proof of Theorem 6.5. Recall that the split steps at the initial stage of 2PP-PREPROC spawn $O(poly(\log|\mathcal{D}|))$ subproblems. We reason about the space and time usage for the j-th subproblem.

First, in the preprocessing phase, if the potential of the PANDA instance, i.e.

$$\mathsf{PANDA}(\mathcal{D}^{(j)}, \mathsf{DC}^{(j)}, (\widetilde{\boldsymbol{\theta}}, \widetilde{\boldsymbol{g}}_S), \mathsf{ProofSeq}(S)),$$

is no larger than $\log S$, then by invariant (4) of PANDA, the output tables $(S^{(j)}_B)_{B\in\mathsf{B}_T}$ (and every intermediate view produced by this PANDA instance) have size $\widetilde{O}(S)$. Otherwise, the j-th subproblem

consumes space $O(|\mathcal{D}|)$ for storing its input $(\mathcal{D}^{(j)}, \mathsf{DC}^{(j)})$ in the index. Thus, the overall space consumption for the data structure is $O(poly(\log|\mathcal{D}|)) \cdot \widetilde{O}(|\mathcal{D}| + S) = \widetilde{O}(S)$.

Next, we are left to show that for any subproblem delegated to the online phase, the PANDA instance created by 2PP-ONLINE, $\mathsf{PANDA}(\mathcal{D}^{(j)} \cup \{Q_A\}, \mathsf{DC}^{(j)} \cup \mathsf{AC}, (\boldsymbol{\lambda}, \boldsymbol{g}_T), \mathsf{ProofSeq}(T))$, terminates in time $\widetilde{O}(T_\rho)$, where

$$\log T_\rho = \ell(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S}) - \log S \cdot \|\boldsymbol{\theta}_{\mathsf{B}_S}\|_1.$$

Recall that $n_{Y|X} := \log N_{Y|X}$ and by strong duality,

$$\ell(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S}) = \sum_{(X,Y)\in\mathsf{DC}} n_{Y|X} \cdot (\delta_S)_{Y|X} + \sum_{(X,Y)\in\mathsf{DC}\cup\mathsf{AC}} n_{Y|X} \cdot (\delta_S)_{Y|X} + \sum_{(X,Y|X)\in\mathsf{SC}} n_{Z|\emptyset} \cdot (\gamma_{X,Y|X} + \gamma_{Y|X,X}).$$

To show this, we look at both PANDA instances of the j-th subproblem, $\rho(\mathcal{D}^{(j)} \cup \{Q_A\}, \mathsf{DC}^{(j)} \cup \mathsf{AC})$, i.e.

$$\begin{aligned}
&\textit{(preprocessing instance)} \qquad \mathsf{PANDA}(\mathcal{D}^{(j)}, \mathsf{DC}^{(j)}, (\widetilde{\boldsymbol{\theta}}, \widetilde{\boldsymbol{g}}_S), \mathsf{ProofSeq}(S)) \\
&\textit{(online instance)} \qquad \mathsf{PANDA}(\mathcal{D}^{(j)} \cup \{Q_A\}, \mathsf{DC}^{(j)} \cup \mathsf{AC}, (\boldsymbol{\lambda}, \boldsymbol{g}_T), \mathsf{ProofSeq}(T))
\end{aligned}$$

The *preprocessing instance* has the potential (6.30), while the *online instance* has the following *potential*:

$$\sum_{(X,Y)} n^{(j)}_{W|Z} \cdot (g_T)_{Y|X} := \sum_{(X,Y)\in\mathsf{DC}\cup\mathsf{AC}} n^{(j)}_{W|Z} \cdot (\delta_T)_{Y|X} + \sum_{(X,Y|X)\in\mathsf{SC}} n^{(j)}_{W|Z} \cdot \gamma_{X,Y|X} + \sum_{(X,Y|X)\in\mathsf{SC}} n^{(j)}_{X|\emptyset} \cdot \gamma_{Y|X,X}$$

where $n^{(j)}_{W|Z} := \log N^{(j)}_{W|Z}$ and $(Z, W, N^{(j)}_{W|Z}) \in \mathsf{DC}^{(j)}$ is the constraint that supports a positive $(g_T)_{Y|X}$. Combining the two potentials, we get

$$\begin{aligned}
\sum_{(X,Y)} n^{(j)}_{W|Z} \cdot (g_T)_{Y|X} + \|\boldsymbol{\theta}\|_1 \cdot \sum_{(X,Y)} n^{(j)}_{W|Z} \cdot (\widetilde{g}_S)_{Y|X} = \sum_{(X,Y)\in\mathsf{DC}} n^{(j)}_{W|Z} \cdot (\delta_S)_{Y|X} + \\
\sum_{(X,Y)\in\mathsf{DC}\cup\mathsf{AC}} n^{(j)}_{W|Z} \cdot (\delta_T)_{Y|X} + \sum_{(X,Y|X)\in\mathsf{SC}} (n^{(j)}_{X|\emptyset} + n^{(j)}_{W|Z}) \cdot \gamma_{X,Y|X} + \sum_{(X,Y|X)\in\mathsf{SC}} (n^{(j)}_{W|Z} + n^{(j)}_{X|\emptyset}) \cdot \gamma_{Y|X,X}
\end{aligned}$$

Recall that 2PP-PREPROC executes a split step for every (Y, X)-pair satisfying $(X, Y|X, N_{Z|\emptyset}) \in \mathsf{SC}$ and $\gamma_{X,Y|X} > 0 \vee \gamma_{Y|X,X} > 0$. So for every such (X, Y)-pair, there are some $(\emptyset, X, N^{(j)}_{X|\emptyset})$ and $(X, Y, N^{(j)}_{Y|X}) \in \mathsf{DC}^{(j)}$ such that $N^{(j)}_{X|\emptyset} \cdot N^{(j)}_{Y|X} \le N_{Z|\emptyset}$. It implies that $n^{(j)}_{X|\emptyset} + n^{(j)}_{W|Z} \le \log N_{Z|\emptyset}$. Moreover, for every $(X, Y) \in \mathsf{DC}$ where $(\delta_S)_{Y|X} > 0$ or $(\delta_T)_{Y|X} > 0$, it holds that $n^{(j)}_{W|Z} \le \log N_{Y|X}$. Therefore, we get

$$\sum_{(X,Y)} n^{(j)}_{W|Z} \cdot (g_T)_{Y|X} + \|\boldsymbol{\theta}\|_1 \cdot \sum_{(X,Y)} n^{(j)}_{W|Z} \cdot (\widetilde{g}_S)_{Y|X} \le \ell(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S})$$

This implies that for the j-th subproblem (thus for any subproblem), either in the preprocessing phase,

$$\sum_{(X,Y)} n^{(j)}_{W|Z} \cdot (g_T)_{Y|X} \leq \log S,$$

or in the online phase

$$\begin{aligned}\sum_{(X,Y)} n^{(j)}_{W|Z} \cdot (g_T)_{Y|X} &\leq \ell(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S}) - \|\boldsymbol{\theta}\|_1 \cdot \sum_{(X,Y)} n^{(j)}_{W|Z} \cdot (\widetilde{g}_S)_{Y|X} \\ &\leq \ell(\boldsymbol{\lambda}_{\mathsf{B}_T}, \boldsymbol{\theta}_{\mathsf{B}_S}) - \|\boldsymbol{\theta}_{\mathsf{B}_S}\|_1 \cdot \log S \\ &= \log T_\rho.\end{aligned}$$

So, all online PANDA instances terminate in time as predicated by (6.28).

□

6.7 Applications

In this section, we apply our framework to obtain state-of-the-art space-time tradeoffs for several specific problems, as well as obtain new tradeoff results.

6.7.1 Tradeoffs for k-Set Intersection

We will first study the CQAP (6.2) that corresponds to the non-Boolean k-Set Disjointness problem (set $y = x_{k+1}$)

$$\varphi(\mathbf{x}_{[k+1]} \mid \mathbf{x}_{[k]}) \leftarrow \bigwedge_{i \in [k]} R(x_{k+1}, x_i)$$

From the decomposition with a single node t with $\chi(t) = [k+1]$, we construct two PMTDs, one with $M_1 = \emptyset$, another with $M_2 = \{t\}$. Thus, $\nu_1(t) = \nu_2(t) = [k+1]$. This gives rise to the following (only) two-phase disjunctive rule:

$$T_{[k+1]} \vee S_{[k+1]} \leftarrow Q_{[k]}(\mathbf{x}_{[k]}) \wedge \bigwedge_{i \in [k]} R(x_{k+1}, x_i)$$

For this rule, we have the following joint Shannon-flow inequality:

$$\begin{aligned} h_S(k, k+1) + \sum_{i \in [k-1]} \{h_S(i|k+1) + h_T(k+1)\} + (k-1) \cdot h_T([k]) \\ \geq h_S([k+1]) + (k-1) \cdot h_T([k+1]). \end{aligned}$$

By Theorem 6.5, we get the tradeoff $S \cdot T^{k-1} \cong |\mathcal{D}|^k \cdot |Q_A|^{k-1}$.

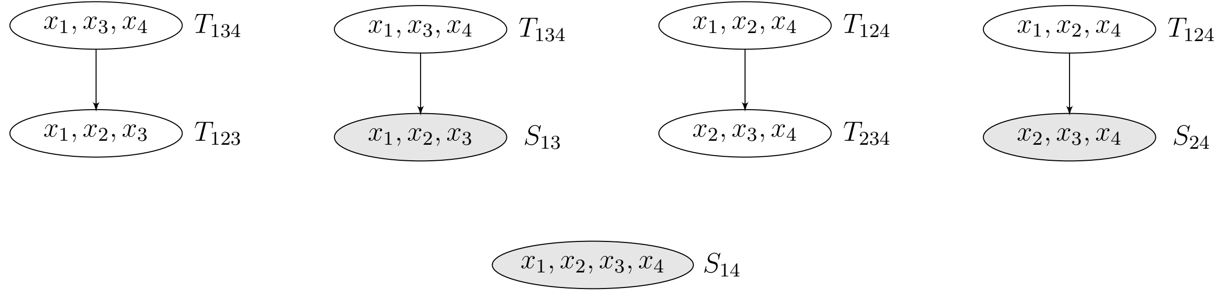


Figure 6.4: The PMTDs for the 3-reachability CQAP.

6.7.2 Tradeoffs via Fractional Edge Covers

Let $\varphi(\mathbf{x}_A \mid \mathbf{x}_A)$ be a CQAP with hypergraph $([n], \mathcal{E})$ of φ. A fractional edge cover of $S \subseteq [n]$ is an assignment $\mathbf{u} = (u_F)_{F \in \mathcal{E}}$ such that (i) $u_F \geq 0$, and (ii) for every $i \in S$, $\sum_{F: i \in F} u_F \geq 1$. For any fractional edge cover $\mathbf{u}$ of $[n]$, we define the *slack* of $\mathbf{u}$ w.r.t. $A \subseteq [n]$:

$$\alpha(\mathbf{u}, A) := \min_{i \notin A} \sum_{F \in \mathcal{E}: i \in F} u_F.$$

In other words, the slack is the maximum factor by which we can scale down the fractional cover $\mathbf{u}$ so that it remains a valid edge cover of the variables not in A. Hence $(u_F/\alpha(\mathbf{u}, A))_{F \in \mathcal{E}}$ is a fractional edge cover of $[n] \setminus A$. We always have $\alpha(\mathbf{u}, A) \geq 1$.

Theorem 6.6. *Let $\varphi(\mathbf{x}_A \mid \mathbf{x}_A)$ be a CQAP. Let $\mathbf{u}$ be any fractional edge cover of the hypergraph of φ. Then, for any input database $\mathcal{D}$, and any access request, the following intrinsic tradeoff holds:*

$$S \cdot T^{\alpha(\mathbf{u}, A)} \cong |Q_A|^{\alpha(\mathbf{u}, A)} \cdot \prod_{F \in \mathcal{E}} |R_F|^{u_F}$$

Before we prove the result, we first present the following lemma which is a generalization of Shearer's lemma (Lemma D.1 in [NRR13]).

Lemma 6.4. *Let $\mathcal{H} = ([n], \mathcal{E})$ be a hypergraph and $\hat{\mathbf{u}}$ be a fractional edge cover of $[n] \setminus A \subseteq [n]$. Then, for any polymatroid function $h \in \Gamma_n$, we have:*

$$\sum_{F \in \mathcal{E}} \hat{u}_F \cdot h(F \mid A \cap F) + h(A) \geq h([n])$$

Proof. Assume w.l.o.g. that $A = \{1, \dots, \ell - 1\}$. Then we can write:

$$h([n]) = \sum_{j=\ell}^{n} h(j \mid i : i < j) + h(A)$$

$$\leq \sum_{j=\ell}^{n} \sum_{F \in \mathcal{E}: j \in F} \hat{u}_F \cdot h(j \mid i : i < j, i \in F) + h(A)$$
$$= \sum_{F \in \mathcal{E}} \hat{u}_F \sum_{j \in F \setminus A} h(j \mid i : i < j, i \in F) + h(A)$$
$$= \sum_{F \in \mathcal{E}} \hat{u}_F \left(h(F) - \sum_{j \in F, j < \ell} h(j \mid i : i < j, i \in F) \right) + h(A)$$
$$= \sum_{F \in \mathcal{E}} \hat{u}_F \left(h(F) - \sum_{j \in F \cap A} h(j \mid i : i < j, i \in F \cap A) \right) + h(A)$$
$$= \sum_{F \in \mathcal{E}} \hat{u}_F \left(h(F) - h(A \cap F) \right) + h(A)$$
$$= \sum_{F \in \mathcal{E}} \hat{u}_F \cdot h(F \mid A \cap F) + h(A)$$

□

Now, we are ready to prove Theorem 6.6.

Proof of Theorem 6.6. To obtain the desired tradeoff, we consider two PMTDs. The first PMTD P_1 has one node t with $\chi_1(t) = [n]$ and $M_1 = \emptyset$, while the second PMTD P_2 has also one bag t with $\chi_2(t) = [n]$ and $M_2 = \{t\}$. P_1 contains the T-view $T_{[n]}(\mathbf{x}_{[n]})$, while P_2 contains the S-view $S_A(\mathbf{x}_A)$. These two PMTDs correspond to the following materialization policy: either store directly the answer of an access request, or compute the access request from scratch. Hence, we only need to consider one disjunctive rule (we use $\mathbf{x}$ to denote the tuple $\mathbf{x}_{12\ldots n}$):

$$T_{[n]}(\mathbf{x}) \vee S_A(\mathbf{x}_A) \leftarrow Q_A(\mathbf{x}_A) \wedge \bigwedge_{F \in \mathcal{E}} R_F(\mathbf{x}_F).$$

Define $\alpha = \alpha(\mathbf{u}, A)$ and $\hat{\mathbf{u}} = \mathbf{u}/\alpha$. We can now write the following proof:

$$\sum_{F \in \mathcal{E}} u_F \cdot \log N_{F|\emptyset} + \alpha \cdot \log |Q_A| \geq \sum_{F \in \mathcal{E}} u_F \cdot \{h_T(F \mid A \cap F) + h_S(A \cap F)\} + \alpha \cdot h_T(A)$$
$$= \sum_{F \in \mathcal{E}} u_F \cdot h_T(F \mid A \cap F) + \alpha \cdot h_T(A) + \sum_{F \in \mathcal{E}} u_F \cdot h_S(A \cap F)$$
$$\geq \sum_{F \in \mathcal{E}} u_F \cdot h_T(F \mid A \cap F) + \alpha \cdot h_T(A) + h_S(A) \quad \textit{(Shearer's Lemma)}$$
$$= \alpha \sum_{F \in \mathcal{E}} \hat{u}_F \cdot h_T(F \mid A \cap F) + \alpha \cdot h_T(A) + h_S(A) \quad \textit{(Lemma 6.4)}$$
$$\geq \alpha \cdot h_T([n]) + h_S(A)$$

The second inequality is a direct application of Shearer's Lemma on the sub-hypergraph $(A, \{A \cap F \mid F \in \mathcal{E}\})$ of H, since $\mathbf{u}$ is a fractional edge cover of A. The last inequality is a direct consequence of Lemma 6.4. By Theorem 6.5, we obtain the desired tradeoff. □

Theorem 6.6 can also be shown as a corollary of Theorem 1 in [DK18]. However, the data structure used in [DK18] is much more involved, since its goal is to also bound the delay during enumeration (while we are interested in total time instead). A simpler construction with the same tradeoff was shown in [DHK23]. Our framework recovers the same result using a simple materialization strategy with two PMTDs.

Example 6.10. Consider $\varphi(\mathbf{x}_{[k]} \mid \mathbf{x}_{[k]}) \leftarrow \bigwedge_{i \in [k]} R(y, x_i)$ (corresponds to the k-Set Disjointness problem) with the fractional edge cover $\mathbf{u}$, where $u_j = 1$ for $j \in \{1, \dots, k\}$. The slack w.r.t. $[k]$ is k, since the fractional edge cover $\hat{\mathbf{u}}$, where $\hat{u}_i = u_i/k = 1/k$ covers y. Applying Theorem 6.6, we obtain a tradeoff of $S \cdot T^k \cong |Q_A|^k \cdot |\mathcal{D}|^k$. When $|Q_A| = 1$, this matches the best-known space-time tradeoff for the k-Set Disjointness problem.

6.7.3 Tradeoffs via Tree Decompositions

Let $\varphi(\mathbf{x}_A \mid \mathbf{x}_A)$ be a CQAP. In the previous section, we recovered a space-time tradeoff using two trivial PMTDs. Here, we will show how our framework recovers a better space-time tradeoff by considering a larger set of PMTDs that corresponds to one decomposition.

Pick any arbitrary non-redundant free-connex decomposition $(\mathcal{T}, \chi, r)$. We start by taking any set of nodes that are not ancestors of each other in the decomposition as a materialization set. Then, for each node t in the materialization set, we merge all bags in the subtree of t into the bag of t (and truncate the subtree). By ranging over all such materialization sets, we construct a fixed (finite) set of PMTDs. We say that this set of PMTDs is *induced* from $(\mathcal{T}, \chi, r)$. We now input the induced set of PMTDs to our general framework. To discuss the obtained space-time tradeoff, take any assignment of a fractional edge cover $\mathbf{u}_t$ to each node $t \in V(\mathcal{T})$ and let u_t^* be its total weight. Let A_t denote the common variables between node t and its parent (for the root, $A_r = A$), and define $\alpha_t = \alpha(\mathbf{u}_t, A_t)$ to be the slack in node t w.r.t. A_t. Now, take the nodes P of any root-to-leaf path in $\mathcal{T}$. We will show that any such path P generates the following intrinsic tradeoff:

$$S^{\sum_{t \in P} 1/\alpha_t} \cdot T \cong |Q_A| \cdot |\mathcal{D}|^{\sum_{t \in P} u_t^*/\alpha_t}$$

To obtain the final space-time tradeoff, we take the worst such tradeoff across all root-to-leaf paths.

To show the above result, let $\mathcal{P}$ be the set of all 2-phase disjunctive rules generated by the induced set of PMTDs. We start our analysis by showing that for any disjunctive rule ρ_a in this set, there is another disjunctive rule ρ_b that is no easier than ρ_a in terms of its intrinsic tradeoff. Interestingly, despite choosing a (possibly) harder rule, we are still able to recover many state-of-the-art tradeoffs. We begin by stating the following observation.

Observation 2. *For any 2-phase disjunctive rules ρ_a and ρ_b, ρ_a is said to be no harder than ρ_b (or equivalently, ρ_b is no easier than ρ_a) if the S-targets of ρ_b are a subset of the S-targets of ρ_a and the T-targets of ρ_b are a subset of the T-targets of ρ_a.*

In other words, Observation 2 states that adding more targets to the head of a disjunctive rule can only make its model evaluation easier since we can always ignore the new targets. The next lemma makes use of the structure of the T-views in a PMTD.

Lemma 6.5. *Let $\varphi(\mathbf{x}_A \mid \mathbf{x}_A)$ be a given CQAP. Let $(\mathcal{T}, \chi, r)$ be a fixed free-connex tree decomposition. Let $\mathcal{P}$ be the set of PMTDs induced from $(\mathcal{T}, \chi, r)$. Let ρ_a be a 2-phase disjunctive rule generated from $\mathcal{P}$. Then, there is a 2-phase disjunctive rule ρ_b that is no easier than ρ_a such that for any two T-targets of ρ_b, their corresponding nodes in $\mathcal{T}$ do not lie in some root-to-leaf path.*

Proof. Let $\mathsf{B}_T(\rho)$ be a set of bags of $\mathcal{T}$ such that for every $B \in \mathsf{B}_T(\rho)$, there is a T-targets T_B picked by the 2-phase disjunctive rule ρ. In other words, ρ contains a T-target for each $B \in \mathsf{B}_T(\rho)$. Similarly, we define $\mathsf{B}_S(\rho)$ for S-views. Note that by definition of views, a T-view for a node implies that its ancestors are all associated with T-views.

We construct the 2-phase disjunctive rule ρ_b in the following way: for each PMTD, if ρ_a picks an S-view, ρ_b follows ρ_a's pick; if ρ_a picks a T-view, ρ_b looks at the path from root to this T-view (picked by ρ_a) and picks the first node t such that $\nu(t) \in \mathsf{B}_T(\rho_a)$, i.e. the top-most T-view ρ_a picked on this branch. It is easy to see that by this construction, it holds that no two bags corresponding to two different T-targets of ρ_b lie on a root-to-leaf path in $\mathcal{T}$. Furthermore, $\mathsf{B}_T(\rho_b) \subseteq \mathsf{B}_T(\rho_a)$ and $\mathsf{B}_S(\rho_b) = \mathsf{B}_S(\rho_a)$. Thus, rule ρ_b is no easier than rule ρ_a. $\square$

We are now ready to show the main property for any tree decomposition.

Lemma 6.6. *Let $\varphi(\mathbf{x}_A \mid \mathbf{x}_A)$ be a given CQAP. Let $(\mathcal{T}, \chi, r)$ be a fixed free-connex tree decomposition. Let $\mathcal{P}$ be the set of PMTDs induced from $(\mathcal{T}, \chi, r)$. Any 2-phase disjunctive rule ρ_a generated from $\mathcal{P}$ is no harder than a 2-phase disjunctive rule of the form*

$$T(\mathbf{x}_{\chi(t_\ell)}) \vee \bigvee_{j \in [\ell]} S(\mathbf{x}_{A_j}) \leftarrow Q_A(\mathbf{x}_A) \wedge \bigwedge_{F \in \mathcal{E}} R_F(\mathbf{x}_F), \tag{6.32}$$

where $t_1 = r, t_2, \ldots, t_\ell$ are nodes of the tree $\mathcal{T}$ that form a path starting from the root node r; and

$$A_j := \begin{cases} A & \text{if } j = 1 \\ \chi(t_j) \cap \chi(t_{j-1}) & \text{if } j = 2, \ldots, \ell \end{cases}$$

Proof. We start by invoking Lemma 6.5 with ρ_a and $(\mathcal{T}, \chi, r)$ as input to get the rule ρ_b that satisfies that no T-target of ρ_b is an ancestor of another. We fix the T-targets of ρ_b, i.e. fix $\mathsf{B}_T(\rho_b) = \{B_1, \dots, B_k\}$. Our goal is to show that ρ_b must contain a subset of S-targets whose corresponding nodes in $\mathcal{T}$ form a path starting from the root and ending at (including) some node t such that $\chi(t) \in \mathsf{B}_T(\rho_b)$. In the following, we will use the function $\chi^{-1}(B)$ to recover the node $t \in V(\mathcal{T})$ such that $\chi(t) = B$ and use $parent(t)$ to denote the parent node of a non-root node t.

We prove that the property holds by allowing an adversary to pick targets in PMTDs, while we adaptively choose the PMTDs that the adversary must pick from. We can choose the ordering of the PMTDs because to construct a 2-phase disjunctive rule, one view must be picked from every PMTD in $\mathcal{P}$ and we are only controlling the order in which they are examined.

Consider the PMTD P_1 where the set for S-targets is exactly $M_1 = \{\chi^{-1}(B_1), \dots, \chi^{-1}(B_k)\}$. We offer the adversary to pick a target from P_1. We claim that the adversary must pick one S-view corresponding to a node in M_1. Indeed, the adversary cannot pick a T-view from P_1 since the T-targets of ρ_b have already been fixed and cannot be changed. Suppose the adversary picks S-view associated with node $\chi^{-1}(B_i)$. For this S-view (since its parent is associated with a T-view),

$$\nu(\chi^{-1}(B_i)) = \chi(\chi^{-1}(B_i)) \cap \chi(parent(\chi^{-1}(B_i))) = B_i \cap \chi(parent(\chi^{-1}(B_i)). \tag{6.33}$$

We will now choose the PMTD P_2 where $M_2 = M_1 \cup parent(\chi^{-1}(B_i))$ and give it to the adversary. Once again, the adversary cannot pick a T-view since that will change $\mathsf{B}_T(\rho_b)$ and must choose an S-view associated with one of the nodes in M_2. Suppose the adversary picks S-view associated with a node $\chi^{-1}(B_{i'}) \in M_2$. We generate the next PMTD P_3 where $M_3 = M_2 \cup parent(\chi^{-1}(B_{i'}))$. In general, the PMTD P_{q+1} generated after q rounds has $M_{q+1} = M_q \cup parent(\chi^{-1}(B_q))$, where $\chi^{-1}(B_q) \in M_q$ is the node ρ_b picked as an S-view in the last round (for PMTD P_q with materialization set M_q). This process terminates when the S-view corresponding to the root is picked by the adversary. It is also guaranteed that this process will terminate after a finite number of steps as we always add nodes in the materialization set by moving up the tree and the tree is of finite size. At every step, the adversary picks an S-view for a bag and can only get closer to the root across all branches. Therefore, when the adversary reaches the root, there must be a subset of picked S-views corresponding to some path starting from the root $t_1 = r$ to some node t_ℓ in $\mathcal{T}$ such that $\chi(t_\ell) \in \mathsf{B}_T(\rho_b)$, the desired property. Also, note that these S-views follow (6.33), thus we complete the proof. □

Lemma 6.6 tells us that it is sufficient to analyze only the 2-phase disjunctive rules that are of the form as shown in (6.32). Now we proceed to find a proof sequence for (6.32). Let the bags for $t_1 = r, t_2, \dots, t_\ell$ have corresponding fractional edge covers $\mathbf{u}^{(1)}, \dots, \mathbf{u}^{(\ell)}$, where $\mathbf{u}^{(j)} = (u_F^{(j)})_{F \in \mathcal{E}}$, for $j \in [\ell]$. We define α_j to be the slack of each bag w.r.t. the variables in A_j (i.e. $\alpha_j = \alpha(\mathbf{u}^{(j)}, A_j)$),

and introduce a factor $\beta_j = \beta^\star/\alpha_j$ where $\beta^\star = \sum_{j\in[\ell]} \alpha_j$. Next, we apply Theorem 6.6 for each of the ℓ bags, multiply the proof sequence obtained for the j-th bag with β_j, and sum up terms as follows:

$$
\begin{aligned}
\sum_{j\in[\ell]} \beta_j \sum_{F\in\mathcal{E}} u_F^{(j)} \cdot \log|R_F| + \beta^\star \cdot \log|Q_A| &\geq \sum_{j\in[\ell]} \beta_j \sum_{F\in\mathcal{E}} u_F^{(j)} \cdot (h_T(F \mid A_j \cap F) + h_S(A_j \cap F)) \\
&\qquad\qquad\qquad + \beta^\star \cdot h_T(A_1) \\
&= \sum_{j\in[\ell]} \beta_j \sum_{F\in\mathcal{E}} u_F^{(j)} \cdot h_T(F \mid A_j \cap F) + \beta^\star \cdot h_T(A_1) \\
&\quad + \sum_{j\in[\ell]} \beta_j \sum_{F\in\mathcal{E}} u_F \cdot h_S(A_j \cap F) \\
&\geq \sum_{j\in[\ell]} \beta_j \cdot \alpha_j \sum_{F\in\mathcal{E}} \hat{u}_F^{(j)} \cdot h_T(F \mid A_j \cap F) + \beta^\star \cdot h_T(A_1) \\
&\qquad\qquad\qquad + \sum_{j\in[\ell]} \beta_j \cdot h_S(A_j) \\
&= \sum_{j\in[\ell]} \beta^\star \sum_{F\in\mathcal{E}} \hat{u}_F^{(j)} \cdot h_T(F \mid A_j \cap F) + \beta^\star \cdot h_T(A_1) \\
&\qquad\qquad\qquad + \sum_{j\in[\ell]} \beta_j \cdot h_S(A_j) \\
&= \sum_{j\in\{2,\ldots,\ell\}} \beta^\star \sum_{F\in\mathcal{E}} \hat{u}_F^{(j)} \cdot h_T(F \mid A_j \cap F) \\
&\quad + \beta^\star \left(\sum_{F\in\mathcal{E}} \hat{u}_F^{(1)} \cdot h_T(F \mid A_1 \cap F) + h_T(A_1) \right) \\
&\quad + \sum_{j\in[\ell]} \beta_j \cdot h_S(A_j) \\
&\geq \sum_{j\in\{2,\ldots,\ell\}} \beta^\star \sum_{F\in\mathcal{E}} \hat{u}_F^{(j)} \cdot h_T(F \mid A_j \cap F) + \beta^\star h_T(\chi(t_1)) \\
&\qquad\qquad\qquad + \sum_{j\in[\ell]} \beta_j \cdot h_S(A_j) \\
&\geq \sum_{j\in\{2,\ldots,\ell\}} \beta^\star \sum_{F\in\mathcal{E}} \hat{u}_F^{(j)} \cdot h_T(F \mid A_j \cap F) + \beta^\star h_T(A_2) \\
&\qquad\qquad\qquad + \sum_{j\in[\ell]} \beta_j \cdot h_S(A_j) \\
&\cdots \\
&\geq \beta^\star h_T(\chi(t_\ell)) + \sum_{j\in[\ell]} \beta_j \cdot h_S(A_j),
\end{aligned}
$$

where the second inequality is a direct application of Shearer's Lemma and the third inequality is a direct consequence of Lemma 6.4. From this proof sequence, we obtain the following intrinsic

tradeoff,

$$|Q_A|^{\beta^\star} \cdot |\mathcal{D}|^{\sum_{j\in[\ell]} \beta_j \cdot u_j^*} \cong S^{\sum_{j\in[\ell]} \beta_j} \cdot T^{\beta^\star}$$

where $u_j^* = \sum_{F\in\mathcal{E}} u_F^{(j)}$. Equivalently, we get the desired expression

$$|Q_A| \cdot |\mathcal{D}|^{\sum_{j\in[\ell]} u_j^*/\alpha_j} \cong S^{\sum_{j\in[\ell]} 1/\alpha_j} \cdot T. \tag{6.34}$$

In the above tradeoff, for a given S, (assume $|Q_A| = 1$), we get

$$\begin{aligned} \log T &= \sum_{j\in[\ell]} \frac{u_j^*}{\alpha_j} \cdot \log|\mathcal{D}| - \sum_{j\in[\ell]} (1/\alpha_j) \log S \\ &= \sum_{j\in[\ell]} (1/\alpha_j) \cdot (u_j^* \log|\mathcal{D}| - \log S) \end{aligned}$$

One observation is that if some bag t_j on the path $t_1, \dots, t_\ell$ has an AGM bound that is not greater than S, then the materialization of t_j's corresponding S-view $S(\mathbf{x}_{A_j})$ can be fully materialized as the model for (6.32). Otherwise, for every $t_j, j \in [\ell]$, we have that $(u_j^* \log|\mathcal{D}| - \log S)$ is non-negative for every $j \in [\ell]$, thus the above expression for $\log T$ monotonically increases as ℓ increases. Therefore, the most expensive tradeoff corresponds to the disjunctive rule of the form in (6.32) that starts from the root and ends at a leaf. To obtain the final space-time tradeoffs, we simply take the worst tradeoffs across all the root-to-leaf paths in $\mathcal{T}$.

