Data-Aware Complexity Analysis and Program Optimization

Kyle Deeds

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

Doctor of Philosophy

University of Washington

2025

Reading Committee:

Dan Suciu, Chair

Magda Balazinska, Chair

Gilbert Bernstein

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science and Engineering

University of Washington

Abstract

Data-Aware Complexity Analysis and Program Optimization

Kyle Deeds

Co-Chairs of the Supervisory Committee:
Dan Suciu
Paul G. Allen School of Computer Science and Engineering

Magda Balazinska
Paul G. Allen School of Computer Science and Engineering

This dissertation explores the problem of analyzing and optimizing data-dependent programs from a theoretical and practical perspective. The performance of these programs depends in a complex manner on the distribution of the input data, and they arise in many contexts, e.g. databases, sparse tensor programming, and graph analytics. By definition, these programs cannot be optimized by considering the code alone, so an optimizer for them must consider information about the data distribution. This dissertation presents two new theoretical approaches for analyzing data-dependent programs by bounding the size of their intermediate results: the degree sequence bound and partition constraints. It then describes two practical systems for producing these bounds: SafeBound and COLOR. Lastly, we present a state-of-the-art optimizer for sparse tensor programs, Galley, that demonstrates the value of data-aware optimization.

TABLE OF CONTENTS

Page

List of Figures iv

List of Tables vi

Chapter 1: Introduction 1
1.1 Motivation and Contributions 5
1.2 Chapter Organization & Publications 10

Chapter 2: Background 12
2.1 A Brief History of Cardinality Estimation 12
2.2 The Sparse Tensor Programming Ecosystem 22

Chapter 3: Degree Sequence Bound 27
3.1 Problem Statement 32
3.2 The Star Query 37
3.3 The Berge-Acyclic Query 47
3.4 Comparison with the AGM and Polymatroid Bounds 51
3.5 Functional Representation 55
3.6 Conclusions 59

Chapter 4: Partition Constraints 60
4.1 Preliminaries 62
4.2 Partition Constraints 64
4.3 The Hexagon Query 69
4.4 Partition Constraints for General Conjunctive Queries 74
4.5 Conclusions and Future Work 82

Chapter 5: SafeBound 83
5.1 Abbreviated Description of the Degree Sequence Bound 85
5.2 SafeBound 87

5.3 Optimizations . . . 100
5.4 Evaluation . . . 103
5.5 Conclusion & limitations . . . 115

Chapter 6: COLOR . . . 116
6.1 Problem Setting . . . 119
6.2 Colorings & Lifted Graphs . . . 121
6.3 Lifted Subgraph Counting . . . 124
6.4 Handling Cycles . . . 127
6.5 Alternate Coloring Methods . . . 129
6.6 Optimization . . . 131
6.7 Evaluation . . . 137
6.8 Conclusion . . . 150

Chapter 7: Galley . . . 151
7.1 Background & Connection to RA . . . 154
7.2 Galley Overview . . . 161
7.3 Logical Optimizer . . . 163
7.4 Physical Optimizer . . . 170
7.5 Sparsity Estimation . . . 173
7.6 Experimental Evaluation . . . 177
7.7 Related Work . . . 186
7.8 Limitations . . . 186

Chapter 8: Conclusion & Future Work . . . 188

Bibliography . . . 191

Appendix A: Degree Sequence Bounds Appendix . . . 209
A.1 Discussion of the Framework . . . 209
A.2 Completing Proofs from Sec. 3.2 . . . 210
A.3 Completing Proofs from Sec. 3.3 . . . 228
A.4 Completing Proofs from Sec. 3.5 . . . 238

Appendix B: Partition Constraints Appendix . . . 251
B.1 Further Details for Section 4.3 . . . 251
B.2 Further Details for Section 4.4 . . . 256

Appendix C: COLOR Appendix 260
C.1 Proof of Thm. 6.3.5 260
C.2 Proof of Theorem 6.6.1 262

LIST OF FIGURES

Figure Number Page

1.1 Sparse Matrix Chain Multiplication (ABC) . . . 3
1.2 Partition Constraints Example. . . . 6

2.1 Example join query. . . . 13
2.2 Fiber-Tree Abstraction . . . 24

3.1 Example Degree Sequence. . . . 28
3.2 Example Algorithm Execution . . . 51
3.3 Incidence Graph . . . 51

4.1 Graph Example. . . . 65
4.2 Access Example. . . . 65
4.3 Example PCs. . . . 68
4.4 The Query $Q_{\bigcirc}$. . . . 68

5.1 An example column, $R.V$, and its degree sequence f. . . . 85
5.2 Example worst-case instance. . . . 86
5.3 Compressed degree sequence example. . . . 89
5.4 A tree decomposition for Q in Example 5.2.8 . . . 97
5.5 Workload runtimes, planning time, and estimation error . . . 106
5.6 The runtime of the 80 longest-running queries across all benchmarks. . . . 107
5.7 The average runtime of queries binned by their runtime using Postgres' estimates. . . . 108
5.8 Statistics size and Construction Time . . . 109
5.9 The frequency and magnitude of FK index performance regressions. . . . 111
5.10 CDS modeling micro-benchmark. . . . 111
5.11 CDS clustering microbenchmark. . . . 112
5.12 Construction scaling experiment. . . . 114

6.1 Lifted counting example. . . . 117
6.2 Accuracy of coloring as the number of colors increases . . . 118
6.3 Relative Error by Estimator . . . 137

6.4 Inference Time by Estimator . . . 138
6.5 Relative Error by Cardinality Bound Method . . . 138
6.6 Inference Time by Cardinality Bound Method . . . 139
6.7 Statistics Size . . . 144
6.8 Build Time . . . 145
6.9 Relative Error by Coloring Method . . . 145
6.10 Construction Scaling . . . 146
6.11 Relative Error vs Proportion Updated (Human) . . . 148
6.12 Relative Error vs Max Cycle Length Stored (Youtube) . . . 148
6.13 Inference Time vs Query Pathwidth (Youtube) . . . 149
6.14 Relative Error vs Samples (Youtube) . . . 149

7.1 Logistic regression implemented in the language of a sparse tensor compiler. . . 152
7.2 Galley overview. . . . 156
7.3 Fibertree format abstraction. . . . 156
7.4 Query plan dialects. . . . 157
7.5 Annotated expression tree for $\sigma(\sum_{jpc} S_{ipc}(P_{pj}\theta_j + C_{cj}\theta_j))$ . . . 167
7.6 ML Inference Over Joins . . . 177
7.7 Linear Algebra Kernels . . . 178
7.8 Subgraph Counting Experiments . . . 182
7.9 BFS Execution Time . . . 185

B.1 Depiction of $R_{X,Y,Z}$ for $n = 27$ . . . 251
B.2 Depiction of $C_{X,Y,Z}$ for $\frac{n}{7} = 16$ . . . 254

LIST OF TABLES

Table Number Page

6.1 Notation Dictionary 124
6.2 Estimator Failure Rates per dataset. 134
6.3 Experimental Datasets. 140

7.1 Experimental Dataset Sizes 179
7.2 Total Subgraph Counting Execution Time (S) 181

ACKNOWLEDGMENTS

The work in this dissertation would not have been possible without the diligent mentorship of my advisors Dan Suciu and Magda Balazinska. At the start of my program, I did not know what a join was, and Dan showed me that it is actually the only important operation. Through long sessions at the whiteboard, Dan would carefully and excitedly demonstrate to me that database joins were connected to the deepest mysteries of mathematics. Equally importantly, through workshops and conferences, Dan introduced me to a wonderful community of database theorists. I had never expected this seemingly obscure field to feel like a home, but I am grateful that it has. Magda's guidance in navigating the concerns of the systems community would prove essential as well, and her advice on writing and presenting has made me a fundamentally better scientific communicator.

However, this work would also not have been possible without the incredible PhD students that I collaborated with along the way. Willow Ahrens, Moe Kayali, Diandre Sabale, and Camillo Merkl were truly indispensable collaborators, generous with both their time and their incredible insights. From Willow, I not only learned everything that I know about compilers, but I also learned valuable lessons in how to produce real, useful software in academia. Galley simply could not have happened without Willow's dedicated work and guidance. Moe was my closest friend in the program, and his work on quasi-stable coloring was the ground upon which we built COLOR. I met Diandre when he was a wee undergrad, and, over the course of building COLOR, it was incredible to see him come into his own as a researcher. I can't wait to see what he'll do next. Camillo is simply one of the most brilliant and personable theoreticians that I have ever met, and his contributions to the partition constraints paper were immense.

DEDICATION

To my mom, who has been an endless source of encouragement and support and whose startup ideas remain promising. To my dad, who taught me to ask the right questions and whose footsteps I am following with a concerning diligence. To my brothers, who instilled a sense of humility in me, likely saving me from becoming truly awful. Finally, to my partner Alison, who lifted me up on hard days, encouraged me to celebrate each little victory, and politely listened to far more database nonsense than any sane person should be forced to.

Chapter 1

INTRODUCTION

It is hard to overstate the importance of query optimization to the widespread adoption of database systems over the past five decades. By automatically producing efficient implementations for declarative user programs, query optimizers shield users from the complex performance problems that arise when manipulating large datasets. This permits an ergonomic, uncluttered programming paradigm that has been readily adopted by users, allowing them to avoid the hazards of general programming languages [134].

The central challenge in query optimization is estimating the runtime of alternative query plans. This is difficult because their execution fundamentally depends on the characteristics of the data. Queries executed on a social graph with close knit communities will have a different performance when executed on a company's hierarchical organizational chart. To find the optimal plan, the optimizer cannot simply examine the query; it must also take into account the distribution of the data, or statistics thereof. While this challenge originates from the study of database query optimizers, many kinds of computation are fundamentally data-dependent and can benefit from this approach.

The object of this dissertation is to **analyze and optimize data-dependent programs by introducing new data-aware complexity analyses and building a data-aware optimizer for sparse tensor programs**.

In Chapters 3- 5 of this dissertation, we address the problem of analyzing the complexity of data-dependent programs. However, rather than directly analyzing the programs, we focus on estimating the size of intermediate results. Once those sizes are known, many kinds of computation become easy to analyze. For example, if we know the number of rows in both the input and output of a merge join, then the complexity of the operation is just their sum. If there's a tree of merge join operations, then the output size of each node fully determines the complexity of the whole tree. This is why estimating these sizes is the core

of the cost model in database query optimizers. Further, the size of intermediate results in many programming paradigms can be defined using relational algebra (i.e. database queries). This allows us to focus on estimating these sizes in relational algebra and translate the results to a wide variety of domains.

In general, estimating the size of queries is referred to as *cardinality estimation*, but we focus on the stricter problem of proving upper bounds on the size of query outputs, which are referred to as *cardinality bounds*. Previous approaches for cardinality bounding relied on coarse statistics which resulted in loose bounds [10, 90]. Suppose we wanted to bound the output of the join between two tables which each have N rows. With only this information, we cannot prove that every row in both tables does not share the same join value, so we cannot prove that the cardinality will be less than N^2. This is a problem because most real-world joins will produce asymptotically smaller outputs. More generally, this kind of coarse statistic cannot be used for practical program optimization because it cannot distinguish between efficient and inefficient implementations.

In this dissertation, we introduce three novel, more granular statistics and show how they can be used to produce tight cardinality bounds. In Ch. 3, we present the *degree sequence bound*, the first cardinality bound to incorporate complete information about the skew of the join columns [38]. We prove that this bound is tight for the most common class of queries, i.e. cannot be made lower without additional information, and that it is efficient to compute, requiring at most $O(\sqrt{N})$ time where N is the tables' size. Following this, Ch. 4 introduces *partition constraints*, a statistic that captures deep structure in tables [37]. This work is inspired by the notion of graph degeneracy which has been shown to be both prevalent in data and useful for improving algorithms like triangle counting[19, 21, 20]. In the database setting, it addresses the issue of tables containing subsets of rows that follow very different distributions. We show that these statistics can be used to produce asymptotically tighter bounds than previous methods, and we design novel join algorithms whose runtime never exceeds these new bounds. Lastly, in Ch. 6, we focus on the graph setting, and we prove that stable colorings are sufficient to exactly compute the number of occurrences of an acyclic pattern graph in a larger data graph.

In Ch. 5 and Ch. 6, we go beyond theory and implement our bounds in real systems,

```sql
SELECT A.I, C.L, SUM(A.V*B.V*C.V)
FROM A, B, C
WHERE A.J = B.J AND B.K = C.K
GROUP BY A.I, C.L
```

(a) SQL Version

```python
import scipy.sparse as sp
D = A @ B @ C
```

(b) Python Version

Figure 1.1: Sparse Matrix Chain Multiplication (ABC)

SafeBound and COLOR [43, 41]. The former implements the degree sequence bound, and the latter adapts an approximation of stable colorings. To produce bounds for real queries, we adapt our statistics to handle complex query predicates and optimize the latency of computing the bounds. By doing this, we show that these bounds can provide competitive accuracy, latency, and memory footprint while ensuring principled guarantees.

To see the necessity of data-aware analysis for understanding and optimizing data-dependent programs, consider the following example based on sparse linear algebra.

Example 1.0.1. *Consider the sparse matrix chain multiplication described in Fig. 1.1a and, equivalently, in Fig. 1.1b. Note that sparse matrices are stored in a compressed format that only records the non-zero entries, and operations over them run in time relative to the number of non-zero entries. There are two alternative query plans, $(AB)C$ and $A(BC)$, for this program that differ in the order that the matrix multiplications are handled. The goal would be for a program optimizer to intelligently choose between these plans to choose the fastest. However, as we'll see below, this is not possible without carefully considering the sparsity patterns (i.e. data distribution) of the input matrices. Now, suppose that you have*

the following data as input,

$$ABC = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

On this data, the $(AB)C$ plan first materializes a dense matrix, then multiplies this with C. Assuming only non-zero entries are iterated over, both steps require $O(n^2)$ floating point operations to compute. The $A(BC)$ plan is much more efficient because it first materializes BC which has only a single non-zero entry, then multiplies this single-entry matrix with C. In this plan, both steps require only $O(n)$ operations.

A traditional complexity analysis that considers the size of the input matrices but not their structure is unable to differentiate between these plans. In the worst case, the multiplication of two sparse matrices with $O(n)$ elements will require $O(n^2)$ operations, resulting in an $O(n^2)$ bound for the first step of both programs. With the techniques described in this dissertation, we can use more detailed statistics on input data to automatically identify the better plan, enabling a new form of program optimization for sparse linear algebra and its generalization, sparse tensor programming.

Chapter 7 builds on the tools and intuition found in the prior chapters and introduces Galley, a state-of-the-art optimizer for sparse tensor programs [36]. To build this system, we begin by performing a deep analysis of sparse tensor algebra (STA) expressions with arbitrary pointwise and aggregate operations, and we use this to create a formal cost model for sparse tensor programs. This model allows us to apply the data-aware complexity analysis that we refined in the previous chapters. Building on this, we formalize the search space for equivalent STA expressions, and we provide an efficient dynamic programming algorithm to optimize over it. Lastly, we show how to translate optimized STA expressions to efficient implementations using a state-of-the-art sparse tensor compiler [3]. The end result is a new kind of optimizer for sparse tensor programs that produces up to 300× faster execution compared to production systems like PyTorch in experiments with our prototype

system.

In the following section, I present the motivation and contributions for each project in more detail. This will go some way towards situating them with respect to prior work, but a full account of related work and necessary background material will be deferred to Ch. 2.

1.1 Motivation and Contributions

Over the last decade, there has been a growing interest in cardinality bounds from the theoretical community. These bounds take in a query and statistics on a database instance, and they produce a bound on the size of the query's output when evaluated on that instance. The interest in them is largely due to their role in bounding the time complexity of query evaluation. The first non-trivial cardinality bound, the AGM bound, only relied on the size of the relations[10]. Later work in this area would incorporate more detailed statistics about the input relations, beginning with functional dependencies then generalizing them to *degree constraints*[89, 90]. This produced exciting developments in query processing as each worst-case bound inspired join algorithms that were optimal relative to it like Generic Join, Leapfrog Triejoin and PANDA [112, 141, 90]. However, these bounds remained too loose for use in practical query optimization [30, 43].

To address this problem, Chapter 3 introduces ***the degree sequence bound*, a novel cardinality bound that incorporates detailed information about the skew of join columns** [38]. As the name implies, the core of this bound is the degree sequence statistic, which records the distribution of frequencies within a join column. In other words, the degree sequence captures the skew within a column, and the bound formalizes the common intuition that joining on a column with low skew cannot produce an explosion in join results. This work pioneered the use of the degree sequence for cardinality estimation, and it inspired several later works which used this idea [87, 158].

For acyclic queries, we show that the bound is tight and strictly lower than prior bounds. Informally, this means that it is impossible to prove a lower bound without gathering more detailed statistics on the tables. On the practical side, the degree sequence bound can be computed in linear time with respect to the size of the statistics. We further show that these can be losslessly compressed to at most $O(\sqrt{N})$ where N is the table size and lossily

Access

PersonID	RoomID
Ava	Beacon Hall
Ben	Beacon Hall
Amir	Beacon Hall
Cole	Delta Hall
Dan	Delta Hall
Emma	Gala Hall
Ella [Janitor]	Beacon Hall
Ella [Janitor]	Delta Hall
Ella [Janitor]	Gala Hall

$\text{Access}^{\text{PersonID}}$

PersonID	RoomID
Ava	Beacon Hall
Ben	Beacon Hall
Amir	Beacon Hall
Cole	Delta Hall
Evelyn	Delta Hall
Dan	Delta Hall
Emma	Gala Hall

$\text{Access}^{\text{RoomID}}$

PersonID	RoomID
Ella [Janitor]	Beacon Hall
Ella [Janitor]	Delta Hall
Ella [Janitor]	Gala Hall

Figure 1.2: Partition Constraints Example.

compressed even further while maintaining a valid bound.

In Chapter 4, we introduce ***partition constraints*, which identify a deeper structure within relations, in order to produce asymptotically tighter cardinality bounds and faster join algorithms** [37]. The intuition is that relations can often be partitioned such that each piece has a join column with low skew. By partitioning all relations in this way, any query can be decomposed into a union of queries over combinations of partitions. The benefit of this is that each partition query has additional constraints on the data, and it can be bounded and executed independently in manner that takes advantage of those additional constraints.

Example 1.1.1. *Consider a table that tracks people's access to classrooms in a university as shown in Fig. 1.2. Each row represents one person having access to one classroom.* `RoomID` *has high skew because many students enter each classroom, e.g. Beacon Hall appears many times because it hosts a class with three students.* `PersonID` *also has high skew because the janitor, Ella, needs to enter every classroom. In this small example, the most frequent person and classroom each only appear three times, but on large datasets this skew could be much worse with elements that appear thousands or millions of times. Now, suppose that*

we want to compute the following query that identifies people with access to rooms that their parents donated to:

$$Q(P, S, R) := Parent(P, S) \bowtie Access(S, R) \bowtie DonatedTo(R, P)$$

Because the full Access table has high skew in both columns, the query optimizer, which is attempting to avoid large intermediate results, will struggle to choose which two tables to join first. By partitioning on the basis of janitors vs students, one partition has low skew in the `RoomID` *column and one has low skew in the* `PersonID` *column. We can now split the query into two parts:*

$$Q^{RoomID}(P, S, R) := Parent(P, S) \bowtie Access^{RoomID}(S, R) \bowtie DonatedTo(R, P)$$

$$Q^{PersonID}(P, S, R) := Parent(P, S) \bowtie Access^{PersonID}(S, R) \bowtie DonatedTo(R, P)$$

When dealing with each one, the query optimizer can leverage the low skew present in each partition. For example, the $Access^{RoomID}(S, R)DonatedTo(R, P)$ *join produces a small output because* $RoomID$ *has low skew in this partition.*

We introduce partition constraints (PCs) to denote how well a table can be partitioned to reduce its skew. A relation with a low PC can be successfully partitioned into pieces with low skew while a table with a high PC cannot be usefully decomposed. Surprisingly, we find that the problem of computing PCs, along with the optimal partitions, is highly tractable. We present a quadratic time exact algorithm and a linear time approximate algorithm.

Chapter 5 focuses on moving cardinality bounds from theory to practice which requires overcoming several challenges. First, the theory community typically considers conjunctive queries without selection predicates, i.e. SQL queries with only SELECT and FROM clauses. Real SQL queries involve complicated selection predicates that drastically impact the output size, as reflected in cardinality estimation benchmarks [99, 71]. For example, a user might want to filter a relation based on an age range or a string prefix. Due to the constraints of database architectures, the query optimizer typically cannot access the relations at query-time, so these predicates need to be handled based on statistics that are gathered offline. Second, query optimizers operate on extremely limited time and space

budgets, so any practical system for cardinality bounding should operate in milliseconds and megabytes.

As an answer to these problems, **Chapter 5 introduces a system called SafeBound that produces tight degree sequence bounds for complex real-world queries** [43]. SafeBound handles selection predicates by adapting traditional data summaries such as histograms, most-common value lists, and n-grams to produce degree sequences rather than selectivity estimates. To maintain strict upper bounds, we needed to fundamentally rework these summaries and provide novel techniques to handle basic issues, e.g. range predicates that span multiple histogram buckets. To minimize the space and time of the estimator, we develop the first algorithm for compressing degree sequences, and we clustering and filtering to save even more space. This results in an estimator which reduces overall runtimes by $20 - 85\%$, uses $3 - 6.8\times$ space, and is $3 - 500\times$ faster compared to competing methods in experiments with our prototype implementation.

Building on exciting recent work in stable colorings (eq. Weisfeiler-Leman colorings) [63], Chapter 6 applies a combination of theory and system techniques to cardinality estimation in graph databases. The workloads associated with these databases typically consist of many more joins per query and tend to include more cyclic joins than traditional relational workloads. Traditional cardinality estimation methods that rely on uniformity and independence assumptions are poorly suited to this challenge, and they tend to drastically underestimate[99, 116]. In general, graph datasets feature complex structures that dramatically affect the cardinality of subgraph matching queries, which form the core of graph query languages like GQL and Cypher[55, 44].

To address this problem, **Chapter 6 presents COLOR, a system that leverages the theory of stable colorings to compute cardinality bounds and estimates for queries over graphs** [41]. Stable colorings were originally introduced to address the problem of graph isomorphism [65, 63]. By checking for isomorphism on the smaller model (i.e. the weighted graph over the colors), non-isomorphic graphs can be quickly rejected without performing a full isomorphism check. In our work, we prove that the model produced by a stable coloring can compute the exact cardinality of all acyclic queries. This is effective when the stable coloring produces a small model. However, in practice, the stable

coloring of a graph often requires a very large number of colors [85]. This results in a large model and poses a problem for cardinality estimation which requires minimal inference latency and memory usage. To reduce the size of the model, we employ approximation techniques, and we show that it remains highly accurate even when graphs with millions of nodes are represented with 10s of colors. To improve estimation time, we adapt state of the art techniques for both aggregation and sampling, reducing the inference runtime exponentially. Lastly, to improve accuracy on cyclic queries, we propose a novel statistic called cycle closure probability and tightly integrate it with our small graph model. This results in a system with a median error < 10 on workloads with queries that have 100s of edges and which requires $80 - 800\times$ less space than competing methods.

Finally, Chapter 7 applies these techniques to optimize sparse tensor programs, resulting in a system called Galley[36]. Tensor programming is typically done using the Python Array API where users define an imperative graph of high-level operators such as taking the maximum over a dimension or multiplying two matrices. Existing dense tensor programming systems, including PyTorch and NumPy, largely respect this compute graph, at best performing straightforward optimizations such as fusing operators or culling dead code[118, 1, 73]. This places users in the role of performance engineer, requiring them to carefully construct an optimized compute graph. In the sparse setting, sparse tensor compilers are the state of the art, and they require the user to make additional decisions about loop ordering and data layout[95, 3]. Producing optimal implementations in these systems can be highly complicated and requires a trial-and-error approach, even for expert users. To automate these decisions, we need to model the cost of different implementations, and, much like in databases, this comes down to estimating the size (i.e. sparsity) of intermediate results.

Building on our prior work, **Galley performs data-aware program optimization for sparse tensor programs, resulting in state-of-the-art performance**. In doing so, it allows users to define their programs using the familiar and concise Python Array API while receiving the same kinds of performance optimization as database users. We divide this optimization problem into two stages; 1) logical optimization which determines the order that aggregates are performed in and when intermediates are materialized 2)

physical optimization which makes lower-level decisions about the loop order and output tensor format for each aggregation. The former extends ideas from database theory like FAQs to the more general setting of arbitrary sparse tensor algebra [88]. The latter leverages the similarity of sparse tensor kernels to worst-case optimal join algorithms to model the cost of different loop orders[112, 142]. Both optimizers rely on a shared system for sparsity estimation that is grounded in the techniques developed throughout this dissertation for data-aware complexity analysis and cardinality bounding. By applying data-aware optimization, Galley achieves up to 300× faster execution for ML inference over joins compared to standard methods and up to 200× faster execution on sparse linear algebra kernels compared with PyTorch.

1.2 Chapter Organization & Publications

1. Chapter 2 provides notation, background, and related work.

2. Chapter 3 introduces and analyzes the Degree Sequence Bound.

3. Chapter 4 defines partition constraints, provides algorithms for computing them, and shows how they can lead to asymptotically improved cardinality bounds and join algorithms.

4. Chapter 5 describes SafeBound, a practical system for computing the degree sequence bound for complex queries over real data.

5. Chapter 6 presents the system COLOR which builds on stable colorings to produce cardinality estimates and bounds for graph databases.