Example 6.11. Consider the 4-reachability CQAP. Here, $H = A = \{x_1, x_5\}$. We will consider the tree decomposition with bags $t_1 = \{x_1, x_2, x_4, x_5\} \to t_2 = \{x_2, x_3, x_4\}$.

Take the edge cover $u_1 = 1, u_4 = 1$ for the bag t_1, and the edge cover $u_2 = 1, u_3 = 1$ for the bag t_2. The first bag has slack $\alpha_1 = 1$ (w.r.t. x_1, x_5), while the second has slack $\alpha_2 = 2$ (w.r.t. x_2, x_4). Here we have one root-to-leaf path, hence we get the tradeoff $S^{1+1/2} \cdot T \cong |Q_A| \cdot |\mathcal{D}|^{2/1+2/2}$, or equivalently $S^{3/2} \cdot T \cong |Q_A| \cdot |\mathcal{D}|^3$.

Before we conclude this section, we show that the tradeoff we obtained in (6.34) across all root-to-leaf paths of a fixed tree decomposition $(\mathcal{T}, \chi)$ recovers (and possibly improves over) Theorem 13 of [DK18], without incurring extra hyper-parameters. Indeed, in [DK18], the authors set a hyper-parameter $\delta(t)$ for every $t \in V(\mathcal{T})$ and let the online answering time to be $T = |\mathcal{D}|^{\sum_{j\in[\ell]} \delta(t_j)}$, where $t_1, \dots, t_\ell$ is a root-to-leaf path of $\mathcal{T}$ that maximizes $\sum_{j\in[\ell]} \delta(t_j)$. Suppose we construct a 2-phase disjunctive rule of the form (6.34) for this root-to-leaf path, $t_1, \dots, t_\ell$, then for any fractional edge cover $u_j^*, j \in [\ell]$, it holds that (by rewriting (6.34))

$$\log_{|\mathcal{D}|} S = \frac{1}{\sum_{j\in[\ell]} (1/\alpha_j)} \cdot \left(\sum_{j\in[\ell]} \frac{u_j^*}{\alpha_j} - \log_{|\mathcal{D}|} T \right)$$

$$= \frac{1}{\sum_{j\in[\ell]}(1/\alpha_j)} \cdot \left(\sum_{j\in[\ell]} \frac{u_j^*}{\alpha_j} - \sum_{j\in[\ell]} \delta(t_j) \right)$$
$$= \frac{1}{\sum_{j\in[\ell]}(1/\alpha_j)} \cdot \sum_{j\in[\ell]} \frac{1}{\alpha_j}(u_j^* - \delta(t_j) \cdot \alpha_j)$$
$$\leq \max_{j\in[\ell]} \left(\sum_{F\in\mathcal{E}} u_F^{(j)} - \delta(t_j) \cdot \alpha_j \right).$$

The last inequality holds because for all $w_i \geq 0$ such that $\sum_i w_i = 1$, $\sum_{i=1}^n \gamma_i w_i \leq \max_i \gamma_i$. Setting $w_i = \frac{1}{\alpha_i} \cdot \frac{1}{\sum_{j\in[\ell]}(1/\alpha_j)}$, we get the desired expression. Thus, it is easy to see that the tradeoff we obtained in (6.34) across all root-to-leaf paths of a fixed tree decomposition recovers results in [DK18].

6.7.4 Tradeoffs for k-Reachability

In this part, we will consider the CQAP that corresponds to the k-reachability problem described in Example 6.2:

$$\phi_k(x_1, x_{k+1} \mid x_1, x_{k+1}) \leftarrow \bigwedge_{i\in[k]} R(x_i, x_{i+1}).$$

Prior work [GKLP17] has shown the following tradeoff for a input $\mathcal{D}$, which was conjectured to be asymptotically optimal for $|Q_A| = 1$:

$$S \cdot T^{2/(k-1)} \cong |\mathcal{D}|^2 \cdot |Q_A|^{2/(k-1)}.$$

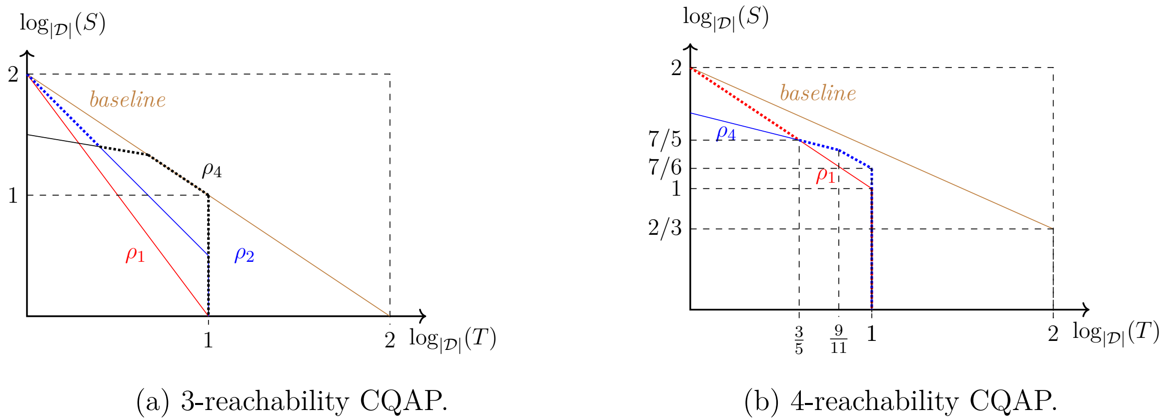


Figure 6.5: Space-time tradeoffs for the 3- and 4-reachability CQAP. The new tradeoffs obtained from our framework are depicted via the dotted segments. The brown lines (baseline) show the previous state-of-the-art tradeoffs.

We will show that the above tradeoff can be significantly improved for $k \geq 3$ by applying our framework.

3-reachability As a first step, we consider the set of all non-redundant and non-dominant PMTDs (five in total), as seen in Figure 6.4. The five PMTDs will lead to $2^4 = 16$ disjunctive rules, but we can reduce the number of rules we analyze by discarding rules with strictly more targets than other rules. For example, the disjunctive rule with head $T_{134} \vee S_{13} \vee T_{124} \vee S_{14}$ can be ignored, since there is another disjunctive rule which has a strict subset of targets, i.e., $T_{134} \vee T_{124} \vee S_{14}$. We list out the heads of the two-phase disjunctive rules we need to consider (we omit the variables for simplicity), along with the intrinsic tradeoffs for each rule in Table 6.1.

Table 6.1: 2-phase disjunctive rules for 3-reachability

rule	head	tradeoff(s)
ρ_1	$T_{134} \vee T_{124} \vee S_{14}$	$S \cdot T^2 \cong \lvert\mathcal{D}\rvert^2 \cdot \lvert Q_A\rvert^2$
ρ_2	$T_{123} \vee S_{13} \vee T_{124} \vee S_{14}$	$S^2 \cdot T^3 \cong \lvert\mathcal{D}\rvert^4 \cdot \lvert Q_A\rvert^3$ $T \cong \lvert\mathcal{D}\rvert \cdot \lvert Q_A\rvert$
ρ_3	$T_{134} \vee T_{234} \vee S_{24} \vee S_{14}$	$S^2 \cdot T^3 \cong \lvert\mathcal{D}\rvert^4 \cdot \lvert Q_A\rvert^3$ $T \cong \lvert\mathcal{D}\rvert \cdot \lvert Q_A\rvert$
ρ_4	$T_{123} \vee S_{13} \vee T_{234} \vee S_{24} \vee S_{14}$	$S \cdot T \cong \lvert\mathcal{D}\rvert^2 \cdot \lvert Q_A\rvert$ $S^4 \cdot T \cong \lvert\mathcal{D}\rvert^6 \cdot \lvert Q_A\rvert$ $T \cong \lvert\mathcal{D}\rvert \cdot \lvert Q_A\rvert$

Note that rules can admit two (or more) tradeoffs that do not dominate each other; hence, we need to pick the best tradeoff depending on the regime we consider (see Zhao et al. [ZDK23b] for how we prove each tradeoff). To understand the combined tradeoff we obtain from our analysis, we plot in Figure 6.5a each tradeoff curve by taking $\log_{|\mathcal{D}|}$ and then taking the x-axis as $\log_{|\mathcal{D}|}(T)$ and the y-axis as $\log_{|\mathcal{D}|}(S)$ (fixing $|Q_A| = 1$). The dotted line in the figure shows the resulting tradeoff, which is a piecewise linear function. Note that each linear segment denotes a different strategy that is optimal for that regime of space. Note that Figure 6.5a is not necessarily optimized for $|Q_A| > 1$. Suppose that $S = |\mathcal{D}|$ and we receive $|\mathcal{D}|$ single-tuple access requests in the online phase. Answering them one-by-one costs time $\widetilde{O}(|\mathcal{D}|^2)$ using the above tradeoffs. However, one better strategy is to batch the $|\mathcal{D}|$ tuples into a single access request (essentially a 4-cycle query where $|Q_A| = |\mathcal{D}|$) and use PANDA to answer it from scratch, which costs time $\widetilde{O}(|\mathcal{D}|^{3/2})$.

4-reachability We also study the CQAP for the 4-reachability problem. We defer the (rather involved) proofs to [ZDK23b], but we include here a plot (Figure 6.5b) similar to the one in Figure 6.5a. One surprising observation is that the new space-time tradeoff is better than the prior state-of-the-art for *every regime of space*. We should also point out that the tradeoff can possibly be further improved by including even more PMTDs (our analysis involved 11 PMTDs!), but the calculations were beyond the scope of this work.

General reachability Analyzing the best possible tradeoff for $k \geq 5$ becomes a very complex proposition. However, from Section 6.7.3 and the analysis of [DHK23], our framework can at least obtain the $S \cdot T^{2/(k-1)} \cong |\mathcal{D}|^2$ tradeoff, and can likely strictly improve it.

6.8 Related Work

Set Intersections and Distance Oracles. Space-time tradeoffs for query answering (either exact or approximate) has been an active study across multiple research communities over the last decade [KRR13, AN16, LMNT15, CP10a]. Cohen et al. [CP10a] introduced the fast set intersection problem and presented a data structure to enumerate the intersection of two sets with guarantees on the total answering time. Goldstein et al. [GKLP17] formulated the k-reachability problem on graphs, and showed a simple recursive data structure which achieves the $S \cdot T^{2/(k-1)} = O(|\mathcal{D}|^2)$ tradeoff. They also conjectured that the tradeoff is optimal and used it to justify the optimality of an approximate distance oracle proposed by [Aga14]. The study of (approximate) distance oracles over graphs was initiated by Patrascu and Roditty [PR10], where lower bounds are shown on the size of a distance oracle for sparse graphs based on a conjecture on the best possible data structure for a set intersection problem. Cohen and Porat [CP10b] also connected set intersection to distance oracles. Agarwal et al. [AGHP11, Aga14] introduced the notion of *stretch* of an oracle that controls the error allowed in the answer. Further, for stretch-2 and stretch-3 oracles, we can achieve tradeoffs $S \cdot T = O(|\mathcal{D}|^2)$ and $S \cdot T^2 = O(|\mathcal{D}|^2)$ respectively, and for any integer $k > 0$, a stretch-$(1+1/k)$ oracle exhibits an $S \cdot T^{1/k} = O(|\mathcal{D}|^2)$ tradeoff. However, no lower bounds are known so far for non-constant query time.

Space-delay Tradeoffs. A different line of work considers the problem of enumerating query results of a join query, with the goal of minimizing the *delay* between consecutive tuples of the output. In constant delay enumeration [Seg13, BDG07], the goal is to achieve constant delay for a CQ after a preprocessing step of linear time (and space); however, only a subset of CQs can achieve such a tradeoff. Factorized databases [OZ15] achieve constant delay enumeration after a more expensive super-linear preprocessing step for any CQ. If we want to reduce preprocessing time further, it is necessary to increase the delay. Kara et al. [KNOZ20] presented a tradeoff between

preprocessing time and delay for enumerating the results of any hierarchical CQ under static (and dynamic) settings. Deng et al. [DLT23b] initiates the study of the space-query tradeoffs for range subgraph counting and range subgraph listing problems. The problem of CQs with access patterns was first introduced by Deep and Koutris [DK18] (under the restriction $|Q_A| = 1$), but the authors only consider full CQAPs. Previous work [XD17] considered the problem of constructing space-efficient views of graphs to perform graph analytics, but did not offer any theoretical guarantees. More recently, Kara et al. [KNOZ23a] studied tradeoffs between preprocessing time, delay, and update time for CQAPs. They characterized the class of CQAPs that admit linear preprocessing time, constant delay enumeration, and constant update time. All of the above results concern the tradeoff between space (or preprocessing time) and delay, while our work focuses on the total time to answer the query. Deep et al. [DHK23] is the closest to our work, where the authors study the problem of building tradeoffs for Boolean CQs specifically. The authors propose two results that slightly improve upon [DK18]. They were also the first to recognize that the k-reachability tradeoff is not optimal by proposing a small improvement for $k \geq 3$. The results in our work are a vast generalization that is achieved using a more comprehensive framework. For the dynamic setting, [BKS17] initiated the study of answering CQs under updates. Recently, [KNN+19] presented an algorithm for counting the number of triangles under updates. [KNOZ23a] proposed dynamic algorithms for CQAPs and provided a syntactic characterization of queries that admit constant time per single-tuple update and whose output tuples can be enumerated with constant delay.

6.9 Conclusion

In this chapter, we present a framework for computing general space-time tradeoffs for answering CQs with access patterns. We show the versatility of our framework by demonstrating how it can capture state-of-the-art tradeoffs for problems that have been studied separately. The application of our framework has also uncovered previously unknown tradeoffs. Many open problems remain, among which are obtaining (conditional) lower bounds that match our upper bounds, and investigating how to make our approach practical.

Chapter 7

Join with Recursion I: An Algorithmic Framework

Datalog is a recursive query language that has gained prominence due to its expressivity and rich applications across multiple domains, including graph processing [SPSL13], declarative program analysis [SJSW16, SB15], and business analytics [GHLZ13]. Most of the prior work focuses on Datalog programs over the Boolean semiring (this corresponds to the standard relational join semantics): popular examples include same generation [BMSU85], cycle finding, and pattern matching. Many program analysis tasks such as Dyck-reachability [Rep98, LZR20, LZR21], context-free reachability [MP21], and Andersen's analysis [And94] can also be naturally cast as Datalog programs over the Boolean domain. However, modern data analytics frequently involve aggregations over recursion. Seminal work by Green et al. [GKT07] established the semantics of Datalog over the so-called K-relations where tuples are annotated by the domain of a fixed semiring. Prior work has also studied the fundamental problem of computing Datalog provenance by leveraging the proof-theoretic and fixpoint-theoretic derivations [RMS21b, RMS21a], as well as through building compact circuits for various semirings [DMRT14]. Recently, Abo Khamis et al. [KNP$^+$22a] proposed Datalog$^\circ$, an elegant extension of Datalog, and established key algebraic properties that governs convergence of Datalog$^\circ$. The authors further made the observation that the convergence rate of Datalog can be studied by confining to the class of *naturally-ordered* semirings (formally defined in Section 7.1).

A parallel line of work has sought to characterize the complexity of Datalog evaluation. The general data complexity of Datalog is P-complete [Gar97, Cos99], with the canonical P-complete Datalog program:

$$T(x_1) \leftarrow U(x_1).$$

$$T(x_1) \leftarrow T(x_2) \wedge T(x_3) \wedge R(x_2, x_3, x_1).$$

Some fragments of Datalog can have lower data complexity: the evaluation for non-recursive Datalog is in AC_0, whereas evaluation for Datalog with linear rules is in NC and thus efficiently parallelizable [UG88, AP93]. However, all such results do not tell us how efficiently we can evaluate a given Datalog program. As an example, Program (7) can be evaluated in linear time with respect to the input size m [GK04]. A deeper understanding of precise upper bounds for Datalog is important, given that it can capture practical problems (all of which are in P) across various domains as mentioned before.

Unfortunately, even for the Boolean semiring, the current general algorithmic techniques for Datalog evaluation typically aim for an imprecise polynomial bound instead of specifying the tightest possible exponent. Semi-naïve or naïve evaluation only provides upper bounds on the number of iterations, ignoring the computational cost of each iteration. For example, Program (7) can be evaluated in at most $O(m)$ iterations, but the cost of an iteration can be as large as $\Theta(m)$. Further, going beyond the Boolean semiring, obtaining tight bounds for evaluating general Datalog programs over popular semirings (such as absorptive semirings with a total order which are routinely used in data analytics [RMS21b, RMS21a]) is unclear. In particular, proving correctness of program evaluation is not immediate and the impact of the semiring on the evaluation time remains unknown.

Endeavors to pinpoint the exact data complexity for Datalog fragments have focused on the class of Conjunctive Queries (CQs) [JR18, AGM08, KNS17] and union of CQs [CK21], where most have been dedicated to develop faster algorithms. When recursion is involved, however, exact runtimes are known only for restricted classes of Datalog. Seminal work of Yannakakis [Yan90] established a $O(n^3)$ runtime for chain Datalog programs (formally defined in Section 7.1.2), where n is the size of the active domain. Such programs have a direct correspondence to context-free grammars and capture a fundamental class of static analysis known as context-free reachability (CFL). When the chain Datalog program corresponds to a regular grammar, the runtime can be further improved to $O(m \cdot n)$[1]. An $O(n^3)$ algorithm was proposed for the Datalog program that captures Andersen's analysis [MP21]. Recently, Casel and Schmid [CS21] studied the fine-grained complexity of evaluation, enumeration, and counting problems over regular path queries (also a Datalog fragment) with similar upper bounds. However, none of the above techniques generalize to arbitrary Datalog programs.

Our Contribution In this chapter, we ask: given a Datalog program P over a naturally-ordered semiring σ, what is the *tightest possible runtime as a function of the input size m and the size of the active domain n*? Our main contributions are as follows.

We propose a general, two-phased framework for evaluation of P over σ. The first phase *grounds* P into an equivalent system of polynomial equations G. Though constructing groundings naïvely

[1] m denotes the size of the input data and n denotes the size of the active domain for a Datalog program throughout this chapter.

is rather straightforward, we show that groundings of smaller size are attainable via tree decomposition techniques. The second phase evaluates the least fixpoint of G over the underlying semiring σ. We show that finite-rank semirings and absorptive semirings with total order, two routinely-used classes of semirings in practice, admit efficient algorithms for least fixpoint computation. We apply our framework to prove state-of-the-art and new results for practical Datalog fragments (e.g. linear Datalog).

Further, we establish tightness of our results by demonstrating a matching lower bound on the running time (conditioned on the popular min-weight ℓ-clique conjecture) and size of the grounded program (unconditional) for a class of Datalog programs.

7.1 Preliminaries

In this section, we describe the concepts used in the rest of this chapter.

7.1.1 Semirings and Their Properties

Recall the definition of Datalog at Section 2.4 and Semirings (Definition 2.8).

ω-complete and ω-continuous Semirings An ω-chain in a poset is a sequence $a_0 \sqsubseteq a_1 \sqsubseteq \dots$ with elements from the poset. A naturally-ordered semiring σ is *ω-complete* if every (infinite) ω-chain $a_0 \sqsubseteq a_1 \sqsubseteq \cdots$ has a least upper bound $\texttt{sup}\ a_i$, and is *ω-continuous* if also $\forall a \in \Sigma$, $a \oplus \texttt{sup}\ a_i = \texttt{sup}\ (a \oplus a_i)$ and $a \otimes \texttt{sup}\ a_i = \texttt{sup}\ (\mathtt{a} \otimes \mathtt{a_i})$.

Rank The *rank* of a strictly increasing ω-chain $a_0 \sqsubset a_1 \sqsubset \cdots \sqsubset a_r$ is r ($x \sqsubset y$ if $x \sqsubseteq y$ and $x \neq y$). We say that a semiring σ has rank r if every strictly increasing sequence has rank at most r. Any semiring over a finite domain has constant rank (e.g. $\mathbb{B}$ has rank 1).

We highlight two special classes of naturally-ordered semirings to be studied in this chapter: dioids and absorptive semirings.

Dioids A *dioid* σ is a semiring where $\oplus$ is idempotent, i.e. $a \oplus a = a$ for any $a \in \Sigma$. A dioid is naturally ordered, i.e. $a \sqsubseteq b$ if $a \oplus b = b$. The natural order satisfies the monotonicity property: for all $a, b, c \in \Sigma, a \sqsubseteq b$ implies $a \oplus c \sqsubseteq b \oplus c$ and $a \otimes c \sqsubseteq b \otimes c$.

Absorptive Semirings A semiring σ is *absorptive* if $\mathbf{1} \oplus a = \mathbf{1}$ for every $a \in \Sigma$. An absorptive semiring is also called 0-*stable* [KNP+22a]. An absorptive semiring is a dioid and $\mathbf{0} \sqsubseteq a \sqsubseteq \mathbf{1}$, for any $a \in \Sigma$. An equivalent characterization of absorptive semirings can be found here [ZDK+24a]. Simple examples of absorptive semirings are the Boolean semiring and the *tropical semiring* $\mathsf{Trop}^+ = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$ (that can be used to measure the shortest path), where its natural order $x \sqsubseteq y$ is the reverse order $x \geq y$ on $\mathbb{R}_+ \cup \{\infty\}$.

7.1.2 Datalog over Semirings

Next we describe the syntax of Datalog over a semiring σ. Here EDBs are considered as σ-relations, i.e., each tuple in each EDB R is annotated by an element from the domain of σ. Tuples not in the EDB are annotated by $\mathbf{0}$ implicitly. Standard relations are essentially $\mathbb{B}$-relations. When a Datalog program is evaluated on such σ-relations, IDB σ-relations are derived. Here, conjunction ($\wedge$) is interpreted as $\otimes$, and disjunction as $\oplus$ of σ, as discussed in [GKT07].

Sum-product Queries The Datalog program (2.12) has two rules of the same head, thus an IDB tuple can be derived by either rule (i.e. a disjunction over rules). Following Abo Khamis et al. [KNP+22a], we combine all rules with the same IDB head into one using disjunction. Hence the program under a semiring σ is written as:

$$T(x_1, x_2) \leftarrow R(x_1, x_2) \oplus \left(\bigoplus_{x_3} T(x_1, x_3) \otimes R(x_3, x_2) \right). \tag{7.1}$$

where $\oplus$ corresponds to alternative usage (or disjunction) [GKT07] and the implicit $\exists x_3$ in the second rule of (2.12) is made explicit by $\bigoplus_{x_3}$. Let ℓ be the number of variables $x_1, x_2, \cdots, x_\ell$ in a Datalog program as in Eqn. (7.1). For $J \subseteq [\ell]$, where $[\ell] = \{1, 2, \ldots, \ell\}$, $\mathbf{x}_J$ denotes the set of variables $\{x_j \mid j \in J\}$. In the rest of the chapter, w.l.o.g., we consider a Datalog program as a set of rules of distinct IDB heads, where each rule has the following form:

$$T(\mathbf{x}_H) \leftarrow \varphi_1(\mathbf{x}_H) \oplus \varphi_2(\mathbf{x}_H) \oplus \ldots \tag{7.2}$$

Here $\mathbf{x}_H$ denotes the set of head variables. The body of a rule is a *sum* (i.e., $\oplus$) over one or more *sum-product queries* (defined below) $\varphi_1(\mathbf{x}_H), \varphi_2(\mathbf{x}_H), \ldots$ corresponding to different derivations of the IDB $T(\mathbf{x}_H)$ over a semiring σ. Formally, a *sum-product (in short, sum-prod) query* φ over σ has the following form:

$$\varphi(\mathbf{x}_H): \quad \bigoplus_{\mathbf{x}_{[\ell] \setminus H}} \bigotimes_{J \in \mathcal{E}} T_J(\mathbf{x}_J), \tag{7.3}$$

where (1) $\mathbf{x}_H$ is the set of head variables, (2) each T_J is an EDB or IDB predicate, and (3) $([\ell], \mathcal{E})$ is the *associated hypergraph* of φ with hyperedges $\mathcal{E} \subseteq 2^{[\ell]}$ and vertex set as $\mathbf{x}_{[\ell]}$.

For every Datalog program in the sum-product query form, we will assume that there is a unique IDB that we identify as the *target* (or *output*) of the program. We use $\mathsf{arity}(P)$ to denote the maximum number of variables contained in any IDB of a program P.

Monadic, Linear, and Chain Datalog We say that a Datalog program P is *monadic* if every IDB is unary (i.e. $\mathsf{arity}(P) = 1$); a Datalog program is *linear* if every sum-prod query in every rule contains at most one IDB (e.g., the program in Eq. (7.1) is linear). A *chain* query is a sum-prod query over

binary predicates as follows:

$$\varphi(x_1, x_{k+1}) : \bigoplus_{\mathbf{x}_{\{2,\dots,k\}}} T_1(x_1, x_2) \otimes T_2(x_2, x_3) \otimes \cdots \otimes T_k(x_k, x_{k+1}).$$

A *chain* Datalog program is a program where every rule consists of one or a sum of multiple chain queries. A chain Datalog program corresponds to a Context-Free Grammar (CFG) [Yan90].

Least Fixpoint Semantics The least fixpoint semantics of a Datalog program P over a semiring σ is defined as the standard Datalog, through its immediate consequence operator (ICO). An ICO applies all rules in P exactly once over a given instance and uses all derived facts (and the instance) for the next iteration. The naive evaluation algorithm iteratively applies ICO until a least fixpoint is reached (if any). In general, naive evaluation of Datalog over a semiring σ may not converge, depending on σ. Kleene's Theorem [DP02] shows that if σ is ω-continuous and the semiring σ is ω-complete, then the naive evaluation converges to the least fixpoint. Following prior work [EKL10, GKT07], we assume that σ is ω-continuous and ω-complete.

σ-equivalent Two Datalog programs P, G are σ-equivalent if for any input EDB instance annotated with elements of a semiring σ, the targets of P and G are identical σ-relations when least fixpoints are reached for both.

Parameters & Computational Model We use m to denote the sum of sizes of the input EDB σ-relations to a Datalog program (henceforth referred to as the "*input size is m*"). We use n to denote the size of the active domain of EDB σ-relations (i.e., the number of distinct constants that occur in the input). If $\mathsf{arity}(P) = k$, then $n \leq m \cdot k$. We assume *data complexity* [Var82], i.e. the program P size (the total number of predicates and the variables) is a constant. We use the standard word-RAM model with $O(\log m)$-bit words and unit-cost operations [CLRS22] for all complexity results. $\widetilde{O}$ hides poly-logarithmic factors in the size of the input.

7.2 Framework and Main Results

This section presents a general framework to analyze the data complexity of Datalog programs. We show how to use this framework to obtain both state-of-the-art and new algorithmic results.

A common technique to measure the runtime of a Datalog program is to multiply the number of iterations until the fixpoint is reached with the cost of each ICO evaluation. Although this method can show a polynomial bound in terms of data complexity, it cannot generate the tightest possible upper bound. Our proposed algorithmic framework allows us to decouple the semiring-dependent fixpoint computation from the structural properties of the Datalog program. It splits the computation

into two distinct phases, where the first phase concerns the *logical structure* of the program, and the second the *algebraic structure* of the semiring.

Grounding Generation In the first phase, the Datalog program is transformed into a σ-equivalent *grounded program*, where each rule contains only constants. A *grounding* of a grounded Datalog program is a system of polynomial equations where we assign to each grounded EDB atom a distinct coefficient e in the semiring and to each grounded IDB atom a distinct variable x. Our goal in this phase is to construct a σ-equivalent grounding G of the smallest possible size $|G|$. The size of G is measured as the total number of coefficients and variables in the system of polynomial equations.

Grounding Evaluation In the second phase, we need to deal with evaluating the least fixpoint of G over the semiring σ. The fixpoint computation now depends on the structure of the semiring σ, and we can completely ignore the underlying logical structure of the program. Here, we need to construct algorithms that are as fast as possible w.r.t. the size of the grounding $|G|$. For example, it is known that over the Boolean semiring, the least fixpoint of G can be computed in time $O(|G|)$ [GK04].

7.2.1 Grounding Generation

The first phase of the framework generates a grounded program.

Example 7.1. We will use as an example the following variation of Program (7.1), over an arbitrary semiring σ:

$$T(x_1, x_2) \leftarrow R(x_1, x_2) \oplus \left(\bigoplus_{x_3} T(x_1, x_3) \otimes U(x_3) \otimes R(x_3, x_2) \right).$$

We introduced an unary atom U to capture properties of the nodes in the graph. Consider an instance where R contains two edges, (a, b) and (b, c), and U contains three nodes, a, b, c. One possible σ-equivalent grounded Datalog program (over any semiring σ) is:

$$T(a, b) \leftarrow R(a, b) \oplus T(a, a) \otimes U(a) \otimes R(a, b), \qquad T(a, a) \leftarrow \mathbf{0}$$
$$T(a, c) \leftarrow T(a, b) \otimes U(b) \otimes R(b, c), \qquad T(b, c) \leftarrow R(b, c).$$

This corresponds to the following system of polynomial equations:

$$x_{ab} = e_{ab} \oplus x_{aa} \otimes f_a \otimes e_{ab} \qquad x_{aa} = \mathbf{0}$$
$$x_{ac} = x_{ab} \otimes f_b \otimes e_{bc} \qquad x_{bc} = e_{bc}.$$

As there is a one-to-one correspondence between a grounded program and its grounding, we will only use the term grounding from now on. The naive way to generate a σ-equivalent grounded

Table 7.1: Summary of the grounding results. The $\widetilde{O}$ notation hides polylog factors in m (total input size) and n (size of the active domain). k denotes the arity of the program. See Section 7.5 for the definitions of $\mathsf{fhw}, \mathsf{subw}$.

Semiring	**Class of Programs**	**IDB Arity**	**Grounding Size & Time**	**Result**
all	rulewise-acyclic	k	$O(n^{k-1} \cdot (m + n^k))$	Theorem 7.1
all	rulewise free-connex acyclic	k	$O(m + n^k)$	Theorem 7.1
all	linear and rulewise-acyclic	2	$O(n \cdot m)$	Theorem 7.2
all	fractional hypertree-width $\leq \mathsf{fhw}$	k	$O(n^{k-1} \cdot (m^{\mathsf{fhw}} + n^{k \cdot \mathsf{fhw}}))$	Theorem 7.5
dioid	submodular width $\leq \mathsf{subw}$	k	$\widetilde{O}(n^{k-1} \cdot (m^{\mathsf{subw}} + n^{k \cdot \mathsf{subw}}))$	Theorem 7.2
dioid	submodular width $\leq \mathsf{subw}$	1	$\widetilde{O}(m^{\mathsf{subw}})$	Corollary of Theorem 7.2
dioid	free-connex submodular width $\leq \mathsf{fsubw}$	k	$\widetilde{O}(m^{\mathsf{fsubw}} + n^{k \cdot \mathsf{fsubw}})$	Theorem 7.3
dioid	linear and submodular width $\leq \mathsf{subw}$	k	$\widetilde{O}(n^{k-1} m^{\mathsf{subw}-1} \cdot (m + n^k))$	Theorem 7.4

program is to take every rule and replace the variables with all possible constants. However, this may create a grounding of very large size. Our key idea is that we can optimize the size of the generated grounding by exploiting the logical structure of the rules.

Upper Bounds Our first main result (Section 7.4) considers the body of every sum-product query in the Datalog program is acyclic (formally Definition 7.4); we call such a program *rulewise-acyclic.*

Theorem 7.1. *Let P be a rulewise-acyclic Datalog program over some semiring σ, with input size m, active domain size n, and $\mathsf{arity}(P) \leq k$. Then, we can construct a σ-equivalent grounding in time (and has size) $O(n^{k-1} \cdot (m + n^k))$.*

A direct result of the above theorem is that for *monadic* Datalog, where $\mathsf{arity}(P) = 1$, we obtain a grounding of size $O(m)$, which is essentially optimal. The main technical idea behind Theorem 7.1 is to construct the join tree corresponding to each rule, and then decompose the rules following the structure of the join tree.

It turns out that, in analogy to conjunctive query evaluation, we can also get good bounds on the size of the grounding by considering a width measure of tree decompositions of the rules. We show the following theorem in Section 7.5, which relates the grounding size to the maximum *submodular width* (Section 7.5) across all rules.

Theorem 7.2. *Let P be a Datalog program where $\mathsf{arity}(P) \leq k$, subw is the maximum submodular width across all rules of P, and σ is a dioid and suppose the input size is m, and the active domain size is n. Then, we can construct a σ-equivalent grounding in time (and has size) $\widetilde{O}(n^{k-1} \cdot (m^{\mathsf{subw}} + n^{k \cdot \mathsf{subw}}))$.*

The above theorem requires the semiring to be idempotent w.r.t. the $\oplus$ operation (i.e. a dioid). For a general semiring, the best we show is that we can replace subw with the weaker notion of fractional hypertree width (Proposition 7.5). Sections 7.4 and 7.5 show refined constructions that improve the grounding size when the Datalog program has additional structure (e.g., linear rules); we summarize these results in Table 7.1.

Lower Bounds One natural question is: do Theorem 7.1 and 7.2 attain the best possible grounding bounds? This is unlikely to be true for specific Datalog fragments (e.g. linear Datalog has a tighter grounding as shown in Proposition 7.4); however, we can show optimality for a class of programs (its formal proof can be found here [ZDK+24a]).

Theorem 7.3. *Take any integer $k \geq 2$ and any rational number $w \geq 1$ such that $k \cdot w$ is an integer. There exists a (non-linear) Datalog program P over the tropical semiring Trop^+ with $\mathsf{arity}(P) \leq k$, such that:*

1. *$\mathsf{subw}(P) = w$,*
2. *any Trop^+-equivalent grounding has size $\Omega(n^{k-1+kw})$,*
3. *under the min-weight ℓ-Clique hypothesis [LWW18], no algorithm that evaluates P has a runtime of $O(n^{k-1+kw-o(1)})$.*

7.2.2 Grounding Evaluation

Given a grounding G for a Datalog program, we now turn to the problem of computing the (least fixpoint) solution for this grounding. In particular, we study *how fast can we evaluate G under different types of semirings as a function of its size.* Ideally, we would like to have an algorithm that computes the fixpoint using $O(|G|)$ semiring operations; however, this may not be possible in general. In Section 7.3, we will show fast evaluation strategies for two different classes of semirings: (i) semirings of finite rank, and (ii) semirings that are totally ordered and absorptive. Our two main results can be stated as follows.

Theorem 7.4. *We can evaluate a grounding G over any semiring of rank r using $O(r \cdot |G|)$ semiring operations.*

Theorem 7.5. *We can evaluate a grounding G over any absorptive semiring with total order using $O(|G| \log |G|)$ semiring operations.*

Many semirings of practical interest are captured by the above two classes. The Boolean semiring in particular is a semiring of rank 1. Hence, we obtain as a direct corollary the result in [GK04]:

Corollary 7.1. *We can evaluate a grounding G over the Boolean semiring in time $O(|G|)$.*

The *set semiring* $(2^K, \cup, \cap, \emptyset, K)$ for a finite set K has rank $|K|$. In fact, all bounded distributive lattices have constant rank. As another example, the *access control* semiring $(\{P, C, S, T, 0\}, \min, \max, 0, P)$ [FGT08] employs a constant number of security classifications, where P = PUBLIC, C = CONFIDENTIAL, S = SECRET, T = TOP-SECRET and $P \sqsubset C \sqsubset S \sqsubset T \sqsubset 0$ is the total order for levels of clearance. For all these semirings, their grounding can be evaluated in time $O(|G|)$ by Theorem 7.4. The class of absorptive semiring with total order contains Trop^+ that has infinite rank. Hence, Theorem 7.4 cannot be applied, but using Theorem 7.5, we can evaluate G over Trop^+ in time $O(|G| \log |G|)$.

7.2.3 Applications

Finally, we discuss algorithmic implications of our general framework. In particular, we demonstrate that our approach captures as special cases several state-of-the-art algorithms for tasks that can be described in Datalog. Table 7.2 summarizes some of our results that are straightforward applications of our framework.

To highlight some of our results, Dikjstra's algorithm for single-source shortest path is a special case of applying Theorem 7.5 after we compute the grounding of the program. As another example, Yannakakis [Yan90] showed that a binary (i.e. EDB/IDB arities are at most 2) rulewise-acyclic

programs can be evaluated in time $O(n^3)$. By Theorem 7.1 (let $k = 2$), an equivalent grounding of size $O(n^3)$ can be constructed in time $O(n^3)$ for such programs. Then by Corollary 7.1, we can evaluate the original program in $O(n^3)$ time, thus recovering the result of Yannakakis.

If the binary rulewise-acyclic Datalog program is also linear, it can be a evaluated in $O(m \cdot n)$ time [Yan90]. We generalize this result to show that the $O(m \cdot n)$ bound holds for any linear rulewise-acyclic Datalog program with IDB arity at most 2, even if the EDB relations are of higher arity (see Table 7.2, Proposition 7.2).

As another corollary of Theorem 7.2 and Corollary 7.1, monadic Datalog can be evaluated in time $\widetilde{O}(m^{\mathsf{subw}})$. This is surprising in our opinion, since $\widetilde{O}(m^{\mathsf{subw}})$ is the best known runtime for Boolean CQs. Hence, this result tells us that the addition of recursion with unary IDB does not really add to the runtime of evaluation!

7.3 Grounding Evaluation

This section presents algorithms for evaluating a grounding over two types of commonly-used semirings [RMS21b, RMS21a, KNP+22a]. First, Section 7.3.1 presents a procedure that transforms the grounding to one with a more amenable structure, called a *2-canonical grounding*. Then, we present evaluation algorithms for semirings of rank r (Section 7.3.2), and for absoprtive semirings that are totally ordered (Section 7.3.3).

7.3.1 2-canonical Grounding

We say that a grounding G is *2-canonical* if every equation in G is of the form $y = x \oplus z$ or $y = x \otimes z$. As a first step of our evaluation, we first transform the given grounding into a 2-canonical form using the following Lemma 7.1.

Lemma 7.1. *A grounding G can be transformed into a σ-equivalent 2-canonical grounding of size at most $4|G|$ in time $O(|G|)$.*

Proof. The construction works in two steps. In the first step, we rewrite all monomials. Consider a monomial $\boldsymbol{m} = x_1 \otimes x_2 \otimes \dots x_n$. If $n = 1$, then simply let $\boldsymbol{m} = x_1 \otimes \mathbf{1}$. Otherwise ($n \geq 2$), we rewrite $\boldsymbol{m}$ as follows. We introduce $n-1$ new variables $y_1, \dots, y_{n-1}$, replace $\boldsymbol{m}$ with y_{n-1} and add the following equations in the system:

$$y_1 = x_1 \otimes x_2, \quad y_2 = y_1 \otimes x_3, \quad \dots \quad y_{n-1} = y_{n-2} \otimes x_n$$

This transformation increases the size of the system by at most a factor of two. Indeed, the monomial contributes n to $|G|$, and the new equations contribute $1 + 2(n-2) \leq 2n$.

Table 7.2: Summary of runtime results. The $\widetilde{O}$ notation hides polylog factors in m (total input size) and n (size of the active domain). k denotes the arity of the program.

Task	Semiring	Program	Runtime
APSP [Flo62, War62]	Tropical	$T(x_1,x_2) \leftarrow R(x_1,x_2) \oplus \bigoplus_{x_3} T(x_1,x_3) \otimes R(x_3,x_3)$	$\widetilde{O}(m \cdot n)$
Single-source Shortest Path (SSSP) [Dij22]	Tropical	$T(x_1) \leftarrow U(x_1) \oplus \bigoplus_{x_2} T(x_2) \otimes R(x_2,x_1)$	$\widetilde{O}(m)$
CFL reachability [Yan90]	Boolean	chain programs	$O(n^3)$
	Boolean	IDB arity ≤ 2 & rulewise-acyclic	$O(n^3)$
Regular Path Queries All-pairs [BB13, CS21]	Boolean	linear chain programs	$O(m \cdot n)$
CFL-APSP [Val74, Yan90]	Tropical	chain programs	$\widetilde{O}(n^3)$
Monadic acyclic Datalog [GGV02, GK04]	Boolean	monadic rulewise-acyclic	$O(m)$
Monadic Datalog [GGV02, GK04]	Boolean	monadic	$\widetilde{O}(m^{\mathsf{subw}})$
Linear Datalog [IMNP24]	Boolean	IDB arity $\leq k$	$\widetilde{O}(n^{k-1} m^{\mathsf{subw}-1} \cdot (m + n^k))$
Datalog	Boolean	IDB arity $\leq k$	$\widetilde{O}(n^{k-1} \cdot (m^{\mathsf{subw}} + n^{k \cdot \mathsf{subw}}))$

After the first step, we are left with equations that are either a product of two elements, or a sum of the form $x = x_1 \oplus x_2 \oplus \dots x_n$ (w.l.o.g., we can assume no equations of the form $x = y$). Here, we apply the same idea as above, replacing multiplication with addition. This transformation will increase the size of the system by another factor of two using the same argument as before. The equivalence of the new grounding follows from the associativity property of both $\oplus, \otimes$. $\square$

We demonstrate Lemma 7.1 with an example.

Example 7.2. Consider the polynomial equation showed up in Section 7.2: $x_{ab} = e_{ab} \oplus x_{aa} \otimes f_a \otimes e_{ab}$. Its rewriting (via substitutions) following the proof of Lemma 7.1 is:

$$x_{ab} = e_{ab} \oplus y_1, \quad y_1 = x_{aa} \otimes y_2, \quad y_2 = f_a \otimes e_{ab}$$

Here y_1, y_2 are new IDB variables introduced to make the system of polynomial equations 2-canonical.

7.3.2 Finite-rank Semirings

We now present a grounding evaluation algorithm (Algorithm 15) over a semiring of rank r, which is used to prove Theorem 7.4 [ZDK+24a].

Algorithm 15: Grounding evaluation over a rank r semiring

```
   Input  : grounding G
 1 transform G to be 2-canonical via Lemma 7.1
 2 construct a hash table E, such that for every variable x in G, E[x] is the set of all
     equations that contain x in the right-hand side ;  init an empty queue q
 3 foreach EDB variables e in G do  h(e) ← e ;
 4 foreach IDB variables x in G do  h(x) ← 0 ;
 5 foreach EDB variables e in G do  insert e into q ;
 6 while q is not empty do
 7 │  pop x from q
 8 │  forall (y = x ⊛ z) ∈ E[x]  do                          // ⊛ ∈ {⊕,⊗}
 9 │  │  if h(y) ≠ h(x) ⊛ h(z) then
10 │  │  └  h(y) ← h(x) ⊛ h(z) insert y into q
11 return h(·)
```

The key idea of the algorithm is to compute the least fixpoint in a fine-grained way. Instead of updating all equations in every iteration, we will carefully choose a subset of equations to update their left-hand side variables. In particular, at every step we will pick a new variable (Line 7) and then only update the equations that contain this variable[2] (Line 8-10). Because the semiring has rank r, the value of each variable cannot be updated more than r times; hence, we are guaranteed that each equation will be visited only $2 \cdot r$ times. Moreover, since the grounding is 2-canonical,

[2]We assume that look-ups, inserts, and deletes on hash tables cost constant time. In practice, hashing can only achieve amortized constant time. Therefore, whenever we claim constant time for hash operations, we mean amortized constant time.

updating each variable needs only one $\oplus$ or $\otimes$ operation; the latter property would not be possible if we had not previously transformed the grounding into a 2-canonical one.

7.3.3 Absorptive Semirings with Total Order

We present a Dijkstra-style algorithm (Algorithm 16) for grounding evaluation over an absorptive semiring with $\sqsubseteq$ being a total order to prove Theorem 7.5 [ZDK+24a]. It builds upon prior work [RMS21a] by further optimizing the original algorithm to achieve the almost-linear runtime.

Algorithm 16: Grounding evaluation over an absorptive semiring with total order

```
   Input  : a grounding G
1  transform G to be 2-canonical via Lemma 7.1
2  construct a hash table E, such that for every variable x in G, E[x] is the set of all
     equations that contain x in the right-hand side ;  init F ← ∅
3  foreach EDB variables e in G do  h(e) ← e ;
4  foreach IDB variables x in G do  h(x) ← 0 ;
5  init an empty priority queue q of decreasing values w.r.t. ⊒
6  forall IDB variables x = y ⊛ z in G do                          // ⊛ ∈ {⊕,⊗}
7  |  h(x) ← h(y) ⊛ h(z) ; insert x into q of value h(x)
8  while q is non-empty do
9  |  pop a variable x of the max value from q
10 |  F ← F ∪ {x} ;
11 |  forall (y = x ⊛ z) ∈ E[x] ∧ y ∉ F do
12 |  |  if h(y) ≠ h(x) ⊛ h(z) then
13 |  |  |  h(y) ← h(x) ⊛ h(z)
14 |  |  |  insert y into q of value h(y)
15 return h(·)
```

The algorithm follows the same idea as Algorithm 15 by carefully updating only a subset of equations at every step. However there are two key differences. First, while Algorithm 15 is agnostic to the order in which the newly updated variables are propagated to the equations (hence the use of a queue), Algorithm 16 needs to always pick the variable with the current maximum value w.r.t. the total order $\sqsubseteq$. To achieve this, we need to use a priority queue instead of a queue, which is the reason of the additional logarithmic factor in the runtime. The second difference is that once a variable is updated once, it gets "frozen" and never gets updated again (see Lines 10-11). We show

in the detailed proof of correctness [ZDK+24a] that it is safe to do this and still reach the desired fixpoint.

7.4 Grounding of Acyclic Datalog

In this section, we study how to find an efficient grounding (in terms of space usage and time required) of a Datalog program over a semiring σ. First, we introduce *rulewise-acyclicity* of a program using the notion of *tree decompositions*.

Example 7.3. We ground the APSP program (7.1). First, $R(x_1, x_2)$ produces $O(|R|)$ groundings, since it is an EDB. The latter sum-prod query has a $O(|R| \cdot n)$ grounding, since for each tuple $R(x_3, x_2)$, we should consider all active domain of x_1. Hence, the equivalent grounding is of size $O(|R| + |R| \cdot n) = O(m \cdot n)$.

Acyclicity A sum-prod query φ is said to be *acyclic* if its associated hypergraph admits a join tree. The GYO reduction [YO79] is a well-known method to construct a join tree for φ. We say that a rule of a Datalog program is *acyclic* if every sum-prod query in its body is acyclic and a program is *rulewise-acyclic* if it has only acyclic rules.

We formally prove Theorem 7.1, by introducing a grounding algorithm for rulewise-acyclic programs attaining the desired bounds; the pseudocode can be found in [ZDK+24a].

Proof of Theorem 7.1. The σ-equivalent grounding G is constructed rule-wise from P. Take any acyclic rule from P with head $T(\mathbf{x}_H)$. Note that its arity $|H| \leq k$. We ground each sum-prod query $\varphi(\mathbf{x}_H)$ in this rule one by one. To ground a single φ (which is of the form (2.10)), we construct a join tree $(\mathcal{T}, \chi)$ of φ. Each bag in the join tree will correspond to an atom in the body of φ. We root $\mathcal{T}$ from any atom that contains at least one of the variables in $\mathbf{x}_H$ (i.e. head variables), say s_0. This orients the join tree. For any node s, define $(V_s, \mathcal{E}_s)$, where $V_s \subseteq [\ell], \mathcal{E}_s \subseteq \mathcal{E}$, to be the hypergraph constructed by only taking the bags from the subtree of $\mathcal{T}$ rooted at t as hyperedges. Note that $(V_{s_0}, \mathcal{E}_{s_0}) = ([\ell], \mathcal{E})$ is the associated hypergraph of φ. Let $H_s \subseteq H$ be the head variables that occur in this subtree (so at root, $H_{s_0} = H$). Further, rename the head atom $T(\mathbf{x}_H)$ to be $U_{e_{(\perp,s_0)}}(\mathbf{x}_H)$, with $\perp$ being an imaginary parent node of the root s and $e_{(\perp,s_0)} = H$.

Next, we use a recursive method Ground to ground φ. Let r be the parent of s in $\mathcal{T}$. A Ground$(s, U_{e_{(r,s)}})$ call at a node s grounds the sum-prod query

$$U_{e_{(r,s)}}(\mathbf{x}_{e_{(r,s)}}) \leftarrow \bigoplus_{\mathbf{x}_{V_s \setminus e_{(r,s)}}} \bigotimes_{J \in \mathcal{E}_s} T_J(\mathbf{x}_J). \tag{7.4}$$

Hence, calling Ground$(s_0, U_{e_{(\perp,s_0)}})$ at s_0 suffices to ground φ. We describe the procedure Ground$(s, U_{e_{(r,s)}})$, which has three steps:

(i) *Refactor.* Define for each child t of s (i.e. $(s,t) \in E(\mathcal{T})$, $E(\mathcal{T})$ being the directed edges of $\mathcal{T}$), $e_{(s,t)} = (\chi(s) \cap \chi(t)) \cup H_t$ and

$$S = \chi(s) \cup \bigcup_{t:(s,t)\in E(\mathcal{T})} e_{(s,t)}.$$

We refactor as follows: $U_{e_{(r,s)}}(\mathbf{x}_{e_{(r,s)}}) = \bigoplus_{\mathbf{x}_{V_s \setminus e_{(r,s)}}} \bigotimes_{J\in\mathcal{E}_s} T_J(\mathbf{x}_J)$

$$\stackrel{(1)}{=} \bigoplus_{\mathbf{x}_{V_s \setminus e_{(r,s)}}} T_{\chi(s)}(\mathbf{x}_{\chi(s)}) \otimes \bigotimes_{t:(s,t)\in E(\mathcal{T})} \bigotimes_{J\in\mathcal{E}_t} T_J(\mathbf{x}_J)$$

$$\stackrel{(2)}{=} \bigoplus_{\mathbf{x}_{S \setminus e_{(r,s)}}} T_{\chi(s)}(\mathbf{x}_{\chi(s)}) \otimes \bigotimes_{t:(s,t)\in E(\mathcal{T})} \left(\bigoplus_{\mathbf{x}_{V_t \setminus e_{(s,t)}}} \bigotimes_{J\in\mathcal{E}_t} T_J(\mathbf{x}_J) \right)$$

$$\stackrel{(3)}{=} \bigoplus_{\mathbf{x}_{S \setminus e_{(r,s)}}} T_{\chi(s)}(\mathbf{x}_{\chi(s)}) \otimes \bigotimes_{t:(s,t)\in E(\mathcal{T})} U_{e_{(s,t)}}(\mathbf{x}_{e_{(s,t)}})$$

where (1) regroups the product into the node s and subtrees rooted at each child t of s, (2) safely pushes aggregation over every child t (since any $i \in V_t \setminus e_{(s,t)}$ only occurs in the subtree rooted at t, otherwise by the running intersection property, $i \in \chi(s) \cap \chi(t) \subseteq e_{(s,t)}$, a contradiction), and (3) replaces every child sum-prod query by introducing a new IDB $U_{e_{(s,t)}}(\mathbf{x}_{e_{(s,t)}})$ and a corresponding query:

$$U_{e_{(s,t)}}(\mathbf{x}_{e_{(s,t)}}) \leftarrow \bigoplus_{\mathbf{x}_{V_t \setminus e_{(s,t)}}} \bigotimes_{J\in\mathcal{E}_t} T_J(\mathbf{x}_J). \tag{7.5}$$

(ii) *Ground.* Instead of grounding the sum-prod query (7.4) as a whole, we ground the σ-equivalent (but refactored) query

$$U_{e_{(r,s)}}(\mathbf{x}_{e_{(r,s)}}) \leftarrow \bigoplus_{\mathbf{x}_{S \setminus e_{(r,s)}}} T_{\chi(s)}(\mathbf{x}_{\chi(s)}) \otimes \bigotimes_{t:(s,t)\in E(\mathcal{T})} U_{e_{(s,t)}}(\mathbf{x}_{e_{(s,t)}}).$$

Notice that S is exactly the set of variables appear in the body of the refactored query. We ground this query as follows: for each possible tuple (say $\mathbf{a}_{\chi(s)}$) in $T_{\chi(s)}$, we add a grounded rule for each tuple (say $\mathbf{a}_{S\setminus\chi(s)}$) that can be formed over the schema $\mathbf{x}_{S\setminus\chi(s)}$ using the active domain of each variable. This is done by taking the attribute values for each variable from $\mathbf{a}_{\chi(s)}$ and $\mathbf{a}_{S\setminus\chi(s)}$, and substituting it in every predicate (EDB, IDB, or the head) of the query.