6. Chapter 7 introduces Galley, a state-of-the-art optimizer for sparse tensor programs.

Chapter 3 contains material from our ICDT paper [38] as well as our extended version in TODS [39]. Chapter 4 contains material from our ICDT 2025 best paper and best student paper [37]. This work was done as co-first author with Timo Camillo Merkl from TU Wien. Chapter 5 contains material from our SIGMOD paper [43] and the accompanying

code can be found at [40]. Chapter 6 contains material from our VDLB paper [41] and the accompanying code can be found at [42]. Chapter 7 contains material from our SIGMOD paper [36] and builds on our OOPSLA paper [3] which introduced the underlying Finch compiler.

Chapter 2
BACKGROUND

In this chapter, we review important background that we will use throughout this dissertation. We will begin with a broad overview of cardinality estimation and bounding. This context will be important for understanding the work in Ch. 3- 6. Next, we briefly describe the history of sparse tensor programming and sparse tensor compilers. This will set the stage for Ch. 7 which describes an optimizer for these programs that operates on top of this compiler infrastructure.

2.1 A Brief History of Cardinality Estimation

2.1.1 Notation and Definitions

We begin by defining some terms and concepts that will be useful throughout this discussion. To begin with, unless otherwise stated, this section will focus on full conjunctive queries which we define below.

Definition 2.1.1. A conjunctive query, Q, is a statement of the following form,

$$Q(\mathbf{X}) := R_1(\mathbf{X}_1) \bowtie \ldots \bowtie R_k(\mathbf{X_k}) \tag{2.1}$$

The left-hand side is the head of the query while the right-hand side is the body, and each $R_i(\mathbf{X}_i)$ is a relation (eq. atom). $\mathbf{X}$ and $\mathbf{X}_i$ are sets of variables such that $\mathbf{X} \subseteq \bigcup_i \mathbf{X}_i$. A full conjunctive query is one where $\mathbf{X} = \bigcup_i \mathbf{X}_i$. Given a concrete database instance, D, we denote the result of evaluating Q on D as $Q[D]$.

Example 2.1.2. *To make this concrete, we present a simple join query in Fig. 2.1 between the* `ActedIn` *and* `FilmedIn` *tables. Because they both have a* `Movie` *column, their natural join occurs on that column. For example, the row (Mark Hamill, Star Wars IV) in ActedIn joins with both (Star Wars IV, Tunisia), (Star Wars IV, California), and (Star Wars IV,*

ActedIn

Person	Movie
Mark Hamill	Star Wars IV
Harrison Ford	Star Wars IV
Carrie Fisher	Star Wars IV
Elijah Wood	Lord of the Rings
Viggo Mortensen	Lord of the Rings

FilmedIn

Movie	Location
Star Wars IV	Tunisia
Star Wars IV	California
Star Wars IV	Arizona
Lord of the Rings	New Zealand

$$Q(Person, Movie, Location) := ActedIn(Person, Movie) \bowtie FilmedIn(Movie, Location)$$

Q

Person	Movie	Location
Mark Hamill	Star Wars IV	Tunisia
Mark Hamill	Star Wars IV	California
Mark Hamill	Star Wars IV	Arizona
Harrison Ford	Star Wars IV	Tunisia
Harrison Ford	Star Wars IV	California
Harrison Ford	Star Wars IV	Arizona
Carrie Fisher	Star Wars IV	Tunisia
Carrie Fisher	Star Wars IV	California
Carrie Fisher	Star Wars IV	Arizona
Elijah Wood	Lord of the Rings	New Zealand
Viggo Mortensen	Lord of the Rings	New Zealand

Figure 2.1: Example join query.

Arizona) in FilmedIn, producing the first three rows in Q. Note that joins can produce results that are much larger than the inputs. For example, three rows in each table contain the movie value Star Wars IV, and the join of these rows results in nine output rows because every combination of them is represented in the output.

This class of queries captures SQL queries restricted to SELECT, FROM, and JOIN clauses, subgraph-matching queries, and boolean tensor products. In the SQL setting, each relation R_i is a table with columns $\mathbf{X}_i$. Lastly, a join algorithm $\mathcal{A}$ is any algorithm that receives a query Q and a database instance I as input and outputs the relation $Q[D]$.

Definition 2.1.3. A cardinality estimation method, $\mathcal{E}$, is a function that takes in a query, Q, and a set of statistics, $\mathbf{s}(D)$, over a database, D, and outputs an estimate of $|Q[D]|$,

$$\mathcal{E}(Q, \mathbf{s}(D)) \approx |Q[D]| \tag{2.2}$$

We call $\mathcal{E}$ a *cardinality bounding method* iff,

$$\mathcal{E}(Q, \mathbf{s}(D)) \geq |Q[D]| \quad \forall \; Q, D \tag{2.3}$$

Informally, a cardinality estimate takes in a query and statistics on the database, and it outputs a number that is *close* to the size of the query output on that database. A cardinality bound is a cardinality estimate with the additional guarantee that it will never underestimate the output size.

2.1.2 The Traditional Approach to Cardinality Estimation

Since the modern query optimization paradigm was introduced by Selinger et al. in their System R work [131], that optimization has been guided by cardinality estimates. In this original work, they established the foundational assumptions that would guide decades of cardinality estimators: 1) that values are preserved during join operations (i.e. to the extent possible, every join value finds a match) 2) that selection predicates are independent (i.e. their joint selectivity is just the product of their individual selectivities) and 3) the uniformity assumption (i.e. join values all have the same frequency). With these three assumptions and some very basic statistics on the data, System R was able to produce serviceable cardinality estimates and perform useful query optimization.

Example 2.1.4 (System R Cardinality Estimation)**.** *Suppose we have two relations R and S where R stores information about employees and S stores sales information by department.*

$$\sigma_{R.tenure>5}(R) \bowtie_{R.dept\_id=S.id} \sigma_{S.sales>500000}(S) \tag{2.4}$$

A traditional query optimizer would have the following statistics available to them:

1. *The number of tuples in R and S: $|R| = 10,000$ and $|S| = 100$*

2. *The number of distinct values: $V(R, dept\_id) = 50$ and $V(S, id) = 100$*

3. *Estimates of the selectivity of each selection predicate, typically derived from a histogram [119]: $s_{R.tenure>5} \approx .7$ and $s_{S.sales>50000} \approx .4$*

Under the uniformity and preservation assumptions, the probability that a row in R joins with a row in S is $1/V(S, id)$, and vice versa. Therefore, the selectivity of the join condition is estimated as follows:

$$s_{R.dept\_id=S.id} \approx \min(\frac{1}{V(S, id)}, \frac{1}{V(R, dept\_id)}) = .01 \tag{2.5}$$

To compute the cardinality estimate, the traditional approach assumes independence between all predicates.

$$|Q| \approx |R| * |S| * s_{R.tenure>5} * s_{S.sales>100000} * s_{R.dept\_id=S.id} \tag{2.6}$$

$$|Q| \approx 2800 \tag{2.7}$$

However, it's easy to see that this may result in an inaccurate estimate. For instance, one may expect that stores with long tenured employees may be more successful at sales. This correlation would result in more rows than expected in the output.

The early work that built on this foundation attempted to loosen these assumptions in small, manageable ways. For example, early work on multidimensional histograms attempted to loosen the independence assumption by jointly estimating the selectivity of two or more predicates [108, 67]. Alternatively, join histograms were proposed to loosen the uniformity and independence assumption for join predicates [81, 97]. This kind of histogram splits the join domain into buckets and tracks the number of rows and distinct values within each one.

2.1.3 Modern Approaches to Cardinality Estimation

While cardinality estimation has always seen a modicum of interest in the database community, there has been a surge of work over the past decade. This "modern era" of cardinality estimation techniques was spurred by a blog post, "Is Query Optimization a 'Solved Problem'?", and a seminal benchmark paper, "How Good are Query Optimizers, Really?" [103, 99]. The former leveraged the author's decades of expertise in query optimization to argue that cardinality estimation was the "Achilles Heel" of query optimization. The latter used a new benchmark and exhaustive experiments to demonstrate the inaccuracy of existing estimates and their drastic impact on query execution.

The techniques introduced over the past decade can be roughly categorized into: 1) machine learning 2) sampling 3) sketching and 4) cardinality bounding. In this sub-section, I'll provide a survey on the first three kinds, and the next one will dive deeply into cardinality bounds because they are core to the work in this dissertation.

Machine Learning Approaches

Modeling the distribution of data within and between tables is the core problem of cardinality estimation, so it is natural to apply machine learning techniques. However, the efficiency of these techniques depends on their ability to take advantage of the unique properties of database queries, and they generally lack support for query features like self-joins and group-by.

Learning From Queries. The first wave of learned cardinality estimators focused on the regression problem where the training data is pairs of queries and their resulting cardinalities [94, 125]. The core challenge of this approach is to determine 1) a good way to represent queries as feature vectors and 2) a good way to collect training data. When the user's queries closely match the distribution of the training data, these approaches can be fast, space efficient, and accurate. However, larger domains and more complex correlations between tables can cause lower accuracy as a small set of training queries can no longer cover the important features of the data distribution [136, 92].

Learning From Data. The alternative approach to learned cardinality estimation at-

tempts to directly learn the distribution of the data in an unsupervised manner. Neurocard does this by sampling the outer join of the database schema and learning a probability distribution where every column is a variable [155]. This requires a very large model, and it requires joins to follow the schema structure (e.g. no self-joins are allowed). DeepDB and FactorJoin both find ways to split the data distribution into disjoint row and column sets [77, 153]. Splitting the data by rows allows for more accurate estimation but increases latency and space. Partitioning the columns assumes independence between them which can increase error but reduce latency and space.

Sampling Approaches

The key challenge in applying sampling to CE is the difficulty of efficiently sampling *any* join results. The naive approach, Bernoulli Sampling, samples each row in the database offline and independently with probability p. Because we sample the two relations independently, the probability that the samples produce any join results is very low. Specifically, the probability of a tuple in the full output occurring in the join of the sample is p^k where k is the number of relations in the join. Based on this probability, this estimator returns the cardinality of the sample join multiplied by $\frac{1}{p^k}$. By the union bound, the probability of at least one output tuple occurring in the sample output is at most $\text{OUT} * p^k$ where OUT is the true size of the output. If $\text{OUT} = 10000$ and $k = 5$, then sampling with probability $p = .1$ would have at most a 10% probability of sampling any output tuples.

To address this problem, each sampling method finds a way to correlate the sampling of tuples across tables. Correlated sampling (later called universe sampling) does this by sampling tuples iff any of their join value's hash values are $\leq p$ [143, 84]. For a two-table join, this ensures that tuples in the output will be sampled with probability p if the input is sampled with the same probability, a factor p improvement over Bernoulli sampling. However, this does not scale to longer chains of joins. For example, the probability of an output row being sampled in a three table join $R(X)S(X,Y)T(Y)$ is p^2 because rows in S must have $hash(X) \leq p$ AND $hash(Y) \leq p$.

The alternative approach is to sample tuples from each table at runtime, conditioned on

the samples from prior tables of the join. WanderJoin is the seminal work of this variety, and it equates this to sampling a random walk from a graph over the tuples in the database [101]. To do this, indices are required on the join columns of each table, and the walk terminates if a sampled tuple has no joining partner or fails the non-join or cyclic-join predicates. When these predicates are highly selective, this can result in a significant amount of wasted effort. To reduce the failure rate from cyclic queries, Alley guides sampling with an index that tracks whether tuples occur in small cyclic patterns, and it adapts worst-case optimal join techniques to reduce the sampling space [93].

Sketch Approaches

Probabilistic sketches were initially popularized for tracking statistics like the number of distinct elements in a streaming setting [54]. This is done by mapping elements to a set of entries in a matrix and either adding or subtracting from them when that element is inserted or deleted in the stream.

Prior work showed that probabilistic sketches could be used as an end-to-end solution for cardinality estimation [7, 127, 83, 75, 140]. They did this by adapting sketches that were originally designed for approximating the inner product of two streams. These approaches support high update throughput and provide concrete confidence intervals. However, they require prior knowledge of the query in two ways. First, they need to know the structure of the joins in order to determine which sketches to maintain. Second, they need to know the query's selection predicates (e.g. $R.Age > 25$). This allows them to filter tuples before inserting them into sketches, avoiding the use of histograms and most-common-value lists [83, 75]. Further, existing sketching techniques lack support for cyclic queries and queries with group-by clauses.

2.1.4 Prior Work on Cardinality Bounds

Cardinality bounding methods are an alternative approach to cardinality estimation that produce deterministic upper bounds on the true cardinality. Once we have an upper bound on the query's output, we can define a notion of optimality for join algorithms: an algorithm

is worst-case optimal if it runs in time linear with respect to this output bound. This kind of analysis was the original motivation for studying cardinality bounds, and it led to the development of worst-case optimal join algorithms, arguably the biggest advancement in query execution in decades[112, 141]. While the bounds developed for this analysis were elegant and insightful, they only considered the size of the input tables, so they were not accurate enough to guide a query optimizer. To improve these bounds, a sequence of theoretical results showed how more granular statistics could be leveraged. Much of the work in this dissertation follows in this vein by proposing novel bounds (Ch. 3 and 4), building real systems for computing them (Ch. 5, 6), and proposing new join algorithms that are optimal relative to them (Ch. 4). In this section, I'll provide a short history of these bounds, prior to the work presented in this dissertation.

The AGM Bound

The pioneering theoretical work in this space was done by Atserias, Grohe, and Marx. They introduced what would come to be known as the AGM bound, which we denote here by $\mathcal{E}_{AGM}$ [10]. To define the AGM bound, we need to introduce the notion of a query's hypergraph.

Definition 2.1.5. Given a conjunctive query Q,

$$Q(\mathbf{X}) := R_1(\mathbf{X}_1) \bowtie \cdots \bowtie R_k(\mathbf{X}_k)$$

The hypergraph of the query, $H_Q(V_Q, E_Q)$, is defined as follows.

1. Each variable in the query is a vertex, $V_Q = \mathbf{X}$

2. Each relation in the query forms an edge over the variables in the relation, $E_Q = \{\mathbf{X}_1, \ldots, \mathbf{X}_k\}$

Next, we need to introduce the notion of the *fractional edge covering number* of a query, $\rho^*(Q)$.

Definition 2.1.6. A fractional edge covering of a hypergraph $H(V, E)$ is a function $\rho : E \to \mathbb{R}^+$ such that:

$$\sum_{e \in E \ s.t. \ v \in E} \rho(v) \geq 1 \ \forall \ v \in V \tag{2.8}$$

We denote the fractional edge covering number of a query as the minimum total weight of a fractional edge covering of its hypergraph:

$$\rho^*(Q) = \min_{\rho} \sum_{e \in E_Q} \rho(e) \tag{2.9}$$

The fractional edge covering of a hypergraph is an assignment of weights to the edges the graph such that every node is covered by a total weight of at least 1. With this, we can define the AGM bound as follows,

Definition 2.1.7. Given a query, Q, and a bound on the size of any relation, N, the AGM bound, $\mathcal{E}_{AGM}$ is,

$$\mathcal{E}_{AGM} = N^{\rho*(Q)} \tag{2.10}$$

For any database D of size N, $|Q[D]| \leq \mathcal{E}_{AGM}$, and there exists a "worst case" database instance D where $|Q[D]|$ is within a constant of $\mathcal{E}_{AGM}$.

Example 2.1.8. *Consider the following query:*

$$Q_\Delta(X, Y, Z) = R(X, Y)S(Y, Z)T(X, Z)$$

The optimal fractional edge covering for this query assigns $\frac{1}{2}$ *to each relation, resulting in* $\rho^*(Q) = \frac{3}{2}$*. This implies,*

$$|Q[D]| \leq N^{\frac{3}{2}}$$

This is surprising because the join of any pair of relations can produce a quadratic result, so any plan based on binary joins will not be worst-case optimal.

This disconnect between the worst-case output size and the runtime of binary join plans was the original motivation for studying worst-case optimal join algorithms. These algorithms eschew relation-at-a-time join processing and instead use a variable-at-a-time approach, enabling more efficient filtering of intermediate join results.

Information Theoretic Bounds

To further improve on the AGM bound, a line of work began to include additional statistics on the data. Prior to the Degree Sequence Bound, the most general form of this was based on *degree constraints*[59, 89, 90].

Definition 2.1.9. A degree constraint $DC_R(\mathbf{X}, \mathbf{Y}, d)$ implies the following where R is a relation, $(\mathbf{X}, \mathbf{Y})$ are sets of variables, and d is a positive integer:

$$\max_{x \in \mathbf{X}} |\pi_{\mathbf{Y}} \sigma_{\mathbf{X}=x}(R)| \leq d$$

In this equation, we are maximizing the number of occurrences over all combinations of values in the join columns $\mathbf{X}$ after projecting the relation R down to the columns $\mathbf{Y}$. Degree constraints are a measure of the skewness and frequency of join values. Considering the `ActedIn` table from Fig. 2.1, the degree constraints $DC_{ActedIn}(\{Person\}, \{Person, Movie\}, 1)$ and $DC_{ActedIn}(\{Movie\}, \{Person, Movie\}, 3)$ hold because every actor appears once and Star Wars appears three times. Degree constraints are important because columns with low skew cannot produce a large explosion in the size of a query result. For example, joining on a relation's key, i.e. a column with a degree constraint of 1, can never increase the size of the result. However, leveraging these more detailed constraints to produce tight upper bounds proved to be challenging. Solving this problem required the introduction of a new framework for cardinality bounding.

The information theoretic framework begins by considering the probability distribution derived from uniformly randomly sampling $Q[D]$. The entropy of this distribution is, by definition, equal to $log(|Q[D]|)$. By maximizing this quantity over the set of entropic functions that are consistent with our statistics, we can bound the cardinality. Unfortunately, the space of entropic functions is very complex and not amenable to optimization, so this space is typically approximated by the set of polymatroid functions which can be defined by a series of linear inequalities. The bound is then computed with an off-the-shelf linear solver that maximizes over this space.

2.2 The Sparse Tensor Programming Ecosystem

In the final chapter of this thesis, we describe a system for optimizing sparse tensor programs based on key insights from database theory. To provide context for this system, this section will provide a brief history of approaches to sparse tensor programming.

2.2.1 Defining Sparse Tensor Programming

Tensor programming is a version of array programming, a classic paradigm in computing. A tensor program is a sequence of bulk operations over multidimensional arrays of, typically, numeric data. Linear algebra is a subset of tensor programming restricted to 0, 1, and 2-dimensional arrays. This style of programming tends to produces regular access patterns, enabling the full utilization of hardware. In this work, I use tensor index notation which consists of aggregations and pointwise operations over tensors with named indices. In this notation, matrix multiplication is defined as:

$$C_{ik} = \sum_{j} A_{ij} * B_{jk}$$

Unfortunately, not all data can fit neatly into the model of dense numeric grids. A network can be viewed as a matrix where each i, j entry is either 0 or 1 depending on the existence of an edge from the node i to the node j. However, most networks are *sparse* because the vast majority of nodes are not directly connected by an edge which results in a matrix where nearly all entries are 0. Representing this explicitly as a dense matrix is infeasible, and these situations have motivated the study of *sparse tensors* which only explicitly represent non-zero elements. Slightly more generally, sparse tensors have a defined fill value which is often 0 but may change depending on the data (e.g. ∞ in distance matrices). Sparse tensor programming extends traditional tensor programming to operate directly over these compressed tensors.

2.2.2 Sparse Linear Algebra

Like their dense counterparts, early systems for sparse tensor computation were restricted to linear algebra. While this area has a long history, the Sparse BLAS Standard was a

seminal attempt to standardize implementations of these operations [47]. Unfortunately, due to the diversity of sparsity patterns and Sparse BLAS' lack of support for general semiring operations, this standard never saw broad adoption. Instead, it was supplanted by the GraphBLAS standard which corrected these problems and demonstrated its utility in the context of graph processing [29]. Outside of these attempts from the HPC community, most users of sparse linear algebra are likely using the SciPy.sparse library [144] which provides basic operations over sparse matrices in basic formats like CSR, which we will define in the next section.

Sparse linear algebra libraries are typically collections of hand-optimized function implementations. Each implementation handles a single operation, which we call a kernel, and a specific set of input formats. When the number of kernels, sparse formats, and inputs-per-kernel increases, this requires exponentially more implementations which costs developer time and often increases the complexity of the API for the end-user. This has led to sparse implementations that lack functionality or only provide complete functionality for simple formats [118, 1].

2.2.3 The Advent of Sparse Tensor Compilers

To overcome this issue, the compilers community has developed *Sparse Tensor Compilers*. These compilers take in a high-level imperative description of a tensor kernel and a separate description of the tensor formats, and they automatically produce efficient low-level implementations[95, 3, 80]. There are a myriad of formats for sparse tensors, but these compilers often rely on the fiber-tree framework to define them. In this framework, a sparse tensor is stored as a tree where each layer of the tree corresponds to a dimension of the tensor. The order of these dimensions and the data structures used to store each layer define the tensor format. You can see an example of this in Fig. 2.2 where the row dimension corresponds to the first level of the tree and the column dimension to the second level. The compressed sparse-row (CSR) representation is depicted to the right of it. It is defined by a row-then-column dimension order with a dense vector storing the row positions and a sorted list representing each non-zero column index.

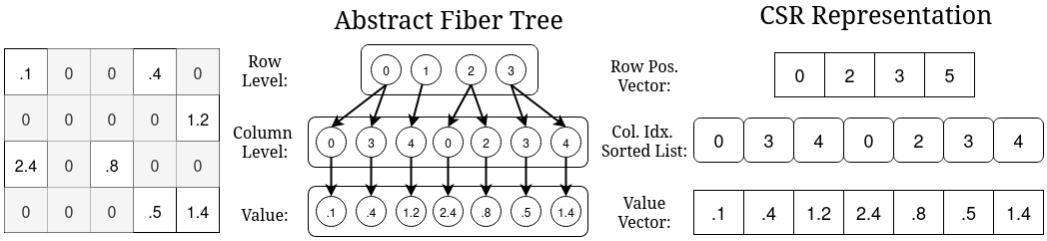


Figure 2.2: Fiber-Tree Abstraction

2.2.4 Prior Work in Optimizing Sparse Tensor Algebra

Optimizing (Sparse) Linear Algebra

Until relatively recently, the optimization of numerical workloads was restricted to linear algebra workloads. Because they don't rely on a compiler for execution, optimizing pure linear algebra expressions is equivalent to finding an efficient mapping of the original problem to a small set of hand-optimized kernels (e.g. matrix-vector multiplication, matrix-matrix multiplication, etc.).

The original optimization problem for dense linear algebra workloads is matrix chain multiplication which maps a matrix chain expression $A^{(1)} \dots A^{(k)}$ to a sequence of binary multiplication operations. This can be done with dynamic programming in $O(k^3)$ time, and the best known algorithm for this optimization requires $O(k \log(k))$ time [78, 79]. However, this problem represents a small subset of possible LA programs. To handle the more general case, Barthels et al. created the system Linnea [13]. Using a combination of rewriting, dynamic programming, and simple FLOP counts as a cost model, Linnea optimizes dense linear algebra programs involving inversion and decomposition. By finding an efficient mapping to BLAS operations, Linnea produced order-of-magnitude speedups over non-optimized, eager implementations on a range of classic LA expressions.

Early work on optimizing sparse linear algebra programs arose from the distributed computing context. SystemML was originally developed by IBM to take in a high-level LA program and automatically create a DAG of map-reduce operators to execute it [57]. The goal was to simplify the programming experience which could become extremely verbose

when translating LA to map-reduce and provide comparable performance to hand-coded implementations. This system saw active development for many years and was the basis for a long line of research papers on sparse LA optimization [16, 24, 26, 49, 25, 50]. As this work advanced, they used a combination of templated kernels to handle arbitrary function application and highly optimized GraphBLAS operators [29].

The most recent work in this direction, SPORES, applied e-graph techniques to exhaustively optimize linear algebra programs involving summation, multiplication, and addition [147]. E-graphs provided an efficient framework for term rewriting which allowed this work to explore a much larger space of implementations[149]. Beyond this, the key insight of this work was that applying linear algebra equalities was not sufficient to find the optimal rewriting. Instead, it is necessary to view linear algebra programs as relations (eq. as tensors) and apply relational equalities which are known to be complete. Because SPORES was built on top of SystemML, it still required the final form to be a valid LA expression. Despite this restriction, SPORES still produced asymptotic speedups on several problems.

Optimizers for Sparse Tensor Compilers

The compilers community has developed sparse tensor optimizers such as SparseAuto and Pigeon [5, 46]. To avoid modeling the complex data interactions, these systems focus on the asymptotic complexity of different implementations. This results in a pareto frontier of plans whose asymptotic complexities are incomparable from which they choose one based on heuristics. This begins with a principled approach, but the number of plans in the pareto frontier can be quite large, in the millions for kernels with a handful of inputs. This increases the importance of the selection heuristic which remains ignorant of the user's data distribution, resulting in poor selection when data is outside of the expected distribution. The other approach from this community, embodied by the WACO system, has been to apply machine learning to estimate the cost of a sparse kernel [152]. To simplify the problem, this work focuses exclusively on the choice of output format and the loop order. Prior to this thesis, this is the extent of the prior work on optimizing these compilers.

Leveraging Database Systems

The final approach takes advantage of the similarity between relational and tensor algebra to execute tensor programs using database systems. In Sec. 7.1.3, we prove this equivalence formally. However, as an example, consider Fig. 1.1 which expresses the same matrix chain multiplication in both SQL and linear algebra. Performing this translation allows the programs to benefit from query optimization and from the database execution engine which is highly optimized for extremely sparse workloads. However, a recent study exploring this approach showed that even the most efficient database systems struggle to come within an order of magnitude of traditional libraries for dense execution [130]. This should not come as a surprise. Database systems are highly tuned for their intended workloads which rarely involve dense attribute domains. Even the SQL extensions that explicitly define and manipulate arrays are typically viewed as a non-core operator, and they have received relatively little optimization effort as compared to the core of joins, projections, and filters. On the other hand, libraries like NumPy provide a minimal interface and can assume that all data consists of dense, rectilinear grids of numbers [73]. This allows them to ignore the overhead of comparing indices explicitly and makes optimizations like vectorization and blocking much more efficient.