(iii) *Recurse.* Now every child t of s has an introduced IDB $U_{e_{(s,t)}}$. For each t, we call into $\textsc{Ground}(t, U_{e_{(s,t)}})$ to recursively ground the new rule (7.5). If there are no children (i.e. s is a leaf node), the $\textsc{Ground}$ call on s terminates immediately.

We argue the grounding size and time at the grounding step (ii), for any node s when grounding the refactored query. Recall that r is the parent node of s. We consider all possible tuples over $\mathbf{x}_S$

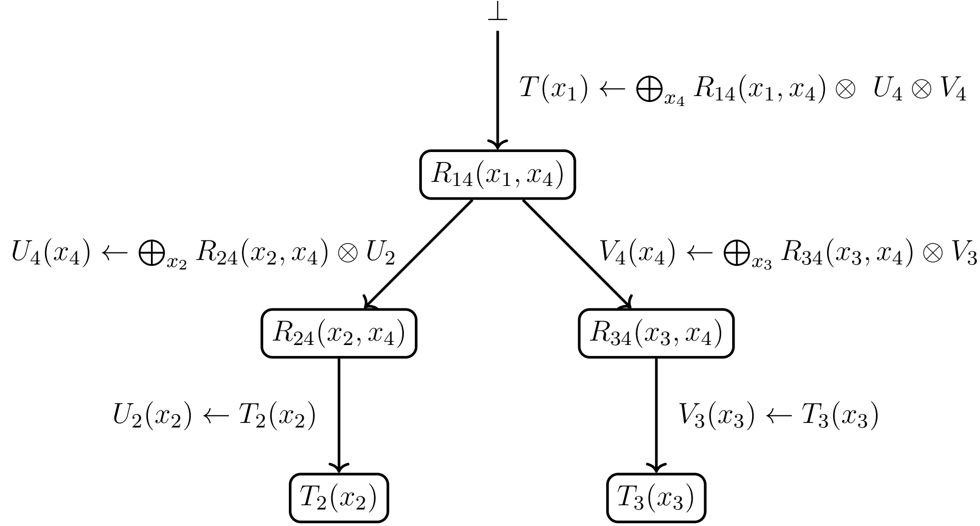


Figure 7.1: A join tree with its corresponding rewriting.

(variables in the body of the refactored query). First, $T_{\chi(s)}$ is of size $O(m)$ if it is an EDB, or $O(n^k)$ if it is an IDB. Next, for the number of tuples that can be constructed over $\mathbf{x}_{S \setminus \chi(s)}$, note that

$$S = \chi(s) \cup \bigcup_{t:(s,t) \in E(\mathcal{T})} e_{(s,t)} = \chi(s) \cup \bigcup_{t:(s,t) \in E(\mathcal{T})} H_t \subseteq \chi(s) \cup H_s$$

where the inclusion holds since $H_t \subseteq H_s$ if t is a child of s. Observe that if $H_s \subsetneq H$, then $|H_s| \leq k-1$; otherwise (i.e., $H_s = H$), since we have rooted the join tree to a node that contains at least one head variable, at least one of the head variables in H_s must belong to $\chi(s)$ (again by the running intersection property), in which case also $|H_s \setminus \chi(s)| \leq k-1$. Thus, $|S \setminus \chi(s)| \leq |(\chi(s) \cup H_s) \setminus \chi(s)| = |H_s \setminus \chi(s)| \leq k-1$. Hence, the groundings inserted for any intermediate rule is of size (and in time) $O(n^{k-1} \cdot (m + n^k))$, i.e. the product of the cardinality of $T_{\chi(s)}$ and the number of possible tuples that can be formed over the schema $\mathbf{x}_{S \setminus \chi(S)}$. □

Example 7.4. We illustrate Theorem 7.1 with the acyclic rule:

$$T(x_1) \leftarrow \bigoplus_{\mathbf{x}_{234}} T_2(x_2) \otimes T_3(x_3) \otimes R_{24}(\mathbf{x}_{24}) \otimes R_{34}(\mathbf{x}_{34}) \otimes R_{14}(\mathbf{x}_{14})$$

A join tree is drawn in Figure 7.1 with new intermediate IDBs and rules (one per edge of the join tree) introduced recursively by the proof of Theorem 7.1. In the transformed program, each rule produces $O(m + n) = O(m)$ groundings. Hence, its overall size is $O(m)$.

We now provide a step-by-step walk-through of how to evaluate the rule by following the tree decomposition in Figure 7.1. For succinctness, we call the nodes in the join tree by their corresponding assigned atoms. Following the rooted join tree in Figure 7.1, we start from calling GROUND at the root node R_{14}, i.e. GROUND$(R_{14}, T(x_1))$:

(*i*) (*Refactor*) The root node R_{14} has two children and thus we introduce two new IDBs $U_4(x_4)$ for the node R_{24} and $V_4(x_4)$ for the node R_{34}. This corresponds to the following *refactoring* of the original rule:

$$\begin{aligned}
T(x_1) &= \bigoplus_{\mathbf{x}_{234}} T_2(x_2) \otimes T_3(x_3) \otimes R_{24}(x_2, x_4) \otimes R_{34}(x_3, x_4) \otimes R_{14}(x_1, x_4) \\
&= \underbrace{\bigoplus_{x_4} R_{14}(x_1, x_4) \otimes \left(\bigoplus_{x_2} R_{24}(x_2, x_4) \otimes T_2(x_2)\right) \otimes \left(\bigoplus_{x_3} R_{34}(x_3, x_4) \otimes T_3(x_3)\right)}_{\textit{(push } \bigoplus \textit{ over two children)}} \\
&= \underbrace{\bigoplus_{x_4} R_{14}(x_1, x_4) \otimes U_4(x_4) \otimes V_4(x_4)}_{\textit{(introduce two new IDBs)}}
\end{aligned}$$

along with two new rules for the intermediate IDBs:

$$\begin{aligned}
U_4(x_4) &\leftarrow \bigoplus_{x_2} R_{24}(x_2, x_4) \otimes T_2(x_2) \\
V_4(x_4) &\leftarrow \bigoplus_{x_3} R_{34}(x_3, x_4) \otimes T_3(x_3).
\end{aligned}$$

(*ii*) (*Ground*) We ground the refactored rule $T(x_1) \leftarrow \bigoplus_{x_4} R_{14}(x_1, x_4) \otimes U_4(x_4) \otimes V_4(x_4)$ (i.e. the top rule shown in Figure 7.1) by taking all tuples in the EDB R_{14} and that naturally subsumes all possible tuples with non-zero annotations (i.e. $R_{14}(x_1, x_4) \otimes U_4(x_4) \otimes V_4(x_4) \neq \mathbf{0}$). Recall that tuples not in the EDB R_{24} has an implicit annotation of $\mathbf{0}$ and thus do not contribute to the sum-product. This gives a grounding of size $O(|R_{14}|) = O(m)$.

(*iii*) (*Recurse*) We recursively call GROUND on the children nodes, i.e. GROUND($R_{24}, U_4(x_4)$) and GROUND($R_{34}, V_4(x_4)$). To avoid repeating, we only demonstrate GROUND($R_{24}, U_4(x_4)$) next.

Again, following the join tree rooted at R_{24}, we call GROUND($R_{24}, U_4(x_4)$):

(*i*) (*Refactor*) The node R_{24} has one child $T_2(x_2)$ and thus we introduce a new IDB $U_2(x_2)$. This corresponds to the following *refactoring* of the original rule:

$$\begin{aligned}
U_4(x_4) &= \bigoplus_{x_2} R_{24}(x_2, x_4) \otimes T_2(x_2) && \textit{(no more variables to push } \bigoplus\textit{)} \\
&= \bigoplus_{x_2} R_{24}(x_2, x_4) \otimes U_2(x_2) && \textit{(introduce a new IDB)}
\end{aligned}$$

along with a new rule for the intermediate IDB:

$$U_2(x_2) \leftarrow T_2(x_2).$$

(*ii*) (*Ground*) We ground the refactored rule

$$U_4(x_4) \leftarrow \bigoplus_{x_2} R_{24}(x_2, x_4) \otimes U_2(x_2)$$

by taking all tuples in the EDB $R_{24}(x_2, x_4)$. This gives a grounding of size $O(|R_{24}|) = O(m)$.

(*iii*) (*Recurse.*) We recursively call $\textsc{Ground}(T_2, U_2(x_2))$ next.

Finally, we call $\textsc{Ground}(T_2, U_2(x_2))$. However, since T_2 is a leaf node, there is no refactoring and further recursion. We simply ground the rule: $U_2(x_2) \leftarrow T_2(x_2)$ by taking all values in the active domain of x_2. This gives a grounding of size $O(n)$ (recall $n \leq m$).

Free-connexity Take a tree decomposition $(\mathcal{T}, \chi)$ of a sum-prod query $\varphi(\mathbf{x}_H)$ rooted at a node $r \in V(\mathcal{T})$. Let $\mathsf{TOP}_r(x)$ be the highest node in $\mathcal{T}$ containing x in its bag. We say that $(\mathcal{T}, \chi)$ is *free-connex w.r.t.* r if for any $x \in H$ and $y \in [\ell] \setminus H$, $\mathsf{TOP}_r(y)$ is not an ancestor of $\mathsf{TOP}_r(x)$ [WY21]. We say that $(\mathcal{T}, \chi)$ is free-connex if it is free-connex w.r.t. some $r \in V(\mathcal{T})$. An acyclic query is free-connex if it admits a free-connex join tree. If every sum-prod query in every rule of a program P admits a free-connex join tree, then P is *rulewise free-connex acyclic*. Theorem 7.1 (see Table 7.1) shows that in such a case, there is a much tighter bound (dropping the n^{k-1} term).

Proposition 7.1. *Let P be a rulewise free-connex acyclic* Datalog *program over some semiring σ with input size m, active domain size n, and* arity(P) *is at most k. Then, a σ-equivalent grounding can be constructed in time (and has size) $O(m + n^k)$.*

Linear Acyclic Programs If arity$(P) \leq 2$, Theorem 7.1 states that a rulewise-acyclic Datalog program admits a grounding of size $O(n^3)$. If the program is also linear (e.g. same generation [BMSU85]), we strengthen the upper bound to $O(m \cdot n)$ (see Table 7.1). The proof can be found in [ZDK+24a] and more examples are included its Appendix.

Proposition 7.2. *Let P be a linear rulewise-acyclic* Datalog *program over some semiring σ with input size m, active domain size n, and* arity(P) *is at most 2. Then, we can construct a σ-equivalent grounding in time (and has size) $O(m \cdot n)$.*

7.5 Grounding of General Datalog

This section introduces an algorithm for grounding any Datalog program over an arbitary dioid via the PANDA algorithm [KNS17] (or PANDA, for short). PANDA is introduced to evaluate CQs (i.e. sum-prod queries over the *Boolean semiring*).

Submodular width A function $f : 2^{[\ell]} \mapsto \mathbb{R}_+$ is a non-negative *set function* on $[\ell]$ ($\ell \geq 1$). The set function is *monotone* if $f(X) \leq f(Y)$ whenever $X \subseteq Y$, and is *submodular* if $f(X \cup Y) + f(X \cap Y) \leq f(X) + f(Y)$ for all $X, Y \subseteq [\ell]$. A non-negative, monotone, submodular set function h such that $h(\emptyset) = 0$ is a *polymatroid*. Let φ be a $\textsf{sum-prod}$ query (2.10). Let Γ_ℓ be the set of all polymatroids h on $[\ell]$ such that $h(J) \leq 1$ for all $J \in \mathcal{E}$. The *submodular width* of φ is

$$\textsf{subw}(\varphi) := \max_{h \in \Gamma_\ell} \min_{(\mathcal{T},\chi) \in \mathcal{F}} \max_{t \in V(\mathcal{T})} h(\chi(t)), \tag{7.6}$$

where $\mathcal{F}$ is the set of all *non-redundant* tree decompositions of φ. A tree decomposition is *non-redundant* if no bag is a subset of another. Abo Khamis et al. [KNS17] proved that non-redundancy ensures that $\mathcal{F}$ is finite, hence the inner minimum is well-defined. We define the *free-connex submodular width* of φ, $\textsf{fsubw}(\varphi)$, by restricting $\mathcal{F}$ to the set of all non-redundant free-connex tree decompositions.

We extend $\textsf{subw}$ and $\textsf{fsubw}$ to $\textsf{Datalog}$. The *submodular width* of a program P (i.e. $\textsf{subw}(P)$) is the maximum $\textsf{subw}(\varphi)$ across all $\textsf{sum-prod}$ queries in P. Likewise, the *free-connex submodular width* of P (i.e. $\textsf{fsubw}(P)$) is the maximum $\textsf{fsubw}(\varphi)$ over all $\textsf{sum-prod}$ queries in P. If P is rulewise-acyclic, $\textsf{subw}(P) = 1$.

This section describes the key ideas of the algorithm underlying Theorem 7.2. We demonstrate the grounding algorithm step-by-step on the following concrete example.

Example 7.5. The following $\textsf{Datalog}$ computes a diamond-pattern reachability starting from some node a set $U(x_1)$,

$$T(x_1) \leftarrow U(x_1) \tag{7.7}$$

$$T(x_1) \leftarrow \bigoplus_{\mathbf{x}_{234}} T(x_3) \otimes R_{32}(\mathbf{x}_{32}) \otimes R_{21}(\mathbf{x}_{21}) \otimes R_{34}(\mathbf{x}_{34}) \otimes R_{41}(\mathbf{x}_{41}) \tag{7.8}$$

A simple (but suboptimal) grounding is to ground the 4-cycle join first, materializing the opposite nodes of the cycle, i.e.

$$\textsf{Cycle}(x_1, x_3) \leftarrow \bigoplus_{\mathbf{x}_{24}} R_{32}(x_3, x_2) \otimes R_{21}(x_2, x_1) \otimes R_{34}(x_3, x_4) \otimes R_{41}(x_4, x_1)$$

$$T(x_1) \leftarrow U(x_1) \oplus \bigoplus_{\mathbf{x}_3} T(x_3) \otimes \textsf{Cycle}(x_1, x_3)$$

$\textsf{PANDA}$ evaluates the first rule (essentially a non-recursive $\textsf{sum-prod}$ query) in time $\widetilde{O}(m^{3/2} + n^2)$, since $\textsf{Cycle}(x_1, x_3)$ has a grounding (or output) of size $O(n^2)$.

We ground the rule in Equation 7.8 that involves a 4-cycle join. In particular, our grounding algorithm constructs a grounding of size $\widetilde{O}(m^{3/2})$ (note the missing n^2 additive factor).

(*i*) *Refactor*. Let $([4], \mathcal{E})$ be the associated hypergraph of the above cyclic recursive rule, where $\mathcal{E} = \{\{3\}, \{3,2\}, \{2,1\}, \{3,4\}, \{4,1\}\}$. We construct the non-redundant tree decompositions $(\mathcal{T}_1, \chi_1), (\mathcal{T}_2, \chi_2)$ (there are only two), both having two nodes $\{v_1, v_2\}$ and one edge $\{(v_1, v_2)\}$, where the bags are

$$\chi_1(v_1) = [3], \chi_1(v_2) = \{3,4,1\};\ \chi_2(v_1) = \{4,1,2\}, \chi_2(v_2) = \{2,3,4\}.$$

This allows us to rewrite and factorize the cyclic body into two acyclic sum-prod queries, one for each decomposition:

$$\begin{aligned}\varphi(x_1) &= \bigoplus_{\mathbf{x}_{234}} T(x_3) \otimes R_{32} \otimes R_{21} \otimes R_{34} \otimes R_{41} \\ &= \bigoplus_{\mathbf{x}_{234}} \underbrace{T \otimes R_{32} \otimes R_{21}}_{B_{[3]}(\mathbf{x}_{[3]})} \otimes \underbrace{R_{34} \otimes R_{41}}_{B_{341}(\mathbf{x}_{341})} \oplus \underbrace{R_{21} \otimes R_{41}}_{B_{412}(\mathbf{x}_{412})} \otimes \underbrace{T \otimes R_{32} \otimes R_{34}}_{B_{234}(\mathbf{x}_{234})}\end{aligned}$$

where the second line uses the idempotence of $\oplus$. The underbraces introduce new IDBs and rules (e.g. $B_{[3]} \leftarrow T \otimes R_{32} \otimes R_{21}$ corresponds to the bag $\chi_1(v_1)$ of $\mathcal{T}_1$ and likewise for B_{341}, B_{234} and B_{412}).

(*ii*) *Grounding Bags*. A naïve grounding of (say) $B_{[3]}$ is of size $O(m \cdot n)$ (cartesian product of R_{32} with domain of x_1), which is suboptimal. Instead, we use PANDA to ground the new IDBs.

Let $Q(\mathbf{x}_{[4]})$ be the set of tuples such that a tuple $\mathbf{a}_{[4]} \in Q$ if and only if its annotation $T(\mathbf{a}_3) \otimes R_{32}(\mathbf{a}_{32}) \otimes R_{21}(\mathbf{a}_{21}) \otimes R_{34}(\mathbf{a}_{34}) \otimes R_{41}(\mathbf{a}_{41}) \neq \mathbf{0}$. Here, $\mathbf{a}_J$ with $J \subseteq [4]$ is a shorthand for the projection of $\mathbf{a}_{[4]}$ onto the variables in J. The PANDA algorithm outputs one table for every new IDB (to differentiate, denote the PANDA output tables as $B^*_{[3]}$, B^*_{341}, B^*_{234} and B^*_{412}) such that: (1) each table is of size $\widetilde{O}(m^{3/2})$ since $\mathsf{subw}(P) = 3/2$; (2) $Q(\mathbf{x}_{[4]})$ is equal to the union of two CQs, one for each decomposition, i.e.

$$(B^*_{[3]}(\mathbf{x}_{[3]}) \bowtie B^*_{341}(\mathbf{x}_{341})) \cup (B^*_{412}(\mathbf{x}_{412}) \bowtie B^*_{234}(\mathbf{x}_{234})) = Q(\mathbf{x}_{[4]})$$

Using PANDA output tables, we ground the four new IDBs, e.g. for each tuple $\mathbf{a}_{[3]} \in B^*_{[3]}$, we add a grounded rule: $B_{[3]}(\mathbf{a}_{[3]}) \leftarrow T(\mathbf{a}_3) \otimes R_{32}(\mathbf{a}_{32}) \otimes R_{21}(\mathbf{a}_{21})$. The same step is applied to all IDBs in total time and size $\widetilde{O}(m^{3/2})$. This ensures that each tuple in $Q(\mathbf{x}_{[4]})$ is present in at least one of $B_{[3]} \otimes B_{341}$ or $B_{412} \otimes B_{234}$, with the correct annotation being preserved, i.e. for any $\mathbf{a}_{[4]} \in Q(\mathbf{x}_{[4]})$,

$$T \otimes R_{32} \otimes R_{21} \otimes R_{34} \otimes R_{41} = (B_{[3]} \otimes B_{341}) \oplus (B_{412} \otimes B_{234}).$$

(*iii*) *Grounding Acyclic Sub-queries*. With the grounded bags, we now rewrite $\varphi(x_1)$ as a sum of two acyclic sum-prod queries:

$$\varphi(x_1) = \left[\bigoplus_{\mathbf{x}_{234}} B_{[3]}(\mathbf{x}_{[3]}) \otimes B_{341}(\mathbf{x}_{341})\right] \oplus \left[\bigoplus_{\mathbf{x}_{234}} B_{412} \otimes B_{234}\right]$$

We ground the two acyclic sub-queries (one per decomposition) using the construction in the proof of Theorem 7.1. For example, if we root both trees at v_1, we get a rewriting:

$$U_{31}(\mathbf{x}_{31}) \leftarrow \bigoplus_{x_4} B_{341} \qquad U_{24}(\mathbf{x}_{24}) \leftarrow \bigoplus_{x_3} B_{234}$$

$$\varphi(x_1) \leftarrow \left[\bigoplus_{\mathbf{x}_{23}} B_{[3]} \otimes U_{31}(\mathbf{x}_{31})\right] \oplus \left[\bigoplus_{\mathbf{x}_{24}} B_{412} \otimes U_{24}(\mathbf{x}_{24})\right]$$

guaranteeing that every intermediate IDB has a grounding of size $\widetilde{O}(m^{3/2})$. Thus, the total size of the grounding for P is $\widetilde{O}(m^{3/2})$.

We show a free-connex version (Proposition 7.3) by restricting the first refactoring step to use only free-connex decompositions. Though $\mathsf{fsubw} \geq \mathsf{subw}$, a n^{k-1} factor is shaved off from the bound.

Proposition 7.3. *Let P be a Datalog program with input size m, active domain size n, $\mathsf{arity}(P) \leq k$. fsubw is its free-connex submodular width. Let σ be a dioid. Then, a σ-equivalent grounding can be constructed in time (and has size) $\widetilde{O}(m^{\mathsf{fsubw}} + n^{k \cdot \mathsf{fsubw}})$.*

Linear Programs For linear programs, a careful analysis yields the following improved result.

Proposition 7.4. *Let P be a linear Datalog program where $\mathsf{arity}(P) \leq k$, input size m, and active domain size n. Let σ be a dioid. Then, a σ-equivalent grounding can be constructed in time (and has size) $\widetilde{O}(n^{k-1} m^{\mathsf{subw}-1} \cdot (m + n^k))$.*

Fractional Hypertree-width So far, all our results only apply to dioids. However, we can extend our results to *any* semiring by using the InsideOut algorithm [AKNR16]. Similar to $\mathsf{subw}(\varphi)$ (7.6), the *fractional hypertree-width* of a sum-prod query φ, i.e. $\mathsf{fhw}(\varphi)$, is defined as

$$\mathsf{fhw}(\varphi) := \min_{(\mathcal{T},\chi) \in \mathcal{F}} \max_{h \in \Gamma_\ell} \max_{t \in V(\mathcal{T})} h(\chi(t))$$

The fractional hypertree-width of a program P is the maximum fhw over all sum-prod queries, denoted as $\mathsf{fhw}(P)$. We show Theorem 7.5 for the grounding size over *any* semiring (see Table 7.1).

Proposition 7.5. *Let P be a Datalog program with input size m, active domain size n, and $\mathsf{arity}(P) \leq k$. Let σ be a naturally-ordered semiring. Then, a σ-equivalent grounding can be constructed in time (and has size) $O(n^{k-1} \cdot (m^{\mathsf{fhw}} + n^{k \cdot \mathsf{fhw}}))$.*

7.6 Related Work

In this section, we present a brief overview of the prior works that are related to our work.

Complexity of Datalog. Data complexity of special fragments of Datalog has been explicitly studied. Gottlob et al. [GK04] showed that monadic acyclic Datalog can be evaluated in linear time w.r.t the size of the program plus the size of the input data. Gottlob et al. [GGV02] defined the fragment *Datalog LITE* as the set of all stratified Datalog queries whose rules are either monadic or guarded. The authors showed that this fragment can also be evaluated in linear time as monadic acyclic Datalog. This result also follows from a generalization of Courcelle's theorem [Cou90] by Flum et al. [FFG02]. Our framework subsumes these results as an application. Lutz et al. [LP22] studied efficient enumeration of *ontology-mediated queries* that are acyclic and free-connex acyclic. The reader may refer to Green at al. [GHLZ13] for an in-depth survey of Datalog rewriting, Datalog with disjunctions, and with integrity constraints. However, there is no principled study on general Datalog programs in parameterized complexity despite its prevalence in the literature of join query evaluation [NRR13, AKNR16, KNS17, Mar10].

Datalog over Semirings. Abo Khamis et al. [KNP+22a] proposed Datalog$^\circ$, an extension of Datalog evaluated over a POPS (short for Partially Ordered Pre-Semirings). POPS subsumes semirings, and in addition, it allows for an imposed partial order other than the natural order. The authors characterize the convergence of Datalog$^\circ$ through grounding and the stability of the underlying POPS. Upper bounds for convergence rate (i.e. the number of Kleene iterations to converge) are also shown. Besides Datalog$^\circ$ [KNP+22a], recent work [IMNP24] has looked at the convergence rate for linear Datalog$^\circ$. The evaluation of Datalog over absorptive semirings with a total order has also been studied [RMS21a] where the key idea is to transform the program (and its input) into a weighted hypergraph and use Knuth's algorithm [Knu77] for evaluation. this chapter recovers and extends this result by formalizing the proof of correctness via the concept of asynchronous Kleene chains, and showing a precise runtime for evaluating a grounding over such semirings. None of the works mentioned above focus on finding the smallest possible grounding, a key ingredient to show the tightest possible bounds.

Datalog Provenance Computation and Circuits. Algorithms for Datalog provenance computation provides an alternative way to think of Datalog evaluation. Deutch et al. [DMRT14] initiated the study of circuits for database provenance. They show that for a Datalog program having $|G|$ groundings for IDBs, a circuit for representing Datalog provenance for the Sorp(X) semiring (absorptive semirings are a special case of Sorp) can be built using only $|G| + 1$ layers. As an example, for APSP, the circuit construction and evaluation cost $O(n^4)$ time. An improvement of the result [Juk15] showed that for APSP, a monotone arithmetic circuit of size $O(n^3)$ can be constructed by mimicking

the dynamic programming nature of the Bellman-Ford algorithm. We use this improvement to show a circuit unconditional lower bound on the grounding size. Ramusat et al. [RMS21b] have also shown that Djikstra's algorithm can be coupled with the semi-naïve evaluation for Datalog provenance.

7.7 Conclusion

This chapter introduces a general two-phased framework that uses the structure of a Datalog program to construct a tight grounding, and then evaluates it using the algebraic properties of the semiring. Our framework successfully recovers state-of-the-art results for popular programs (e.g. chain programs, APSP), and uncovers new results (e.g. for linear Datalog). We also show a matching lower bound for a class of Datalog programs. Future work includes efficient evaluation over broader classes of semirings [KNP+22a], circuit complexity of Datalog over semirings, and general grounding lower bounds.

Chapter 8

Join with Recursion II: A System Design

The rapid expansion of data-intensive applications has underscored the need for query languages that are both simple and expressive. As a declarative language, Datalog adds recursion to relational algebra in a concise syntax, making it especially well-suited for domains such as graph processing [LGS13, FZZ+19], network monitoring [AAHM05], program analysis [SJSW16, SEV16], and distributed systems [CPL+24]. A classic illustration is the graph reachability query, which determines all nodes reachable from a source node a. Its Datalog formulation is:

```
reach(a) :- true.
reach(y) :- reach(x), edge(x, y).
```

Here, `reach(y)` is the accumulating set of nodes `y` reachable from `a`, and `edge(x,y)` is a relation that contains the edges of the graph, from node `x` to `y`. Starting from the base fact `reach(a)`, the program iteratively joins `edge(x,y)` to discover the newly reachable nodes.

In recent years, academic advances and industry demands have spurred the development of Datalog engines. In program analysis, Datalog has been proven powerful for static analysis tasks such as bug detection and security checks. This momentum has led to systems such as Soufflé [SJSW16], Flix [MYL16], and Ascent [SGM22], which are designed from the ground up, often augmented by domain-specific optimizations [SJC+18, JSZS19]. Although these systems excel in their intended applications, they tend to sacrifice flexibility. For example, adding incremental maintenance for continuous updates—highly desirable for rapid prototyping—typically requires major system modifications [ZSRS21]. Similarly, many of these systems were initially designed in single-node architectures; scaling up to accommodate larger datasets frequently demands extensive engineering.

Other systems sidestep the from-scratch approach by layering Datalog functionality atop established databases for out-of-the-box deployment. A prominent example is RecStep [FZZ+19], which compiles Datalog into SQL queries, then delegates its execution to QuickStep [PDZ+18]—a single-node in-memory parallel DBMS—one iteration at a time. RecStep inherits mature database features from QuickStep, such as the cost-based optimizer and parallelism. However, reusing a black-box

DBMS complicates Datalog-specific optimizations, particularly those spanning multiple iterations, such as semi-naïve evaluation and index maintenance. While RecStep interjects between iterations to impose some of these optimizations, the back-and-forth control flow incurs non-negligible synchronization overheads. A similar design underpins DDlog [RB19], an incremental-only Datalog engine that directly translates Datalog into Differential Dataflow[1] (DD) programs. Despite its simplicity, studies have observed that DDlog incurs significant memory overhead–often orders of magnitude higher than alternative approaches [MWC23, ZSRS21, HZJS21, LSZ22].

Balancing flexibility and efficiency for Datalog remains an open challenge. In this paper, we propose a design that unifies (*i*) efficient, off-the-shelf relational operators as execution primitives, and (*ii*) fine-grained controls over optimizations. To achieve the first goal, we use DD's streaming operators as building blocks, positioning DDlog as a baseline that code-generates Datalog into lower-level DD without any optimization layers. However, the second goal of optimizing Datalog systematically is a non-trivial task. Datalog's recursive semantics often render conventional SQL optimizations inapplicable or error-prone. For example, cost-based planning [LGM+15] lacks accurate static statistics in recursive settings. As such, most systems (e.g. Soufflé and DDlog) rely on ad-hoc heuristics or manual tuning. Instead, we introduce an intermediate representation (IR) that separates the logical structure of Datalog from its execution. This decoupling allows us to reason and automate a suite of specialized optimizations on the IR to curb memory usage, a primary bottleneck of DDlog. Furthermore, drawing from database theory (e.g., worst-case optimal joins [NRR13, ZDK+24a]), we build a novel Datalog-tailored optimizer that avoids intermediate materialization blow-ups and is robust against poor join plans. Our effort culminates in FlowLog: a highly efficient Datalog system that targets both batch and incremental processing. Its simple architecture enables developers to integrate new operators, cater to niche workloads, and prototype large-scale recursive queries—all with minimal system overhaul. In summary, this paper makes the following **contributions**.

1. **System Architecture.** Sec. 8.2 presents the system design of FlowLog, centered around an IR that decouples the Datalog logical structure from its physical execution of DD.

2. **Memory-saving Techniques.** FlowLog automates a suite of optimizations, including logic fusion (Sec. 8.3), subplan sharing (Sec. 8.6), and boolean specialization (Sec. 8.7), which reduce the memory overhead of DD's internal states.

3. **Structural Query Optimization.** Sec. 8.4 presents FlowLog's optimizer that leverages only the structural properties of Datalog to avoid worst-case blow-ups in intermediate results.

[1] Differential dataflow [MMII13] programs chain up a set of DD's streaming operators that continuously maintain internal states for efficient incremental computation as data evolves (see Sec. 8.1.3 for a formal introduction.)

Sec. 8.5 complements this with a pre-filtering algorithm that stabilizes the execution of recursive queries against poor join orders. Both techniques draw inspiration from recent advances in database theory on join algorithms.

4. **Extensions.** Sec. 8.8 discusses FlowLog's extensible nature that eases deploying new applications like incremental maintenance, recursive aggregation or scale-out executions.

5. **Experiments.** Sec. 8.9 conducts extensive experiments on a wide set of benchmarks we have collected across multiple domains. Results show that FlowLog often substantially outperforms state-of-the-art Datalog engines, in terms of latency, memory usage, and scalability.

Related Work. As [FZZ+19, KK22] pointed out, most Datalog engines are purpose-built for specific domains [SJSW16, HZJS21, SEV16, MYL16, SGM22, AXR24, HMAO24, SYI+16, SSG+25]. Over time, they have introduced a range of optimizations and extensions tailored to Datalog [HZJS21, GAK12, SSSN24, WWZ22, KNP+22a, ZWF+23, GSS+24], such as incremental Datalog [ZSRS21, RB19]. We incorporate some of these techniques into FlowLog (e.g., index sharing [SJC+18, MLSR20], unified IDB evaluation [FZZ+19]), while others, such as magic sets [WKN+22], de-specialized relations [HZJS21], customized data structures [SBMM23] and worst-case optimal joins [AXR24], remain promising future explorations. Notably, major gaps in the Datalog literature persist, particularly the lack of effective query planning and limited scalability in highly iterative workloads, both being critical challenges for large-scale Datalog applications [AHZ+22, FMK22].

8.1 Background

In this section, we provide a brief overview of standard Datalog, its evaluation, and extensions.

8.1.1 Datalog Basics

A standard Datalog program [AHV95] is a set of *rules*. A rule is an expression of the following form:

```
h :- p1, p2, ..., pk.
```

The terms $h, p_1, \ldots, p_k$ are atoms, i.e., formulas of the form $R(x, y, \ldots)$, where R is the atom (or relation) name and $(x, y, \ldots)$ is its variables (or attributes). The atom h is the *head* and the atoms $p_1, \ldots, p_k$ are the *body* of the rule. A rule can be interpreted as a logical implication: if $p_1, \ldots, p_k$ are true, then so is the head h. We assume that every attribute of h occurs in some p_i The atoms of a Datalog program are of two types: IDB and EDB. An atom that represents an input relation is an EDB (extensional database); an EDB comprises a set of (base) facts/tuples and is never the head of

a rule. A atom that represents a derived relation is an IDB (intensional database, in **bold** font); an IDB must appear in the head of at least one rule.

Example 8.1. Consider the following task over a directed graph: find all nodes that can reach a target via an even number of hops [ZKD24]. We represent the graph as a binary EDB edge(x, y), where (x, y) is a fact if there is an edge from node x to node y. We use another EDB target(x) as the unary relation containing the target node a. The task can then be expressed in Datalog as follows:

```
r1.  reach(x) :- target(x).
r2.  reach(x) :- edge(x,y), edge(y,z), reach(z).
```

Here, the atom reach(x) is the IDB to output. r_1 initializes the trivial case: the target node is reachable in zero hop. The second rule r_2 is recursive and states that if there is a length-2 path from x to z, and z can reach the target using an even number of hops, then so can x.

Dependency Graph and Stratification. A *dependency graph* of a Datalog program is a directed graph where every rule is a node and there is an edge from rule r_1 to r_2 if the head of r_1 occurs in the body of r_2. A rule is *recursive* if it belongs to a directed cycle, and *non-recursive* otherwise. A *stratification* of a Datalog program is a partition of the rules into strata, where each stratum is the set of rules that belongs to the same strongly connected component of the dependency graph. The topological ordering of the strongly connected components defines the ordering among the strata. The dependency graph for Example 8.1 has two nodes r_1 and r_2, and two edges $r_1 \to r_2$ and $r_2 \to r_2$. Thus, the program has two strata, $\{r_1\}$ and $\{r_2\}$ in the topological order.

Common Datalog Extensions. To enrich Datalog for practical usage, we incorporate common syntactic extensions as [FZZ+19]–constraints, stratified negations (where negated atoms are either EDB or an IDB from a lower stratum), and (possibly recursive) aggregations. For example, this allows finding two hops (x, z) in a graph that are:

```
edge(x,y), edge(y,z), x ≠ z       // not loops
edge(x,y), edge(y,z), ¬edge(x,z) // not one hop
```

or for each x and z, counting the number of possible two hops:

```
two_hops(x,z,COUNT(y)) :- edge(x,y), edge(y,z).
```

8.1.2 Datalog Evaluation

The straightforward way to implement Datalog is via *naïve* evaluation. Starting with the set of all EDB facts, we iteratively apply every rule as a join query to derive new facts, adding them to the head IDBs until no new facts can be derived, i.e., a fixpoint is reached.

Although simple, naïve evaluation is usually wasteful because each iteration executes rules on all historical data, leading to rediscovery of facts derived in previous iterations. Hence, modern implementations of Datalog typically use *semi-naïve* evaluation, which only uses new tuples from the last iteration to derive facts in the current iteration. A standard practice is to further exploit stratification: each stratum gets evaluated in order, and the results are used as input for the next stratum. In Example 8.1, the first stratum simply inserts the target node t to $\mathsf{reach}(x)$. Then, at each iteration $i = 1, 2, \ldots$, we only consider the new facts derived in the $(i-1)$-th iteration, denoted as $\Delta\mathsf{reach}^i$ where $\Delta\mathsf{reach}^0 = \{\mathsf{a}\}$, to populate new facts by computing the join $\mathsf{edge}(x, y) \bowtie \mathsf{edge}(y, z) \bowtie \Delta\mathsf{reach}^{(i-1)}$.

8.1.3 Differential Dataflow

Differential Dataflow (or DD) [MMII13] is a data-parallel programming model for large-scale incremental data processing. Its Rust implementation[2] compiles down to Timely Dataflow, a lower-level generic distributed streaming system introduced by [MMI+13].

Collections. DD abstracts a relation as a stream of rows, termed as a collection. A row in a collection is a triple $(\mathtt{data}, \mathtt{time}, \mathtt{diff})$, where `data` is the raw tuple from the relation, `time` is the timestamp when the tuple is ingested, and `diff` is its multiplicity. The `diff` field is for DD to annotate duplications and track incremental changes (i.e., $+\delta$ represents an insertion of δ copies of the tuple and $-\delta$ represents a retraction of δ copies). In Example 8.1, DD constructs corresponding input collections for $\mathsf{target}(x)$ and $\mathsf{edge}(x, y)$, where the former is initialized as a single row $(\mathsf{a}, 0, 1)$ for the target a; the latter is a collection of rows $((\mathsf{a}, \mathsf{b}), 0, 1)$ for each edge (a, b) of the graph. Here, 0 is the initial timestamp.

Differential Operators. DD uses a set of incremental operators such as `map`, `filter`, and `join`, that each imposes a relational operation on input collection(s) and outputs a collection. We can compose them to compute SQL queries and quickly respond to input changes. At its core, every differential operator maintains a minimal set of changes to the output when the input changes, and propagates these updates in a semi-naïve manner – only considering changes since the last timestamp. Let $R(x, y)$ and $S(x, z)$ be two collections. Suppose $((\mathsf{a}, \mathsf{b}), 0, 4)$ and $((\mathsf{c}, \mathsf{b}), 0, 1)$ are two rows of $R(x, y)$ and $((\mathsf{a}, \mathsf{d}), 0, 3)$ is a row of $S(x, z)$. A rule `T(y,z) :- R(x,y), S(x,z)` maps to a composition of `join` and `map` as follows:

```
R.join(&S).map(|t| (t.y, t.z));
```

[2] https://github.com/TimelyDataflow/differential-dataflow

Here, join emits a row $((\mathsf{a}, (\mathsf{b}, \mathsf{d})), 0, 12)$, where the output data is formatted as $(x, (y, z))$ (i.e. join keys followed by a grouping of values) and diff is the product of two input diffs, i.e., $4 \times 3 = 12$. The downstream map projects to (y, z) and gets $((\mathsf{b}, \mathsf{d}), 0, 12)$. If there is an insertion of $((\mathsf{a}, \mathsf{b}), 0, +2)$ in the collection, only a delta change $((\mathsf{b}, \mathsf{d}), 0, +2 \times 3 = +6)$ will propagate through.

DD allows users to define custom operators (beyond standard SQL) without worrying about low-level incremental mechanisms. This makes DD a powerful backend for Datalog applications. A unique but essential operator is iterate, which repeatedly applies an enclosed DD closure to input collections. The following code snippet shows how to implement Example 8.1 as a DD program.

```
// (1) iterate starting from target
target.iterate(|reach| {
    // (2) derive new reach(x) from joins
    edge.map(|t| (t.y, t.x))
        .join(&edge)
        .map(|t| (t.z, t.x))
        .join(&reach)
        .map(|t| t.x)
        // (3) union and dedup
        .concat(&reach)
        .distinct()             });
```

The iterate operator initializes reach as $(\mathsf{a}, (0, 0), 1)$, i.e. target, and sets a series of nested timestamps $(0, i)$, where 0 is the outer timestamp and $i = 0, 1, \ldots$ is the iteration counter. Then, it repeatedly applies the inner DD logic as i increments–for each i, the concat operator adds new output into reach and distinct de-duplicates the results, i.e. for each row, diff maps to 1 if diff > 1. When no new rows are derived (i.e. fixpoint), iterate collects all results and returns the final reach to the outer scope/timestamp 0.

The inner logic is verbose but necessary. An idiosyncratic feature of DD is the use of map and join; map is not only a projection, but also a way to re-organize data into a key-value pair, e.g., the first map swaps (x, y) to (y, x), and that designates y as the key and x as the value. This is because the join operator of DD requires its two input collections to be explicitly pre-indexed on the join keys, using an arrange operator described next.

Arrangements. An arrangement is an in-memory index for DD collections [MLSR20]. It can be considered as a sorted dictionary that allows efficient concurrent access. An arrangement indexes batches of historical changes, maintains them over time, and merges them into compact representations as appropriate (e.g., when a timestamp is advanced). The first join in the above code pre-arranges both operands by imposing an arrange operation internally for each and then uses a

more primitive join_core operator to join on arrangements. Indeed, the first join is executed under the hood as

```
edge.map(|t| (t.y, t.x))
    .arrange()          // k: (y), v: (x)
    .join_core(
        &edge.arrange() // k: (y), v: (z)
    ); // output schema k: (y), v: (x, z)
```

8.2 System Design

In this section, we present the architecture of FlowLog. Like DDlog, FlowLog leverages DD as its execution backend. However, rather than compiling Datalog directly into a DD program, FlowLog first translates the rules into a more amenable intermediate representation (IR) that represents the logical structure of the input program, before compiling into DD. This design choice enables us to reason about optimizations only at the logical level by hiding away the low-level intricacies of DD, such as nested scopes, arrangements, and de-duplications, making it simpler to modularize and extend. This disentanglement is analogous to the idea of *grounding*, which was used to derive better theoretical guarantees for the runtime of Datalog evaluation [KNP+22a, ZDK+24a]. We will incorporate some of these theoretical ideas when optimizing plans (see Section 8.4).

Overview. The architecture of FlowLog consists of three main components: the front-end, the optimizer, and the executor. The **front-end** parses the input program from a `.dl` file, following the popular grammar of Soufflé, checks for syntax, and then stratifies the rules and creates a per-rule catalog for meta information, such as the join graphs (formally defined in Section 8.4.2). Next, **the optimizer** populates an IR from the catalog and applies a set of novel optimizations. Finally, **the executor** renders the optimized IR handed from the optimizer into a full-fledged DD program, reads the input data, and spins up the iterative execution.

Query Optimization (IR). The IR used in FlowLog is a directed acyclic graph where each node is a logical transformation, such as `Join`, `Map`, and `Filter`. The edges between nodes represent the data that flows between transformations and its label is the data schema. A `Join` must take two inputs with the join keys aligned, as it eases us to translates the IR into the physical DD program.

Figure 8.2a shows a full IR for Example 8.1. The lowest `Join` transformation binds both inputs on the variable y and sends the result of schema (y, x, z) upstream. The IR is now much more concise and readable, almost like a relational algebra plan. We will drop the edge labels if the schema is obvious from the context.

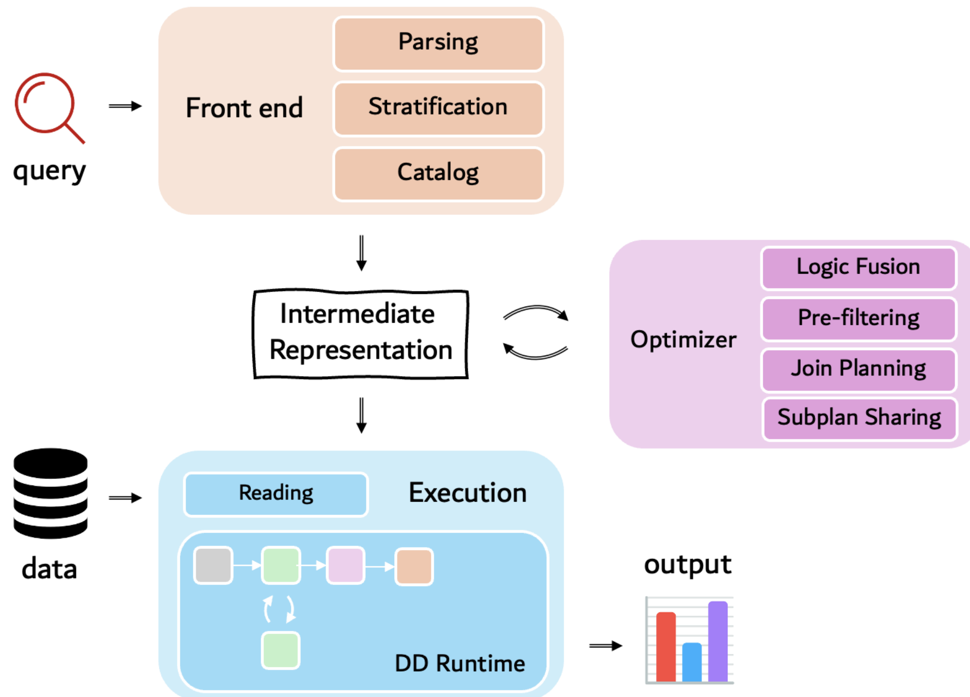


Figure 8.1: System Architecture of FlowLog

Query Execution. FlowLog inherits the properties of DD automatically by using it as the execution layer—semi-naïve execution is built-in, and differential operators are highly efficient, parallel (in fact, asynchronous), and naturally incremental. However, as exhibited by DDlog against other systems such as Soufflé, a key side-effect is the memory overhead of maintaining the incremental states, which can be prohibitive when the intermediate results are large. In the next sections, we introduce a suite of optimizations that FlowLog applies to the IR. These optimizations are geared towards minimizing the intermediate memory footprint, thus making FlowLog much more competitive for large-scale recursive queries.

8.3 Logic Fusion

Logic fusion is a query optimization technique that merges multiple small and adjacent logic (of frequent occurrence) into a single operator to reduce dispatches during interpretation. For systems built atop DD, logic fusion can also reduce the intermediate memory usage. We will use common logic fusion in our IR extensively.

(a) Initial IR (a DAG that represents the logical plans of the rules)

(b) Fused IR (using `Join-FlatMap` and `UnionAll`)

Figure 8.2: Logic Fusion for Example 8.1

Unary Transformations. We introduce a new `FlatMap` transformation into the IR that fuses consecutive `Map` and `Filter` (similar fusions are applied to optimize incremental SQL[3]). A `FlatMap` translates into a lower-level `flat_map` operator of DD that filters and projects tuples to the desired schema in one pass. For example, the following rule and code snippet finds the neighbors of node `a`:

```
// neighbor(y) :- edge(x, y), x = a.
let neighbor = edge.flat_map(|t|
  if t.x == a { vec![(t.y)] } else { vec![] } );
```

[3] https://materialize.com/blog/generalizing-linear-operators/#fusing-logic

If in addition the `FlatMap` (or `Map`) is an identity transformation, we remove it from the IR, such as the `Map` for target(x) in Figure 8.2a.

Binary Transformations. We introduce two fused binary transformations to the IR: `Join-FlatMap` and `UnionAll`.

A `Join-FlatMap` fuses a `Join` with a subsequent `Map` and `Filter` (if any). It thus avoids materializing the full join output that is later projected or filtered. It is heavily used across our IR. The initial IR of r_2 can be optimized by fusing any consecutive `Join` and `Map`. In the fused IR (see Figure 8.2b), the lowest `Join-FlatMap` directly emits (z, x) tuples. At the execution layer, `Join-FlatMap` maps to the `join_core` operator in the DD library, which works as follows.

```
edge.join_core(&edge, |t|
    if some filters are passed // if any
        vec![(t.z, t.x)] // map to (z, x)
    else vec![]  );      // filtered out
```

A `Union` transformation merges two input into one. However, in the context of Datalog, if there are multiple rules deriving the same IDB, instead of merging two at a time and maintaining every intermediate `Union`, we fuse them into a single yet multi-way `UnionAll` transformation to merge all inputs at once. It compiles down to a customized `union_all` operator. This is analogous to the unified IDB evaluation optimization proposed in RecStep [FZZ+19].

8.4 Join Plan Optimization

Cost-based join plan optimization is a standard practice in DBMS to minimize intermediate results. Traditional optimizers estimate costs using approximated statistics (e.g., cardinalities, selectivities) and select the cheapest plan in a prescribed search space. However, optimizing join plans for Datalog is much more strenuous as these statistics are often unavailable or difficult to estimate: IDB (s) are only populated at runtime, and the delta sizes of IDBs vary across iterations. On top of that, industrial-strength applications such as DOOP [BS09] and DDISASM [FS20] usually involve highly complex join topologies (e.g. cyclic joins) than those typically handled by traditional DBMS. Consequently, systems such as Soufflé and DDlog use join orders hard-coded by the user. That is, if a user writes a rule with body p_1, p_2, p_3, the system executes the joins in that order even if there are cross products, i.e. $(p_1 \bowtie p_2) \bowtie p_3$. On the other hand, RecStep collects runtime statistics periodically and leverages the underlying DBMS's optimizer to re-optimize join plans on the fly, paying a synchronization and catalog maintenance overhead.

FlowLog introduces a novel lightweight optimizer that finds a static join plan for every rule containing multi-way joins. Rather than relying on runtime statistics, our approach analyzes the

join topology of a rule using a hybrid of heuristics and worst-case analysis [ZDK+24a]. Despite its simplicity, our experiments show that this optimizer effectively avoids asymptotically poor join orders. It lays the foundation for designing more refined cost models for Datalog, potentially considering factors such as EDB statistics in the future.

8.4.1 Structural Cost Model

Since high-quality statistics are typically unattainable for Datalog, FlowLog adopts a structural cost model that uses the distinct number of participating variables as a proxy for the asymptotic cost of a transformation. For example, in Figure 8.2b, the lower `Join-FlatMap` involves x, y, z variables and hence bears a cost of 3, while the `Join-FlatMap` above involves z, x and is assigned a cost of 2.

The overall cost of a join plan is defined as the maximum cost of any transformation in the plan. The cost of the plan in Figure 8.2b is 3, dominated by the lowest `Join-FlatMap`. Intuitively, for an input graph with N nodes, if using this plan, the worst-case intermediate size (and so is the runtime) of the program in Example 8.1 can be bounded as $O(N^3)$. Formal theorems can be found in [ZDK+24a]. Similar techniques are proposed for subgraph pattern matching [MPPV04] and have shown effectiveness for multi-way many-to-many joins.

8.4.2 Search Strategy

We establish the necessary definitions to describe the search space of join plans. Our optimizer picks the plan in the search space that minimizes the cost according to the structural cost model.

A **(weighted) join graph** of a rule is a graph where every node is an atom and an edge exists if the two atoms join on at least one variable, and its weight is the number of variables they join on. We only consider inner joins for now, and simply push down semijoins, antijoins and filters as deep as possible when the IR is finally constructed from the decided join order.

A **join tree** [Yan81] is a spanning tree of the join graph such that for every variable x, the atoms containing x (in its schema) induce a connected subtree of the spanning tree. The join tree is then used to define acyclicity of a rule, i.e. a rule is acyclic if and only if there is a join tree for it. Zhao et al. [ZDK+24a] showed that for acyclic rules, bottom-up join orders (rooted arbitrarily) along a join tree, after early projections, usually yield tight intermediate size bounds.

A **join spanning tree** (JST) extends this notion to cyclic rules. A JST is a maximum spanning tree of the weighted join graph and reduces to a join tree when the rule is acyclic [Mai83]. A **rooted JST** defines a join-project plan by post-order traversal: at each step, an atom is joined with its parent, followed by projecting out variables no longer needed. The only JST (also a join graph or a join tree) for Example 8.1 is shown in Figure 8.3. Here we root the JST at edge(x, y), and it maps

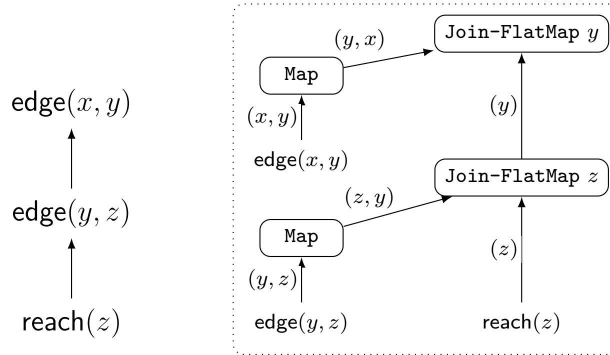


Figure 8.3: A rooted JST for Example 8.1 (left) and its translated IR (right) from a post order traversal of the rooted JST.

into the IR on the right, i.e. a bottom-up join order. If we root the JST at $\mathsf{reach}(z)$ instead, we recover the IR in Figure 8.2b.

Search Space. Our search space excludes semijoins (atoms whose variables are subsumed by another atom), antijoins and filters, as they are pushed down to the lowest possible transformation when the IR is constructed from the selected (optimized) join plan. This narrows down to a multi-way inner join where the search space is defined as all rooted JSTs of its join graph. There are three main reasons for such a choice: (1) JSTs naturally avoid cross products when possible, since cross products correspond to zero-weight edges in the join graph, (2) this space is reasonably small, and thus tractable and conservative, and (3) the search space collapses down to rooted join trees if the rule is acyclic, for which we can provably show tight bounds on intermediate sizes [ZDK+24a]. In our running example (which is acyclic), Figure 8.3 will be selected against Figure 8.2b as it has a cost of 2, instead of 3 in our cost model. Intuitively, it avoids computing $\mathsf{edge}(x, y) \bowtie \mathsf{edge}(y, z)$ by using two semi-joins.

Next, we show a more involved example and demonstrate that JSTs can optimize join orders for rules with cyclic multi-way joins.

Example 8.2. Consider an expensive rule in the DOOP program analysis framework [BS09], which contains recursive 8-way joins.

```
VarPointsTo(heap, to) :-
      Reach(inm),
      LoadArrayIdx(base, to, inm),
      VarPointsTo(bh, base),
      ArrayIndexPointsTo(bh, heap),
      VarType(to, tp),
```

```
    HeapAllocationType(bh, bht),
    ComponentType(bht, bct),
    SupertypeOf(tp, bct).
```

where `VarType`, `HeapAllocationType`, `ComponentType` are EDBs; others (in bold) are IDBs. Figure 8.4 displays its join graph. `Reach(inm)` is a semijoin subsumed by `LoadArrayIdx` and `inm` is projected away after the semijoin. When constructing the final IR, the semijoin is pushed down to the read of `LoadArrayIdx`. The JST selected by the optimizer is rooted at `LoadArrayIdx` and is highlighted on top of the join graph using thick, directed edges. The dotted edges are edges in the join graph, but are not part of the selected JST.

We now discuss the plans for Example 8.2. Soufflé uses the given listing order: it joins `Reach` with `LoadArrayIdx`, then the result joins with `VarPointsTo`, and so on. The listing order here has been manually tuned for practical efficiency. In our cost model, the cost of this listing order is 5, dominated by the 5th join with `HeapAllocationType`. Indeed, when we finish the first four joins up to `VarType`, the resulting schema, projecting away unnecessay variables, is (`bh`, `tp`, `heap`, `to`), where `bh` and `tp` are join keys for the subsequent joins with `HeapAllocationType` and `SupertypeOf` respectively, whereas (`heap`, `to`) are the desired output variables. The next join with `HeapAllocationType` involves 5 variables, i.e. `bh`, `tp`, `heap`, `to` and `bht`. By contrast, the rooted JST FlowLog selected in Figure 8.4(left) translates to the IR (right), where we use `Jn` as a shorthand for `Join-FlatMap`. The leaf `LoadArrayIdx` $\ltimes$ `Reach` results from semijoin pushdown. The optimize favors this plan as it has a cost of 3–no transformation involves more than 3 distinct variables–which is much lower than the listing order.