Chapter 3

DEGREE SEQUENCE BOUND

In this chapter, we propose a new cardinality bound (Def. 2.1.3) based on *degree sequences*, called the *Degree Sequence Bound*. As we discussed in the previous two chapters, traditional cardinality estimation methods attempt to predict the number of rows in the output of a query based on statistics about the input tables, typically in the database setting. In contrast to this, cardinality bounding methods attempt to use statistics to prove an upper bound on the size of the query output. Prior approaches to computing upper bounds were only able to use coarse statistics about the data (e.g. table sizes and the maximum degree of join columns). This lack of detailed information about the data resulted in loose bounds. With a more granular view of the data distribution, the degree sequence bound achieves much higher accuracy than previous bounds.

To define degree sequences, we need a few more pieces of notation than we provided in Ch. 2. Given a relation R, an attribute X, and a value $u \in \Pi_X(R)$, the *degree* of u is the number of tuples in R with u in the X attribute, formally $d^{(u)} = |\sigma_{X=u}(R)|$. The *degree sequence* of an attribute X in relation R is the sorted sequence of all degrees for the values of that attribute, $d^{(u_1)} \geq d^{(u_2)} \geq \cdots \geq d^{(u_n)}$. Going forward, we drop any reference to values and instead refer to degrees by their index in this sequence, also called their *rank*, i.e. $d_1 \geq \cdots \geq d_n$. The degree sequence is very similar to a rank-frequency distribution in the probability literature and has been extensively used in graph analysis [15, 70]. A degree sequence can easily be computed offline and compressed effectively, with a good space/accuracy tradeoff due to its monotonicity; see Fig. 3.1 for an illustration.

The main insight in this chapter is that degree sequences offer more information on the database instance than the statistics used by previous upper bounds. For example, the AGM bound uses only the cardinality of the relations, which is $\sum_i d_i$, while the extension to degree constraints [90] uses the cardinality, $\sum_i d_i$, and the maximum degree, d_1. We improve

Name	...
Alice	...
Alice	...
Bob	...
Carlos	...
Carlos	...
Carlos	...
David	...
Eseah	...
Eseah	...
Eseah	...
Eseah	...
Eseah	...
Vivek	...
Vivek	...
Vivek	...
Gael	...
Hans	...
Hans	...
John	...
Karl	...
Lee	...

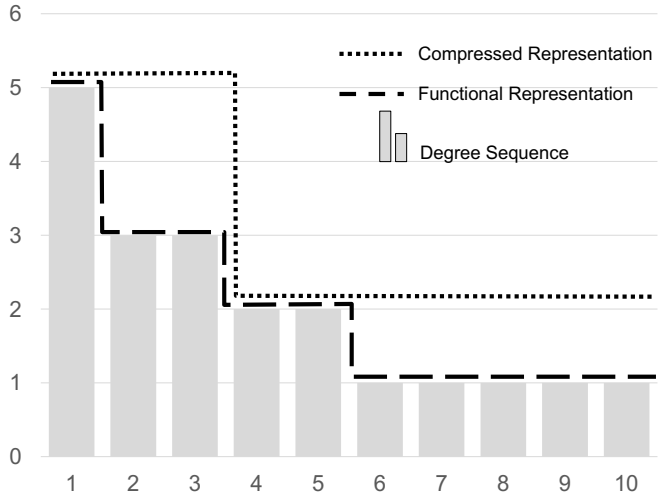


Figure 3.1: Example Degree Sequence.

significantly over these bounds because the degree sequence dramatically constrains the set of possible instances compared to these prior statistics.

For this new bound we had to develop entirely new techniques over those used for the AGM and the polymatroid bounds. Previous techniques are based on information theory. If some relation $R(X, Y)$ has cardinality N, then any probability space over R has an entropy, H, that satisfies $H(XY) \leq \log N$; if the degree sequence of the attribute X is $d_1 \geq d_2 \geq \ldots$, then $H(Y|X) \leq \log d_1$. Both the AGM and the polymatroid bound start from such constraints on the entropy. Unfortunately, these constraints do not extend to degree sequences. Because H is ignorant of $d_2, d_3, \ldots$, Information theory gives us only three degrees of freedom, namely $H(XY), H(X), H(Y)$, while the degree sequence has an arbitrary number of degrees of freedom. Rather than using information theory, our new framework models relations as tensors, formulates the upper bound as a linear optimization problem, and describes a highly efficient means of solving this problem. This framework is restricted to *Berge-acyclic, full conjunctive queries* [52] (reviewed in Sec. 3.1); throughout

the chapter we will assume that queries are in this class. As we explain in appendix A, these are the most common queries found in applications.

The Worst-Case Instance Our main result (Theorems 3.2.2 and 3.3.1) is a tight cardinality bound given the degree sequences of all relations. This bound is obtained by evaluating the query on a *worst-case instance* that satisfies those degree constraints.[1] Intuitively, each relation of the worst-case instance is obtained by matching the highest degree values in the different columns, and the same principle is applied across relations. For example, consider the join $R(X,\ldots) \bowtie S(X,\ldots)$, where the degree sequences of $R.X$ and $S.X$ are $a_1 \geq a_2 \geq \cdots$ and $b_1 \geq b_2 \geq \cdots$ respectively. The true cardinality of the join is $\sum_i a_i b_{\tau(i)}$ for some unknown permutation τ. But this sum is maximized when τ sorts the degree sequence b_i in decreasing order,[2] which implies that $\sum_i a_i b_i$ is a tight upper bound on the size of the join. Thus, the upper bound is obtained when the highest degree values match. Our degree sequence bound holds even when the input relations are allowed to be bags. Furthermore, we prove (Theorem 3.4.4) that this bound is always below the AGM and polymatroid bounds. This result required us to develop a new, explicit formula for the polymatroid bound for Berge-acyclic queries, which is of independent interest, given in Theorem 3.4.1.

Compact Representation A full degree sequence is about as large as the relation instance, while cardinality estimators need to run in sub-linear time. Fortunately, a degree sequence can be represented compactly using a piece-wise constant function, called a *staircase function*, with at most $O(\min(\sqrt{N}, d_1))$ parts (Lemma 3.5.1). This is illustrated in Fig. 3.1. Our next result, Theorem 3.5.3, is an algorithm for the degree sequence bound that runs in quasi-linear time (i.e. linear plus a logarithmic factor) in the size of the representation, independent of the size of the instance. The algorithm makes some rounding errors (Lemma 3.5.2), so its output may be slightly larger than the exact bound. However we prove that it is still lower than the AGM and polymatroid bounds (Theorem 3.5.5). The

---

[1] In graph theory, the problem of computing a graph satisfying a given degree sequence is called the *realization problem*.

[2] For a simple example, if $a_1 \geq a_2$, $b_1 \geq b_2$, then $a_1b_1 + a_2b_2 \geq a_1b_2 + a_2b_1$.

algorithm can even operate on a lossily compressed representation of the degree sequence and produce a valid bound. By using few buckets and upper-bounding the degree sequence one can trade off the memory size and estimation time for accuracy. At one extreme, we could upper bound the entire sequence using a single bucket with the constant d_1, at the other extreme we could keep the complete sequence. Neither the AGM bound nor the polymatroid bound have this ability to tradeoff space and time.

Max Tuple Multiplicity Despite using more information than previous upper bounds, our bound can still be overly pessimistic, because it needs to match the most frequent elements in all attributes. For example, suppose a relation has two attributes whose highest degrees are a_1 and b_1, respectively. Its worst-case instance is a bag and must include some tuple that occurs $\min(a_1, b_1)$ times. Usually, a_1 and b_1 are large, since they represent the frequencies of the worst heavy hitters in the two columns, but in practice they rarely occur together $\min(a_1, b_1)$ times. To avoid such worst-case matchings, we use one additional piece of information on each base table: the max multiplicity over all tuples, denoted B. Usually, B is significantly smaller than the largest degrees, and, by imposing it as an additional constraint, we can significantly improve the query's upper bound; in particular, when $B = 1$ then the relation is restricted to be a set. Our main results in Theorems 3.2.2 and 3.3.1 extend to max tuple multiplicities, but in some unexpected ways. The worst-case relation may no longer be a conventional relation: it may have tuples that occur more than B times, and, when the relation has 3 or more attributes it may even have tuples with negative multiplicities. This arises from viewing the worst-case relation as a tensor that satisfies a particular linear program. Nevertheless, these unconventional worst-case relations provide an even better degree sequence bound than by ignoring B.

Example 3.0.1. *Consider the full conjunctive query* $Q(\cdots) = R(X, \cdots) \bowtie S(X, Y, \cdots) \bowtie T(Y, \cdots)$*, where we omit showing attributes that appear in only one of the relations. Alternatively, we can write* $Q(X, Y) = R(X) \bowtie S(X, Y) \bowtie T(Y)$ *where* R, S, T *are bags rather than sets. Assume the following degree sequences:*

$$\boldsymbol{d}^{(R.X)} = (3, 2, 2) \qquad \boldsymbol{d}^{(T.Y)} = (2, 1, 1, 1) \qquad \boldsymbol{d}^{(S.X)} = (5, 1) \qquad \boldsymbol{d}^{(S.Y)} = (3, 2, 1) \qquad (3.1)$$

The AGM bound uses only the cardinalities, which are:

$$|R| = 7 \qquad |S| = 6 \qquad |T| = 5$$

The AGM bound[3] *is* $|R| \cdot |S| \cdot |T| = 210$. *The extension to degree constraints in [90] uses in addition the maximum degrees:*

$$\texttt{deg}(R.X) = 3 \qquad \texttt{deg}(S.X) = 5 \qquad \texttt{deg}(S.Y) = 3 \qquad \texttt{deg}(T.Y) = 2$$

and the bound is the minimum between the AGM bound and the following quantities:

$$|R| \cdot \texttt{deg}(S.X) \cdot \texttt{deg}(T.Y) = 7 \cdot 5 \cdot 2 = 70$$
$$\texttt{deg}(R.X) \cdot |S| \cdot \texttt{deg}(T.Y) = 3 \cdot 6 \cdot 2 = 36$$
$$\texttt{deg}(R.X) \cdot \texttt{deg}(S.Y) \cdot |T| = 3 \cdot 3 \cdot 5 = 45$$

Thus, the degree-constraint bound is improved to 36.

Our new bound is given by the answer to the query on the worst-case instance of the relations R, S, T, *shown here together with their multiplicities (recall that they are bags):*

$$R = \begin{array}{|c|}\hline a \\ \hline b \\ \hline c \\ \hline \end{array}\ \begin{array}{l} 3 \\ 2 \\ 2 \end{array}, \qquad S = \begin{array}{|c|c|}\hline a & u \\ \hline a & v \\ \hline b & w \\ \hline \end{array}\ \begin{array}{l} 3 \\ 2 \\ 1 \end{array}, \qquad T = \begin{array}{|c|}\hline u \\ \hline v \\ \hline w \\ \hline z \\ \hline \end{array}\ \begin{array}{l} 2 \\ 1 \\ 1 \\ 1 \end{array}, \tag{3.2}$$

The three relations have the required degree sequences, for example $S.X$ *consists of 5* a*'s and 1* b, *thus has degree sequence* $(5, 1)$. *Notice the matching principle: we assumed that the most frequent element in* $R.X$ *and* $S.X$ *are the same value* a, *and that the most frequent values in* $S.X$ *and in* $S.Y$ *occur together. On this instance, we compute the query and obtain the answer* Q.

$$Q = \begin{array}{|c|c|}\hline a & u \\ \hline a & v \\ \hline b & w \\ \hline \end{array}\ \begin{array}{l} 3 \cdot 3 \cdot 2 = 18 \\ 3 \cdot 2 \cdot 1 = 6 \\ 2 \cdot 1 \cdot 1 = 2 \end{array}$$

The upper bound is the size of the answer on this instance, which is $18 + 6 + 2 = 26$, *and it improves over 36. Here, the improvement is relatively minor, but this is a consequence*

[3]To apply this bound in the bag setting, we model each relation as having private variables, e.g. $R(X, U), S(X, Y, V, W), T(Y, Z)$. Due to this, the only fractional edge cover is $1, 1, 1$.

of the short example. In practice, degree sequences often have a long tail, i.e. with a few large leading degrees $d_1, d_2, \ldots$ followed by very many small degrees $d_m, d_{m+1}, \ldots, d_n$ (with a large n). In that case the improvements of the new bound can be very significant.

Now, suppose that we have one additional piece of information about S: every tuple occurs at most $B = 2$ times. Then, we need to reduce the multiplicity of (a, u), and the new worst-case instance, denoted S', is the following relation which decreases the cardinality bound to 25.

$$S' = \begin{array}{|c|c|l} \hline a & u & 2 \\ a & v & 2 \\ a & w & 1 \\ b & u & 1 \\ \hline \end{array}$$

Discussion This chapter is the extended version of a conference paper that appeared in ICDT'2023 [38] and a journal paper that appeared in ACM TODS[39]. Recently in [158], the information-theoretic bound in [90] has been extended to also use ℓ_p-norms of the degree sequences. Their approach does not use the degree sequence $a_1 \geq \cdots \geq a_n$, but instead uses some of its ℓ_p norms, i.e. $\ell_p = (\sum_i a_i^p)^{1/p}$. The connection between the ℓ_p-bounds and our degree sequence is quite subtle. On one hand, given complete information, these two methods use are (in theory) equivalent: knowing the full degree sequence allows us to compute the first ℓ_p-norms, $\ell_1, \ldots, \ell_n$, and, conversely, from the first n ℓ_p-norms one can recover the entire degree sequence. Somewhat surprisingly given this, it was shown in [158] (Appendix C.3) that, even if given full access to n ℓ_p-norms, the degree sequence bound can be asymptotically better than the ℓ_p-bound. However, in practice both methods are generally compressed: the degree sequence via approximation by a staircase function, and ℓ_p norms via restriction to just the first few norms, making the two methods somewhat incomparable.

3.1 Problem Statement

Tensors In this paper, it is convenient to define tensors using a named perspective, where each coordinate is associated with a variable. We write variables with capital letters $X, Y, \ldots$ and sets of variables with boldface, $\boldsymbol{X}, \boldsymbol{Y}, \ldots$ We assume that each variable X has an

associated finite domain $D_X \stackrel{\text{def}}{=} [n_X]$ for some number $n_X \geq 1$ where $[n_X] = \{1, \ldots, n_X\}$. For any set of variables $\boldsymbol{X}$, we denote by $D_{\boldsymbol{X}} \stackrel{\text{def}}{=} \prod_{Z \in \boldsymbol{X}} D_Z$. We use lower case for values, e.g. $z \in D_Z$, and boldface for tuples, e.g. $\boldsymbol{x} \in D_{\boldsymbol{X}}$. An $\boldsymbol{X}$-*tensor*, or simply a tensor when $\boldsymbol{X}$ is clear from the context, is $\boldsymbol{M} \in \mathbb{R}^{D_{\boldsymbol{X}}}$. For example, when $\boldsymbol{X} = \{X_1, X_2\}$, an $\boldsymbol{X}$-tensor is a matrix $\boldsymbol{M} \in R^{n_{X_1} \times n_{X_2}}$, and $\boldsymbol{M}_{(1,2)}$ is the entry at row 1 and column 2. We say that $\boldsymbol{M}$ has $|\boldsymbol{X}|$ coordinates. Given two $\boldsymbol{X}$-tensors $\boldsymbol{M}, \boldsymbol{N}$, we write $\boldsymbol{M} \leq \boldsymbol{N}$ for the component-wise order ($M_{\boldsymbol{x}} \leq N_{\boldsymbol{x}}$, for all $\boldsymbol{x}$). If $\boldsymbol{X}, \boldsymbol{Y}$ are two sets of variables, then we denote their union by $\boldsymbol{XY}$. If $\boldsymbol{X}, \boldsymbol{Y}$ are disjoint, and $\boldsymbol{x} \in D_{\boldsymbol{X}}, \boldsymbol{y} \in D_{\boldsymbol{Y}}$, then we denote by $\boldsymbol{xy} \in D_{\boldsymbol{XY}}$ the concatenation of the two tuples.

Definition 3.1.1. Let $\boldsymbol{M}, \boldsymbol{N}$ be an $\boldsymbol{X}$-tensor, and a $\boldsymbol{Y}$-tensor respectively. Their tensor product is the following $\boldsymbol{XY}$-tensor:

$$\forall \boldsymbol{z} \in D_{\boldsymbol{XY}} : \qquad (\boldsymbol{M} \otimes \boldsymbol{N})_{\boldsymbol{z}} \stackrel{\text{def}}{=} M_{\pi_{\boldsymbol{X}}(\boldsymbol{z})} \cdot N_{\pi_{\boldsymbol{Y}}(\boldsymbol{z})} \tag{3.3}$$

If $\boldsymbol{X}, \boldsymbol{Y}$ are disjoint and $\boldsymbol{M}$ is an $\boldsymbol{XY}$-tensor, then we define its $\boldsymbol{X}$-summation to be the following $\boldsymbol{Y}$-tensor:

$$\forall \boldsymbol{y} \in D_{\boldsymbol{Y}} : \qquad (\texttt{SUM}_{\boldsymbol{X}}(\boldsymbol{M}))_{\boldsymbol{y}} \stackrel{\text{def}}{=} \sum_{\boldsymbol{x} \in D_{\boldsymbol{X}}} M_{\boldsymbol{xy}} \tag{3.4}$$

If $\boldsymbol{M}, \boldsymbol{N}$ are $\boldsymbol{XY}$ and $\boldsymbol{YZ}$ tensors, where $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z}$ are disjoint sets of variables, then their dot product is the $\boldsymbol{XZ}$-tensor:

$$\forall \boldsymbol{x} \in D_{\boldsymbol{X}}, \boldsymbol{z} \in D_{\boldsymbol{Z}} : \qquad (\boldsymbol{M} \cdot \boldsymbol{N})_{\boldsymbol{xz}} \stackrel{\text{def}}{=} \texttt{SUM}_{\boldsymbol{Y}}(\boldsymbol{M} \otimes \boldsymbol{N})_{\boldsymbol{xz}} = \sum_{\boldsymbol{y} \in D_{\boldsymbol{Y}}} M_{\boldsymbol{xy}} N_{\boldsymbol{yz}} \tag{3.5}$$

In other words, we use $\otimes$ like a natural join. For example, if $\boldsymbol{M}$ is an IJ-tensor (i.e. a matrix) and $\boldsymbol{N}$ is an KL-tensor, then $\boldsymbol{M} \otimes \boldsymbol{N}$ is the Kronecker product; if $\boldsymbol{P}$ is an IJ-tensor (like $\boldsymbol{M}$), then $\boldsymbol{M} \otimes \boldsymbol{P}$ is the element-wise product. The dot product sums out the common variables. For example, if $\boldsymbol{a}$ is a J-tensor, then $\boldsymbol{M} \cdot \boldsymbol{a}$ is the standard matrix-vector multiplication, and its result is an I-tensor. You can then express the push-down of a summation over a product as follows. If $\boldsymbol{M}$ is an $\boldsymbol{X}$-tensor, $\boldsymbol{N}$ is a $\boldsymbol{Y}$-tensor and $\boldsymbol{X}, \boldsymbol{Y}$ are disjoint sets of variables, then:

$$\forall \boldsymbol{X}_0 \subseteq \boldsymbol{X}, \forall \boldsymbol{Y}_0 \subseteq \boldsymbol{Y} : \qquad \texttt{SUM}_{\boldsymbol{X}_0 \boldsymbol{Y}_0}(\boldsymbol{M} \otimes \boldsymbol{N}) = \texttt{SUM}_{\boldsymbol{X}_0}(\boldsymbol{M}) \otimes \texttt{SUM}_{\boldsymbol{Y}_0}(\boldsymbol{N}) \tag{3.6}$$

Permutations A permutation on $D = [n]$ is a bijective function $\sigma : D \to D$; the set of permutations on D is denoted S_D, or simply S_n. If $\boldsymbol{D} = D_1 \times \cdots \times D_k$ then we denote $S_{\boldsymbol{D}} \stackrel{\text{def}}{=} S_{D_1} \times \cdots \times S_{D_k}$. Given an $\boldsymbol{X}$-tensor $\boldsymbol{M} \in \mathbb{R}^{D_{\boldsymbol{X}}}$ and permutation $\boldsymbol{\sigma} \in S_{D_{\boldsymbol{X}}}$, the *$\boldsymbol{\sigma}$-permuted $\boldsymbol{X}$-tensor* is $\boldsymbol{M} \circ \boldsymbol{\sigma} \in \mathbb{R}^{D_{\boldsymbol{X}}}$:

$$\forall \boldsymbol{x} \in D_{\boldsymbol{X}} : \qquad (\boldsymbol{M} \circ \boldsymbol{\sigma})_{\boldsymbol{x}} \stackrel{\text{def}}{=} \boldsymbol{M}_{\boldsymbol{\sigma}(\boldsymbol{x})}$$

Sums are invariant under permutations, for example if $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^{D_Z}$ are Z-vectors and $\sigma \in S_{D_Z}$, then $(\boldsymbol{a} \circ \sigma) \cdot (\boldsymbol{b} \circ \sigma) = \boldsymbol{a} \cdot \boldsymbol{b}$, because $\sum_{i \in D_Z} a_{\sigma(i)} b_{\sigma(i)} = \sum_{i \in D_Z} a_i b_i$. Viewing relations as tensor is convenient in this work because it provides an ordering on the domains of join variables, and we rely on permutations to manipulate this ordering. This allow us to guarantee that the ordering of a domain satisfies properties like monotonicity w.r.t. different quantities.

Queries A full conjunctive query Q is:

$$Q(\boldsymbol{X}) = \bowtie_{R \in \boldsymbol{R}} R(\boldsymbol{X}_R) \tag{3.7}$$

where $\boldsymbol{R} \stackrel{\text{def}}{=} \boldsymbol{R}(Q)$ denotes the set of its relations, $\boldsymbol{X}$ is a set of variables, and $\boldsymbol{X}_R \subseteq \boldsymbol{X}$ for each relation $R \in \boldsymbol{R}$. This is a formal way of representing relational queries without projection. A tuple, $\boldsymbol{x}$, appears in the result, $Q(\boldsymbol{X})$, iff its subset occurs in each relation $\boldsymbol{x}[\boldsymbol{X}_R] \in R(\boldsymbol{X}_R) \forall R \in \boldsymbol{R}$.

The *incidence graph* of Q is the following bipartite graph: $T \stackrel{\text{def}}{=} (\boldsymbol{R} \cup \boldsymbol{X}, E \stackrel{\text{def}}{=} \{(R, Z) \mid Z \in \boldsymbol{X}_R\})$. A query Q is *Berge-acyclic* [52] if its incidence graph is a tree. This definition is an equivalent formulation to what is usually found in the literature, see Appendix A. Unless otherwise stated, all queries in this paper are assumed to be full, Berge-acyclic conjunctive queries. We use bag semantics for query evaluation, and represent an *instance* of a relation $R \in \boldsymbol{R}$ by an $\boldsymbol{X}_R$-tensor, $\boldsymbol{M}^{(R)}$, where $M^{(R)}_t$ is defined to be the multiplicity of the tuple $t \in D_{\boldsymbol{X}_R}$ in the bag R. The number of tuples in the answer to Q is:

$$|Q| = \texttt{SUM}_{\boldsymbol{X}} \left(\bigotimes_{R \in \boldsymbol{R}} \boldsymbol{M}^{(R)} \right) \tag{3.8}$$

Example 3.1.2. *Consider the following query:*

$$Q(X, Y, Z, U, V, W) = R(X, Y) \bowtie S(Y, Z, U) \bowtie T(U, V) \bowtie K(Y, W)$$

Its incidence graph is $T = (\{R, \ldots, K\} \cup \{X, \ldots, W\}, \{(R, X), (R, Y), (S, Y), \ldots, (K, W)\})$ *and is an undirected tree. An instance of* $R(X, Y)$ *is represented by a matrix* $\boldsymbol{M}^{(R)} \in \mathbb{R}^{D_X \times D_Y}$*, where* $M^{(R)}_{xy}$ *= the number of times the tuple* (x, y) *occurs in* R*. Similarly,* S *is represented by a tensor* $\boldsymbol{M}^{(S)} \in \mathbb{R}^{D_Y \times D_Z \times D_U}$*. The size of the query's output is:*

$$|Q| = \sum_{x,y,z,u,v,w} M^{(R)}_{xy} M^{(S)}_{yzu} M^{(T)}_{uv} M^{(K)}_{yw}$$

Degree Sequences We denote by $\mathbb{R}_+ \stackrel{\text{def}}{=} \{x \mid x \in \mathbb{R}, x \geq 0\}$ and we say that a vector $f \in \mathbb{R}_+^{[n]}$ is non-increasing if $f_{r-1} \geq f_r$ for $r \in [2, \ldots, n]$.