8.4.3 Plan Execution

Left-deep Plans. Join planning is tied closely to the underlying execution. The listing order Soufflé, DDlog and others used is equivalent to a left-deep join plan in a DBMS and is incapable of representing bushy plans, such as Figure 8.4. This is a necessary choice for Soufflé as it always compiles the listing order down to an indexed nested loop join, where indexes (e.g. B-trees) are built on every atom except the first one and it iterates over each tuple of the first atom (as the outer loop) and probes into the indexes of the rest. Left-deep plans allow pipelining without intermediate materialization, which makes Soufflé highly memory-efficient. However, scaling indexed nested loop joins to a multi-threaded setting is non-trivial and modern Datalog systems such as Soufflé and Flan [AXR24] only distribute the outer for-loop among threads. As shown in our experiments, this level of parallelism is insufficient to saturate resources even for simple recursive queries like reachability, under data skew.

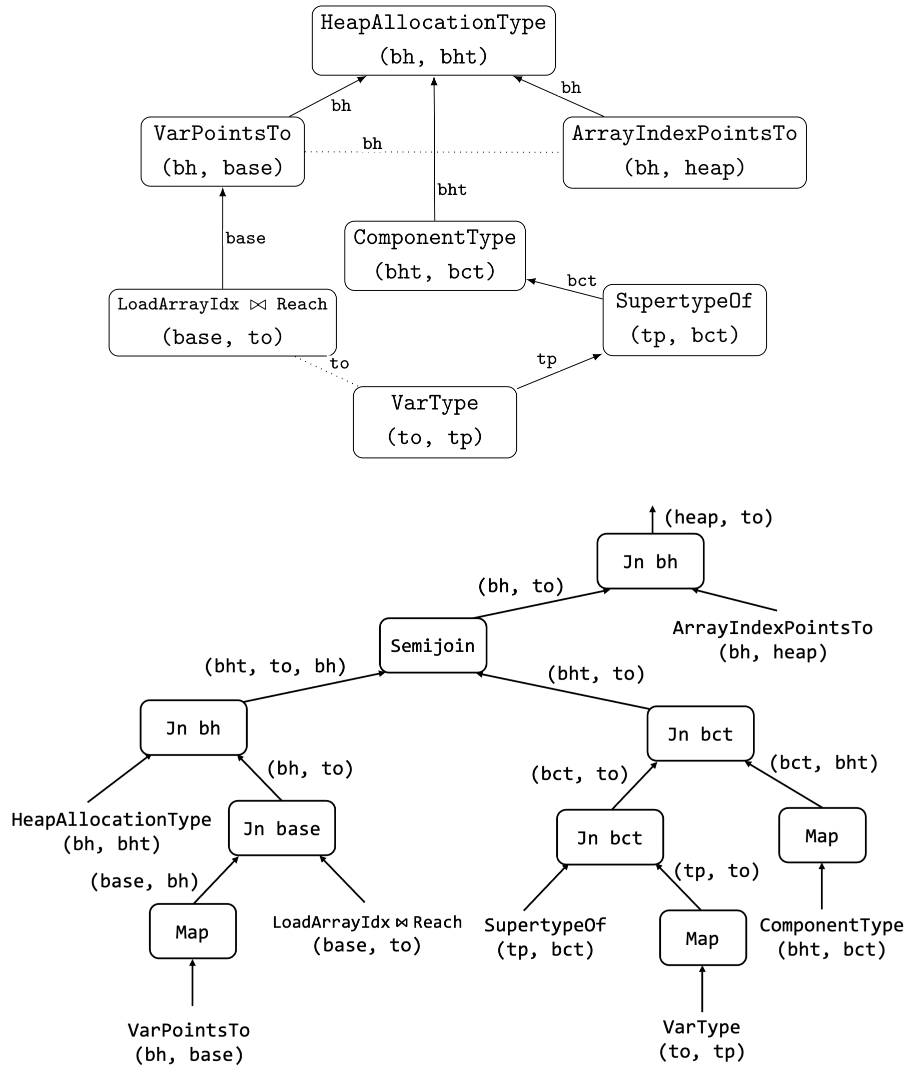


Figure 8.4: The cyclic join graph and a rooted JST (highlighted via directed edges) of Example 8.2 (top) and its translated IR (bottom)

Bushy Plans. FlowLog, in contrast, utilizes DD, which employs a fundamentally different execution model. Differential operators such as `join`, `arrange` are inherently stateful and hence intermediate materialization occurs by design. However, a key advantage is that DD is asynchronous: changes propagate through differential operators and are reacted concurrently, without waiting for previous stages to complete. Therefore, DD naturally exploits parallelism across multiple cores, as computation is dynamically scheduled based on available updates rather than a predetermined execution sequence. Given this execution model, FlowLog can take advantage of bushy join plans without worrying about compromising parallelism. To control memory usage and avoid excessive blow-ups, FlowLog leverages

insights from database theory (e.g. tree decompositions [AKNR16, KNS17, ZDK+24a], worst-case optimal joins [NRR13, WWS23], etc.) to select join plans that have smallest possible (worst-case) intermediate sizes. Based on these observations, FlowLog favors *bushy*, rooted JSTs (measured by tree depth) whenever multiple candidate join plans have the same estimated cost under its cost model.

8.5 Making Datalog Robust

Unlike most SQL queries, Datalog exhibits high sensitivity to data distribution, mutual recursion, and join plans. The volatility makes standard query optimization techniques error-prone. When a suboptimal plan is chosen for a rule, the iterative nature of Datalog can further exacerbate the slowdown, often stalling execution altogether. In contrast, there has been a growing interest in techniques that makes SQL queries robust, such as sideways information passing (SIP) [IT08, ZPSP17], predicate transfer [YZYK24], and diamond hardened joins [BKN24]. These approaches advocate a more pessimistic stance, prioritizing resilience against worst-case scenarios to ensure stable performance. However, robustness techniques are largely missing in Datalog systems. Integrating such techniques into FlowLog is a first step toward our vision: making Datalog execution robust and predictable. In fact, for a simple $R \bowtie S$, DD's `join` already offers robustness as it runs a specialized hash-join that balances both input streams (i.e. no build-probe asymmetry) [MLSR20]. For multi-way joins, we use a SIP-style algorithm to stabilize its iterative execution.

SIP-style Algorithms. The key idea of SIP-style algorithms is to pre-filter dangling tuples before the actual joins. Dangling tuples are tuples that do not participate in the final output. The Yannakakis algorithm [Yan81] is a classic example. It requires the join graph to be acyclic to be able to construct a join tree. Following the post-order traversal over the join tree, it applies two sequence of semijoins to pre-filter base tables: (1) a bottom-up pass that semijoins the child atoms with the parent, followed by (2) a top-down pass that uses the reduced parent atoms to further semijoin-reduce the children. The semijoins provably prune all dangling tuples and the subsequent joins exhibit robustness against bad join orders in practice [ZPSP17, YZYK24].

Many expensive Datalog rules, however, involve cyclic join graphs, where no join tree exists. Inspired by Yannakakis, we apply a similar two-pass semijoin reduction in FlowLog, but for arbitrary join graphs. Our approach is as follows. We pick any atom in the join graph to start a breadth-first search (BFS). As we visit an atom, we semijoin-reduce it using its already-visited neighboring atoms, on the join keys. We conclude the first pass when all atoms are visited. Then we traverse the join graph in the reverse order of the first pass with another round of semijoin reduction (i.e. the second pass).

Example 8.3 (Galen)**.** Consider the following program from [LT14] that describes an inference task in medical ontologies:

```
r1.  p(x,z) :- p(x,y), p(y,z).
r2.  p(x,z) :- p(y,w), u(w,r,z), q(x,r,y).
r3.  p(x,z) :- c(y,w,z), p(x,w), p(x,y).
r4.  q(x,r,z) :- p(x,y), q(y,r,z).
r5.  q(x,u,z) :- q(x,r,z), s(r,u).
r6.  q(x,e,o) :- q(x,y,z), r(y,u,e), q(z,u,o).
```

where the atoms `u`, `c`, and `s` are EDBs. The listing orders here are tuned manually on the given dataset [LT14]. Among them, r_2, r_3, r_6 dominate the runtime and they are highly susceptible to bad join orders. The join plan optimization in Sec. 8.4 can effectively handle r_2 – it avoids joining `u` and `q` upfront (due to a high cost of 6 in our cost model). However, the optimal join orders for r_3 and r_6 are obscure (note that p, q are IDBs in a mutual recursion). Take r_3 for example, it has a triangular join graph and all join orders are indistinguishable under our cost model (i.e. cost of 4) and yet the chosen order is an order of magnitude faster than, say, `c(y,w,z), p(x,y), p(x,w)` (we call it the bad listing order henceforth).

We next show how our `SIP` algorithm is applied to r_3 in Example 8.3. We implement it via Datalog rule rewriting. Without loss of generality, we pick `c` as the starting point and BFS visits the atoms in the bad listing order. Then the rewritten `SIP` rules for r_3 are the following (underscores are placeholders for unused variables):

```
// (1) first pass c(y,w,z) -> p(x,w) -> p(x,y)
p1(x,y) :- p(x,y), c(y,_,_).
p2(x,w) :- p(x,w), c(_,w,_), p1(x,_).

// (2) 2nd pass c(y,w,z) <- p1(x,w) <- p2(x,y)
p3(x,y) :- p1(x,y), p2(x,_).
c4(y,w,z) :- c(y,w,z), p2(_,w), p3(_,y).

// (3) reduced join for r3
r3'.  p(x,z) :- c4(y,w,z), p3(x,y), p2(x,w).
```

The rewriting is semantically equivalent to r_3, but it is more robust against poor join orders. In most iterations, the bad listing order incurs a substantial blow-up when joining `c` and `p(x,y)`, but the final output shrinks significantly at the last join with `p(x,w)`. The rewriting, without such runtime knowledge, passes the selective semijoins of `p2(x,w)` to the first two atoms and reduces them into

p3, c4 in the second pass. Even though r'_3 uses the same join order, c4 $\bowtie$ p3 is now drastically smaller to compute and maintain. Similar techniques can be applied to r_6 to stabilize its execution.

Discussion Robust planning usually has overheads, such as extra probing and maintaining of intermediate indexes for semijoins. In the extreme of no dangling tuples, we pay nearly 2× more memory and CPU resources (linear to input sizes). However, the overhead is outweighed when a significant number of dangling tuples are pruned. Techniques like predicate transfer [YZYK24] further mitigate the semijoin costs leveraging Bloom filters. For FlowLog, we keep it simple and offer `SIP` as an option that can be turned on/off by users by adding a `.sip` directive beneath rules. When turned on, FlowLog prefilters input tables before running the optimized join plans from Sec. 8.4. We show in our experiments (Sec. 8.9.4) that both techiques are often complementary in that prefiltering leaves the join planner with a closer to worst-case scenario (most remaining tuples join) and the latter by design aims for worst-case optimalities.

8.6 Subplan Sharing

McSherry et al. [MLSR20] introduced arrangements into DD to enable efficient index sharing across concurrent queries without redundant reconstruction. They demonstrated the benefits on simple Datalog programs by manually enforcing arrangement sharing.

In FlowLog, we extend this idea by automatically identifying and sharing common subplans within and across rules to reduce memory consumption. Our sharing algorithm is greedy, leveraging the fact that DD maintains the output of every intermediate operator. The input we get is a set of IR, one for each rule, where each IR is a tree (i.e. no sharing yet). Given a set of IR —one per rule—where no sharing is initially present (hence tree-shaped), we canonicalize and hash every subtree. When a duplicate hash is detected, we replace the subtree by a pointer to the output of the first occurrence of that subtree. This coalesces disconnected IR trees into a DAG structure.

Figure 8.5 illustrates this process, applied to the IR in Figure 8.3. On the left, the IR is canonicalized by encoding variable positions relative to their atoms. For example, in edge(x, y) and edge(y, z), variables are rewritten as $(e.0, e.1)$, while reach maps to $r.0$. This reveals that two `Map` subplans are structurally identical (up to variable renaming). Consequently, we reuse the output of the first `Map` —an arrangement on key $e.1$ and value $e.0$—for the second instance. The right side of Figure 8.5 presents the resulting IR after sharing.

Our approach subsumes shared arrangements [MLSR20] as a special case and further extends it to sharing common table subexpressions (CTEs), e.g. if identical joins are recognized in multiple rules, a common scenario in large-scale Datalog programs such as DOOP.

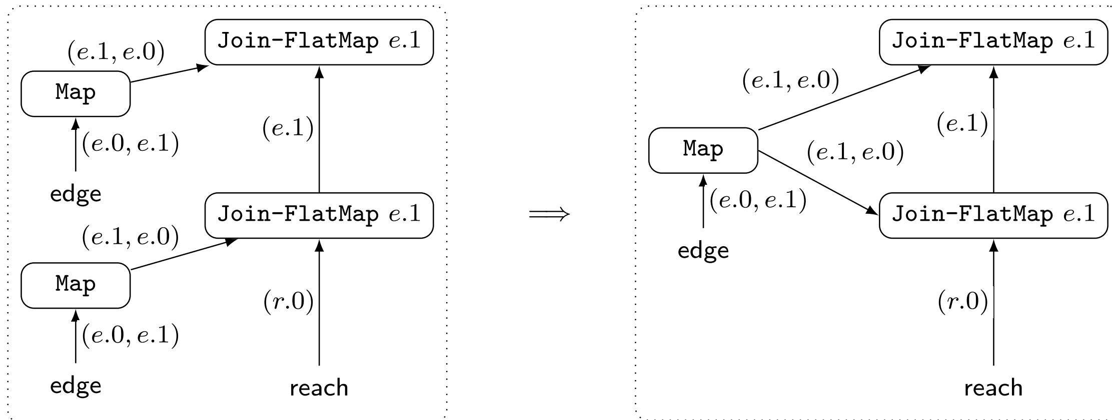


Figure 8.5: Subplan sharing for Figure 8.3

8.7 Boolean Specialization

Recall that each row (`data`, `time`, `diff`) of DD uses an integral (typically 64 bits) `diff` to encode the number of copies of `data`, 0 being the absence of the tuple and negative values indicating deletions or subtractions. Differential operators apply integer arithmetics (e.g. $+, \cdot, -$) to track output `diffs`. A `join` multiplies `diffs` of the two joining rows. A `concat` of $((\mathsf{a}, \mathsf{b}), 0, 4)$ and $((\mathsf{a}, \mathsf{b}), 0, 3)$ results in $((\mathsf{a}, \mathsf{b}), 0, 7)$ by adding the `diffs`, while an `antijoin` yields $((\mathsf{a}, \mathsf{b}), 0, 4 - 3 = 1)$ by subtracting the right `diff` from the left.

Integer arithmetic on `diff` fits well for incremental execution, but it is not always necessary for many Datalog applications, where the mere presence of a tuple is enough. In DD, it corresponds to restricting the `diff` to the Boolean domain, i.e. `true` for presence and `false` for absence. Then `join` implies a logical AND, as the output row is present only if both inputs are. A `concat` of $((\mathsf{a}, \mathsf{b}), 0, \texttt{true})$ and $((\mathsf{a}, \mathsf{b}), 0, \texttt{false})$ should return $((\mathsf{a}, \mathsf{b}), 0, \texttt{true} \vee \texttt{false} = \texttt{true})$ by a logical OR on the `diffs`—preserving presence when at least one exists. However, an `antijoin` is not expressible because subtraction is undefined for Booleans. In fact, DD encodes the presence of a tuple as a zero-bit struct, whereas a `false` (or absent) tuple is immediately discarded when encountered. A Boolean `diff` type has two benefits: (1) it reduces memory footprint by storing `diff` as a zero-bit presence struct, and (2) the logical simplifications enable compiler optimizations that short-circuit Boolean operations.

As a result, FlowLog enforces Boolean `diff` type (as the zero-bit presence struct) whenever possible by coercing the input `diff` values into Booleans and each differential operator, such as `join`, `concat`, into Boolean semantics, i.e. AND, OR operations. To handle non-monotonic operators (i.e. those that require negation or deletion operations), e.g. antijoins, we introduce a custom

lift operator that transitions diff values between Booleans and integers—lift true to 1 and false to 0. In our example, $((\mathsf{a}, \mathsf{b}), 0, \texttt{true})$ antijoin $((\mathsf{a}, \mathsf{b}), 0, \texttt{true})$ is lifted to an integer subtraction $((\mathsf{a}, \mathsf{b}), 0, 1 - 1 = 0)$, and then casted down to a Boolean diff of false.

8.8 Extensibility

We now discuss the extensible nature of FlowLog.

Algebraic Semantics A first extension is on the algebraic types for diff. Sec. 8.7 encodes diff as a Boolean value, as an optimization for batch Datalog queries. To support *incremental Datalog* (EDBs have insertions/deletions over time), we simply fall back on integer arithmetic as DD does by default. Recursive aggregation is another extension that arises frequently in modern data analytics [MYL16, WKN+22, KNP+22a, SSSN24]. For example, the FlowLog program below computes connected components (CC) of an undirected graph via label propagation:

```
CC(x,MIN(x)):- edge(x,_). // initialize labels i
CC(x,MIN(i)):- edge(y,x), CC(y,i). // propagate i
```

The program iteratively propagates labels—discarding larger ones as smaller labels are discovered—until all nodes in a connected component share the same label. Standard semi-naïve evaluation is not applicable in such cases, and most Datalog systems require substantial changes to their execution logic to support this. For example, Soufflé assumes set semantics and relies exclusively on semi-naïve evaluation. In contrast, FlowLog implements recursive aggregation with minimal changes by leveraging incremental aggregation in DD (see CC in Table 8.1). A future optimization is to encode the aggregation semantics directly into the diff field (a generalization over Sec. 8.7), using an algebraic structure called *semiring* [GKT07] that defines the type and valid operations—e.g., Booleans $(\texttt{false}, \texttt{true}, \vee, \wedge)$. By assigning diff to the desired semiring, optionally using a lift operator to cast between semirings for alternative semantics, FlowLog supports efficient and general recursive aggregation with ease.

Distributed Execution A natural extension is scaling FlowLog to distributed environments. DD operators are automatically sharded across workers (i.e. threads or machines) to attain resource saturation. Logical timestamps require minimal synchronization while managing task dependencies, allowing concurrent execution. Unlike Soufflé and RecStep, which are grounded in single-machine execution because of their designs, or BigDatalog [SYI+16] that carefully re-designs distributed execution for recursive settings, FlowLog, as future work, can seamlessly extend beyond single-node setups.

8.9 Benchmarking and Experiments

This section presents experimental results of FlowLog.

Programs and Datasets We curate a suite of queries and datasets as benchmarks—arguably one of the most comprehensive collections evaluated to date. (1) Bipartite is a custom program that decides if a connected undirected graph is bipartite (i.e., two-colorable), starting from a randomly initialized blue node:

```
red(y)  :- edge(x, y), blue(x).
blue(y) :- edge(x, y), red(x).
answer() :- red(x), blue(x).
```

(2) Graph queries such as Single-source Reachability (Reach), Same Generation (SG), Transitive Closure (TC), Connected Components (CC). (3) Program analysis tasks like Andersen, CSPA, and CSDA (Context-sensitive Point-to and Dataflow Analysis) from RecStep's experimental suite [FZZ+19] (so are the datasets); (3) Dyck represents Dyck-2 reachability [LSZ22] whose datasets (kernel and postgre) are sampled from CFPQ_Data[4]; (4) Galen (Example 8.3), and CRDT for conflict-free replicated data types come from Frank McSherry's blog[5]; (5) Polonius encodes an alias-based version of Rust's borrow checker[6]; (6) DOOP [BS09], a popular points-to analysis micro benchmark, features 136 rules with intricate recursive dependencies. Its datasets are sampled from the DaCapo suite [BGH+06]; (7) DDISASM is a simplified disassembly analysis program from [FS20] and its datasets are synthesized from the CVC5 [BBB+22] and Z3 SMT solvers [dMB08].

Competing Engines We compare FlowLog against several state-of-the-art, open-source Datalog engines: Soufflé (both compiled and interpreted modes), RecStep, and DDlog. Soufflé is the most adopted Datalog engine, backed by years of development and a rich set of optimizations, e.g. [HZJS21, SJC+18, JSZS19, AHZ+22]. RecStep attains strong performance and scalability on large-scale data over similar architectures such as BigDatalog. DDlog shares the same DD backend as FlowLog but differs in design and lacks key optimizations discussed in the paper, making it a fair baseline. We assess all engines on their latest stable releases in three dimensions: execution time, scalability with increasing thread count, and memory comsumption. All experiments run on a CloudLab [Clo18] VM with dual-socket AMD EPYC 7543 32-core processors (64 physical cores, 128 hardware threads), running Ubuntu 20.04 LTS with 256 GB RAM.

Soufflé (compiled) and DDlog require per-program compilation, whereas FlowLog, Soufflé (interpreted), and RecStep are interpreters. Soufflé typically incurs a 10s compilation overhead for small

[4] https://formallanguageconstrainedpathquerying.github.io/CFPQ_Data
[5] https://github.com/frankmcsherry/dynamic-datalog
[6] https://github.com/rust-lang/polonius

Program	#rules	Dataset	FlowLog	Souffle (compiled)	Souffle (interpreted)	RecStep	DDlog
Bipartite	4	mind	**6.6** | **3.2**	19.8 | 16.2	21.6 | 16.7	11.1 | 5.1	31.1 | 26.3
		netflix [Yan22]	**32.3** | **7.7**	113.8 | 102.7	119.9 | 107.7	39.6 | 8.0	174.5 | 148.3
Reach	2	livejournal	**13.1** | **8.1**	20.8 | 18.7	22.9 | 20.2	27.9 | 9.0	110.8 | 99.8
		orkut	**22.5** | 13.0	36.4 | 32.3	40.1 | 35.2	43.3 | **11.8**	185.6 | 165.4
		twitter [FZZ+19]	**323.8** | **104.6**	tout	tout	tout	tout
TC	2	G5K-0.001	13.7 | **2.4**	**9.2** | 4.7	11.0 | 5.9	35.8 | 16.5	39.2 | 22
		G10K-0.001 [FZZ+19]	87.5 | **10.4**	**52.2** | 15.7	63.6 | 20.5	193.5 | 70.1	206.8 | 94.6
SG	2	G5K-0.001	27.8 | **4.1**	**26.2** | 5.9	31.5 | 7.3	63.9 | 23.9	61.2 | 27.1
		G10K-0.001 [FZZ+19]	**198.8** | **19.6**	380.7 | 39.9	459.9 | 49.1	535.7 | 137.7	412.3 | 110.5
CC	2	livejournal	**89.7** | **14.8**	×	×	127.1 | 28.3	199 | 107.9
		orkut [FZZ+19]	**141.6** | **22.6**	×	×	171.9 | 25.4	308.1 | 176.7
Andersen	4	medium (500000)	**6.3** | **4.6**	125 | 33.3	130 | 34.1	43.3 | 12.8	51.4 | 50.7
		large (1000000) [FZZ+19]	**12.8** | **5.6**	311 | 76.3	318.9 | 75.5	93.9 | 18.6	107.3 | 103.3
CSDA	2	httpd	**5.4** | **2.6**	8.7 | 9.6	9.2 | 11.5	64.7 | 50.1	23.7 | 21.2
		linux	**26.2** | **9.6**	40.5 | 59.7	44.2 | 65.5	tout | 278.5	123.9 | 102.2
		postgresql [FZZ+19]	**14.6** | **6.2**	22.9 | 26.7	24.1 | 29.5	377.3 | 204.9	73.3 | 65.4
CSPA	10	httpd	101.7 | **14.0**	**59.5** | 42.6	91.5 | 60.5	547.5 | 149.1	320.2 | 270.0
		linux	**20.6** | **4.8**	20.8 | 12.1	27.4 | 15.6	73.7 | 49.0	68.4 | 51.3
		postgresql [FZZ+19]	102.7 | **14.7**	**69.3** | 50.9	101.7 | 63.7	469.8 | 152.4	321.0 | 262.4
Dyck	7	kernel	**4.8** | **2.3**	17.3 | 8.9	19.4 | 9.0	31.6 | 12.2	26.2 | 27.5
		postgre	**3.7** | **1.8**	11.8 | 5.4	13.3 | 5.8	16.4 | 11.3	15.4 | 14.8
Galen	8	galen [ZSRS21]	**33.2** | **8.9**	54.7 | 29.5	62.8 | 28.4	tout	145.2 | 60.2
CRDT	23	crdt	290.8 | **109.9**	**209.9** | 232.9	240.4 | 286.8	×	tout | 465.9
Polonius	37	polonius	242.3 | **45.9**	**188.6** | 283.8	209.9 | 357.0	×	tout | 477.8
DOOP	136	batik	43.4 | **14.9**	**39.7** | 34.2	62.0 | 42.8	×	150.2 | 100.2
		biojava	**8.6** | **5.1**	14.1 | 11.1	27.8 | 14.9	×	38.5 | 42.4
		eclipse	45.0 | **16.3**	**29.4** | 24.1	47.4 | 31.2	×	143.2 | 93.0
		xalan	**5.5** | **3.9**	9.5 | 7.2	16.0 | 10.1	×	22.9 | 29.2
		zxing [BGH+06]	**7.2** | **4.3**	10.8 | 9.2	16.6 | 12.8	×	27.8 | 35.4
DDISASM	28	cvc5 [BBB+22]	78.2 | **13.9**	**33.2** | 16.8	53.9 | 18.1	×	577.9 | 103.5
		z3 [dMB08]	**93.6** | **28.1**	144.2 | 104.2	165.4 | 107.1	×	tout | 462.8

Table 8.1: Runtime results (seconds) under 4 and 64 threads, separated by '|' and FlowLog results are shaded. The best performance for each row is colored in **blue** for 4 threads and **red** for 64 threads; × indicates unsupported syntax and `tout` indicates a timeout after 10 minutes.

programs and 30s for larger ones such as DOOP and DDISASM. In contrast, as [HZJS21] points out, DDlog exhibits significantly higher Rust compilation times—often exceeding 100s—even for small programs.

8.9.1 Runtime Summary

Table 8.1 reports end-to-end runtimes (in seconds) of all competing engines across our benchmark suite, measured on both 4 and 64 threads. All results reflect the best of 3 runs, with the fastest per row in **bold**. To isolate core execution efficiency and study the effect of our memory-saving techniques, we disable FlowLog's join planning optimizations (Sec. 8.4–8.5) and fix the join order of each rule to the best-performing listing order, determined through preliminary profiling on the datasets. This setup, discussed at Sec. 8.4, enforces a consistent left-to-right join order for FlowLog, Soufflé, and DDlog across all programs. By contrast, RecStep, in its default behavior, reoptimizes join orders on-the-fly through its underlying DBMS optimizer. Logic fusion (Sec. 8.3), subplan sharing (Sec. 8.6), and boolean specialization (Sec. 8.7) remain enabled for FlowLog, in stark contrast to DDlog, being a baseline that directly translates Datalog into DD programs.

On 4-thread runs, FlowLog outperforms all competitors in 21 out of 31 program-dataset pairs. It consistently leads on programs such as CC, Dyck, and CSDA. A striking example is Andersen (large), where FlowLog runs 24.4× faster than Soufflé (both modes), 7.3× than RecStep, and 8.1× than DDlog. A key reason for this advantage is subplan sharing (Sec. 8.6): all indexes for IDBs in Andersen are constructed once and reused in multiple rules, avoiding a large amount of redundant work. In Reach, FlowLog attains 1.6× speedups over Soufflé, 2× over RecStep, and 8× over DDlog on livejournal and orkut; on the much larger twitter graph (over 25 GB), all others timeout except FlowLog. That said, FlowLog does not always excel on **batch-oriented workloads**—programs with few but expensive iterations, like TC on G10K-0.001 (6 iter.). In these cases, its incremental design (built atop DD) incurs overheads, as large intermediate results are maintained but used sparingly. Another such example is CSPA on `httpd` (29 expensive iter.), where Soufflé (compiled) runs in 59s—1.7× faster than FlowLog's 101s runtime. Still, FlowLog outperforms RecStep and DDlog by 5.4× and 3.1×, respectively, and matches Soufflé interpreter, which lacks compiler-time optimizations such as staged compilation [SJSW16].