Definition 3.1.3. Fix a set of variables $\boldsymbol{X}$, with domains D_Z, $Z \in \boldsymbol{X}$. A *degree sequence* associated with the coordinate $Z \in \boldsymbol{X}$ is a non-increasing vector $\boldsymbol{f}^{(Z)} \in \mathbb{R}_+^{D_Z}$. We call a particular entry in this vector, $f^{(Z)}_r$, the *degree at rank* r. An $\boldsymbol{X}$-tensor $\boldsymbol{M}$ is *consistent* w.r.t. $\boldsymbol{f}^{(Z)}$ if:

$$\texttt{SUM}_{\boldsymbol{X}-\{Z\}}(\boldsymbol{M}) \leq \boldsymbol{f}^{(Z)} \tag{3.9}$$

We call a tensor, $\boldsymbol{M}$, consistent with a tuple of degree sequences $\boldsymbol{f}^{(\boldsymbol{X})} \stackrel{\text{def}}{=} (\boldsymbol{f}^{(Z)})_{Z \in \boldsymbol{X}}$, if it is consistent with every $\boldsymbol{f}^{(Z)}$. Furthermore, given $B \in \mathbb{R}_+ \cup \{\infty\}$, called the *max tuple multiplicity*, we say that $\boldsymbol{M}$ is *consistent* w.r.t. B if $\boldsymbol{M}_t \leq B$ for all $t \in D_{\boldsymbol{X}}$. We will often call vectors $\boldsymbol{f}^{(\boldsymbol{X})}$ *degree constraints*, and we say that $\boldsymbol{M}$ *satisfies* the degree constraints $\boldsymbol{f}^{(\boldsymbol{X})}$ (and the bound B). We denote:

$$\begin{aligned}
\mathcal{M}_{\boldsymbol{f}^{(\boldsymbol{X})}, B} &\stackrel{\text{def}}{=} \{\boldsymbol{M} \in \mathbb{R}^{D_{\boldsymbol{X}}} \mid \boldsymbol{M} \text{ is consistent with } \boldsymbol{f}^{(\boldsymbol{X})}, B\} \\
\mathcal{M}^+_{\boldsymbol{f}^{(\boldsymbol{X})}, B} &\stackrel{\text{def}}{=} \{\boldsymbol{M} \in \mathbb{R}_+^{D_{\boldsymbol{X}}} \mid \boldsymbol{M} \text{ is non-negative and consistent with } \boldsymbol{f}^{(\boldsymbol{X})}, B\}
\end{aligned} \tag{3.10}$$

For a simple illustration consider two degree sequences $\boldsymbol{f} \in \mathbb{R}^{[m]}, \boldsymbol{g} \in \mathbb{R}^{[n]}$. $\mathcal{M}_{\mathbf{f},\mathbf{g},\infty}$ is the set of matrices $\boldsymbol{M}$ whose row-sums and column-sums are $\leq \boldsymbol{f}$ and $\leq \boldsymbol{g}$ respectively; $\mathcal{M}^+_{\mathbf{f},\mathbf{g},\infty}$ is the subset of non-negative matrices; $\mathcal{M}_{\mathbf{f},\mathbf{g},B}$ is the subset of matrices in $\mathcal{M}_{\mathbf{f},\mathbf{g},\infty}$ that also satisfy $M_{ij} \leq B$, $\forall i, j$.

Problem Statement Fix a query Q. For each relation R, we are given a set of degree sequences $\boldsymbol{f}^{(R,\boldsymbol{X}_R)} \stackrel{\text{def}}{=} \left(\boldsymbol{f}^{(R,Z)}\right)_{Z \in \boldsymbol{X}_R}$, and a tuple multiplicity $B^{(R)} \in \mathbb{R}_+ \cup \{\infty\}$. We are asked to find the maximum size of Q over all database instances consistent with all degree

sequences and tuple multiplicities. To do this, we represent a relation instance R by an unknown tensor $\boldsymbol{M}^{(R)} \in \mathcal{M}^{+}_{\mathbf{f}^{(R,\boldsymbol{X}_R)},B^{(R)}}$ and an unknown set of permutations $\boldsymbol{\sigma}^{(R)} \in S_{D_{\boldsymbol{X}_R}}$, and solve the following problem:

Problem 1 (Degree Sequence Bound)**.** Solve the following optimization problem:

$$\text{Maximize: } |Q| = \texttt{SUM}_{\boldsymbol{X}} \left(\bigotimes_{R \in \boldsymbol{R}} (\boldsymbol{M}^{(R)} \circ \boldsymbol{\sigma}^{(R)}) \right) \tag{3.11}$$

$$\text{Where: } \forall R \in \boldsymbol{R},\ \boldsymbol{\sigma}^{(R)} \in S_{D_{\boldsymbol{X}_R}},\ \boldsymbol{M}^{(R)} \in \mathcal{M}^{+}_{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}}$$

This is a non-linear optimization problem: while the set $\mathcal{M}^{+}$ defined in Eq. (3.10) is a set of linear constraints, the objective (3.11) is non-linear. In the rest of the paper we describe an explicit formula for the degree sequence bound, which is optimal (i.e. tight) when $B^{(R)} = \infty$, for all R, and is optimal in a weaker sense in general.

Example 3.1.4. *We revisit Example 3.0.1 in our new notation. The worst-case instances R, S, T in Eq.* (3.2) *are represented by the following tensors, where we also show their degree sequences on the margins, and denote them by $\boldsymbol{f}^{(R.X)}$, $\boldsymbol{f}^{(S.X)}$, $\boldsymbol{f}^{(S.Y)}$, $\boldsymbol{f}^{(T.Y)}$:*

$$\boldsymbol{M}^R = \begin{array}{c} \boldsymbol{f}^{R.X} \\ \begin{pmatrix} 3 \\ 2 \\ 2 \end{pmatrix} \end{array} \begin{array}{c} 3 \\ 2 \\ 2 \end{array} \qquad \boldsymbol{M}^S = \begin{array}{c} \\ \begin{pmatrix} 3 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \\ \boldsymbol{f}^{(S.Y)} \quad 3 \quad 2 \quad 1 \quad 0 \end{array} \begin{array}{c} \boldsymbol{f}^{(S.X)} \\ 5 \\ 1 \\ 0 \end{array} \qquad \boldsymbol{M}^T = \begin{array}{c} \boldsymbol{f}^{(T.Y)} \\ \begin{pmatrix} 2 \\ 1 \\ 1 \\ 1 \end{pmatrix} \end{array} \begin{array}{c} 2 \\ 1 \\ 1 \\ 1 \end{array} \tag{3.12}$$

The values a, b, c are now encoded as numbers $1, 2, 3$, similarly u, v, w, z become $1, 2, 3, 4$. The tensors store the multiplicities, for example, $\boldsymbol{M}^R_1 = 3$ means that $R(a)$ occurs 3 times, and $\boldsymbol{M}^S_{23} = 1$ means that $S(b, w)$ occurs once. The degree sequences of the arrays (one-coordinate tensors) $\boldsymbol{M}^R, \boldsymbol{M}^T$ are the same as the arrays, while those of $\boldsymbol{M}^S$ are the row-sums and the column-sums. One can immediately check that the size of the output to the conjunctive query in Example 3.0.1 on this worst-case instance is:

$$|Q| = \sum_{ij} \boldsymbol{M}^R_i \boldsymbol{M}^S_{ij} \boldsymbol{M}^T_j \tag{3.13}$$

In general, we are not given the instance, but only the degree constraints $\boldsymbol{f}^{(R.X)}$, $\boldsymbol{f}^{(S.X)}$, $\boldsymbol{f}^{(S.Y)}$, $\boldsymbol{f}^{(T.Y)}$, *and our goal is to maximize the following quantity:*

$$\sum_{ij} \boldsymbol{M}^R_i \boldsymbol{M}^S_{\sigma(i)\tau(j)} \boldsymbol{M}^T_j \tag{3.14}$$

where $\boldsymbol{M}^R \le \boldsymbol{f}^{(R.X)}$, $\boldsymbol{M}^T \le \boldsymbol{f}^{(T.Y)}$, *the row-sums and column sums of* $\boldsymbol{M}^S$ *are upper bounded by* $\boldsymbol{f}^{(S.X)}, \boldsymbol{f}^{(S.Y)}$ *respectively, and* σ, τ *are arbitrary permutations on the rows, and columns of* $\boldsymbol{M}$ *respectively; this is a special case of Eq.* (3.11)*. In our simple example we did not need to permute the elements of* $\boldsymbol{M}^R$ *and of* $\boldsymbol{M}^T$*, because that can be captured by permuting the rows and columns of* $\boldsymbol{M}^S$*. One can check that the maximum value of* (3.14) *is attained when the matrices* $\boldsymbol{M}^R, \boldsymbol{M}^S, \boldsymbol{M}^T$ *are those in* (3.12)*, and the permutations* σ, τ *are the identity.*

3.2 The Star Query

In this section we present the degree-sequence upper bound for a simple type of query, called a star query. Then, in Section 3.3 we show how to use this to compute an upper bound for any Berge-acyclic conjunctive query.

A *star query* is defined as follows:

$$Q_{\texttt{star}} = S(X_1, \ldots, X_d) \bowtie R^{(1)}(X_1) \bowtie \cdots \bowtie R^{(d)}(X_d) \tag{3.15}$$

Thus, there is one relation S containing all variables, and a unary relation $R^{(p)}$ for each variable X_p, $p = 1, d$. Assume that the domain of each variable X_p is $[n_p]$ for some $n_p > 0$, and denote by $[\boldsymbol{n}] \stackrel{\text{def}}{=} [n_1] \times \cdots \times [n_d]$. We denote by $\boldsymbol{a}^{(X_p)} \in \mathbb{R}_+^{[n_p]}$ the vector (1-coordinate tensor) associated to the relation $R^{(p)}$, and with $\boldsymbol{M} \in \mathbb{R}_+^{[\boldsymbol{n}]}$ the d-coordinate tensor associated to the relation S. By re-ordering the domain of the variable X_P, we can assume w.l.o.g. that the vector $\boldsymbol{a}^{(X_p)}$ is non-increasing; therefore, the degree sequence of $R^{(p)}$ is $\boldsymbol{a}^{(X_p)}$ itself.

3.2.1 Problem Statement

Problem 1 specializes to our star query $Q_{\texttt{star}}$ as follows. Fix degree sequence constraints $\boldsymbol{f}^{(\boldsymbol{X})} \stackrel{\text{def}}{=} (\boldsymbol{f}^{(X_1)}, \ldots \boldsymbol{f}^{(X_d)})$ for $\boldsymbol{M}$ (the tensor of S): we will require $\boldsymbol{M}$ to satisfy the degree

sequences $\boldsymbol{f}^{(\boldsymbol{X})}$. For the unary relations $R^{(p)}$ we will assume w.l.o.g. that their instances $\boldsymbol{a}^{(X_p)}$ are equal to their degree constraints (since we can always make $R^{(p)}$ larger, and only increase $|Q_{\texttt{star}}|$). Hence, we will use the same notation $\boldsymbol{a}^{(p)}$ for both the degree constraint and the instance for $R^{(p)}$. Then, Problem 1 asks us to maximize the expression:

$$\max_{\boldsymbol{M},\boldsymbol{\sigma}} |Q_{\texttt{star}}| = \max_{\boldsymbol{M},\boldsymbol{\sigma}} \sum_{(i_1,\ldots,i_d)\in[\boldsymbol{n}]} (\boldsymbol{M}\circ\boldsymbol{\sigma})_{i_1\cdots i_d} \cdot a_{i_1}^{(X_1)}\cdots a_{i_d}^{(X_d)} = (\boldsymbol{M}\circ\sigma)\cdot \boldsymbol{a}^{(X_1)}\cdots\boldsymbol{a}^{(X_d)} \tag{3.16}$$

where $\boldsymbol{M}$ ranges over all tensors consistent with the degree constraints $\boldsymbol{f}^{(X_p)}$, $p=1,d$, and $\boldsymbol{\sigma}$ ranges over the permutations of its domains. The "dot" operation in (3.16) denotes the dot product in (3.5). To find $\max_{\boldsymbol{M},\boldsymbol{\sigma}}$ in (3.16), we will compute $\text{argmax}_{\boldsymbol{M},\boldsymbol{\sigma}}$: we will call $\boldsymbol{M}$ and σ the *worst-case tensor* and *permutation* respectively. We show in this section that the worst case instance is independent of the arrays $a^{(X_p)}$, $p=1,d$, i.e. it depends only on the degree sequences $\boldsymbol{f}^{(X_p)}, p=1,d$, while the worst case permutation is the identity. We strengthen Problem 1 to:

Problem 2 (Worst-Case Tensor)**.** Fix $\boldsymbol{f}^{(\boldsymbol{X})} = (\boldsymbol{f}^{(X_1)},\ldots,\boldsymbol{f}^{(X_d)}) \in \mathbb{R}_+^{n_1}\times\cdots\times\mathbb{R}_+^{n_d}$, and $B\geq 0$. Find a tensor $\boldsymbol{C}\in\mathcal{M}_{\boldsymbol{f}^{(\boldsymbol{X})},\infty}$ such that, for all $\boldsymbol{\sigma}\in S_{[\boldsymbol{n}]}, \boldsymbol{M}\in\mathcal{M}^+_{\boldsymbol{f}^{(\boldsymbol{X})},B}$, and all non-increasing vectors $\boldsymbol{a}^{(X_1)}\in\mathbb{R}_+^{[n_1]},\ldots,\boldsymbol{a}^{(X_d)}\in\mathbb{R}_+^{[n_d]}$:

$$(\boldsymbol{M}\circ\boldsymbol{\sigma})\cdot\boldsymbol{a}^{(X_1)}\cdots\boldsymbol{a}^{(X_d)} \leq \boldsymbol{C}\cdot\boldsymbol{a}^{(X_1)}\cdots\boldsymbol{a}^{(X_d)} \tag{3.17}$$

In the rest of this section we describe the worst-case tensor $\boldsymbol{C}$. If its entries are ≥ 0 and $\leq B$, then $\boldsymbol{C}\in\mathcal{M}^+_{\boldsymbol{f}^{(\boldsymbol{X})},B}$ and, by setting $\boldsymbol{M}\stackrel{\text{def}}{=}\boldsymbol{C}$ and $\boldsymbol{\sigma}\stackrel{\text{def}}{=}$ the identity permutations, the relation S represented by $\boldsymbol{M}\circ\boldsymbol{\sigma}$ maximizes $|Q_{\texttt{star}}|$, achieving our goal. But, somewhat surprisingly, we found that sometimes this worst-case $\boldsymbol{C}$ has entries $> B$ or < 0, yet it still achieves our goal of a tight upper bound for $|Q_{\texttt{star}}|$. This is why we allow $\boldsymbol{C}\in\mathcal{M}_{\boldsymbol{f}^{(\boldsymbol{X})},\infty}$.

3.2.2 Main Result

Let Δ_Z denote the *discrete derivative* of an $\boldsymbol{X}$-tensor w.r.t. a variable $Z\in\boldsymbol{X}$, and Σ_Z denote the *discrete integral*. Formally, if $\boldsymbol{a}\in\mathbb{R}^{[n]}$ is a Z-vector, then, setting $a_0\stackrel{\text{def}}{=}0$:

$$\forall i\in[n]: \qquad (\Delta_Z\boldsymbol{a})_i \stackrel{\text{def}}{=} a_i - a_{i-1} \qquad\qquad (\Sigma_Z\boldsymbol{a})_i = \sum_{j=1,i} a_j \tag{3.18}$$

Notice that:

$$\Sigma_Z(\Delta_Z \boldsymbol{a}) = \Delta_Z(\Sigma_Z \boldsymbol{a}) = \boldsymbol{a} \qquad \texttt{SUM}_Z(\Delta_Z \boldsymbol{a}) = a_n \tag{3.19}$$

We apply Δ, Σ to multi-coordinate tensors, e.g. if $\boldsymbol{M}$ is an XYZ-tensor, then $(\Delta_Y \boldsymbol{M})_{xyz} \stackrel{\text{def}}{=} M_{xyz} - M_{x(y-1)z}$. One should think of the three operators $\Delta_X, \Sigma_X, \texttt{SUM}_X$ as analogous to the continuous operators $\frac{d\cdots}{dx}$, $\int \cdots dx$, $\int_0^n \cdots dx$.

Definition 3.2.1. Given $\boldsymbol{f}^{(\boldsymbol{X})}$, B as in Problem 2, the *value* tensor, $\boldsymbol{V}^{\boldsymbol{f}^{(\boldsymbol{X})},B} \in \mathbb{R}_+^{[\boldsymbol{n}]}$, is defined by the following linear optimization problem:

$$\forall \boldsymbol{m} \in [\boldsymbol{n}]: \qquad V_{\boldsymbol{m}}^{\boldsymbol{f}^{(\boldsymbol{X})},B} \stackrel{\text{def}}{=} \text{ Maximize: } \sum_{\boldsymbol{s} \le \boldsymbol{m}} M_{\boldsymbol{s}} \tag{3.20}$$

$$\text{Where: } \boldsymbol{M} \in \mathbb{R}_+^{[\boldsymbol{m}]}; \boldsymbol{M} \in \mathcal{M}^+_{\boldsymbol{f}^{(\boldsymbol{X})},B}$$

The *worst-case* tensor, $\boldsymbol{C}^{\boldsymbol{f}^{(\boldsymbol{X})},B} \in \mathbb{R}^{[\boldsymbol{n}]}$, is defined as:

$$\boldsymbol{C}^{\boldsymbol{f}^{(\boldsymbol{X})},B} \stackrel{\text{def}}{=} \Delta_{X_1} \cdots \Delta_{X_d} \boldsymbol{V}^{\boldsymbol{f}^{(\boldsymbol{X})},B} \tag{3.21}$$

In other words, for any prefix of dimensions $m_1 \le n_1, \ldots, m_d \le n_d$, the value $V_{m_1,\ldots,m_d}$ is the maximum volume that can be stored in the hyper-rectangle $[m_1] \times \cdots \times [m_d]$ while satisfying the degree constraints[4] $\boldsymbol{f}^{(\boldsymbol{X})}$. We will drop the superscripts when clear from the context, and write simply $\boldsymbol{V}, \boldsymbol{C}$.

Intuitively, our proofs proceed by permuting dimensions so that lower indexed entries match with larger values in the join partners. This means that the tensor which maximizes the join size will necessarily push the weight to the "upper left" corner as much as possible. By definition, the value tensor describes how much weight can be pushed into that corner without breaking the degree constraints, and the worst-case tensor $\boldsymbol{C}^{\boldsymbol{f}^{(\boldsymbol{X})},B} \in \mathbb{R}^{[\boldsymbol{n}]}$ is a concrete tensor which realizes it.

Our main result in this section is:

[4]Since the tensor $\boldsymbol{M} \in \mathbb{R}_+^{[\boldsymbol{m}]}$ has only $m_p \le n_p$ dimensions in coordinate X_p, the notation $\boldsymbol{M} \in \mathcal{M}^+_{\boldsymbol{f}^{(\boldsymbol{X})},B}$ is with some abuse, because the vector $f^{(X_p)}$ has n_p dimensions. We simply assume that we take the restriction of $f^{(X_p)}$ to the first m_p dimensions.

Theorem 3.2.2. *Let $\boldsymbol{f}^{(\boldsymbol{X})}, B$ be given as above, and let $\boldsymbol{V}, \boldsymbol{C}$ defined by* (3.20)-(3.21). *Then:*

1. *$\boldsymbol{C}$ is a solution to Problem 2, i.e. $\boldsymbol{C} \in \mathcal{M}_{\boldsymbol{f}^{(\boldsymbol{X})},\infty}$ and it satisfies Eq.* (3.17). *Furthermore, it is tight in the following sense: there exists a tensor $\boldsymbol{M} \in \mathcal{M}^{+}_{\boldsymbol{f}^{(\boldsymbol{X})},B}$ and non-increasing vectors $\boldsymbol{a}^{(p)} \in \mathbb{R}_{+}^{[n_p]}$, $p = 1, ..., d$, such that inequality* (3.17) *(with $\boldsymbol{\sigma}$ the identity) is an equality.*

2. *When the number of coordinates is $d = 2$ then $\boldsymbol{C}$ is integral and non-negative. If $d \geq 3$, $\boldsymbol{C}$ may have negative entries.*

3. *If $B < \infty$, then $\boldsymbol{C}$ may not be consistent with B, even if $d = 2$.*

4. *If there exists any solution $\boldsymbol{C}' \in \mathcal{M}^{+}_{\boldsymbol{f}^{(\boldsymbol{X})},B}$ to Problem 2, then $\boldsymbol{C}' = \boldsymbol{C}$.*

5. *For any non-increasing vectors $\boldsymbol{a}^{(X_p)} \in \mathbb{R}_{+}^{[n_p]}$, $p = 2, ..., d$, the vector $\boldsymbol{C} \cdot \boldsymbol{a}^{(X_2)} \cdots \boldsymbol{a}^{(X_d)}$ is in $\mathbb{R}_{+}^{[n_1]}$ and non-increasing.*

6. *Assume $B = \infty$. Then the following holds:*

$$\forall \boldsymbol{m} \in [\boldsymbol{n}]: \qquad V_{\boldsymbol{m}} = \min\left(F^{(X_1)}_{m_1}, \ldots, F^{(X_d)}_{m_d}\right) \tag{3.22}$$

where $F^{(X_p)}_r \stackrel{def}{=} \sum_{j \leq r} f^{(X_p)}_j$ is the CDF associated to the PDF $\boldsymbol{f}^{(X_p)}$, for $p = 1, ..., d$. Moreover, $\boldsymbol{C}$ can be computed by Algorithm 1, which runs in time $\mathbf{O}(\sum_p n_p)$. This further implies that $\boldsymbol{C} \geq 0$, in other words $\boldsymbol{C} \in \mathcal{M}^{+}_{\boldsymbol{f}^{(\boldsymbol{X})},\infty}$.

This theorem is the main technical result of our paper, and provides all the ingredients necessary for the upper bound in Sec. 3.3 and the compression in Sec. 3.5. We explain the theorem here, provide some selected proofs, and defer the rest of the proofs to Appendix B.

Item 1 of the theorem states that the tensor $\boldsymbol{C}$ in (3.21), is indeed the optimal solution to Problem 2. Items 2 and 3 state that, somewhat surprisingly, $\boldsymbol{C}$ may be inconsistent w.r.t. B, and may even be negative. Item 4 proves that, when this were to happen, then no consistent solution exists to Problem 2, hence we have to use $\boldsymbol{C}$, as defined by (3.21).

Algorithm 1 Efficient construction of $\boldsymbol{C}$ when $B = \infty$

$\forall p = 1, d : s_p \leftarrow 1; \quad \boldsymbol{C} = 0;$

while $\forall p : s_p < n_p$ **do**

$p_{min} \leftarrow \text{argmin}_p(f_{s_p}^{(X_p)}) \qquad d_{min} \leftarrow \min_p(f_{s_p}^{(X_p)})$

$C_{s_1,\ldots,s_d} \leftarrow d_{min}$

$\forall p = 1, d : f_{s_p}^{(X_p)} \leftarrow f_{s_p}^{(X_p)} - d_{min}$

$s_{p_{min}} \leftarrow s_{p_{min}} + 1$

end while

return C

Item 5 is an important technical lemma needed in Sec. 3.3. Recall that the dot product $\boldsymbol{C} \cdot \boldsymbol{a}^{(X_1)} \cdots \boldsymbol{a}^{(X_d)}$ is a scalar value, since all coordinates $X_1, \ldots, X_d$ have been summed out. The dot product $\boldsymbol{C} \cdot \boldsymbol{a}^{(X_2)} \cdots \boldsymbol{a}^{(X_d)}$ is a n_1-dimensional array, since we do not sum out X_1. Item 5 says that this is always non-negative and non-increasing.

Finally, the last item gives more insight into $\boldsymbol{V}$ and, by extension, into $\boldsymbol{C}$. Recall that $V_{\boldsymbol{m}}$, defined by (3.20), is the largest possible sum of a consistent $m_1 \times m_2 \times \cdots \times m_d$ tensor $\boldsymbol{M}$. By construction, $\boldsymbol{M}$ satisfies the degree constraints $\boldsymbol{f}^{(p)}$, in other words (see (3.9)):

$$\texttt{SUM}_{\boldsymbol{X}-\{X^{(p)}\}}(\boldsymbol{M}) \leq \boldsymbol{f}^{(p)}$$

By definition:

$$\boldsymbol{V_m} = \texttt{SUM}_{\boldsymbol{X}}(\boldsymbol{M}) = \texttt{SUM}_{X^{(p)}}\left(\texttt{SUM}_{\boldsymbol{X}-\{X^{(p)}\}}(\boldsymbol{M})\right) \leq \texttt{SUM}_{X^{(p)}}(\boldsymbol{f}^{(p)}) = \boldsymbol{F}_{m_p}^{(p)}$$

Since this holds for every $p = 1, d$, it follows that $\boldsymbol{V_m} \leq \min(\boldsymbol{F}_{m_1}^{(1)}, \ldots, \boldsymbol{F}_{m_d}^{(d)})$. Item 6 states that this inequality becomes an equality, when $B = \infty$.

We illustrate the worst-case tensor with three examples.

Example 3.2.3. *We start with a simple example. Suppose that we want to maximize* $\boldsymbol{a}^T \cdot \boldsymbol{M} \cdot \boldsymbol{b}$*, where* $\boldsymbol{M}$ *is a* 3×4 *matrix satisfying the degree sequences* $\boldsymbol{f} = (6, 3, 1)$ *and* $\boldsymbol{g} = (4, 3, 2, 1)$*; here we assume* $B = \infty$*. If we view* $\boldsymbol{M}, \boldsymbol{a}, \boldsymbol{b}$ *as* XY*-,* X*-, and* Y*-tensors respectively, then, in the notation of Problem 2, our goal is to find* $\boldsymbol{M}$ *to maximize* $\boldsymbol{M} \cdot \boldsymbol{a} \cdot \boldsymbol{b}$*, but we prefer to call the coordinates* $1, 2$ *instead of* X, Y*, to reduce clutter. The vectors* $\boldsymbol{a}, \boldsymbol{b}$

are non-negative and non-increasing, but otherwise unknown. The optimal matrix $\boldsymbol{M}$ *is the matrix* $\boldsymbol{C}$ *given by item 6 of the theorem. To compute it, we start by computing the CDFs of* $\boldsymbol{f}, \boldsymbol{g}$*:*

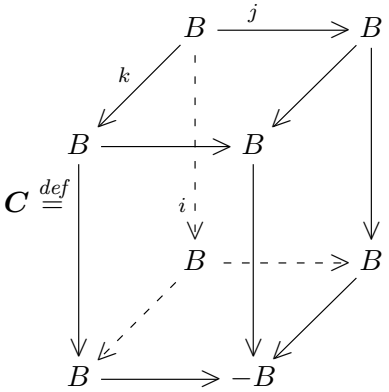
$$\boldsymbol{F} = \Sigma \boldsymbol{f} = (6, 9, 10) \qquad\qquad \boldsymbol{G} = \Sigma \boldsymbol{g} = (4, 7, 9, 10)$$

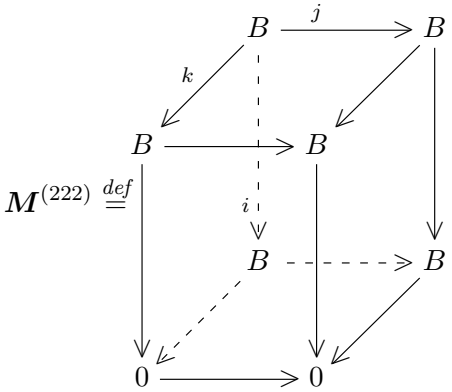
Then we compute $\boldsymbol{V}_{m_1 m_2} = \min(F_{m_1}, G_{m_2})$ *(as per* (3.22)*), and derivate it to obtain* $\boldsymbol{C} = \Delta_1 \Delta_2 \boldsymbol{V}$*:*

$$\boldsymbol{C} = \begin{array}{c} \\ \\ \\ \boldsymbol{g} \end{array} \begin{array}{c} \begin{array}{cccc} & & & \end{array} \boldsymbol{f} \\ \begin{pmatrix} 4 & 2 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{array}{c} 6 \\ 3 \\ 1 \end{array} \\ \begin{array}{cccc} 4 & 3 & 2 & 1 \end{array} \end{array} \qquad \boldsymbol{V} = \begin{array}{c} \\ \\ \\ \boldsymbol{G} \end{array} \begin{array}{c} \begin{array}{cccc} & & & \end{array} \boldsymbol{F} \\ \begin{pmatrix} 4 & 6 & 6 & 6 \\ 4 & 7 & 9 & 9 \\ 4 & 7 & 9 & 10 \end{pmatrix} \begin{array}{c} 6 \\ 9 \\ 10 \end{array} \\ \begin{array}{cccc} 4 & 7 & 9 & 10 \end{array} \end{array}$$

We can check that $V_{m_1 m_2} = \min(F_{m_1}, G_{m_2})$*, for example* $V_{31} = \min(F_3, G_1) = \min(10, 4) = 4$*. Similarly, we can check that* $C_{m_1 m_2} = V_{m_1 m_2} - V_{m_1 - 1, m_2} - V_{m_1, m_2 - 1} + V_{m_1 - 1, m_2 - 1}$*, for example,* $C_{23} = V_{23} - V_{13} - V_{22} + V_{12} = 9 - 6 - 7 + 6 = 2$*. Alternatively,* $\boldsymbol{C}$ *can be computed greedily, using Algorithm 1: start with* $C_{11} \leftarrow \min(f_1, g_1) = 4$*, decrease both* f_1, g_1 *by 4, set the rest of column 1 to 0 (because now* $g_1 = 0$*) and continue with* C_{12}*, etc.*

Example 3.2.4. *This example proves part of Item 2 of the theorem: that* $\boldsymbol{C}$ *may have negative entries when* $d = 3$*. Consider the degree constraints* $\boldsymbol{f} = (\infty, 2B)$*,* $\boldsymbol{g} = \boldsymbol{h} = (\infty, \infty)$*, and the max tuple multiplicity is* $B > 0$*. Here* ∞ *means "a very large number", much larger than* B*. Define the following tensors with* $d = 3$ *coordinates:*





We claim the tensor $\boldsymbol{C}$ *above is precisely* $\boldsymbol{C} = \boldsymbol{C}^{(\boldsymbol{f},\boldsymbol{g},\boldsymbol{h}),B}$*, as given by definition* (3.21)*. Since* $C_{222} = -B$*, this represents an example of a worst-case tensor that is negative. We prove the claim. By definition* (3.21)*,* $\boldsymbol{C}^{(\boldsymbol{f},\boldsymbol{g},\boldsymbol{h}),B} = \Delta_1\Delta_2\Delta_3 V^{(\boldsymbol{f},\boldsymbol{g},\boldsymbol{h}),B}$*, where* $\boldsymbol{V}$ *is given by* (3.20)*. To prove the claim, it suffices to prove that* $\boldsymbol{C} = \Delta_1\Delta_2\Delta_3 V^{(\boldsymbol{f},\boldsymbol{g},\boldsymbol{h}),B}$ *or, equivalently, that* $\Sigma_1\Sigma_2\Sigma_3\boldsymbol{C} = V^{(\boldsymbol{f},\boldsymbol{g},\boldsymbol{h}),B}$*. Let* $\boldsymbol{M}^{(m_1m_2m_3)}$ *denote the optimal* $m_1 \times m_2 \times m_3$*-tensor that maximizes the quantity in Eq.* (3.20) *and satisfies the constraints* $\boldsymbol{f}, \boldsymbol{g}, \boldsymbol{h}, B$*. In other words,* $V_{m_1m_2m_3} = \sum_{i\le m_1, j\le m_2, k\le m_3} M^{(m_1m_2m_3)}_{ijk}$*. We observe that, for all* (m_1, m_2, m_3) *other than* $(2, 2, 2)$*,* $\boldsymbol{M}^{(m_1m_2m_3)}$ *is precisely the* $m_1 \times m_2 \times m_3$*-subtensor of the tensor* $\boldsymbol{C}$ *defined above. This is because* $\boldsymbol{C}$ *satisfies the constraints* $\boldsymbol{f}, \boldsymbol{g}, B$*, is non-negative (except for* C_{222}*), and maximizes the sum in the* $m_1 \times m_2 \times m_3$ *subtensor. Therefore, we have* $(\Sigma_1\Sigma_2\Sigma_3\boldsymbol{C})_{m_1m_2m_3} = \sum_{i\le m_1, j\le m_2, k\le m_3} C_{ijk} = \sum_{i\le m_1, j\le m_2, k\le m_3} M^{(m_1m_2m_3)}_{ijk} = V^{(\boldsymbol{f},\boldsymbol{g},\boldsymbol{h}),B}_{m_1m_2m_3}$ *for all* $(m_1, m_2, m_3) \neq (2, 2, 2)$*. It remains to prove* $(\Sigma_1\Sigma_2\Sigma_3\boldsymbol{C})_{222} = V^{(\boldsymbol{f},\boldsymbol{g},\boldsymbol{h}),B}_{222}$*. On one hand,* $(\Sigma_1\Sigma_2\Sigma_3\boldsymbol{C})_{222} = B + B + B + B + B + B + B - B = 6B$*. On the other hand, one can check that the optimal matrix* $\boldsymbol{M}^{(222)}$ *is the one shown above: it is non-negative, it satisfies the* $\boldsymbol{f}, \boldsymbol{g}, \boldsymbol{h}, B$*-constraints, and its total sum is* $6B$*, which is maximal because, for any consistent tensor* $\boldsymbol{M}$ *we have:*

$$\sum_{i,j,k} M_{ijk} = M_{111} + M_{112} + M_{121} + M_{122} + M_{211} + M_{212} + M_{221} + M_{222} \le B + B + B + B + f_2 = 6B$$

Therefore $V^{(\boldsymbol{f},\boldsymbol{g},\boldsymbol{h}),B}_{222} = \sum_{i,j,k} M^{(222)}_{ijk} = 6B$*, which completes the proof.*

Example 3.2.5. *This example proves item 3 of the theorem: if* $B < \infty$*, then even when* $d = 2$*, the worst-case matrix* $\boldsymbol{C}$ *may fail the constraint* B*, i.e. may have entries* $> B$*. Note that, by item 2,* $\boldsymbol{C}$ *is non-negative. Let* $d = 2$*, let* $\boldsymbol{f} = \boldsymbol{g} = (2B, 2B, B + 1)$ *be degree sequences, and let* $B < \infty$ *be the max tuple multiplicity. Consider the two matrices below:*

$$\mathbf{C} = \begin{array}{r} \\ 2B \\ 2B \\ B+1 \end{array} \begin{array}{c} \begin{array}{ccc} 2B & 2B & B+1 \end{array} \\ \begin{pmatrix} B & B & 0 \\ B & B & 0 \\ 0 & 0 & B+1 \end{pmatrix} \end{array} \qquad \mathbf{M}^{(33)} = \begin{array}{r} \\ 2B \\ 2B \\ B+1 \end{array} \begin{array}{c} \begin{array}{ccc} 2B & 2B & B+1 \end{array} \\ \begin{pmatrix} B & B & 0 \\ B & B-1 & 1 \\ 0 & 1 & B \end{pmatrix} \end{array}$$

Following an argument similar to that in Example 3.2.4, we claim that the matrix $\boldsymbol{C}$ *above is precisely* $\boldsymbol{C} = \boldsymbol{C}^{(\boldsymbol{f},\boldsymbol{g}),B}$*, as given by definition* (3.21)*. Since* $C_{33} > B$*, this represents an example of a worst-case matrix that is inconsistent. We prove the claim. By*

definition (3.21), $\boldsymbol{C}^{(\boldsymbol{f},\boldsymbol{g}),B} = \Delta_1\Delta_2 V^{(\boldsymbol{f},\boldsymbol{g}),B}$, *where* $\boldsymbol{V}$ *is given by* (3.20). *To prove the claim it suffices to prove that* $\boldsymbol{C} = \Delta_1\Delta_2 V^{(\boldsymbol{f},\boldsymbol{g}),B}$ *or, equivalently, that* $\Sigma_1\Sigma_2\boldsymbol{C} = V^{(\boldsymbol{f},\boldsymbol{g}),B}$. *Let* $\boldsymbol{M}^{(m_1m_2)}$ *denote the optimal* $m_1 \times m_2$*-matrix that maximizes the quantity in Eq.* (3.20) *and satisfies the constraints* $\boldsymbol{f}, \boldsymbol{g}, B$. *In other words,* $V_{m_1m_2} = \sum_{i\le m_1, j\le m_2} M_{ij}^{(m_1m_2)}$. *We observe that, for all* (m_1, m_2) *other than* $(3,3)$, $\boldsymbol{M}^{(m_1m_2)}$ *is precisely the* $m_1 \times m_2$*-submatrix of the matrix* $\boldsymbol{C}$ *defined above. This is because,* $\boldsymbol{C}$ *satisfies the constraints* $\boldsymbol{f}, \boldsymbol{g}, B$ *(except for the element* C_{33}*), and it maximizes the sum in the* $m_1 \times m_2$ *submatrix. Therefore, we have* $(\Sigma_1\Sigma_2\boldsymbol{C})_{m_1m_2} = \sum_{i\le m_1, j\le m_2} C_{ij} = \sum_{i\le m_1, j\le m_2} M_{ij}^{(m_1m_2)} = V_{m_1m_2}^{(\boldsymbol{f},\boldsymbol{g}),B}$ *for all* $(m_1, m_2) \neq (3,3)$. *It remains to prove* $(\Sigma_1\Sigma_2\boldsymbol{C})_{33} = V_{33}^{(\boldsymbol{f},\boldsymbol{g}),B}$, *On one hand,* $(\Sigma_1\Sigma_2\boldsymbol{C})_{33} = 5B + 1$. *On the other hand, one can check that the optimal matrix* $\boldsymbol{M}^{(33)}$ *is the one shown above: it satisfies the* $\boldsymbol{f}, \boldsymbol{g}, B$*-constraints, and its total sum is* $5B + 1$, *which is maximal because* $f_1 + f_2 + f_3 = 5B + 1$. *Therefore* $V_{33}^{(\boldsymbol{f},\boldsymbol{g}),B} = \sum_{i\le 3, j\le 3} M_{ij}^{(33)} = 5B + 1$, *which completes the proof.*

3.2.3 Proof

In this section we present some of the main ingredients of the proof of Theorem 3.2.2, and defer the rest to the appendix. This section can be skipped at a first reading, since we only need the statement of Theorem 3.2.2, and not its proof, in the rest of the paper.

Proof of Item 1 First, we show that it suffices to restrict $\boldsymbol{\sigma}$ in Problem 2 to be the identity permutations. More precisely, we prove:

Lemma 3.2.6. *For any tensor* $\boldsymbol{M} \in \mathcal{M}^+_{\boldsymbol{f}(\boldsymbol{X}),B}$ *and any tuple of permutations* $\boldsymbol{\sigma} \in S_{[\boldsymbol{n}]}$ *there exists a tensor* $\boldsymbol{N} \in \mathcal{M}^+_{\boldsymbol{f}(\boldsymbol{X}),B}$ *such that, for all non-increasing vectors* $\boldsymbol{a}^{(X_p)} \in \mathbb{R}_+^{[n_p]}$, $p = 1, d$, *the following holds:*

$$(\boldsymbol{M} \circ \boldsymbol{\sigma}) \cdot \boldsymbol{a}^{(X_1)} \cdots \boldsymbol{a}^{(X_d)} \le \boldsymbol{N} \cdot \boldsymbol{a}^{(X_1)} \cdots \boldsymbol{a}^{(X_d)}$$

Intuitively, if $\boldsymbol{M}$ were a matrix, then we would rearrange its rows and columns such that the row-sums and column-sums are non-decreasing: the resulting tensor $\boldsymbol{N}$ proves the lemma. The formal proof is more subtle than that, and is deferred to Appendix B.1. The lemma implies that $\boldsymbol{\sigma}$ can be chosen to be the identity in Problem 2, because we can simply

replace $\boldsymbol{M}$ and $\boldsymbol{\sigma}$, with $\boldsymbol{N}$ and the identity permutation. In other words that, in order to compute the upper bound to $Q_{\texttt{star}}$, it suffices to restrict the relation instance S to have the highest degrees aligned with the highest degrees of the unary relations $R^{(p)}$, $p = 1, d$.

Therefore, condition (3.17) simplifies, because we can assume that $\boldsymbol{\sigma}$ is the identity. To check the condition we need to show that, for all non-decreasing vectors $\boldsymbol{a}^{(X_p)} \in \mathbb{R}_+^{[n_p]}$, $p = 1, d$, and for every tensor $\boldsymbol{M} \in \mathcal{M}^+_{\boldsymbol{f}^{(\boldsymbol{X})},B}$, the following holds, where $\boldsymbol{C}$ is defined by (3.21):

$$\boldsymbol{M} \cdot \boldsymbol{a}^{(X_1)} \cdots \boldsymbol{a}^{(X_d)} \leq \boldsymbol{C} \cdot \boldsymbol{a}^{(X_1)} \cdots \boldsymbol{a}^{(X_d)} \tag{3.23}$$

Call $\boldsymbol{b} \in \mathbb{R}^n$ a *one-zero vector* if $\boldsymbol{b} = (1, 1, \ldots, 1, 0, \ldots, 0)$. For $m = 1, n$, denote by $\boldsymbol{b}^{(m)}$ the one-zero vector with exactly m 1's. Every non-increasing $\boldsymbol{a} \in \mathbb{R}^n_+$ is a positive linear combination of $\boldsymbol{b}^{(m)}$'s.

$$\boldsymbol{a} = (a_1 - a_2)\boldsymbol{b}^{(1)} + (a_2 - a_3)\boldsymbol{b}^{(2)} + \cdots + (a_{n-1} - a_n)\boldsymbol{b}^{(n-1)} + a_n\boldsymbol{b}^{(n)}$$

This implies that, in order to prove (3.23), it suffices to check it when each vector $\boldsymbol{a}^{(X_p)}$ is some one-zero vector $\boldsymbol{b}^{(X_p,m)}$, where we include X_p in the superscript to indicate that this is an X_p-vector. In that case, the LHS of (3.23) is:

$$\boldsymbol{M} \cdot \boldsymbol{b}^{(X_1,m_1)} \cdots \boldsymbol{b}^{(X_d,m_d)} = (\Sigma_{X_1} \cdots \Sigma_{X_d}\boldsymbol{M})_{\boldsymbol{m}} \quad \boldsymbol{C} \cdot \boldsymbol{b}^{(X_1,m_1)} \cdots \boldsymbol{b}^{(X_d,m_d)} = (\Sigma_{X_1} \cdots \Sigma_{X_d}\boldsymbol{C})_{\boldsymbol{m}} \tag{3.24}$$

By the definition of $V_{\boldsymbol{m}}$ in Eq. (3.20), $(\Sigma_{X_1} \cdots \Sigma_{X_d}\boldsymbol{M})_{\boldsymbol{m}} = \sum_{\boldsymbol{s} \leq \boldsymbol{m}} M_{\boldsymbol{s}} \leq V_{\boldsymbol{m}}$, therefore:

$$(\Sigma_{X_1} \cdots \Sigma_{X_d}\boldsymbol{M})_{\boldsymbol{m}} \leq V_{\boldsymbol{m}} \tag{3.25}$$

Recall that $\boldsymbol{C} \stackrel{\text{def}}{=} \Delta_{X_1} \cdots \Delta_{X_d}\boldsymbol{V}$. By repeatedly applying Eq. (3.19), we derive:

$$\Sigma_{X_1} \cdots \Sigma_{X_d}\boldsymbol{C} = \boldsymbol{V} \tag{3.26}$$

Therefore, we obtain that, for all $\boldsymbol{M} \in \mathcal{M}^+_{\boldsymbol{f}^{(\boldsymbol{X})},B}$:

$$\boldsymbol{M} \cdot \boldsymbol{b}^{(X_1,m_1)} \cdots \boldsymbol{b}^{(X_d,m_d)} \overset{(3.24),(3.25)}{\leq} V_{\boldsymbol{m}} \overset{(3.26)}{=} (\Sigma_{X_1} \cdots \Sigma_{X_d}\boldsymbol{C})_{\boldsymbol{m}} \overset{(3.24)}{=} \boldsymbol{C} \cdot \boldsymbol{b}^{(X_1,m_1)} \cdots \boldsymbol{b}^{(X_d,m_d)}$$

which proves (3.23). Furthermore, if $\boldsymbol{M}$ is the optimal solution to the optimization problem defining $\boldsymbol{V}$, then (3.25) is an equality, and so is (3.23), proving that $\boldsymbol{C}$ is tight. This completes the proof of item 1.

This almost completes the proof of item 1. It remains to prove that $\boldsymbol{C} \in \mathcal{M}_{\boldsymbol{f}^{(\boldsymbol{X})},\infty}$, in other words, that $\boldsymbol{C}$ in (3.21) satisfies the degree constraints $\boldsymbol{f}^{(\boldsymbol{X})}$. For this purpose we use the dual of linear program to (3.20): we include the discussion of the dual linear program and the rest of the proof in Appendix B.2.

Proof of Item 2 We saw in Example 3.2.4 that the entries of $\boldsymbol{C}$ can be negative when $d \geq 3$. We include the rest of the proof in Appendix B.3. For integrality, we show that the matrix of the linear program to (3.20) is totally unimodular. For proving $\boldsymbol{C} \geq 0$, we had to do a deeper analysis of the dual of (3.20). In both cases we used the assumption that $d = 2$. As we saw, when $d = 3$ then the condition $\boldsymbol{C} \geq 0$ fails; we leave open the question of whether integrality fails when $d \geq 3$.

Proof of Item 3 This follows from Example 3.2.5.

Proof of Item 4 Assume that $\boldsymbol{C}'$ satisfies the assumptions in item 4. We will use again one-zero vectors introduced earlier. On one hand, $\boldsymbol{C}$ is a solution to Problem 2, and $\boldsymbol{C}' \in \mathcal{M}^{+}_{\boldsymbol{f}^{(\boldsymbol{X})},B}$, therefore:

$$\boldsymbol{C}' \cdot \boldsymbol{b}^{(X_1,m_1)} \cdots \boldsymbol{b}^{(X_d,m_d)} \leq \boldsymbol{C} \cdot \boldsymbol{b}^{(X_1,m_1)} \cdots \boldsymbol{b}^{(X_d,m_d)} \tag{3.27}$$

This implies $\Sigma_{X_1} \cdots \Sigma_{X_d} \boldsymbol{C}' \leq \Sigma_{X_1} \cdots \Sigma_{X_d} \boldsymbol{C}$. On the other hand, $\boldsymbol{C}'$ is a solution to Problem 2, therefore, $\forall \boldsymbol{M} \in \mathcal{M}^{+}_{\boldsymbol{f}^{(\boldsymbol{X})},B}$:

$$\boldsymbol{M} \cdot \boldsymbol{b}^{(X_1,m_1)} \cdots \boldsymbol{b}^{(X_d,m_d)} \leq \boldsymbol{C}' \cdot \boldsymbol{b}^{(X_1,m_1)} \cdots \boldsymbol{b}^{(X_d,m_d)}$$

We choose $\boldsymbol{M}$ to maximize the LHS, then the LHS becomes $V_{\boldsymbol{m}}$:

$$V_{\boldsymbol{m}} \leq \boldsymbol{C}' \cdot \boldsymbol{b}^{(X_1,m_1)} \cdots \boldsymbol{b}^{(X_d,m_d)} = (\Sigma_{X_1} \cdots \Sigma_{X_d} \boldsymbol{C}')_{\boldsymbol{m}} \tag{3.28}$$

We have obtained $\Sigma_{X_1} \cdots \Sigma_{X_d} \boldsymbol{C} = \boldsymbol{V} \leq \Sigma_{X_1} \cdots \Sigma_{X_d} \boldsymbol{C}'$. The two inequalities imply $\Sigma_{X_1} \cdots \Sigma_{X_d} \boldsymbol{C} = \Sigma_{X_1} \cdots \Sigma_{X_d} \boldsymbol{C}'$, therefore $\boldsymbol{C} = \boldsymbol{C}'$. This completes the proof of item 4.

Proof of items 5, 6 This is deferred to Appendix B.4 and B.5.

3.3 The Berge-Acyclic Query

We now turn to the general problem (1) of bounding acyclic queries. To do this, we orient the query as a tree then leverage our findings in Thm. 3.2.2 to proceed bottom up through this tree, handling each internal node as a modified star query. The additional complexity comes from the fact that we are no longer simply producing a bound from the star query. Instead, we must produce a worst-case unary relation which we can use to proceed further up the tree. For the remainder of the section, we fix a Berge-acyclic query Q with relations $\boldsymbol{R} \stackrel{\text{def}}{=} \boldsymbol{R}(Q)$, degree sequences $\boldsymbol{f}^{R,Z}$, and max tuple multiplicities $B^{(R)}$ as in problem 1.

3.3.1 The Degree Sequence Bound

Theorem 3.3.1. *For any tensors $\boldsymbol{M}^{(R)} \in \mathcal{M}^{+}_{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}}$ and permutations $\boldsymbol{\sigma}^{(R)}$, for $R \in \boldsymbol{R}$, the following holds:*

$$\mathit{SUM}_{\boldsymbol{X}} \left(\bigotimes_{R \in \boldsymbol{R}} (\boldsymbol{M}^{(R)} \circ \boldsymbol{\sigma}^{(R)}) \right) \leq \mathit{SUM}_{\boldsymbol{X}} \left(\bigotimes_{R \in \boldsymbol{R}} \boldsymbol{C}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}} \right) \stackrel{def}{=} DSB(Q) \tag{3.29}$$

where $\boldsymbol{C}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}}$ is the worst-case tensor from Def. 3.2.1.

The theorem says that the upper bound to the query Q can be computed by evaluating Q on the worst case instances, represented by the worst case tensors $\boldsymbol{C}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}}$. We call this quantity the *degree sequence bound* and denote it by $DSB(Q)$. When all max tuple multiplicities, $B^{(R)}$, are ∞, the bound is tight, because in that case every worst-case tensor $\boldsymbol{C}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},\infty}$ is in $\mathcal{M}^{+}_{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},\infty}$ (by Th. 3.2.2 item 6). Otherwise, the bound may not be tight, but it is locally tight, in the sense of Th. 3.2.2 item 1.

Proof. The proof of Thm. 3.3.1 proceeds by selecting an arbitrary root relation $R_{\texttt{ROOT}}$, orienting the incidence graph as a tree, T, directed away from $R_{\texttt{ROOT}}$, and proceeding inductively from the leaves upward. At each inductive step, we show that replacing the relations of the current sub-tree with their worst-case instance can only increase the summation on the LHS of (3.29). Note that because Q is Berge-acyclic, relations in the incidence graph share a single variable with the tree above, therefore the output of any subtree is a unary relation.

Fix tensors $\boldsymbol{M}^{(R)}$ and permutations $\boldsymbol{\sigma}^{(R)}$, for each $R \in \boldsymbol{R}$. Choose any non-root relation, $S \in \boldsymbol{R}$, and assume it has k variables $X_1, \ldots, X_k$. Without loss of generality, we assume X_1 is S's parent in T and $X_2, ..., X_k$ lead to the child relations $R_2, \ldots, R_k$. Further, denote the sub-tree rooted at S as T_S and the set of variables which appear in T_S as $\boldsymbol{X}_{T_S}$. Using summation pushdown as defined in Eq. (3.6), we rewrite the LHS of (3.29) as:

$$\texttt{SUM}_{X_1}\left(\texttt{SUM}_{\boldsymbol{X}\setminus\boldsymbol{X}_{T_S}}\left(\bigotimes_{R\notin T_S}\boldsymbol{M}^{(R)}\circ\boldsymbol{\sigma}^{(R)}\right)\otimes\texttt{SUM}_{\boldsymbol{X}_{T_S}\setminus X_1}\left(\bigotimes_{R'\in T_S}\boldsymbol{M}^{(R')}\circ\boldsymbol{\sigma}^{(R')}\right)\right) \tag{3.30}$$

Given this, the following claim is the core of the proof.