Nonetheless, FlowLog exhibits significantly superior scalability compared to others. At 64 threads, it achieves substantial speedups in all benchmarks and becomes the fastest system in 30 out of 31 cases. In all CSPA runs, FlowLog runtimes drop sharply (e.g., 7.3× speedup on `httpd`), while Soufflé (compiled) sees only modest gains (e.g., 1.4×)—becoming 3× slower than FlowLog. A similar trend appears in Polonius, where Soufflé even degrades from 188s to 283s as thread count increases, whereas FlowLog accelerates from 242s to 46s. In DDISASM (`cvc5`), FlowLog underperforms at 4 threads but surpasses Soufflé (compiled) at 64 threads—reversing a 2.3× slowdown into a 1.2× speedup. Both RecStep and DDlog exhibit poor scalability, lagging behind FlowLog in almost all cases.

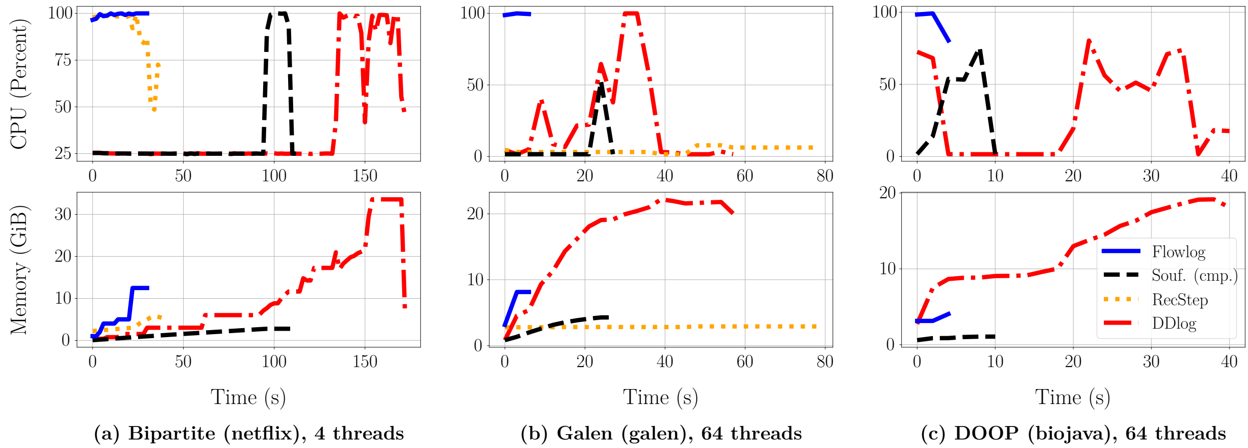


Figure 8.6: Resources consumption on 3 workloads; each figure shows CPU (top) and memory usage (bottom) over time. Soufflé (interp.) shows similar (slightly worse) efficiency to Soufflé (comp.) and is omitted. RecStep does not support DOOP.

8.9.2 CPU and Memory Usage

Figure 8.6 reports real-time CPU and memory usage of all systems on three representative workloads in Table 8.1. In all cases, FlowLog consistently saturates all available cores (near 100% for 4 and 64 threads). FlowLog's high CPU efficiency stems from its asynchronous execution atop DD, where all computation stages—including `iterate`—remain active and concurrent throughout execution (see Sec. 8.4.3). This fully pipelined parallelism exploits all cores continuously. In contrast, Soufflé primarily employs outer-loop parallelism in its index nested loop joins. RecStep leverages the DBMS's internal parallelism, but suffers from synchronization between iterations. As a result, both systems show moderate CPU usage for batch-oriented workloads such as Bipartite (Figure 8.6a, 4 iter.), but their CPU usage becomes erratic and drops sharply—often below 20%—on more iterative workloads like Galen (Figure 8.6b, 32 iter.). Further inefficiencies are evident in Figure 8.6a, where Soufflé and DDlog exhibit long single-threaded phases for index construction over large inputs. On large programs such as DOOP, which feature many recursive strata of heterogeneous characteristics, Soufflé averages $<50\%$ CPU usage. DDlog—lacking dedicated support for parallelism [WWZ22, FMK22]—displays volatile and inefficient CPU usage.

A major source of inefficiency for DDlog lies in its large memory footprint, stemming from direct invocation of DD, which incrementally maintains intermediate results. In all bottom plots of Figure 8.6, DDlog exhibits prohibitively high memory usage (beyond 20GB). In contrast, Soufflé and RecStep incur much lower memory overheads, generally staying below 5GB. Although FlowLog also builds on DD, it benefits heavily from the memory-saving techniques proposed at Sec. 8.3, Sec. 8.6,

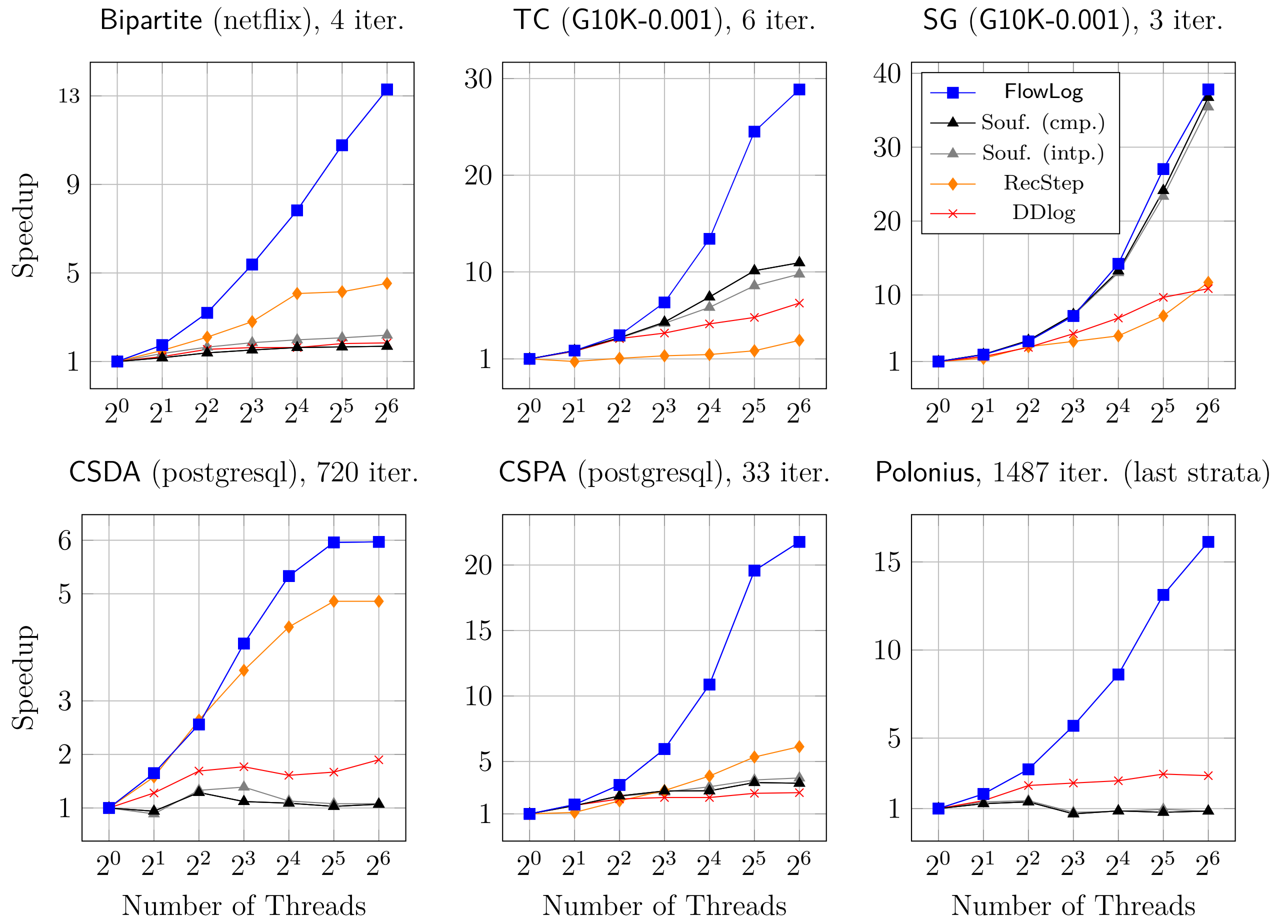
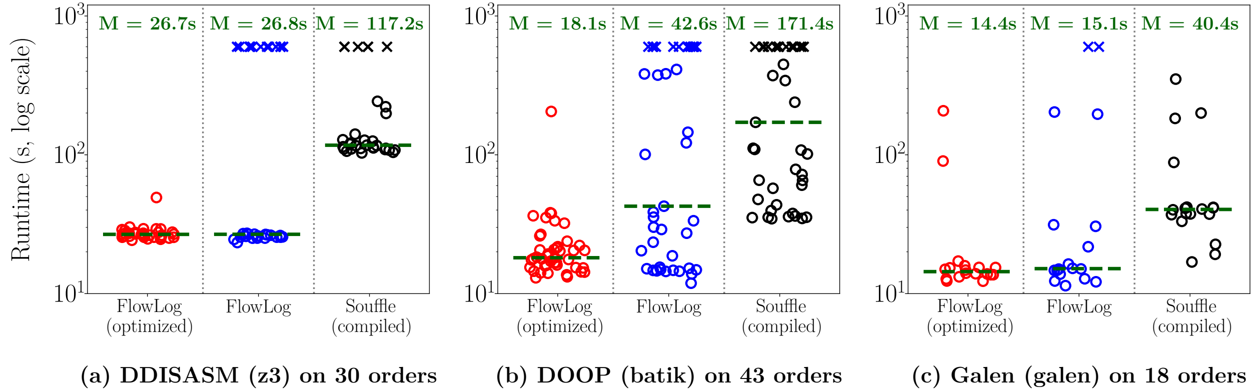


Figure 8.7: Scalability of all systems across 6 program-data pairs, where each subplot reports the speedup relative to single-threaded execution, up to $2^6 = 64$ threads. RecStep does not support the Polonius program and is excluded from the last subplot.

and Sec. 8.7. As a result, FlowLog is much more memory-efficient—using roughly 3× less memory than DDlog on average across the three workloads, yet still 2–3× more than Soufflé.

8.9.3 Parallel Scalability

Scalability of Datalog engines varies heavily by workload. Figure 8.7 presents the scalability of all systems across 6 representative cases. These include graph queries (top row), and program analyses (bottom row). All speedups are relative to single-threaded execution (1×). Across all cases, FlowLog exhibits consistently superior scalability. Its speedup continues to increase up to 32 threads and only slightly plateaus at 64 threads. On batch-oriented workloads such as TG or SG, most engines benefit from increased parallelism due to ample intra-iteration work. In the TC (G10K-0.001) case, FlowLog



(a) DDISASM (z3) on 30 orders **(b) DOOP (batik) on 43 orders** **(c) Galen (galen) on 18 orders**

Figure 8.8: Runtime variability across different join orderings and engines (64 threads). Dark green numbers and dotted lines show the median runtime per system. Timeouts (over 10 min) and OOM cases are marked with a cross (×), all truncated at 10 min.

reaches nearly 30× speedup at 64 threads, while the other engines also scale modestly (up to 10×). On SG, Soufflé matches FlowLog's scalability. However, parallelism shrinks on **long-tail workloads** with many lightweight iterations; and overheads of thread management and data exchange can outweigh the benefits. Examples include CSDA (postgresql, 720 iter.) and Polonius (1487 iter.!), where Soufflé has virtually no speedup, DDlog stays below 2×, and RecStep reaches 5× (for CSDA) but from a much slower baseline. This is largely due to RecStep's lack of continuously maintained indexes, forcing it to reconstruct them at every iteration [FMK22]. In contrast, FlowLog still achieves 6× and 16× speedup, respectively.

8.9.4 Join Planning and Robust Execution

Thus far, we have enforced optimal join plans to compare the best possible performance. However, sub-optimal plans can significantly jeopardize the efficiency of Datalog engines.

First, we revisit RecStep's strategy of relying on its DBMS optimizer to optimize join orders on the fly. In programs like Reach and CSDA, where optimal join orders are evident (e.g., two-way joins or easily inferred from semantics), RecStep repeatedly invokes the optimizer at each iteration, wastefully reproducing the same plan. This overhead undermines RecStep's overall performance compared to FlowLog (Table 8.1). Galen further exposes its shortcomings: RecStep times out due to data skew in IDBs —a pattern difficult for a DBMS to detect by cardinality estimates—leading to poor plan decisions.

We now turn to the consequences of bad join orders in FlowLog and evaluate our techniques introduced in Sec. 8.4 (worst-case planning) and Sec. 8.5 (robust Datalog execution). Figure 8.8 takes

three benchmarks that contain expensive multi-way joins: Galen, DOOP (batik), and DDISASM (z3). For each program, we repeatedly sample one of the top five most costly rules and replace its listing order with a new, unused variant, avoiding cross joins (since Soufflé executes listing orders as-is and they usually lead to significant slowdowns). Figure 8.8 compares the runtime variability of FlowLog (optimized)—FlowLog enabling optimizations of Sec. 8.4 and Sec. 8.5—against default FlowLog and Soufflé (compiled). We will call them FlowLog (opt.), FlowLog and Soufflé henceforth, for brevity. We omit DDlog, as FlowLog can be regarded as a memory-optimized variant of it.

Robust Execution In Figure 8.8, both FlowLog and Soufflé exhibit significant runtime sensitivity to join ordering. Out of 91 distinct listing orders sampled across all programs, while many yield reasonable runtimes, FlowLog times out after 10 minutes or exhausts all RAM (OOM) on 25 instances, and Soufflé times out on 23—over 25% of all cases! In FlowLog, poor join orders force DD to incrementally maintain substantially larger intermediate join outputs, leading to timeouts or OOM. Soufflé, while avoiding materializing intermediates through pipelined left-deep execution (see Sec. 8.4.3), still suffers from long runtimes due to expensive nested-loop joins. In contrast, FlowLog (opt.) never times out or runs OOM across all plans, and its runtime distributions are much more stable.

The most expensive Datalog rules often involve multi-way joins where optimal orders are elusive; here, our SIP-style algorithms aggressively prune non-joining tuples to prevent catastrophic slowdowns. Taking the bad listing order of r_3 in Example 8.3, FlowLog takes 203s and peaks at 235 GB; FlowLog (opt.), though not able to identify the best order, leverages SIP to reduce runtime to 89s and memory to 195 GB. In other cases, such as a five-way join in DDISASM, our cost-based planner (Sec. 8.4) runs itself into bad join orders that suffer from severe data skew, causing FlowLog to run OOM. Yet, SIP filters out most skewed non-joining data, enabling FlowLog (opt.) to still complete in only 29 seconds with 18 GB peak memory. Nonetheless, as discussed in Sec.8.5, SIP is not a panacea: while it effectively mitigates worst cases, it introduces moderate overhead that slightly slows down the fastest plans (visible at the bottom of each subplot in Figure 8.8), albeit by a very small margin.

Join Planning Figure 8.8 reports median runtimes across all join orders, counting timeouts and OOM conservatively as 10 minutes to favor FlowLog and Soufflé. In all programs, FlowLog (opt.) consistently achieves lower median runtimes than the other two. The largest speedup is in DOOP, where FlowLog (opt.) achieves a 2.4× speedup over FlowLog and 9.5× over Soufflé. These gains are largely due to the prevalence of expensive multi-way joins, such as Example 8.2, where our cost-based planner (Sec. 8.4) enables FlowLog (opt.) to select much better worst-case join orders. For example, for the rule of Example 8.2, poorly sampled listing orders cause both FlowLog and Soufflé to time

out, whereas FlowLog (opt.) chooses the order in Figure 8.4 (right), completing in 13s—faster than even the manually-tuned listing order used in Table 8.1 (14.9s).

Lastly, consider the listing orders of Galen's r_2, (see Example 8.3),

```
p(x,z) :- p(y,w), u(w,r,z), q(x,r,y). // best order
p(x,z) :- u(w,r,z), q(x,r,y), p(y,w). // variant 1
p(x,z) :- q(x,r,y), u(w,r,z), p(y,w). // variant 2
```

We record the actual runtimes of these three orders below (in order), including FlowLog runtimes that only enables SIP in the optimizer.

FlowLog (opt.)	FlowLog (sip only)	FlowLog	Soufflé (cmp.)
10.2s	10.4s	8.9s	32.2s
12.3s	OOM	OOM	182.9s
13.7s	OOM	OOM	199.9s

This rule features dense joins, i.e. joins of few dangling tuples, so SIP incurs overheads without offering much pruning benefit. However, such dense joins exhibit near worst-case behavior, precisely the setting where our join planner (Sec. 8.4) excels. In the latter two orders, FlowLog (opt.) avoids OOMs by starting the join from p.

8.10 Conclusion

We introduced FlowLog, a new Datalog engine that decouples the logical optimizations from its physical execution. On the physical side, FlowLog uses DD's streaming primitives to have out-of-the-box efficient execution. However, DD incurs high memory overhead when maintaining large intermediate states. To address this, we introduced a suite of IR-level optimizations, including query planning techniques that avoid poor join orders and stabilize the volatility across iteration. Our experiments show that FlowLog consistently outperforms existing Datalog systems such as Soufflé, RecStep, and DDlog in diverse benchmarks, sometimes by an order of magnitude.

Chapter 9

Conclusion and Future Outlook

This dissertation dicussed the current landscape of join algorithms, introducing on two prominent examples: Yannakakis and PANDA. We emphasized their core insights: Yannakakis pre-filters non-joining tuples before actual joins, whereas PANDA partitions data and selects the most efficient query plan for each partition. Notably, both algorithms are primarily tailored for join-only queries.

This dissertation investigated join algorithms in the context of real-world queries, which often include richer relational constructs beyond simple joins. Chapters 4-7 presented novel algorithmic extensions to existing join techniques that account for these complexities. We introduced the output-sensitive Yannakakis algorithm, which generalizes the classic Yannakakis algorithm to handle arbitrary projections and aggregations—arguably the most common patterns in practical query workloads. We established a linear-time characterization of join queries involving antijoins. We also explored the multi-query optimization setting of join queries with access patterns, which frequently arise in practice. Lastly, we studied recursive queries through the lens of Datalog, where existing join algorithms and system designs remain inadequate. We tackled both issues by proposing a general-purpose grounding-based algorithm that provides strong runtime guarantees. We further built a modular prototype system that seamlessly integrates advanced join algorithms, demonstrating consistent performance gains over existing Datalog implementations.

The story of join algorithms is far from over. Recent work [Hu24b] has advanced beyond the current state of output-sensitive algorithms, showing even stronger runtime guarantees for join-aggregate queries upon ours. While PANDA remains technically intricate, recent efforts have sought to simplify its design [KNS24], though significant challenges persist. The evolution of join algorithms in response to real-world queries is inherently iterative. As new data volumes, query workloads and computational paradigms evolve, join processing must continuously adapt to meet changing demands. For example, recent work has shown that harnessing fast matrix multiplication libraries can lead to faster join algorithms [DHK20]. The research community has been actively expanding its repertoire of specialized

join algorithms to address new challenges in query optimization—such as joins with comparison predicates [WY23, KCM+20], quantile joins [TCG+23], and top-k queries [CTG+21, DK21, TAG+20]. Beyond algorithmic approaches, adaptive or learning-based techniques have also appeared as a promising avenue for data-driven join optimization [ZPSP17, KJS+22, MNM+19, MNM+21, WBHB25]. Pivoting towards distributed and parallel environments, research has increasingly focused on multi-way join algorithms tailored for massive parallelism [BKS13, BKS14, Hu21].

From a lower bound perspective, our understanding of the fundamental limits of join algorithms remains incomplete. Existing lower bounds are narrowly scoped and fail to capture the full complexity of modern join queries. Addressing this gap is essential for a more principled foundation for join algorithm design, but doing so will likely require deep insights from diverse areas, including circuit complexity and information theory. As an example, an interesting open question is whether the results of [FKZ24] can be translated to the all conjunctive queries. Resolving this question would provide a vast generalization on the seminal result by Alon et al. [AB87] that shows that monotone Boolean circuits for the k-clique problem over a graph with n vertices have size $\Omega(n^k/(\log n)^k)$.

From a practical standpoint, many of these algorithmic advances are starting to influence real-world relational engines. While practical implementations often adopt simplified variants rather than exact replicas, they retain key ideas of these join algorithms that drive performance gains. For example, the Predicate transfer technique [YZYK24] draws from the pre-filtering principle of Yannakakis to eliminate non-joining tuples early through semijoins. To avoid the overhead of full semijoins, Predicate transfer uses lightweight Bloom filters as a more efficient pre-filtering mechanism. It has been further demonstrated that this approach also strengthens robustness of query optimizers against suboptimal query plans [ZSY+25]. An exciting future direction is to explore how such pre-filtering techniques can be leveraged to reduce data movement (e.g. data shuffles by hashing) in industry-scale distributed databases. Hu et al. [HW23] introduced a `SQL` rewriting approach to optimize antijoins—an application directly tied to the principles discussed in Chapter 5. Worst-case optimal join (WCOJ) algorithms have also found their way into real systems through a variety of implementations. Leapfrog Triejoin [Vel12], originally deployed in LogicBlox, was among the first, followed by modern adaptations such as those by Freitag et al. [FBS+20]. FreeJoin [WWS23] integrates WCOJ logic with traditional binary joins into a unified execution framework. Meanwhile, in the domain of cardinality estimation, researchers have been leveraging the entropic foundations of PANDA to derive tight bounds on intermediate result sizes [CBS19, DSBC23], guiding optimizers toward better plan choices. A particularly *intriguing* future direction is to explore data partitioning techniques central to PANDA and output-sensitive Yannakakis (Chapter 4), which could enable the deployment of multiple query plans tailored to different partitions of the data—further enhancing query robustness and performance.

LIST OF REFERENCES

[AAHM05] Serge Abiteboul, Zoë Abrams, Stefan Haar, and Tova Milo. Diagnosis of asynchronous discrete event systems: datalog to the rescue! PODS '05, page 358–367, New York, NY, USA, 2005. Association for Computing Machinery.

[AB87] Noga Alon and Ravi B. Boppana. The monotone circuit complexity of boolean functions. *Comb.*, 7(1):1–22, 1987.

[ABDN18] Amir Abboud, Karl Bringmann, Holger Dell, and Jesper Nederlof. More consequences of falsifying SETH and the orthogonal vectors conjecture. In *STOC*, pages 253–266. ACM, 2018.

[ACFS00] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. Gambling in a rigged casino: The adversarial multi-armed bandit problem. *Electron. Colloquium Comput. Complex.*, TR00-068, 2000.

[AG94] Miklós Ajtai and Yuri Gurevich. Datalog vs first-order logic. *J. Comput. Syst. Sci.*, 49(3):562–588, 1994.

[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.

[AGM08] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *FOCS*, pages 739–748. IEEE Computer Society, 2008.

[AHSY24] Pankaj K Agarwal, Xiao Hu, Stavros Sintos, and Jun Yang. On reporting durable patterns in temporal proximity graphs. *Proceedings of the ACM on Management of Data*, 2(2):1–26, 2024.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AHZ+22] Samuel Arch, Xiaowen Hu, David Zhao, Pavle Subotic, and Bernhard Scholz. Building a join optimizer for soufflé. In *LOPSTR*, volume 13474 of *Lecture Notes in Computer Science*, pages 83–102. Springer, 2022.

[AKLS22] Amir Abboud, Seri Khoury, Oree Leibowitz, and Ron Safier. Listing 4-cycles. *arXiv preprint arXiv:2211.10022*, 2022.

[AKNR16] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. Faq: Questions asked frequently. In *Proceedings of the 35th ACM Symposium on Principles of Database Systems*, PODS '16, New York, NY, USA, 2016.

[AN16] Peyman Afshani and Jesper Asbjørn Sindahl Nielsen. Data structure lower bounds for document indexing problems. In *ICALP*, 2016.

[And94] Lars Ole Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, Citeseer, 1994.

[AP93] Foto Afrati and Christos H Papadimitriou. The parallel complexity of simple logic programs. *Journal of the ACM (JACM)*, 40(4):891–916, 1993.

[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, 2009.

[AXR24] Supun Abeysinghe, Anxhelo Xhebraj, and Tiark Rompf. Flan: An expressive and efficient datalog compiler for program analysis. *Proc. ACM Program. Lang.*, 8(POPL):2577–2609, 2024.

[BB13] Pablo Barceló Baeza. Querying graph databases. In *PODS*, pages 175–188, 2013.

[BB16] Johann Brault-Baron. Hypergraph acyclicity revisited. *ACM Comput. Surv.*, 49(3), Dec 2016.

[BBB+22] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In *TACAS (1)*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[BDG07] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007.

[BG81] Philip A Bernstein and Nathan Goodman. Power of natural semijoins. *SIAM Journal on Computing*, 10(4):751–771, 1981.

[BG24] Karl Bringmann and Egor Gorbachev. A fine-grained classification of subquadratic patterns for subgraph listing and friends. *CoRR*, abs/2404.04369, 2024.

[BGH+06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA*, pages 169–190. ACM, 2006.

[BGS20] Christoph Berkholz, Fabian Gerhardt, and Nicole Schweikardt. Constant delay enumeration for conjunctive queries: a tutorial. *ACM SIGLOG News*, 7(1):4–33, 2020.

[BK80] Andries E Brouwer and Antoon W. J. Kolen. A super-balanced hypergraph has a nest point. *Technical report, Math. centr. report ZW146, Amsterdam*, 1980.

[BKN24] Altan Birler, Alfons Kemper, and Thomas Neumann. Robust join processing with diamond hardened joins. *Proc. VLDB Endow.*, 17(11):3215–3228, 2024.

[BKS13] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *PODS*, pages 273–284. ACM, 2013.

[BKS14] Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in parallel query processing. In *PODS*, pages 212–223. ACM, 2014.

[BKS17] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *PODS*, pages 303–318. ACM, 2017.

[BMSU85] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, 1985.

[BPWZ14] Andreas Björklund, Rasmus Pagh, Virginia Vassilevska Williams, and Uri Zwick. Listing triangles. In *International Colloquium on Automata, Languages, and Programming*, pages 223–234. Springer, 2014.

[Bra12] Johann Brault-Baron. A negative conjunctive query is easy if and only if it is beta-acyclic. In *CSL*, volume 16 of *LIPIcs*, pages 137–151. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

[Bra13] Johann Brault-Baron. *De la pertinence de l'énumération : complexité en logiques propositionnelle et du premier ordre.* PhD thesis, University of Caen Normandy, France, 2013.

[BS09] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243–262. ACM, 2009.

[BS19] Christoph Berkholz and Nicole Schweikardt. Constant delay enumeration with fpt-preprocessing for conjunctive queries of bounded submodular width. In *MFCS*, volume 138 of *LIPIcs*, pages 58:1–58:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[BSFN24] Altan Birler, Tobias Schmidt, Philipp Fent, and Thomas Neumann. Simple, efficient, and robust hash tables for join processing. In *DaMoN*, pages 4:1–4:9. ACM, 2024.

[BTAÖ13] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, pages 362–373. IEEE Computer Society, 2013.

[CBS19] Walter Cai, Magdalena Balazinska, and Dan Suciu. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *SIGMOD Conference*, pages 18–35. ACM, 2019.

[CGMV16] Alessio Conte, Roberto Grossi, Andrea Marino, and Luca Versari. Sublinear-space bounded-delay enumeration for massive network analytics: Maximal cliques. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 148, pages 1–148, 2016.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43. ACM Press, 1998.

[CI24] Florent Capelli and Oliver Irwin. Direct access for conjunctive queries with negation. *27th International Conference on Database Theory, ICDT 2024, March 25-28, 2024, Paestum, Italy*, pages 13:1–13:20, 2024.

[CK21] Nofar Carmeli and Markus Kröll. On the enumeration complexity of unions of conjunctive queries. *ACM Transactions on Database Systems (TODS)*, 46(2):1–41, 2021.

[CL15] Timothy M Chan and Moshe Lewenstein. Clustered integer 3sum via additive combinatorics. In *STOC*, pages 31–40, 2015.

[Clo18] https://www.cloudlab.us/, 2018.

[CLRS22] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[Cos99] Stavros Cosmadakis. Inherent complexity of recursive queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 148–154, 1999.

[Cou90] Bruno Courcelle. Graph rewriting: An algebraic and logic approach. In *Formal Models and Semantics*, pages 193–242. Elsevier, 1990.

[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.

[CPL+24] David C.Y. Chu, Rithvik Panchapakesan, Shadaj Laddad, Lucky E. Katahanas, Chris Liu, Kaushik Shivakumar, Natacha Crooks, Joseph M. Hellerstein, and Heidi Howard. Optimizing distributed protocols with query rewrites. *Proc. ACM Manag. Data*, 2(1), March 2024.

[CR89] B. Chazelle and B. Rosenberg. Computing partial sums in multidimensional arrays. In *Proceedings of the Fifth Annual Symposium on Computational Geometry*, SCG '89, page 131–139, New York, NY, USA, 1989. ACM.

[CR91] Bernard Chazelle and Burton Rosenberg. The complexity of computing partial sums off-line. *Int. J. Comput. Geom. Appl.*, 1(1):33–45, 1991.

[CS21] Katrin Casel and Markus L. Schmid. Fine-grained complexity of regular path queries. In *ICDT*, volume 186 of *LIPIcs*, pages 19:1–19:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[CTG+21] Nofar Carmeli, Nikolaos Tziavelis, Wolfgang Gatterbauer, Benny Kimelfeld, and Mirek Riedewald. Tractable orders for direct access to ranked answers of conjunctive queries. In *Proceedings of the 40th ACM Symposium on Principles of Database Systems*, pages 325–341, 2021.

[CTG+23] Nofar Carmeli, Nikolaos Tziavelis, Wolfgang Gatterbauer, Benny Kimelfeld, and Mirek Riedewald. Tractable orders for direct access to ranked answers of conjunctive queries. *ACM Trans. Database Syst.*, 48(1):1:1–1:45, 2023.

[CYQ13] Lijun Chang, Jeffrey Xu Yu, and Lu Qin. Fast maximal cliques enumeration in sparse graphs. *Algorithmica*, 66:173–186, 2013.

[Dat89] Chris J Date. *A Guide to the SQL Standard.* Addison-Wesley Longman Publishing Co., Inc., 1989.

[DCZ+16] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The snowflake elastic data warehouse. In *SIGMOD Conference*, pages 215–226. ACM, 2016.

[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*, SIGMOD '20, pages 1213–1223, New York, NY, USA, 2020. Association for Computing Machinery.

[DHK22] Shaleen Deep, Xiao Hu, and Paraschos Koutris. Ranked enumeration of join queries with projections. *Proc. VLDB Endow.*, 15(5):1024–1037, 2022.

[DHK23] Shaleen Deep, Xiao Hu, and Paraschos Koutris. General space-time tradeoffs via relational queries. In *Algorithms and Data Structures Symposium*, pages 309–325. Springer, 2023.

[Dij22] Edsger W Dijkstra. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, pages 287–290. 2022.

[DK18] Shaleen Deep and Paraschos Koutris. Compressed representations of conjunctive query results. In *PODS*, pages 307–322. ACM, 2018.

[DK21] Shaleen Deep and Paraschos Koutris. Ranked enumeration of conjunctive query results. In *ICDT*, volume 186 of *LIPIcs*, pages 5:1–5:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[DLT23a] Shiyuan Deng, Shangqi Lu, and Yufei Tao. On join sampling and the hardness of combinatorial output-sensitive join algorithms. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 99–111, 2023.

[DLT23b] Shiyuan Deng, Shangqi Lu, and Yufei Tao. Space-query tradeoffs in range subgraph counting and listing. In *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece*, pages 6:1–6:25, 2023.

[DM25] Kyle Deeds and Timo Camillo Merkl. Partition constraints for conjunctive queries: Bounds and worst-case optimal joins. In *ICDT*, volume 328 of *LIPIcs*, pages 17:1–17:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

[dMB08] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[DMRT14] Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. Circuits for datalog provenance. In *ICDT*, volume 3, page 2014. Citeseer, 2014.

[DP02] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

[DRH11] AnHai Doan, Raghu Ramakrishnan, and Alon Y. Halevy. Crowdsourcing systems on the world-wide web. *Commun. ACM*, 54(4):86–96, 2011.

[DSBC23] Kyle Deeds, Dan Suciu, Magda Balazinska, and Walter Cai. Degree sequence bound for join cardinality estimation. In *ICDT*, volume 255 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[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, 2018.

[DZFK24] Shaleen Deep, Hangdong Zhao, Austen Z. Fan, and Paraschos Koutris. Output-sensitive conjunctive query evaluation. *Proc. ACM Manag. Data*, 2(5):220:1–220:24, 2024.

[EKL10] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. Newtonian program analysis. *J. ACM*, 57(6), nov 2010.

[FBS+20] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.*, 13(11):1891–1904, 2020.

[FFG02] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *Journal of the ACM (JACM)*, 49(6):716–752, 2002.

[FGT08] J. Nathan Foster, Todd J. Green, and Val Tannen. Annotated xml: Queries and provenance. In *PODS*, page 271–280. ACM, 2008.

[FHM05] Michael J. Franklin, Alon Y. Halevy, and David Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Rec.*, 34(4):27–33, 2005.

[FKKV25] Johannes Freischuetz, Konstantinos Kanellis, Brian Kroth, and Shivaram Venkataraman. Tuna: Tuning unstable and noisy cloud applications. *arXiv preprint arXiv:2503.01801*, 2025.

[FKOW23] Zhiwei Fan, Paraschos Koutris, Xiating Ouyang, and Jef Wijsen. Lincqa: Faster consistent query answering with linear time guarantees. *Proc. ACM Manag. Data*, 1(1):38:1–38:25, 2023.

[FKZ23] Austen Z. Fan, Paraschos Koutris, and Hangdong Zhao. The fine-grained complexity of boolean conjunctive queries and sum-product problems. In *ICALP*, volume 261 of *LIPIcs*, pages 127:1–127:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[FKZ24] Austen Z. Fan, Paraschos Koutris, and Hangdong Zhao. Tight bounds of circuits for sum-product queries. *Proc. ACM Manag. Data*, 2(2):87, 2024.

[Flo62] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[FMK22] Zhiwei Fan, Sunil Mallireddy, and Paraschos Koutris. Towards better understanding of the performance and design of datalog systems. In *Proceedings of the 4th International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog-2.0 2022)*, CEUR Workshop Proceedings, pages 166–180. CEUR-WS.org, 2022.

[FS20] Antonio Flores-Montoya and Eric M. Schulte. Datalog disassembly. In *USENIX Security Symposium*, pages 1075–1092. USENIX Association, 2020.

[FZZ+19] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. Scaling-up in-memory datalog processing: Observations and techniques. *Proc. VLDB Endow.*, 12(6):695–708, 2019.

[GAK12] Todd J. Green, Molham Aref, and Grigoris Karvounarakis. Logicblox, platform and language: A tutorial. In *Datalog*, volume 7494 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 2012.

[Gal14] François Le Gall. Powers of tensors and fast matrix multiplication. In *ISSAC*, pages 296–303. ACM, 2014.

[Gar97] Michael R Garey. Computers and intractability: A guide to the theory of np-completeness, freeman. *Fundamental*, 1997.

[GBT84] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, page 135–143, New York, NY, USA, 1984. Association for Computing Machinery.

[GGS14] Georg Gottlob, Gianluigi Greco, and Francesco Scarcello. Treewidth and hypertree width. *Tractability: Practical Approaches to Hard Problems*, 1, 2014.

[GGV02] Georg Gottlob, Erich Grädel, and Helmut Veith. Datalog lite: A deductive query language with linear time model checking. *ACM Transactions on Computational Logic (TOCL)*, 3(1):42–79, 2002.

[GHLZ13] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Found. Trends Databases*, 5(2):105–195, 2013.

[GK04] Georg Gottlob and Christoph Koch. Monadic datalog and the expressive power of languages for web information extraction. *J. ACM*, 51(1):74–113, 2004.

[GKLP17] Isaac Goldstein, Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Conditional lower bounds for space/time tradeoffs. In *WADS*, pages 421–436. Springer, 2017.

[GKT07] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, pages 31–40. ACM, 2007.

[GLS99] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In *SEBD*, pages 275–289, 1999.

[Gra95] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[GS13] Gianluigi Greco and Francesco Scarcello. Structural tractability of enumerating csp solutions. *Constraints*, 18(1):38–74, 2013.

[GSS+24] Thomas Gilray, Arash Sahebolamri, Yihao Sun, Sowmith Kunapaneni, Sidharth Kumar, and Kristopher K. Micinski. Datalog with first-class facts. *CoRR*, abs/2411.14330, 2024.

[HMAO24] Anna Herlihy, Guillaume Martres, Anastasia Ailamaki, and Martin Odersky. Adaptive recursive query optimization. In *ICDE*, pages 368–381. IEEE, 2024.

[Hu21] Xiao Hu. Cover or pack: New upper and lower bounds for massively parallel joins. In *PODS*, pages 181–198. ACM, 2021.

[Hu24a] Xiao Hu. Fast matrix multiplication for query processing. *Proceedings of the ACM on Management of Data*, 2(2):1–25, 2024.

[Hu24b] Xiao Hu. Output-optimal algorithms for join-aggregate queries. *CoRR*, abs/2406.05536, 2024.

[HW23] Xiao Hu and Qichen Wang. Computing the difference of conjunctive queries efficiently. *Proc. ACM Manag. Data*, 2023.

[HZJS21] Xiaowen Hu, David Zhao, Herbert Jordan, and Bernhard Scholz. An efficient interpreter for datalog by de-specializing relations. In *PLDI*, pages 681–695. ACM, 2021.

[IMNP24] Sungjin Im, Benjamin Moseley, Hung Q. Ngo, and Kirk Pruhs. On the convergence rate of linear datalogo over stable semirings. In *ICDT*, volume 290 of *LIPIcs*, pages 11:1–11:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[IT08] Zachary G. Ives and Nicholas E. Taylor. Sideways information passing for push-style query processing. In *ICDE*, pages 774–783. IEEE Computer Society, 2008.

[JR18] Manas Joglekar and Christopher Ré. It's all a matter of degree - using degree information to optimize multiway joins. *Theory Comput. Syst.*, 62(4):810–853, 2018.

[JS15] Riko Jacob and Morten Stöckel. Fast output-sensitive matrix multiplication. In *Algorithms-ESA 2015: 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pages 766–778. Springer, 2015.

[JSZS19] Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. Brie: A specialized trie for concurrent datalog. In *PMAM@PPoPP*, pages 31–40. ACM, 2019.

[Juk15] Stasys Jukna. Lower bounds for tropical circuits and dynamic programs. *Theory Comput. Syst.*, 57(1):160–194, 2015.

[JWZ24] Ce Jin, Virginia Vassilevska Williams, and Renfei Zhou. Listing 6-cycles. In *2024 Symposium on Simplicity in Algorithms (SOSA)*, pages 19–27. SIAM, 2024.

[JX23] Ce Jin and Yinzhan Xu. Removing additive structure in 3sum-based reductions. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, pages 405–418, 2023.

[KCM+20] Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, Xuanlong Nguyen, Dan Olteanu, and Maximilian Schleich. Functional aggregate queries with additive inequalities. *ACM Trans. Database Syst.*, 45(4), 2020.

[KDFZ25] Paraschos Koutris, Shaleen Deep, Austen Z. Fan, and Hangdong Zhao. The quest for faster join algorithms (invited talk). In *ICDT*, volume 328 of *LIPIcs*, pages 1:1–1:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

[KDOS24] Mahmoud Abo Khamis, Kyle Deeds, Dan Olteanu, and Dan Suciu. Pessimistic cardinality estimation. *SIGMOD Rec.*, 53(4):1–17, 2024.

[KJS+22] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. Learned cardinality estimation: An in-depth study. In *SIGMOD Conference*, pages 1214–1227. ACM, 2022.

[KK22] Bas Ketsman and Paraschos Koutris. Modern datalog engines. *Found. Trends Databases*, 12(1):1–68, 2022.

[KMRS17] Paraschos Koutris, Tova Milo, Sudeepa Roy, and Dan Suciu. Answering conjunctive queries with inequalities. *Theory Comput. Syst.*, 61(1):2–30, 2017.

[KNN+18] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. In-database learning with sparse tensors. In *PODS*, pages 325–340. ACM, 2018.

[KNN+19] Ahmet Kara, Hung Q Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Counting triangles under updates in worst-case optimal time. In *ICDT*, 2019.

[KNOS24] Mahmoud Abo Khamis, Vasileios Nakos, Dan Olteanu, and Dan Suciu. Join size bounds using l_p-norms on degree sequences. *Proc. ACM Manag. Data*, 2(2):96, 2024.

[KNOZ20] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. In *PODS*, pages 375–392, 2020.

[KNOZ23a] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Conjunctive queries with free access patterns under updates. In *ICDT*, 2023.

[KNOZ23b] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Evaluation trade-offs for acyclic conjunctive queries. In *CSL*, volume 252 of *LIPIcs*, pages 29:1–29:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[KNP+22a] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. Convergence of datalog over (pre-) semirings. In *PODS*, pages 105–117. ACM, 2022.

[KNP+22b] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. Datalog in wonderland. *SIGMOD Rec.*, 51(2):6–17, 2022.

[KNS17] 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 *PODS*, pages 429–444. ACM, 2017.

[KNS24] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. PANDA: query evaluation in submodular width. *CoRR*, abs/2402.02001, 2024.

[Knu77] Donald E Knuth. A generalization of dijkstra's algorithm. *Information Processing Letters*, 6(1):1–5, 1977.

[KOW21] Paraschos Koutris, Xiating Ouyang, and Jef Wijsen. Consistent query answering for primary keys on path queries. In *PODS*, pages 215–232. ACM, 2021.

[KOW24] Paraschos Koutris, Xiating Ouyang, and Jef Wijsen. Consistent query answering for primary keys on rooted tree queries. *Proc. ACM Manag. Data*, 2(2):76, 2024.

[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.

[Lan21] Matthias Lanzinger. Tractability beyond ß-acyclicity for conjunctive queries with negation. In *PODS 2021*, pages 355–369. ACM, 2021.

[LBC+24] Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E. Gonzalez, Ion Stoica, and Matei Zaharia. Optimizing LLM queries in relational workloads. *CoRR*, abs/2403.05821, 2024.

[LBKN14] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *SIGMOD Conference*, pages 743–754. ACM, 2014.

[LGM+15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[LGS13] Monica S. Lam, Stephen Guo, and Jiwon Seo. Socialite: Datalog extensions for efficient social network analysis. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, page 278–289, USA, 2013. IEEE Computer Society.

[LKP+18] Harald Lang, Andreas Kipf, Linnea Passing, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. In *DaMoN*, pages 5:1–5:8. ACM, 2018.

[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.

[LP22] Carsten Lutz and Marcin Przybylko. Efficiently enumerating answers to ontology-mediated queries. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 277–289, 2022.

[LSZ22] Yuanbo Li, Kris Satya, and Qirun Zhang. Efficient algorithms for dynamic bidirected dyck-reachability. *Proc. ACM Program. Lang.*, 6(POPL):1–29, 2022.

[LT14] John Liagouris and Manolis Terrovitis. Efficient identification of implicit facts in incomplete OWL2-EL knowledge bases. *Proc. VLDB Endow.*, 7(14):1993–2004, 2014.

[LWW18] Andrea Lincoln, Virginia Vassilevska Williams, and R. Ryan Williams. Tight hardness for shortest cycles and paths in sparse graphs. In *SODA*, pages 1236–1252. SIAM, 2018.

[LZR20] Yuanbo Li, Qirun Zhang, and Thomas Reps. Fast graph simplification for interleaved dyck-reachability. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 780–793, 2020.

[LZR21] Yuanbo Li, Qirun Zhang, and Thomas Reps. On the complexity of bidirected interleaved dyck-reachability. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–28, 2021.

[Mai83] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[Mar79] HG Marc. On the universal relation. Technical report, Technical report, University of Toronto, 1979.

[Mar10] Dániel Marx. Can you beat treewidth? *Theory Comput.*, 6(1):85–112, 2010.

[Mar13] Dániel Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *J. ACM*, 60(6):42:1–42:51, 2013.

[MLSR20] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared arrangements: practical inter-query sharing for streaming dataflows. *Proc. VLDB Endow.*, 13(10):1793–1806, 2020.

[MMI+13] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455. ACM, 2013.

[MMII13] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.

[MNM+19] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, 2019.

[MNM+21] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In *SIGMOD Conference*, pages 1275–1288. ACM, 2021.

[MP21] Anders Alnor Mathiasen and Andreas Pavlogiannis. The fine-grained and parallel complexity of andersen's pointer analysis. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

[MPPV04] Benjamin J. McMahan, Guoqiang Pan, Patrick Porter, and Moshe Y. Vardi. Projection pushing revisited. In *EDBT*, volume 2992 of *Lecture Notes in Computer Science*, pages 441–458. Springer, 2004.

[MTKW18] David Maier, K. Tuncay Tekle, Michael Kifer, and David Scott Warren. Datalog: concepts, history, and outlook. In *Declarative Logic Programming*, volume 20 of *ACM Books*, pages 3–100. ACM / Morgan & Claypool, 2018.

[MU04] Kazuhisa Makino and Takeaki Uno. New algorithms for enumerating all maximal cliques. In *Algorithm Theory-SWAT 2004: 9th Scandinavian Workshop on Algorithm Theory, Humlebæk, Denmark, July 8-10, 2004. Proceedings 9*, pages 260–272. Springer, 2004.

[MWC23] Muhammad Numair Mansur, Valentin Wüstholz, and Maria Christakis. Dependency-aware metamorphic testing of datalog engines. In *ISSTA*, pages 236–247. ACM, 2023.

[MYL16] Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. From datalog to flix: a declarative language for fixed points on lattices. In *PLDI*, pages 194–208. ACM, 2016.

[ND] Joris Nix and Jens Dittrich. Extending sql to return a subdatabase.

[NK15] Thomas Neumann and Alfons Kemper. Unnesting arbitrary queries. In *BTW*, volume P-241 of *LNI*, pages 383–402. GI, 2015.

[NPRR18] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3), 2018.

[NRR13] Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013.

[OS16] Dan Olteanu and Maximilian Schleich. Factorized databases. *ACM SIGMOD Record*, 45(2):5–16, 2016.

[OZ15] Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2:1–2:44, 2015.

[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. *Proc. VLDB Endow.*, 11(6):663–676, 2018.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in starburst. In *SIGMOD Conference*, pages 39–48. ACM Press, 1992.

[PR10] Mihai Patrascu and Liam Roditty. Distance oracles beyond the thorup-zwick bound. In *FOCS*, pages 815–823. IEEE, 2010.

[Raz03] Ran Raz. On the complexity of matrix product. *SIAM J. Comput.*, 32(5):1356–1369, 2003.

[RB19] Leonid Ryzhyk and Mihai Budiu. Differential datalog. In *Datalog*, volume 2368 of *CEUR Workshop Proceedings*, pages 56–67. CEUR-WS.org, 2019.

[Rep98] Thomas W. Reps. Program analysis via graph reachability. *Inf. Softw. Technol.*, 40(11-12):701–726, 1998.

[RMS21a] Yann Ramusat, Silviu Maniu, and Pierre Senellart. A practical dynamic programming approach to datalog provenance computation. *CoRR*, abs/2112.01132, 2021.

[RMS21b] Yann Ramusat, Silviu Maniu, and Pierre Senellart. Provenance-based algorithms for rich queries over graph databases. In *EDBT*, pages 73–84. OpenProceedings.org, 2021.

[RPE+17] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César A. Galindo-Legaria, and Conor Cunningham. Froid: Optimization of imperative programs in a relational database. *Proc. VLDB Endow.*, 11(4):432–444, 2017.

[RS84] Neil Robertson and Paul D. Seymour. Graph minors. III. planar tree-width. *J. Comb. Theory B*, 36(1):49–64, 1984.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *SIGMOD Conference*, pages 23–34. ACM, 1979.

[SB15] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, 2015.

[SBMM23] Arash Sahebolamri, Langston Barrett, Scott Moore, and Kristopher K. Micinski. Bring your own data structures to datalog. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1198–1223, 2023.

[Seg13] Luc Segoufin. Enumerating with constant delay the answers to a query. In *ICDT*, pages 10–20. ACM, 2013.

[SEV16] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. Inca: a dsl for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 320–331, New York, NY, USA, 2016. Association for Computing Machinery.

[SGM22] Arash Sahebolamri, Thomas Gilray, and Kristopher K. Micinski. Seamless deductive inference via macros. In *CC*, pages 77–88. ACM, 2022.

[SJC+18] Pavle Subotic, Herbert Jordan, Lijun Chang, Alan D. Fekete, and Bernhard Scholz. Automatic index selection for large-scale datalog computation. *Proc. VLDB Endow.*, 12(2):141–153, 2018.

[SJSW16] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC '16, pages 196–206, New York, NY, USA, 2016. ACM.

[SOC16] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *SIGMOD Conference*, pages 3–18. ACM, 2016.

[SPSL13] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proc. VLDB Endow.*, 6(14):1906–1917, 2013.

[SSG+25] Yihao Sun, Ahmedur Rahman Shovon, Thomas Gilray, Sidharth Kumar, and Kristopher K. Micinski. Optimizing datalog for the GPU. In *ASPLOS (1)*, pages 762–776. ACM, 2025.

[SSSN24] Amir Shaikhha, Dan Suciu, Maximilian Schleich, and Hung Q. Ngo. Optimizing nested recursive queries. *Proc. ACM Manag. Data*, 2(1):16:1–16:27, 2024.

[SYI+16] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *SIGMOD*, pages 1135–1149. ACM, 2016.

[TAG+20] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *Proc. VLDB Endow.*, 13(9):1582–1597, 2020.

[TCG+23] Nikolaos Tziavelis, Nofar Carmeli, Wolfgang Gatterbauer, Benny Kimelfeld, and Mirek Riedewald. Efficient computation of quantiles over joins. In *PODS*, pages 303–315. ACM, 2023.

[TGR20] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Optimal join algorithms meet top-k. In *SIGMOD Conference*, pages 2659–2665. ACM, 2020.

[UG88] Jeffrey D. Ullman and Allen Van Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3:5–42, 1988.

[Val74] Leslie Valiant. General context-free recognition in less than cubic time. 1974.

[Var82] Moshe Y Vardi. The complexity of relational query languages. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 137–146, 1982.

[Vel12] Todd L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR*, abs/1210.0481, 2012.

[Vui80] Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, Apr 1980.

[War62] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962.

[WBHB25] Johannes Wehrstein, Tiemo Bang, Roman Heinrich, and Carsten Binnig. Graceful: A learned cost estimator for udfs. *arXiv preprint arXiv:2503.23863*, 2025.

[Wil12] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *STOC*, pages 887–898, 2012.

[WKN+22] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, and Dan Suciu. Optimizing recursive queries with progam synthesis. In *SIGMOD Conference*, pages 79–93. ACM, 2022.

[WW18] Virginia Vassilevska Williams and R Ryan Williams. Subcubic equivalences between path, matrix, and triangle problems. *Journal of the ACM (JACM)*, 65(5):1–38, 2018.

[WWS23] Yisu Remy Wang, Max Willsey, and Dan Suciu. Free join: Unifying worst-case optimal and traditional joins. *Proc. ACM Manag. Data*, 1(2):150:1–150:23, 2023.

[WWZ22] Jiacheng Wu, Jin Wang, and Carlo Zaniolo. Optimizing parallel recursive datalog evaluation on multicore machines. In *SIGMOD Conference*, pages 1433–1446. ACM, 2022.

[WXXZ24] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In *SODA*, pages 3792–3835. SIAM, 2024.

[WY21] Yilei Wang and Ke Yi. Secure yannakakis: Join-aggregate queries over private data. In *SIGMOD Conference*, pages 1969–1981. ACM, 2021.

[WY22] Yilei Wang and Ke Yi. Query evaluation by circuits. In *PODS '22*, pages 67–78. ACM, 2022.

[WY23] Qichen Wang and Ke Yi. Conjunctive queries with comparisons. *SIGMOD Rec.*, 52(1):54–62, 2023.

[XD17] 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, 2017.

[Yan81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, page 82–94. VLDB Endowment, 1981.

[Yan90] Mihalis Yannakakis. Graph-theoretic methods in database theory. In *PODS*, pages 230–242. ACM Press, 1990.

[Yan22] Renchi Yang. Efficient and effective similarity search over bipartite graphs. In *Proceedings of the ACM Web Conference 2022*, pages 308–318, 2022.

[Yao82] Andrew Chi-Chih Yao. Space-time tradeoff for answering range queries (extended abstract). In *STOC*, pages 128–136. ACM, 1982.

[Yao85] Andrew Chi-Chih Yao. On the complexity of maintaining partial sums. *SIAM J. Comput.*, 14(2):277–288, 1985.

[YO79] Clement Tak Yu and Meral Z Ozsoyoglu. An algorithm for tree-query membership of a distributed query. In *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979.*, pages 306–312. IEEE, 1979.

[YZYK24] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. Predicate transfer: Efficient pre-filtering on multi-join queries. In *CIDR*. www.cidrdb.org, 2024.

[ZDK23a] Hangdong Zhao, Shaleen Deep, and Paraschos Koutris. Space-time tradeoffs for conjunctive queries with access patterns. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '23, pages 59–68, New York, NY, USA, 2023. Association for Computing Machinery.

[ZDK23b] Hangdong Zhao, Shaleen Deep, and Paraschos Koutris. Space-time tradeoffs for conjunctive queries with access patterns. *arXiv preprint arXiv:2304.06221*, 2023.

[ZDK+24a] Hangdong Zhao, Shaleen Deep, Paraschos Koutris, Sudeepa Roy, and Val Tannen. Evaluating datalog over semirings: A grounding-based approach. *Proceedings of the ACM on Management of Data*, 2(2):1–26, 2024.

[ZDK+24b] Hangdong Zhao, Shaleen Deep, Paraschos Koutris, Sudeepa Roy, and Val Tannen. Evaluating datalog over semirings: A grounding-based approach, 2024.

[ZFOK23] Hangdong Zhao, Austen Z. Fan, Xiating Ouyang, and Paraschos Koutris. Conjunctive queries with negation and aggregation: A linear time characterization. *CoRR*, abs/2310.05385, 2023.

[ZKD24] Hangdong Zhao, Paraschos Koutris, and Shaleen Deep. Evaluating datalog via structure-aware rewriting. In *Datalog*, volume 3801 of *CEUR Workshop Proceedings*, pages 48–53. CEUR-WS.org, 2024.

[ZMK+25] Haozhe Zhang, Christoph Mayer, Mahmoud Abo Khamis, Dan Olteanu, and Dan Suciu. Lpbound: Pessimistic cardinality estimation using l_p-norms of degree sequences. *CoRR*, abs/2502.05912, 2025.

[ZPSP17] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. Looking ahead makes query plans robust. *Proc. VLDB Endow.*, 10(8):889–900, 2017.

[ZSRS21] David Zhao, Pavle Subotic, Mukund Raghothaman, and Bernhard Scholz. Towards elastic incrementalization for datalog. In *PPDP*, pages 20:1–20:16. ACM, 2021.

[ZSY+25] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. Debunking the myth of join ordering: Toward robust sql analytics. *arXiv preprint arXiv:2502.15181*, 2025.

[ZWF+23] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. Better together: Unifying datalog and equality saturation. *Proc. ACM Program. Lang.*, 7(PLDI):468–492, 2023.

[ZXW+16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.