Claim 1. If the following statements hold for all relations below S in T, then they hold for S.

1. $\texttt{SUM}_{\boldsymbol{X}_{T_S}\setminus X_1}\left(\bigotimes_{R\in T_S} C^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}}\right)$ is a non-increasing vector

2. For some permutation on X_1, denoted $\sigma^*_{X_1}$, the following holds

$$(3.30) \leq \texttt{SUM}_{X_1}\left(\texttt{SUM}_{\boldsymbol{X}\setminus\boldsymbol{X}_{T_S}}\left(\bigotimes_{R\notin T_S}\boldsymbol{M}^{(R)}\circ\boldsymbol{\sigma}^{(R)}\right)\otimes\left(\texttt{SUM}_{\boldsymbol{X}_{T_S}\setminus X_1}\left(\bigotimes_{R'\in T_S} C^{\boldsymbol{f}^{(R',\boldsymbol{X}_{R'})},B^{(R')}}\right)\circ\sigma^*_{X_1}\right)\right)$$

The first statement is straightforward to prove inductively. If it holds for all children, then, by pushing the summation down to the children, we can leverage Thm. 3.2.2 Item 5 to show that it holds for S as well. In the base case, it is immediately true that the worst-case tensor of leaf nodes is a sorted vector. The second statement is proved by taking advantage of the invariance of summation to permutations on dimensions. We begin by using the inductive assumption to replace the relations in each child sub-tree, T_{R_i}, with their worse-case tensors and permutation $\sigma^*_{X_i}$. We denote the result of summing over each child's sub-tree as $a^{(X_i)}\circ\sigma^*_{X_i}$ where $a^{(X_i)}$ is a non-increasing vector due to the first part of the claim,

$$a^{(X_i)}\circ\sigma^*_{X_i} = \texttt{SUM}_{\boldsymbol{X}_{T_{R_i}}\setminus X_i}\left(\bigotimes_{R\in T_{R_i}} C^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}}\right)\circ\sigma^*_{X_i}$$

At this point, we can upper bound (3.30) by,

$$(3.30) \leq \texttt{SUM}_{X_1}\left(\texttt{SUM}_{\boldsymbol{X}\setminus\boldsymbol{X}_{T_S}}\left(\bigotimes_{R\notin T_S}\boldsymbol{M}^{(R)}\circ\boldsymbol{\sigma}^{(R)}\right)\otimes\texttt{SUM}_{\boldsymbol{X}_{T_S}\setminus X_1}\left(\left(\boldsymbol{M}^{(S)}\circ\boldsymbol{\sigma}^{(S)}\right)\otimes\bigotimes_{2\leq i\leq k}a^{(X_i)}\circ\sigma^*_{X_i}\right)\right)$$

We push the permutations induced by S's children, $\sigma^*_{X_2},...,\sigma^*_{X_k}$, into $\boldsymbol{\sigma}^{(S)}$ producing a new permutation, $\boldsymbol{\sigma'}$.

$$(3.30) \leq \texttt{SUM}_{X_1}\left(\texttt{SUM}_{\boldsymbol{X}\setminus\boldsymbol{X}_{T_S}}\left(\bigotimes_{R\notin T_S}\boldsymbol{M}^{(R)}\circ\boldsymbol{\sigma}^{(R)}\right)\otimes\texttt{SUM}_{\boldsymbol{X}_{T_S}\setminus X_1}\left(\left(\boldsymbol{M}^{(S)}\circ\boldsymbol{\sigma'}\right)\otimes\bigotimes_{2\leq i\leq k}a^{(X_i)}\right)\right)$$

Next, we introduce a permutation, σ_{X_1}, to the expression such that $\texttt{SUM}_{\boldsymbol{X}\setminus\boldsymbol{X}_{T_S}}\left(\bigotimes_{R\notin T_S}\boldsymbol{M}^{(R)}\circ\boldsymbol{\sigma}^{(R)}\right)$ is non-increasing. Given that all the children of S are non-increasing by the inductive assumption, we can then apply Thm. 3.2.2 Item 1 to show that replacing $M^{(S)}\circ\boldsymbol{\sigma'}$ with $C^{\boldsymbol{f}^{(S,\boldsymbol{X}_S)},B^{(S)}}$ strictly increases the sum. At this point, all relations in the sub-tree rooted at T_S have been replaced with their worst-case tensor, resulting in the following expression,

$$(3.30) \leq \texttt{SUM}_{X_1}\left(\left(\texttt{SUM}_{\boldsymbol{X}\setminus\boldsymbol{X}_{T_S}}\left(\bigotimes_{R\notin T_S}\boldsymbol{M}^{(R)}\circ\boldsymbol{\sigma}^{(R)}\right)\circ\sigma_{X_1}\right)\otimes\texttt{SUM}_{\boldsymbol{X}_{T_S}\setminus X_1}\left(\bigotimes_{R'\in T_S}C^{\boldsymbol{f}^{(R',\boldsymbol{X}_{R'})},B^{(R')}}\right)\right)$$

To finish the proof of the claim, we apply the inverse of σ_{X_1}, denoted $\sigma^*_{X_1}$, to the expression in order to retrieve the original parent summation, $\texttt{SUM}_{\boldsymbol{X}\setminus\boldsymbol{X}_{T_S}}\left(\bigotimes_{R\notin T_S}\boldsymbol{M}^{(R)}\circ\boldsymbol{\sigma}^{(R)}\right)$. Because this claim holds for all non-root relations, it holds for all immediate children of $R_{\texttt{ROOT}}$. Therefore, we can replace each child of $R_{\texttt{ROOT}}$ with its worst-case tensor and a permutation which we push into $\boldsymbol{\sigma}^{(R_{\texttt{ROOT}})}$. After summations are pushed down to children, the resulting expression is a star query with non-increasing vectors on the points, so we can apply Thm. 3.2.2 Item 1 to replace $M^{(R_{\texttt{ROOT}})}\circ\boldsymbol{\sigma}^{(R_{\texttt{ROOT}})}$ with its worst-case tensor, completing the proof. □

3.3.2 Computing The Degree Sequence Bound

As an immediate consequence of this theorem, we can compute the degree sequence bound in linear time by exhaustively pushing the summations down into the products (note that this is a special case of the FAQ algorithm [88]). We describe this algorithm briefly in Algorithm 2 and provide intuition here. Recall that the incidence graph, T, of a query Q

has a vertex for each variable and relation, and it contains an edge whenever a variable is present within a relation in Q. Because Q is a Berge-acyclic query, T must be a tree. Choose an arbitrary relation $\texttt{ROOT} \in \boldsymbol{R}(Q)$ and designate it as root, then make T a directed tree by orienting all its edges away from the root. Denote by $\texttt{parent}(R) \in \boldsymbol{X}_R$ the parent node of a relation $R \neq \texttt{ROOT}$, associate an X-vector $\boldsymbol{a}^{(X)}$ to each variable X, and a $\texttt{parent}(R)$-vector $\boldsymbol{w}^{(R)}$ to each relation name R, then compute these vectors by traversing the tree bottom-up, as shown in Algorithm 2. Notice that, when X is a leaf variable, then $\text{children}(X) = \emptyset$ and $\boldsymbol{a}^{(X)} = (1, 1, \ldots, 1)^T$; similarly, if $R(X)$ is leaf relation of arity 1 with variable X, then $\boldsymbol{w}^{(R)}$ is the degree sequence of its variable, because $\boldsymbol{w}^{(R)} = \boldsymbol{C}^{(\boldsymbol{f}^{(R,X)}, B^{(R)})} = \boldsymbol{f}^{(R,X)}$. From this discussion, it follows that,

Corollary 3.3.2. *The degree sequence bound $DSB(Q)$ can be computed in time linear in the size of the worst-case input and the size of the query (combined complexity). When $B = \infty$ for all relations, Thm. 3.2.2.6 implies that this is linear in the size of the largest domain.*

We illustrate Algorithm 2 with an example.

Example 3.3.3. *Let:*

$$Q(X, Y, Z, U) := R(X, Y) \bowtie S(Y, Z) \bowtie T(Z, U) \bowtie K(U)$$

Assume we are given degree sequences for each attribute of each relation, and we compute the corresponding worst case tensors (one for each relation): $\boldsymbol{C}^{(R)}, \boldsymbol{C}^{(S)}, \boldsymbol{C}^{(T)}, \boldsymbol{C}^{(K)}$: the first three are matrices, the last is a vector. The degree sequence bound is:

$$\texttt{SUM}_{XYZU} \left(\boldsymbol{C}^{(R)} \otimes \boldsymbol{C}^{(S)} \otimes \boldsymbol{C}^{(T)} \otimes \boldsymbol{C}^{(K)} \right) = \sum_{xyzu} C^{(R)}_{xy} C^{(S)}_{yz} C^{(T)}_{zu} C^{(K)}_{u}$$

Algorithm 2 allows us to compute this expression as follows. Fig. 3.3 shows the incidence graph of this query which will guide the execution of the algorithm. We choose any relation to designate as root: we chose (arbitrarily) S as the root, associate a vector $\boldsymbol{a}^{(X)}, \boldsymbol{a}^{(Y)}, \boldsymbol{a}^{(Z)}, \boldsymbol{a}^{(U)}$ to each variable, and a vector $\boldsymbol{w}^{(R)}, \boldsymbol{w}^{(S)}, \boldsymbol{w}^{(T)}, \boldsymbol{w}^{(K)}$ to each relation. Then, we compute the degree sequence bound as follows, where we show the type of

Algorithm 2 Computing $DSB(Q) = \texttt{SUM}_{\boldsymbol{X}} \left(\bigotimes_{R \in \boldsymbol{R}} \boldsymbol{C}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)}, B^{(R)}} \right)$

for each variable $X \in \boldsymbol{X}$ and non-root relation $R \in \boldsymbol{R}$, $R \neq \texttt{root}$, in bottom-up order **do**

$\boldsymbol{a}^{(X)} \stackrel{\text{def}}{=} \bigotimes_{R \in \text{children}(X)} \boldsymbol{w}^{(R)}$ // element-wise product

$\boldsymbol{w}^{(R)} \stackrel{\text{def}}{=} \boldsymbol{C}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)}, B^{(R)}} \cdot \boldsymbol{a}^{(X_2)} \cdots \boldsymbol{a}^{(X_k)}$ // where $\boldsymbol{X}_R = (X_1, \ldots, X_k)$, $X_1 = \texttt{parent}(R)$

end for

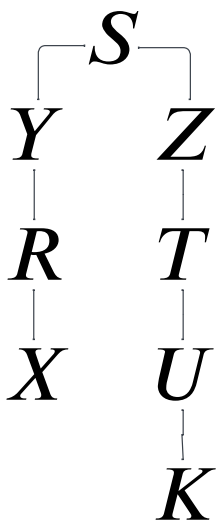
return $C^{\boldsymbol{f}^{(\texttt{root},\boldsymbol{X}_{\texttt{ROOT}})}, B^{(\texttt{ROOT})}} \cdot \boldsymbol{a}^{(X_1)} \cdot \boldsymbol{a}^{(X_2)} \cdots \boldsymbol{a}^{(X_k)}$

each vector, assuming that the domains of the variables X, Y, Z, U are $[n_X], [n_Y], [n_Z], [n_U]$ respectively. This execution is depicted line-by-line in fig. 3.2.

Figure 3.2: Example Algorithm Execution

$$
\begin{aligned}
a^{(U)} &:= C^{(K)} && \in \mathbb{R}_+^{[n_U]} \\
w^{(T)} &:= C^{(T)} \cdot a^{(U)} && \in \mathbb{R}_+^{[n_Z]} \\
a^{(Z)} &:= w^{(T)} && \in \mathbb{R}_+^{[n_Z]} \\
a^{(X)} &:= \mathbf{1} && \in \mathbb{R}_+^{[n_X]} \\
w^{(R)} &:= C^{(R)} \cdot a^{(X)} && \in \mathbb{R}_+^{[n_Y]} \\
a^{(Y)} &:= w^{(R)} && \in \mathbb{R}_+^{[n_Y]} \\
&\textit{return} \left(C^{(S)} \cdot a^{(Y)} \cdot a^{(Z)} \right)
\end{aligned}
$$

Figure 3.3: Incidence Graph



3.4 Comparison with the AGM and Polymatroid Bounds

We now prove that $DSB(Q)$ is always below the AGM [10] and the polymatroid bounds [90, 112]. Note that these bounds are originally constructed using set semantics, and $DSB(Q)$ is defined under bag semantics. To bridge this gap, we will show that under bag semantics the DSB is below both prior bounds, and a minor extension, denoted $DSB_{\texttt{SET}}$, improves

on both prior bounds even under set semantics.

The AGM bound is expressed in terms of the cardinalities of the relations. For each relation R, let N_R be an upper bound on its cardinality. Then, the AGM bound is $AGM(Q) \stackrel{\text{def}}{=} \min_{\boldsymbol{w}} \prod_R N_R^{w_R}$, where the vector $\boldsymbol{w} = (w_R)_{R \in \boldsymbol{R}}$ ranges over the fractional edge covers of the hypergraph associated to Q. If a database instance satisfies $|R| \leq N_R$ for all R, then the size of the query is $|Q| \leq AGM(Q)$, and this bound is tight, i.e. there exists an instance satisfying the cardinality constraints for which we have equality.

The polymatroid bound uses both the cardinality constraints, N_R, and the maximum degrees, $f_1^{(R,X)}$. The general bound in [90] considers maximum degrees for any subset of variables, but throughout this paper we restrict to degrees of single variables, in which case the polymatroid bound is expressed in terms of the quantities N_R and $f_1^{(R,X)}$, one for each relation R and each of its variables X. The AGM bound is the special case where we only have access to the relations' sizes. We review the general definition of the polymatroid bound in Sec. A.3.1, but will mention that no closed formula is known for the polymatroid bound, similar to the AGM bound. We give here the first such closed formula, for the case of Berge-acyclic queries. Let Q be a Berge-acyclic query with incidence graph T (which is a tree). We choose an arbitrary relation, $\texttt{ROOT} \in \boldsymbol{R}(Q)$, to designate as the root of T. For each other relation R, we denote its unique parent variable by $Z_R \stackrel{\text{def}}{=} \texttt{parent}(R)$. Denote by:

$$PB(Q, \texttt{ROOT}) \stackrel{\text{def}}{=} N_{\texttt{ROOT}} \prod_{R \neq \texttt{ROOT}} f_1^{(R,Z_R)} \tag{3.31}$$

One can immediately check that the query answer on any database instance consistent with the statistics satisfies $|Q| \leq PB(Q, \texttt{ROOT})$. A *vertex cover* of Q is set $\boldsymbol{W}_V = \{Q_1, Q_2, \ldots, Q_m\}$, for some $m \geq 1$, where each Q_i is a connected subquery of Q, and each variable of Q occurs in at least one Q_i. Similarly, we define an *edge cover* of Q as a set $\boldsymbol{W}_E = \{Q_1, Q_2, \ldots, Q_m\}$ where each relation of Q occurs in at least one Q_i.

$$PB(\boldsymbol{W}) \stackrel{\text{def}}{=} \prod_{i=1,m} \min_{\texttt{ROOT} \in \boldsymbol{R}(Q_i)} PB(Q_i, \texttt{ROOT}_i) \tag{3.32}$$

Because $|Q| \leq |Q_1| \cdot |Q_2| \cdots |Q_m|$, we have $|Q| \leq PB(\boldsymbol{W})$ where $\boldsymbol{W}$ is an edge or vertex cover. We prove in Sec. A.3.1:

Theorem 3.4.1. *Under set semantics, the polymatroid bound of a Berge-acyclic query Q is the following where $\boldsymbol{W}_V$ ranges over all vertex covers.*

$$PB_{\texttt{SET}}(Q) \stackrel{def}{=} \min_{\boldsymbol{W}_V} PB(\boldsymbol{W}_V) \tag{3.33}$$

Under bag semantics, it is the following where $\boldsymbol{W}_E$ ranges over all edge covers.

$$PB_{\texttt{BAG}}(Q) \stackrel{def}{=} \min_{\boldsymbol{W}_E} PB(\boldsymbol{W}_E) \tag{3.34}$$

Example 3.4.2. *Let $Q = R(X,Y), S(Y,Z), T(Z,U), K(U,V)$. Then $PB(Q,S) = f_1^{(R,Y)} N_S f_1^{(T,Z)} f_1^{(K,U)}$, $PB(\{R,TK\}) = N_R \cdot \min\left(N_T f^{(K,U)}, f^{(T,U)} N_k\right)$, and $PB(\{R,T,K\}) = N_R N_T N_K$. Given this, we can compute*
$PB_{\texttt{SET}}(Q) = \min(PB(\{RSTK\}), PB(\{RS,K\}), PB(\{R,TK\})$ and $PB_{\texttt{BAG}}(Q) = PB(\{RSTK\})$.

If we restrict the formula to the AGM bound, i.e. all max degrees are equal to the cardinalities, $f_1^{(R,X)} = N_R$, then Eq. (3.31) becomes $\prod_{R \in \boldsymbol{R}(Q)} N_R$, while the polymatroid bound (3.32) becomes $\min_{\boldsymbol{W}} \prod_{R \in \boldsymbol{W}} N_R$, where $\boldsymbol{W}$ ranges over integral covers of Q. In particular, the AGM bound of a Berge-acyclic query can be obtained by restricting to integral edge covers, although this property fails for α-acylic queries. For example, consider the query $R(X,Y), S(Y,Z), T(Z,X), K(X,Y,Z)$; when $|R| = |S| = |T| = |K|$ then the AGM bound is obtained by the edge cover $0,0,0,1$, but when $|R| = |S| = |T| \ll |K|$ one needs the fractional cover $1/2, 1/2, 1/2, 0$. Next, we prove next that the degree sequence bound is never worse.

Lemma 3.4.3. *(1) For any choice of root relation, $\texttt{ROOT} \in \boldsymbol{R}(Q)$: $DSB(Q) \leq PB(Q, \texttt{ROOT})$. (2) For any edge cover $\boldsymbol{W}_E = Q_1, \ldots, Q_m$ of Q, $DSB(Q) \leq DSB(Q_1) \cdots DSB(Q_m)$*

Proof. (1) Referring to Algorithm 2, we prove by induction on the tree that, for all $R \neq \texttt{ROOT}$, and every index i, $w_i^{(R)} \leq \prod_{S \in \texttt{tree}(R)} f_1^{(S,Z_S)}$. In other words, each element of the vector $\boldsymbol{w}^{(R)}$ is $\leq$ the product of all max degrees in the sub-tree rooted at R. Assuming this holds for all children of R, consider the definition of $\boldsymbol{w}^{(R)}$ in Algorithm 2. By induction hypothesis, for each vector $\boldsymbol{a}^{(X_p)}$ we have $a_{i_p}^{(X_p)} \leq \prod_{S \in \texttt{tree}(X_p)} f_1^{(S,Z_S)}$, a quantity that is independent

of the index i_p, and therefore we obtain the following:

$$w_{i_1}^{(R)} = \left(\boldsymbol{C}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}} \cdot \boldsymbol{a}^{(X_2)} \cdots \boldsymbol{a}^{(X_k)}\right)_{i_1} \leq \left(\sum_{i_2 i_3 \cdots i_k} C_{i_1 i_2 i_3 \cdots i_k}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}}\right) \cdot \prod_{S \in \mathtt{tree}(R), S \neq R} f_1^{(S,Z_S)}$$

and we use the fact that $\sum_{i_2 i_3 \cdots i_k} C_{i_1 i_2 \cdots i_k}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}} \leq f_{i_1}^{(R,X_1)}$ because, by Theorem 3.2.2 item 1, $\boldsymbol{C}^{\boldsymbol{f}^{(R,\boldsymbol{X}_R)},B^{(R)}}$ is consistent with the degree sequence $f_1^{(R,X_1)}$, and, finally, $f_{i_1}^{(R,X_1)} \leq f_1^{(R,X_1)}$. This completes the inductive proof. The algorithm returns $C^{\boldsymbol{f}^{(\mathtt{root},\boldsymbol{X}_{\mathtt{ROOT}})},B^{(\mathtt{ROOT})}} \cdot \boldsymbol{a}^{(X_1)} \cdot \boldsymbol{a}^{(X_2)} \cdots \boldsymbol{a}^{(X_k)} \leq \mathtt{SUM}(C^{\boldsymbol{f}^{(\mathtt{root},\boldsymbol{X}_{\mathtt{ROOT}})},B^{(\mathtt{ROOT})}}) \cdot \prod_{R \neq \mathtt{ROOT}} f_1^{(R,Z_R)} \leq |\mathtt{ROOT}| \cdot \prod_{R \neq \mathtt{ROOT}} f_1^{(R,Z_R)}$, which is $= PB(Q, \mathtt{ROOT})$, as required.

(2) We prove the statement only for $m = 2$ (the general case is similar) and show that $DSB(Q) \leq DSB(Q_1) \cdot DSB(Q_2)$. Since DSB is the query answer on the worst case instance, we need to show that $|Q_1 \bowtie Q_2| \leq |Q_1| \cdot |Q_2|$. This is not immediately obvious because the worst case instance may have negative multiplicities. Let X be the unique common variable of Q_1, Q_2, and let $\boldsymbol{a}$, $\boldsymbol{b}$ be the X-vectors representing the results of Q_1 and Q_2 respectively. It follows from Theorem 3.2.2 item 5 that $\boldsymbol{a}, \boldsymbol{b}$ are non-negative, therefore, $|Q| = \sum_i a_i b_i \leq (\sum_i a_i)(\sum_i b_i) = |Q_1| \cdot |Q_2|$. □

To extend the DSB to set semantics, we now define $DSB_{\mathtt{SET}}(Q)$ in the natural way,

$$DSB_{\mathtt{SET}}(Q) = \min_{\boldsymbol{W}_V} \prod_{Q_i \in \boldsymbol{W}_V} DSB(Q_i) \tag{3.35}$$

Our discussion implies:

Theorem 3.4.4. *Let Q be a Berge-acyclic query. We denote by $DSB(Q, \boldsymbol{f}, \boldsymbol{B})$ the DSB computed on the statistics $\boldsymbol{f} \stackrel{def}{=} (\boldsymbol{f}^{R,Z})_{R \in \boldsymbol{R}(Q), Z \in \boldsymbol{X}_R}$ and $\boldsymbol{B} \stackrel{def}{=} (B^{(R)})_{R \in \boldsymbol{R}(Q)}$. Then, when Q is evaluated under bag semantics:*

$$|Q| \leq DSB(Q, \boldsymbol{f}, \boldsymbol{B}) \leq DSB(Q, \boldsymbol{f}, \boldsymbol{\infty}) \leq PB_{BAG}(Q) \leq AGM_{BAG}(Q) \tag{3.36}$$

where $|Q|$ is the answer to the query on a database instance consistent with the given statistics. Equivalently, under set semantics,

$$|Q| \leq DSB_{SET}(Q, \boldsymbol{f}, \boldsymbol{1}) \leq DSB_{SET}(Q, \boldsymbol{f}, \boldsymbol{\infty}) \leq PB_{SET}(Q) \leq AGM_{SET}(Q) \tag{3.37}$$

3.5 Functional Representation

A degree sequence requires, in general, $\Omega(n)$ space, where $n = \max_{X \in \boldsymbol{X}} n_X$ is the size of the largest domain. However, one of the most important applications of these bounds is query optimization where they can be used to guide choices about things like join order selection. This setting requires a bounding procedure that is sublinear in space and time relative to the data size because it will be invoked many, many times during a single optimization. A linear procedure could cause the optimization process to become more expensive than the actual execution. Fortunately, a degree sequence can be *represented* compactly, using a staircase function as illustrated in Fig. 3.1, and we show that a very tight bound that operates directly on these compact representations can be calculated in quasi-linear time w.r.t. their compressed size.

We call this the Functional Degree Sequence Bound (FDSB), and we show that DSB $\leq$ FDSB $\leq PB_{BAG}$. We also find that these staircase functions can be further compressed, allowing a clean trade-off between computation and accuracy. Note that in this section, we restrict our discussion to $B^{(R)} = \infty$.

For convenience, we denote a vector element by $f(i)$ rather than f_i. For a non-decreasing vector $\boldsymbol{f} \in \mathbb{R}_+^{[n]}$, we denote by $\boldsymbol{f}^{-1} : \mathbb{R}_+ \to \mathbb{R}_+$ any function satisfying the following, for all v, $0 \leq v \leq f(n)$: if $f(i) < v$ then $i < f^{-1}(v)$, and if $f(i) > v$ then $i > f^{-1}(v)$. Such a function always exists[5], but is not unique.

We say that a vector $\boldsymbol{f} \in \mathbb{R}_+^n$ is *represented* by a function $\hat{f} : \mathbb{R}_+ \to \mathbb{R}_+$ if $f(i) = \hat{f}(i)$ for all $i = 1, n$. A function $\hat{f}$ is a *staircase function with k steps*, in short a *k-staircase*, if there exists dividers $m_0 \stackrel{\text{def}}{=} 0 < m_1 < \cdots < m_k \stackrel{\text{def}}{=} n$ such that $\hat{f}(x)$ is a nonnegative constant on each interval $\{x \mid m_{q-1} < x \leq m_q\}$, $q = 1, k$. The sum or product of an k_1-staircase with an k_2-staircase is an $(k_1 + k_2)$-staircase. We denote the summation of a staircase $\hat{f}(x)$ as $\hat{F}(x) = \int_0^x \hat{f}(t)dt$ which is then an increasing piecewise-linear function. Its standard inverse $\hat{F}^{-1} : \mathbb{R}_+ \to \mathbb{R}_+$ is also increasing and piecewise-linear. If $\hat{F}$ represents the vector F, then $\hat{F}^{-1}$ is an inverse F^{-1} of that vector (as discussed above).

---

[5]E.g. define it as follows: if $\exists i$ s.t. $f(i-1) < v < f(i)$ then set $f^{-1}(v) \stackrel{\text{def}}{=} i - 1/2$, otherwise set $f^{-1}(v) = i$ for some arbitrary i s.t. $f(i) = v$.

We begin by noting that every relation's degree sequence can be compressed efficiently using these staircase functions,

Lemma 3.5.1. *Suppose R is a relation with attribute $X \in \boldsymbol{X}_R$ and cardinality N_R. Then $f^{(R,X)}$ can be represented by a staircase function with $\min(\sqrt{2N_R}, f_1^{(R,X)})$ segments.*

Proof. We have $f_1^{(R,X)} \geq f_2^{(R,X)} \geq \cdots \geq f_{n_X}^{(R,X)}$, where n_X is the size of the domain of X. Assume w.l.o.g. that $f_{n_X}^{(R,X)} > 0$. Consider its natural division into k segments, $0 = i_0 < i_1 < \ldots < i_k \stackrel{\text{def}}{=} n_X$, such that $f_j^{(R,X)}$ is constant for $i_{\ell-1} < j \leq i_\ell$. Since $f^{(R,X)}$ takes non-negative integer values, it follows that there are at most $f_1^{(R,X)}$ unique values, so $k \leq f_1^{(R,X)}$. On the other hand, $N_R = \sum_i f_i^{(R,X)} \geq f_{i_1}^{(R,X)} + \cdots + f_{i_k}^{(R,X)} \geq k + (k-1) + \cdots + 1 + 0 = k(k+1)/2 \geq k^2/2$, which implies $k \leq \sqrt{2N}$. This proves $k \leq \min(\sqrt{2N_R}, f(1))$. □

Given that these degree sequences are highly compressible, we now demonstrate how to compute the bound using these compressed representations directly when $B = 1$ for all relations. Broadly, this simply requires adapting Algorithm 2 to operate on functional representations of the degree sequences. The following lemma shows that doing this still results in a valid bound:

Lemma 3.5.2. *Let $\boldsymbol{F}_1 \in \mathbb{R}_+^{[n_1]}, \ldots, \boldsymbol{F}_d \in \mathbb{R}_+^{[n_d]}$ be non-decreasing vectors satisfying $F_1(0) = 0$ and, for all $p = 2, d$, $F_1(n_1) \leq F_p(n_p)$. Let $a_1 \in \mathbb{R}_+^{[n_1]}, \ldots, a_d \in \mathbb{R}_+^{[n_d]}$ be non-increasing vectors. Denote by $\boldsymbol{C}, \boldsymbol{w}$ the following tensor and vector:*

$$C_{i_1 \cdots i_d} \stackrel{def}{=} \Delta_{i_1} \cdots \Delta_{i_d} \max(F_1(i_1), \ldots, F_d(i_d)) \tag{3.38}$$

$$w(i_1) \stackrel{def}{=} \sum_{i_2=1}^{n_2} \cdots \sum_{i_d=1}^{n_d} C_{i_1 \cdots i_d} \prod_{p \in [2,d]} a_p(i_p) \tag{3.39}$$

Then the following inequalities hold:

$$w(i_1) \geq (\Delta_{i_1} F_1(i_1)) \prod_{p \in [2,d]} a_p\left(\left\lfloor F_p^{-1}(F_1(i_1)) \right\rfloor + 1\right) \tag{3.40}$$

$$w(i_1) \leq (\Delta_{i_1} F_1(i_1)) \prod_{p \in [2,d]} a_p\left(\left\lceil F_p^{-1}(F_1(i_1 - 1)) \right\rceil\right) \tag{3.41}$$

Algorithm 3 $FDSB(Q, \texttt{ROOT})$

for each variable $X \in \boldsymbol{X}$ and non-root relation $R \in \boldsymbol{R}$, $R \neq \texttt{root}$, in bottom-up order **do**

$\hat{\boldsymbol{a}}^{(X)} \stackrel{\text{def}}{=} \bigotimes_{R \in \text{children}(X)} \hat{\boldsymbol{w}}^{(R)}$

$\forall i_1 : \quad \hat{w}^{(R)}(i_1) \stackrel{\text{def}}{=} \left(\hat{f}^{(R,X_1)}(i_1)\right) \prod_{p \in [2,d]} a^{(X_p)} \left(\max(1, (\hat{F}^{(R,X_p)})^{-1}(\hat{F}^{(R,X_1)}(i_1 - 1)))\right)$

end for

return $\int_{i=1}^{|\texttt{ROOT}|} \prod_{p=1,k} \hat{a}^{(X_p)}(\max(1, (\hat{F}^{\texttt{ROOT},X_p})^{-1}(i-1)))$

We give the proof in Appendix D.3. The lemma implies that, in Algorithm 2, we can use inequality (3.41) to upper bound the computation $w^{(R)} = \boldsymbol{C} \cdot \boldsymbol{a}^{(X_2)} \cdots \boldsymbol{a}^{(X_k)}$. In that case, each $F_p(r) \stackrel{\text{def}}{=} \sum_{i=1,r} f_p(r)$ is the cdf of a degree sequence f_p, hence $F_p(0) = 0$ and $F_p(n_p) =$ the cardinality of R, while the tensor $\boldsymbol{C}$ is described in item 6 of Theorem 3.2.2, hence the assumptions of the lemma hold.

Fix a Berge-acyclic query Q, and let each degree sequence $\boldsymbol{f}^{(R,Z)}$ be represented by some $k_{R,Z}$-staircase $\hat{f}^{R,Z}$, and we denote by $\hat{F}^{(R,Z)}$ its summation. Fix any relation $\texttt{ROOT} \in \boldsymbol{R}(Q)$ to designated as root. The *Functional Degree Sequence Bound at* $\texttt{ROOT}$, $FDSB(Q, \texttt{ROOT})$, is the value returned by Algorithm 3. This algorithm is identical to Algorithm 2, except that it replaces both $\boldsymbol{w}^{(R)}$ with a functional upper bound justified by the inequality 3.41 of Lemma 3.5.2, and similarly for the returned result. All functions $\hat{\boldsymbol{a}}^{(X)}$ and $\hat{\boldsymbol{w}}^{(R)}$ are staircase functions, and can be computed in linear time, plus a logarithmic time need for a binary search to lookup a segment in a staircase. Using this, we prove the following in Appendix D.4:

Theorem 3.5.3. *(1)* $FDSB(Q, \texttt{ROOT}) \geq DSB(Q)$. *(2)* $FDSB(Q, \texttt{ROOT})$ *can be computed in time* $T_{FDSB} \stackrel{def}{=} \tilde{O}(m \cdot \sum_{R,Z}(arity(R) \cdot k_{R,Z}))$ *by Alg. 3, where* $\tilde{O}$ *hides a logarithmic term, and* $m = |\boldsymbol{R}(Q)|$ *is the number of relations in* Q.

The theorem says that $FDSB(Q, \texttt{ROOT})$ is still an upper bound on $|Q|$, and can be computed in quasi-linear time in the size of the functional representations of the degree sequences. Next, we check if $FDSB$ is below the polymatroid bound. Consider the computation of $\hat{w}^{(R)}(i_1)$ by the algorithm. On one hand $\hat{f}^{(R,X_1)}(i_1) \leq \hat{f}^{(R,X_1)}(1)$; on the other

hand $a^{(X_p)}(\max(1,\ldots)) \leq a^{(X_p)}(1)$. This allows us to prove (inductively on the tree, in D.2):

Lemma 3.5.4. *$FDSB(Q, \texttt{ROOT}) \leq PB(Q, \texttt{ROOT})$, where PB is defined in* (3.31)*.*

When we proved $DSB \leq PB$ in Lemma 3.4.3, we used two properties of DSB: $DSB(Q, \texttt{ROOT})$ is independent of the choice of ROOT, and $DSB(Q_1 \bowtie \cdots \bowtie Q_m) \leq DSB(Q_1) \cdots DSB(Q_m)$, for any edge cover $\boldsymbol{W} = \{Q_1, \ldots, Q_m\}$. Both hold because $DSB(Q)$ is standard query evaluation: it is independent of the query plan (i.e. choice of ROOT) and it can only increase if we remove join conditions. But $FDSB$ is no longer standard query evaluation and these properties may fail. For that reason we introduce two stronger versions of the functional degree sequence bound:

$$FDSB_{\texttt{BAG}}(Q) = \min_{\boldsymbol{W}_E} \prod_{i=1,m} \min_{ROOT \in \boldsymbol{R}(Q_i)} FDSB(Q_i, ROOT) \tag{3.42}$$

$$FDSB_{\texttt{SET}}(Q) = \min_{\boldsymbol{W}_V} \prod_{i=1,m} \min_{ROOT \in \boldsymbol{R}(Q_i)} FDSB(Q_i, ROOT) \tag{3.43}$$

where $\boldsymbol{W}_V$ range over the covers of Q. The definitions above combined with Lemma 3.5.4 immediately imply that the $FDSB$ is superior to previous bounds:

Theorem 3.5.5. *Suppose Q is a Berge-acyclic query. Then when Q is evaluated under bag semantics,*

$$|Q| \leq FDSB_{BAG}(Q) \leq PB_{BAG}(Q) \leq AGM_{BAG}(Q) \tag{3.44}$$

When Q is evaluated under set semantics,

$$|Q| \leq FDSB_{SET}(Q) \leq PB_{SET}(Q) \leq AGM_{SET}(Q) \tag{3.45}$$

In Appendix A.4.3, we provide a dynamic programming algorithm which computes $FDSB_{BAG}$ and analyze it to achieve the following complexity:

Theorem 3.5.6. *Let T_{FDSB} and m be defined as in Theorem 3.5.3. $FDSB_{BAG}(Q)$ and $FDSB_{SET}(Q)$ can be computed in time $O(2^m \cdot (2^m + m \cdot T_{FDSB}))$.*

Together, Theorems 3.5.6 and 3.5.5 imply that, in quasi-linear time w.r.t. the size of the representation, we can compute in an upper bound to the query Q that is guaranteed to improve over the polymatroid bound. In practice, we this bound is significantly lower than the polymatroid bound, because it accounts for the entire degree sequence $\boldsymbol{f}$, not just f_1.

Finally, we show that one can tradeoff the size of the representation for accuracy. Naively, it's possible to do this by choosing coarser staircase approximations of the degree sequences. They only need to be non-increasing, and lie above the true degree sequences. However, we further prove in Appendix D.3 that your approximation only needs to upper bound the degree sequence in a cumulative sense, a strictly weaker property.

Theorem 3.5.7. *Fix a query Q, let $\boldsymbol{f} = \{\boldsymbol{f}^{(R,X)} | R \in \boldsymbol{R}(Q), Z \in \boldsymbol{X}_R\}$ be the degree sequences for every relation in the query. Let $\hat{\boldsymbol{f}} = \{\hat{\boldsymbol{f}}^{(R,X)} | R \in \boldsymbol{R}(Q), Z \in \boldsymbol{X}_R\}$ be another set of degree sequences. If $\boldsymbol{F}^{(R,\boldsymbol{X}_R)} \leq \hat{\boldsymbol{F}}^{(R,\boldsymbol{X}_R)}$ then,*

$$FDSB(Q, \boldsymbol{f}) \leq FDSB(Q, \hat{\boldsymbol{f}}) \tag{3.46}$$

3.6 Conclusions

We have described the *degree sequence bound* of a conjunctive query, which is an upper bound on the size of its answer, given in terms of the degree sequences of all its attributes. Our results apply to Berge-acyclic queries, and strictly improve over previously known AGM and polymatroid bounds [10, 90]. On one hand, our results represent a significant extension, because they account for the full degree sequences rather than just cardinalities or just the maximum degrees. On the other hand, they apply only to a restricted class of acyclic queries, although, we argue, this class is the most important for practial applications. While the full degree sequence can be as large as the entire data, we also described how to approximate the cardinality bound very efficiently, using compressed degree sequences. Finally, we have argued for using the max tuple multiplicity for each relation, which can significantly improve the accuracy of the cardinality bound.

Chapter 4

PARTITION CONSTRAINTS

In the same vein as the prior chapter, this chapter presents a novel statistic, *partition constraints*, for characterizing the distribution of input data. Where the degree sequence captured the skew of join columns, partition constraints capture the correlation between columns within a table. This is done by partitioning the table into multiple pieces such that each piece has low degree in one column or set of columns. By creating a unique bound or execution plan for every combination of partitions, this enables asymptotically lower cardinality bounds and algorithmic runtimes. We refer readers back to Ch. 2 Sec. 2.1.4 for a description of prior work in cardinality bounds and worst-case optimal join (WCOJ) algorithms.

Recall that a DC for a relation R on the attributes $\mathbf{Y}$ and a subset $\mathbf{X} \subseteq \mathbf{Y}$ asserts that for each fixed instantiation $\mathbf{x}$ of $\mathbf{X}$ there are only a limited number of completions to the whole set of attributes $\mathbf{Y}$. Thus, $\mathbf{X}$ functions like a weak version of a key. However, DCs are a brittle statistic; a single high frequency value per attribute can dramatically loosen a relation's DCs even if all other values only occur once. In the graph setting, where this corresponds to bounded (in- or out-)degree graphs, this has long been viewed as an overly restrictive condition because graphs often have highly skewed degree distributions. To overcome this, the graph theory community identified degeneracy as a natural graph invariant that allows for graphs of unbounded degree while permitting fast algorithms. For example, pattern counting on low degeneracy graphs has recently received considerable attention [18, 19, 20, 21, 28, 58].

In this context, partition constraints (PCs) can be viewed as a declarative version of graph degeneracy for higher-arity data. PCs naturally extend DCs in the sense that every DC can be expressed as PC, but not the other way around. Informally, for a PC to hold over a relation R, we require it to be possible to split R such that each partition satisfies

at least one DC. Again, for a binary edge relation $R = E$ of a directed graph $G = (V, E)$, this means that we can partition the edges into two sets E_1, E_2, such that the subgraph $G_1 := (V, E_1)$ has bounded out-degree while $G_2 := (V, E_2)$ has bounded in-degree.

We aim to provide a thorough analysis of the effect of PCs on conjunctive query answering. To that end, we show that PCs can provide asymptotic improvements to query bounds and evaluation, and we demonstrate how they can gracefully incorporate work on cardinality bounds and WCOJ algorithms for DCs. Further, we provide both exact and approximate algorithms for computing PCs and inspect to what extent PCs are present in well-known benchmark datasets.

Summary of results.

- We introduce PCs as a generalization of DCs and provide two algorithms to determine PCs in polynomial time as well as to partition the data to witness these constraints. One is exact and runs in quadratic time, while the second runs in linear time and provides a constant factor approximation.

- We develop new bounds on the output size of a join query. These bounds use DC-based bounds as a black box and naturally extend them to incorporate PCs. We show that these bounds are asymptotically tighter than bounds that rely on DCs alone. Further, we show that if the DC-based bounds are tight, then the PC-based bounds built on top of them will be tight as well.

- Using WCOJ algorithms for DCs as a black box, we provide improved join algorithms that are worst-case-optimal relative to the tighter PC-based bound. Notably, if an algorithm were proposed that is worst-case optimal relative to a tight DC-based bound, then this would immediately result in a WCOJ algorithm relative to a tight PC-based bound.

Structure of the Chapter. In Section 4.1, we provide some basic definitions and background. We formally introduce and discuss PCs in Section 4.2 and compute them on common benchmark data. In Section 4.3, we illustrate the benefits of the PC framework

by thoroughly analyzing a concrete example. The developed techniques are then extended in Section 4.4 and applied to arbitrary CQs. We conclude and give some outlook to future work in Section 4.5. Full proofs of some results are deferred to the appendix, and we instead focus on providing intuition in the main body.

4.1 Preliminaries

We refer the reader to Ch. 2 Sec. 2.1.1 for a definition of conjunctive queries. We make one small change to these definitions for convenience in this chapter. For a particular database instance I, we denote the answers to a CQ, i.e., the join of the R_i, as a relation $Q^I(\mathbf{Z})$.

In this section, we provide some additional terms that will be relevant in this chapter.

Degree constraints and bounds. This chapter presents a generalization of degree constraints to account for the benefits of partitioning relations. So, we begin by repeating the definition of degree constraints from Ch. 2 Sec. 2.1.1.

Definition 4.1.1. Fix a particular relation $R(\mathbf{Z})$. Given two sets of variables $\mathbf{X}, \mathbf{Y}$ where $\mathbf{X} \subseteq \mathbf{Y} \subseteq \mathbf{Z}$, a degree constraint of the form $DC_R(\mathbf{X}, \mathbf{Y}, d)$ implies the following,

$$\max_{\mathbf{x} \in dom(\mathbf{X})} |\pi_{\mathbf{Y}} \sigma_{\mathbf{X}=\mathbf{x}} R| \leq d.$$

A database instance I satisfying a (set of) degree constraints $\mathbf{DC}$ is denoted by $I \vDash \mathbf{DC}$. For convenience, we denote the minimal d such that $DC_R(\mathbf{X}, \mathbf{Y}, d)$ holds by $DC_R(\mathbf{X}, \mathbf{Y})$.

$$DC_R(\mathbf{X}, \mathbf{Y}) := min(\{d \mid d \in \mathbb{N},\ DC_R(\mathbf{X}, \mathbf{Y}, d) \text{ is true}\}) \tag{4.1}$$

If $\mathbf{Y} = \mathbf{Z}$ and $\mathbf{X} = \emptyset$ the constraint simply bounds the cardinality of R and we write $CC_R(d)$ or CC_R.

In addition to generalizing DCs, our work is able to refine any of the previous bounding methods for DCs by incorporating them into our framework. Therefore, we introduce a general notation for DC-based bounds.

Definition 4.1.2. Given a conjunctive query Q and a set of degree constraints $\mathbf{DC}$, a cardinality bound $CB(Q, \mathbf{DC})$ is any function where the following is true,

$$|Q^I(\mathbf{Z})| \leq CB(Q, \mathbf{DC}) \quad \forall\ I \vDash \mathbf{DC}.$$

Throughout this chapter, we will make specific reference to the combinatorics bound which we describe below. While it is impractical, this bound is computable and tight which makes it useful for theoretical analyses.

Definition 4.1.3. Given a conjunctive query Q and a set of degree constraints $\mathbf{DC}$, we define the combinatorics bound (for degree constraints) as

$$CB_{\text{Comb}}(Q, \mathbf{DC}) = \sup_{I \vDash \mathbf{DC}} |Q^I(\mathbf{Z})|.$$

In this chapter, we make two minimal assumptions about bounds and statistics. First, we assume that cardinality bounds are finite by assuming that every variable in the query is covered by at least one cardinality constraint. Second, we assume that there is a bound on the growth of the bounds as our statistics increase. Formally, for a fixed query Q, we assume that every cardinality bound CB has a function f_Q such that for any $\alpha \in \mathbb{R}^+$ and degree constraints $\mathbf{DC}$ we have $CB(Q, \alpha \cdot \mathbf{DC}) \leq f_Q(\alpha) \cdot CB(Q, \mathbf{DC})$. Usually, cardinality bounds are expressed in terms of power products of the degree constraints [11, 60, 87]. In that case, $f_Q = O(\alpha^c)$ for some constant $c \geq 1$.

Lastly, we will also incorporate existing work on worst-case optimal join (WCOJ) algorithms. Each WCOJ algorithm is optimal relative to a particular cardinality bound, and we describe that relationship formally as follows:

Definition 4.1.4. Denote the runtime of a join algorithm $\mathcal{A}$ on a conjunctive query Q and database instance I as $\mathcal{T}(\mathcal{A}, Q, I)$. $\mathcal{A}$ is worst-case optimal (WCO) relative to a cardinality bound $CB_{\mathcal{A}}$ if the following is true for all queries Q,

$$\mathcal{T}(\mathcal{A}, Q, I) = O(|I| + CB_{\mathcal{A}}(Q, \mathbf{DC})), \quad \forall\ \mathbf{DC},\ I \vDash \mathbf{DC}.$$

Note that the hidden constant may only depend on the query Q and, importantly, neither on the set of degree constraints $\mathbf{DC}$ nor on the database instance I. When $\mathcal{A}$ is optimal relative to CB_{Comb}, we simply call it worst-case optimal (relative to degree constraints).

Note that some algorithms fulfill a slightly looser definition of worst-case optimal by allowing additional poly-logarithmic factors [91]. These algorithms can still be incorporated

into this framework, but we choose the stricter definition to emphasize that we do not induce additional factors of this sort.

Much of the recent work on WCOJ algorithms can be categorized as *variable-at-a-time* (VAAT) algorithms. Intuitively, a VAAT computes the answers to a join query by answering the query for an increasingly large number of variables. This idea is at the heart of Generic Join [114], Leapfrog Triejoin [142], and NPRR [113]. We define this category formally as follows. For an arbitrary query $Q(\mathbf{Z}) \leftarrow R_1 \bowtie \cdots \bowtie R_k$ and an ordering $Z_1, \ldots, Z_r = \mathbf{Z}$, let Q_i denote the sub-query

$$Q_i(Z_1, \ldots Z_i) \leftarrow \pi_{Z_1,\ldots,Z_i} R_1 \bowtie \cdots \bowtie \pi_{Z_1,\ldots,Z_i} R_k$$

Definition 4.1.5. A join algorithm $\mathcal{A}$ that solves conjunctive queries $Q(\mathbf{Z}) \leftarrow R_1(\mathbf{Z}_1) \bowtie \cdots \bowtie R_k(\mathbf{Z}_k)$ is a VAAT algorithm if there is some ordering $Z_1, \ldots, Z_r = \mathbf{Z}$ such that $\mathcal{A}$ takes time $\Omega(\max_i |Q_i^I|)$ for databases I. The choice of the ordering is allowed to depend on I.

4.2 Partition Constraints

Partition constraints (PCs) extend the concept of DCs by allowing for the partitioning of relations. For clarity, we start with the binary setting, revisiting the example from the introduction. To that end, let $E(X, Y)$ be the edge relation of a directed graph $G = (V, E)$. For a degree constraint to hold on E, either the in- or the out-degree of E must be bounded. However, this is a strong requirement and often may not be satisfied, e.g. on a highly skewed social network graph. In these cases, it makes sense to look for further latent structure in the data. We suggest partitioning E such that each part satisfies a (different) degree constraint. We are interested in the following quantity where the minimum is taken over all bi-partitions of E, i.e. $E = E^X \cup E^Y$ (implicitly $E^X \cap E^Y = \emptyset$),

$$\min_{E^X \cup E^Y = E} \max\{DC_{E^X}(X, XY), DC_{E^Y}(Y, XY)\}. \tag{4.2}$$

This corresponds to a splitting of the graph into two graphs. In one of them, $G^X = (V, E^X)$, we attempt to minimize the maximum out-degree of the graph while in the other, $G^Y = (V, E^Y)$, we attempt to minimize the maximum in-degree. It is important to note that both

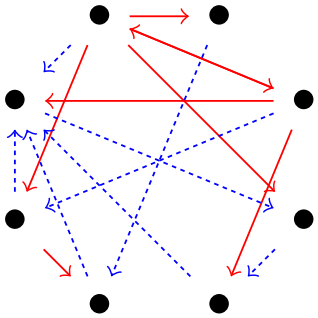


Figure 4.1: Graph Example.

Access

PersonID	RoomID
Ava	Beacon Hall
Ben	Beacon Hall
Cole	Delta Hall
Dan	Delta Hall
Emma	Gala Hall
Finn	Jade Hall
Porter	Beacon Hall
Porter	Delta Hall
Porter	Gala Hall
Porter	Jade Hall

$\text{Access}^{\text{PersonID}}$

PersonID	RoomID
Ava	Beacon Hall
Ben	Beacon Hall
Cole	Delta Hall
Dan	Delta Hall
Emma	Gala Hall
Finn	Jade Hall

$\text{Access}^{\text{RoomID}}$

PersonID	RoomID
Porter	Beacon Hall
Porter	Delta Hall
Porter	Gala Hall
Porter	Jade Hall

Figure 4.2: Access Example.

of these quantities can be arbitrarily lower than the maximum in-degree and out-degree of the original graph. This is where the benefit of considering partitions comes from.

An example of such a partitioning is depicted in Figure 4.1. There, the dashed blue edges represent E^X while the solid red edges represent E^Y. The maximum out- and in-degree of the full graph is 5 while $G^X = (V, E^X)$ has a maximum out-degree of 1 and $G^Y = (V, E^Y)$ has a maximum in-degree of 1. In general, a class of graphs can have an unbounded out- and in-degree but always admit a bi-partitioning with an out- and in-degree of 1, respectively.

This approach is originally motivated by the graph property *degeneracy*. Intuitively, this property tries to allocate each edge of an undirected graph to one of its incident vertices such that no vertex is "responsible" for too many edges. Each partition $E = E^X \cup E^Y$ can be seen as a possible allocation where the edges in E^X are allocated to the X part of the tuples while the edges in E^Y are allocated to the Y part of the tuples. Thus, in general, if the undirected version of a graph class $\mathcal{G}$ has bounded degeneracy, the Quantity 4.2 must

also be bounded for $\mathcal{G}$. In fact, the converse is true as well if the domain of the X and Y attributes are disjoint.

Formally, we define a PC on a relation R as below. Note, we say a collection of subrelations $(R^1, \ldots, R^k)$ partition R when they are pairwise disjoint and $\bigcup_j R^j = R$.

Definition 4.2.1. Fix a particular relation $R(\mathbf{Z})$, a subset $\mathbf{Y} \subseteq \mathbf{Z}$, and let $\mathcal{X} = \{\mathbf{X}^1, ..., \mathbf{X}^{|\mathcal{X}|}\} \subseteq 2^{\mathbf{Y}}$ be a set of sets of variables. Then, a partition constraint of the form $PC_R(\mathcal{X}, \mathbf{Y}, d)$ implies,

$$\min_{(R^{\mathbf{X}})_{\mathbf{X}\in\mathcal{X}} \text{ partition } R} \max\{DC_{R^{\mathbf{X}}}(\mathbf{X}, \mathbf{Y}) \mid \mathbf{X} \in \mathcal{X}\} \leq d.$$

Again, a database instance I satisfying a (set of) partition constraints $\mathbf{PC}$ is denoted by $I \vDash \mathbf{PC}$. We denote the minimal d such that $PC_R(\mathcal{X}, \mathbf{Y}, d)$ holds by $PC_R(\mathcal{X}, \mathbf{Y})$ and we omit R if clear from the context.

This definition says that one can split the relation R into disjoint subsets $R^j \subseteq R, \bigcup_j R^j = R$ and associate each degree constraint over a set of variables $\mathbf{X}^j \in \mathcal{X}$ with a subset R^j such that the maximum of each constraint on its associated relation is bounded by d. From an algorithmic point of view, each part R^j should be handled differently to make use of its unique, tighter DC. The core of this work is in describing how these partitions can be computed and how to make use of the new constraints on each part. Broadly, we show how these new constraints produce tighter bounds on the join size and how algorithms can meet these bounds. Note PCs are a strict generalization of DCs since we can define an arbitrary degree constraint $DC(\mathbf{X}, \mathbf{Y}, d)$ as $PC(\{\mathbf{X}\}, \mathbf{Y}, d)$. For simplicity, when we discuss a collection of PCs, we assume that there is at most one PC per $(\mathcal{X}, \mathbf{Y})$ pair, i.e., there are never two $PC_R(\mathcal{X}, \mathbf{Y}, d), PC_R(\mathcal{X}, \mathbf{Y}, d'), d \neq d'$, as it suffices to keep the stronger constraint.

Next, we present an example of how relations with small PCs might arise in applications.

Example 4.2.2. *Consider a relation* `Access(PersonID, RoomID)` *that records who has access to which room at a university. This could be used to control the key card access of all faculty members, students, security, and cleaning personnel. Most people (faculty members and students) only need access to a limited number of rooms (lecture halls and offices). On the other hand, porters and other caretaker personnel need access to many*

different rooms, possibly all of them. However, each room only needs a small number of people taking care of it. Thus, it makes sense to partition `Access` *into* $\texttt{Access}^{\texttt{PersonID}} \cup \texttt{Access}^{\texttt{RoomID}}$ *with the former tracking the access restrictions of the faculty members and students, and the latter tracking the access restrictions of the caretaker personnel. With this partition, both* $DC_{\texttt{Access}^{\texttt{PersonID}}}(\texttt{PersonID}, \texttt{RoomID})$ *and* $DC_{\texttt{Access}^{\texttt{RoomID}}}(\texttt{RoomID}, \texttt{PersonID})$ *should be small. Thus,* $PC_{\texttt{Access}}(\{\texttt{PersonID}, \texttt{RoomID}\}, \texttt{PersonID RoomID})$ *should be small as well.*

Figure 4.2 depicts a small example instance of the situation. There, the students each only need access to a single lecture hall to attend their courses, and all the rooms are taken care of by a single porter. Thus, following the suggested splitting, the respective DC for both subrelations $\texttt{Access}^{\texttt{PersonID}}$ *and* $\texttt{Access}^{\texttt{RoomID}}$ *is 1 and, therefore, also the PC for the whole relation* `Access` *is 1.*

To see how these statistics manifest in real world data, we calculated the PC of relations from some standard benchmarks which are displayed in Figure 4.3 [137, 72, 99]. We computed the PC using Algorithm 6 from Section 4.4. To model the interesting case of many-to-many joins, we first removed any attributes which are primary keys from each relation. Specifically, we computed the partition constraint $PC(\{X \mid X \in \mathbf{Y}\}, \mathbf{Y})$ where $\mathbf{Y}$ is the set of non-key attributes. We compare this with the minimum and maximum degree of these attributes before partitioning. Naively, by partitioning the data randomly, one would expect a PC roughly equal to the maximum DC divided by $|\mathcal{X}|$. Alternatively, by placing all tuples in the partition corresponding to the minimum DC, one can achieve a PC equal to the minimum DC. However, the computed PC is often much lower than both of these quantities. This implies that the partitioning is uncovering useful structure in the data rather than simply distributing high-degree values over multiple partitions.

4.2.1 Further Partitioning

At this point, one might wonder if further partitioning the data can meaningfully reduce a PC. That is, whether for a given relation $R(\mathbf{Z})$ and a particular set of degree constraints $\{DC_R(\mathbf{X}, \mathbf{Y}) \mid \mathbf{X} \in \mathcal{X}\}$, is it possible to decrease the maximum DC significantly by partition-

Dataset	**Max DC**	**Min DC**	**PC**	$\lvert\mathcal{X}\rvert$
aids	11	11	3	2
yeast	154	119	9	2
dblp	321	113	34	2
wordnet	526	284	3	2
Stats/badges	899	456	8	2
Stats/comments	134887	45	15	3
Stats/post_links	10186	13	2	4
Stats/post_history	91976	32	3	4
Stats/votes	326320	427	33	4
IMDB/keywords	72496	540	71	2
IMDB/companies	1334883	94	13	4
IMDB/info	13398741	2937	123	4
IMDB/cast	25459763	1741	52	6

Figure 4.3: Example PCs.

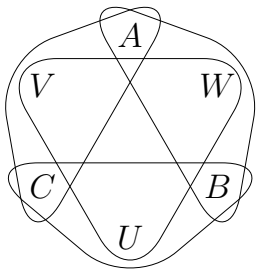


Figure 4.4: The Query $Q_{\hexagon}$.

ing R into more than $|\mathcal{X}|$ parts? We show that this is not the case; neither pre-partitioning nor post-partitioning the data into k parts can reduce the PC by more than a factor of k. We prove the former first.

Theorem 4.2.3. *Given a relation $R(\mathbf{Z})$, a subset $\mathbf{Y} \subseteq \mathbf{Z}$, variable sets $\mathcal{X} \subseteq 2^{\mathbf{Y}}$, and subrelations $(R^1, \ldots, R^k)$ that partition R. Then,*

$$\max_{i=1,\ldots,k} PC_{R^i}(\mathcal{X}, \mathbf{Y}) \geq PC_R(\mathcal{X}, \mathbf{Y})/k.$$

Proof. For the sake of contradiction, we assume that there exists a partitioning $(R^1, \ldots, R^k)$ of R such that,

$$\max_{i=1,\ldots,k} PC_{R^i}(\mathcal{X}, \mathbf{Y}) < PC_R(\mathcal{X}, \mathbf{Y})/k.$$

Then, we can partition each R^i to witness $PC_{R^i}(\mathcal{X}, \mathbf{Y})$. Let $\bigcup_{\mathbf{X}\in\mathcal{X}} R^{i,\mathbf{X}} = R^i$ be such that $DC_{R^{i,\mathbf{X}}}(\mathbf{X}, \mathbf{Y}) \leq PC_{R^i}(\mathcal{X}, \mathbf{Y})$ holds for each part $R^{i,\mathbf{X}}$. For each fixed $\mathbf{X} \in \mathcal{X}$, we

can combine the sub-relations $R^{1,\mathbf{X}}, \ldots, R^{k,\mathbf{X}}$ into a single relation $R^{\mathbf{X}}$. These relations, $(R^{\mathbf{X}})_{\mathbf{X}\in\mathcal{X}}$, form a partition of R. The DC for each $R^{\mathbf{X}}$ is at most the sum of the DCs of $R^{1,\mathbf{X}}, \ldots, R^{k,\mathbf{X}}$. Further, this sum must be less than our initial PC by our assumption,

$$DC_{R^{\mathbf{X}}}(\mathbf{X}, \mathbf{Y}) \leq k \cdot PC_{R^i}(\mathcal{X}, \mathbf{Y}) < PC_R(\mathcal{X}, \mathbf{Y}).$$

This directly implies that

$$\max_{\mathbf{X}\in\mathcal{X}} DC_{R^{\mathbf{X}}}(\mathbf{X}, \mathbf{Y}) < PC_R(\mathcal{X}, \mathbf{Y}).$$

Because the PC is defined as the minimum value of this maximum DC over all possible partitions of R into $|\mathcal{X}|$ parts, this is a contradiction. □

Next, we show that post-partitioning cannot super-linearly reduce the degree either.

Proposition 4.2.4. *Let $(R^1, \ldots, R^k)$ partition a relation $R(\mathbf{Z})$, and let $\mathbf{X} \subseteq \mathbf{Y} \subseteq \mathbf{Z}$ be two subsets of variables. Then,*

$$\max_{i=1,\ldots,k} DC_{R^i}(\mathbf{X}, \mathbf{Y}) \geq DC_R(\mathbf{X}, \mathbf{Y})/k.$$

Proof. Let $\mathbf{x}$ be the value of $\mathbf{X}$ within R that occurs in tuples with $d = DC_R(\mathbf{X}, \mathbf{Y})$ unique values $\mathbf{y}$ of $\mathbf{Y}$. For each of these values $\mathbf{y}$, a tuple containing $\mathbf{y}$ must be associated with one partition R^i. By the pigeonhole principle and the fact that there are $DC_R(\mathbf{X}, \mathbf{Y})$ of these tuples, it follows that some partition must contain at least $DC_R(\mathbf{X}, \mathbf{Y})/k$ of these tuples. Therefore, for some $i \in \{1, \ldots, k\}$, we have $DC_{P_i}(\mathbf{X}, \mathbf{Y}) \geq DC_R(\mathbf{X}, \mathbf{Y})/k$. □

Theorems 4.2.3 and Proposition 4.2.4 together show that for a given relation $R(\mathbf{Y})$ and a particular set of degree constraints $\{DC_R(\mathbf{X}, \mathbf{Z}) \mid \mathbf{X} \in \mathcal{X}\}$ deemed relevant, we only have to consider partitionings of R into at most $|\mathcal{X}|$ pieces. Thus, if the query size is viewed as a constant, then the number of useful partitions is also a constant.

4.3 The Hexagon Query

In this section, we show the benefits of the PC framework by demonstrating how it can lead to asymptotic improvements for both cardinality bounds and conjunctive query evaluation.

Concretely, there is an example query and class of database instances where bounds based on **PC** are asymptotically tighter than those based on **DC**. Further, all VAAT algorithms (See Definition 4.1.5) are asymptotically slower than a **PC**-aware algorithm on this query and instance class. Formally, our aim is to show the following:

Theorem 4.3.1. *There exists a query Q and a set of partition constraints* **PC** *with the degree constraint subset* $\mathbf{DC} \subset \mathbf{PC}$ *such that the following are true:*

1. *Bounds based on degree constraints are asymptotically sub-optimal, i.e.*

$$\sup_{I \vDash \mathbf{PC}} |Q^I(\mathbf{Z})| = o(CB_{Comb}(Q, \mathbf{DC})) = o(\sup_{I \vDash \mathbf{DC}} |Q^I(\mathbf{Z})|).$$

2. *There is an algorithm that enumerates Q^I for instances $I \vDash \mathbf{PC}$ in time $O(\sup_{I \vDash \mathbf{PC}} |Q^I(\mathbf{Z})|)$.*

3. *No VAAT algorithms can enumerate Q^I for instances $I \vDash \mathbf{PC}$ in time $O(\sup_{I \vDash \mathbf{PC}} |Q^I(\mathbf{Z})|)$.*

To prove these claims, we consider the hexagon query (also depicted in Figure 4.4)

$$Q_{\hexagon}(A, B, C, U, V, W) \leftarrow R_1(A, W, B) \bowtie R_2(B, U, C) \bowtie R_3(C, V, A) \bowtie R_4(U, V, W)$$

and impose the following set of PCs on the relations:

$$\begin{array}{llll} DC_{R_1}(AW, AWB, 1), & DC_{R_1}(WB, AWB, 1), & CC_{R_1}(n), & CC_{R_2}(n) \\ DC_{R_2}(BU, BUC, 1), & DC_{R_2}(UC, BUC, 1), & CC_{R_3}(n), & CC_{R_4}(n), \\ DC_{R_3}(CV, CVA, 1), & DC_{R_3}(VA, CVA, 1), & PC_{R_4}(\{U, V, W\}, UVW, 1). & \end{array}$$

We denote the whole set as **PC** and the subset of DCs as **DC**. We now prove the theorem's first claim. That is, the combinatorics bound on $Q_{\hexagon}$ is super linear when only considering **DC**, while there are only a linear number of answers to $Q_{\hexagon}$ over any database satisfying **PC**. We begin by providing a lower bound on the combinatorics bound of $Q_{\hexagon}$.

Lemma 4.3.2. *The combinatorics bound of $Q_{\hexagon}$ based on* **DC** *is in $\Omega(n^{\frac{4}{3}})$.*

Proof Sketch. It suffices to provide a collection of databases $\mathcal{I}$ such that $I \vDash \mathbf{DC}$ and $|Q^I_{\hexagon}| = \Omega(n^{\frac{4}{3}})$ for $I \in \mathcal{I}$. To accomplish this, we introduce a new relation $R_{X,Y,Z}(X, Y, Z)$

with $|dom(X)| = |dom(Z)| = n^{\frac{2}{3}}$ and $|dom(Y)| = n^{\frac{1}{3}}$. Intuitively, think of $R_{X,Y,Z}$ as a bipartite graph from the domain of X to the domain of Z where Y identifies the edge for a given $x \in dom(X)$ or $z \in dom(Z)$. Thus, every $x \in dom(X)$ is connected to $n^{\frac{1}{3}}$ neighbors in $dom(Z)$ and, due to symmetry, also the other way around.

Therefore, the constraints $DC(XY, XYZ, 1)$, $DC(YZ, XYZ, 1)$, and $CC(n)$ are satisfied over $R_{X,Y,Z}$. Thus, we can use a relation of this type for R_1, R_2, R_3. Concretely, we use this relation in the following way: $R_1 = R_{A,W,B}, R_2 = R_{B,U,C}, R_3 = R_{C,V,A}$. Thus, for each ($n^{\frac{2}{3}}$ many) $a \in dom(A)$ there are $n^{\frac{1}{3}}$ many matching $b \in dom(B)$ and possibly up to $n^{\frac{1}{3}}$ many matching $c \in dom(C)$. Thus, in total, there may be up to $n^{\frac{4}{3}}$ answers to $R_1 \bowtie R_2 \bowtie R_3$. To accomplish this also for $Q_{\hexagon}$, we only have to make sure that the variables U, V, W also join in R_4. For that, we simply set $R_4 = dom(U) \times dom(V) \times dom(W)$. This relation then satisfies its cardinality constraint. (Note that it does not satisfy its PC.) □

We now provide an algorithm (Algorithm 4) that enumerates $Q_{\hexagon}$ in linear time for databases with the PC on R_4. This proves that the output size is linear, simultaneously completing our proof of claim 1 and claim 2. Algorithm 4 first decomposes R_4 into three parts with one DC on each part. For this, we apply a linear time greedy partitioning algorithm (for more details see Algorithm 5) and show that it results in partitions whose constraints are within a factor of 3 from the optimal **PC**. Then, for each part, a variable order is selected to take advantage of all DCs. As an example, for the part $R_4^W(U, V, W)$, there are at most 3 matching $(u, v) \in dom(U, V)$ pair for each value of $w \in dom(W)$. Thus, starting with a tuple (a, w, b) of R_1, we can use this fact to determine these values for u, v and then combine this with $DC_{R_2}(BU, BUC, 1)$ to determine the unique value for c. By this reasoning, all of the inner for-loops iterate over a single element or 3 pairs of elements. Thus, the nested loops are linear in their total runtime. We defer the formal proof to App. B.1.

Lemma 4.3.3. *Algorithm 4 enumerates $Q^I_{\hexagon}$ in time $O(n)$ for databases $I \vDash$ **PC**.*

Lemma 4.3.2 and Lemma 4.3.3 together prove the first two claims of Theorem 4.3.1. This shows that PCs have an asymptotic effect on query bounds and that we can take advantage of PCs to design new WCOJ algorithms to meet these bounds. Nevertheless, one might wonder whether the variable elimination idea of established WCOJ algorithms can

Algorithm 4 Linear Hexagon Algorithm

1: $R_4^U, R_4^V, R_4^W \leftarrow$ **decompose**$(R_4, \{U, V, W\}, UVW)$
2: $\{DC_{R_4^U}(U, UVW, 3), DC_{R_4^V}(V, UVW, 3), DC_{R_4^W}(W, UVW, 3)\}$
3: **for** $(a, w, b) \in R_1$ **do**
4: **for** $(u, v) \in \pi_{U,V}\sigma_{W=w}R_4^W$ **do**
5: **for** $c \in (\pi_C\sigma_{B=b\wedge U=u}R_2 \cap \pi_C\sigma_{A=a\wedge V=v}R_3)$ **do**
6: **output** (a, b, c, u, v, w)
7: **end for**
8: **end for**
9: **end for**
10: **for** $(b, u, c) \in R_2$ **do**
11: **for** $(v, w) \in \pi_{V,W}\sigma_{U=u}R_4^U$ **do**
12: **for** $a \in (\pi_A\sigma_{B=b\wedge W=w}R_1 \cap \pi_A\sigma_{C=c\wedge V=v}R_3)$ **do**
13: **output** (a, b, c, u, v, w)
14: **end for**
15: **end for**
16: **end for**
17: **for** $(c, v, a) \in R_3$ **do**
18: **for** $(u, w) \in \pi_{U,W}\sigma_{V=v}R_4^V$ **do**
19: **for** $b \in (\pi_B\sigma_{A=a\wedge W=w}R_1 \cap \pi_B\sigma_{C=c\wedge U=u}R_2)$ **do**
20: **output** (a, b, c, u, v, w)
21: **end for**
22: **end for**
23: **end for**

already meet the improved bound and, in fact, achieve optimal runtimes. Proving the third claim in Theorem 4.3.1, we show that they cannot and, thus, the new techniques have to be employed. Specifically, we show that VAAT algorithms (Definition 4.1.5) require time $\Omega(n^{1.5})$ to compute $Q_{\hexagon}$ on database instances satisfying the PCs.

Lemma 4.3.4. *VAAT algorithms require time* $\Omega(n^{1.5})$ *to enumerate* $Q^I_{\hexagon}$ *for databases* $I \vDash$ **PC**.

Proof Sketch. It suffices to provide a collection of databases $\mathcal{I}$ such that $I \vDash \mathbf{PC}$ and $\max_i |Q^I_{\hexagon i}| = \Omega(n^{1.5})$ for databases $I \in \mathcal{I}$ and arbitrary ordering of the variables. To that end, we introduce two relations, a relation $C_{X,Y,Z}(X,Y,Z)$ and a set of disjoint paths $P_{X,Y,Z}(X,Y,Z)$. For $P_{X,Y,Z}(X,Y,Z)$, the domains of X,Y,Z are of size $\Theta(n)$ and $P_{X,Y,Z}$ simply matches X to Y and Z such that each $d \in dom(X) \cup dom(Y) \cup dom(Z)$ appears in exactly one tuple of $P_{X,Y,Z}$. On the other hand, think of $C_{X,Y,Z}$ as a complete bipartite graph from the domain of X to the domain of Y and Z uniquely identifies the edges. Thus, $|dom(X)| = |dom(Y)| = \Theta(\sqrt{n})$ while $|dom(Z)| = \Theta(n)$. Notice that for both relations, $DC(XY, XYZ, 1), DC(Z, XYZ, 1)$ hold. I.e., any pair of variables determine the last variable and there is a variable that determines the whole tuple on its own.

Consequently, the disjoint union of relations C and P multiple times (with permutated versions of C), e.g.,

$$R(X,Y,Z) = C_{X,Y,Z}(X,Y,Z) \cup C_{Y,Z,X}(Y,Z,X) \cup P_{X,Y,Z}(X,Y,Z),$$

satisfies $PC_R(\{X,Y,Z\}, XYZ, 1)$ and $DC(S_1S_2, XYZ, 1)$ for any $S_1S_2 \subseteq XYZ$.

Thus, we can set R_1, R_2, R_3, R_4 to be the disjoint union of P and all permutations of the relation C. Now, let $X_1, \ldots, X_6$ be an arbitrary variable order for $Q_{\hexagon}$. Then, a VAAT algorithm based on this variable order at least needs to compute the sets:

$$\bowtie_i \pi_{X_1} R_i, \bowtie_i \pi_{X_1X_2} R_i, \bowtie_i \pi_{X_1 \cdots X_3} R_i, \bowtie_i \pi_{X_1 \cdots X_4} R_i, \bowtie_i \pi_{X_1 \cdots X_5} R_i, \bowtie_i \pi_{X_1 \cdots X_6} R_i.$$

Let us consider the set $\bowtie_i \pi_{X_1 \cdots X_4} R_i$. There are two cases:

Case 1: X_5 and X_6 appear conjointly in a relation. Due to the symmetry of the query and the database, we can assume w.l.o.g, $X_5X_6 = UV$ and $\bowtie_i \pi_{X_1 \cdots X_4} R_i = \bowtie_i$

$\pi_{ABCW}R_i$. Furthermore, $\pi_{ABW}C_{A,B,W} \subseteq \pi_{ABW}R_1$, $\pi_{BC}C_{B,C,U} \subseteq \pi_{BC}R_2$, $\pi_{AC}C_{A,C,V} \subseteq \pi_{AC}R_3, \pi_W P_{U,V,W} \subseteq \pi_W R_4$. Thus, intuitively, for at least $\Omega(\sqrt{n})$ elements $a \in dom(A)$ there are $\Omega(\sqrt{n})$ elements $b \in dom(B)$ and $\Omega(\sqrt{n})$ elements $c \in dom(C)$ that all join, and for each pair a, b there is an element $w \in dom(W)$ that fits. In total, $|\bowtie_i \pi_{ABCW}R_i| = \Omega(n^{1.5})$.

Case 2: X_5 and X_6 do not appear conjointly in a relation. Due to the symmetry of the query and the database, we can assume w.l.o.g, $X_5X_6 = AU$ and $\bowtie_i \pi_{X_1 \cdots X_4}R_i$ $=\bowtie_i \pi_{BCVW}R_i$. Furthermore, $\pi_{BW}C_{B,W,A} \subseteq \pi_{BW}R_1$, $\pi_{BC}C_{B,C,U} \subseteq \pi_{BC}R_2$, $\pi_{CV}C_{C,V,A} \subseteq \pi_{CV}R_3, \pi_{VW}C_{V,W,U} \subseteq \pi_{VW}R_4$. Thus, intuitively, for at least $\Omega(\sqrt{n})$ elements $b \in dom(B)$ there are $\Omega(\sqrt{n})$ elements $w \in dom(W)$, $\Omega(\sqrt{n})$ elements $v \in dom(V)$ and, $\Omega(\sqrt{n})$ elements $c \in dom(C)$ that all join. In total, $|\bowtie_i \pi_{BCVW}R_i| = \Omega(n^2)$. □

4.4 Partition Constraints for General Conjunctive Queries

In the following, we extend the ideas of Section 4.3 to an arbitrary full conjunctive queries $Q(\mathbf{Z}) \leftarrow R_1(\mathbf{Z}_1) \bowtie \ldots \bowtie R_k(\mathbf{Z}_k)$ and an arbitrary set of partition constraints $\mathbf{PC} = \{PC_{P_1}(\mathcal{X}_1, \mathbf{Y}_1, d_1), \ldots, PC_{P_l}(\mathcal{X}_l, \mathbf{Y}_l, d_l)\}$. Recall, Algorithm 4 proceeds by decomposing R_4 in accordance with the PC and then, executes a VAAT style WCOJ algorithm over the decomposed instances. We proceed in the same way and start by concentrating on decomposing relations. After partitioning the relations, we show how to lift DC-based techniques for bounding and enumerating conjunctive queries to PCs.

4.4.1 Computing Constraints and Partitions

To take advantage of PCs, we need to be able to decompose an arbitrary relation $R(\mathbf{Z})$ according to a given partition constraint $PC(\mathcal{X}, \mathbf{Y}, d)$. For this task, we propose two poly-time algorithms; a linear approximate algorithm and a quadratic exact algorithm. Crucially, these algorithms do not need d to be computed beforehand, so these algorithms can also be used to compute PC constraints themselves, i.e. to compute the value of $PC(\mathcal{X}, \mathbf{Y})$. We start with the faster approximation algorithm before describing the exact algorithm.

Concretely, Algorithm 5 partitions a relation $R(\mathbf{Z})$ by distributing the tuples from the relation to partitions in a greedy fashion. At each point, it selects the set of variables $\mathbf{X} \in \mathcal{X}$ and particular value $\mathbf{x} \in \pi_{\mathbf{X}}R$ which occurs in the fewest tuples in the relation R. It then

adds those tuples to the partition $R^{\mathbf{X}}$ and deletes them from the relation R. Intuitively, high degree pair $(\mathbf{X}, \mathbf{x})$ will be distributed to partitions late in this process. At this point, most of the matching tuples will already have been placed in different partitions. Formally, we claim the following runtime and approximation guarantee for Algorithm 5.

Algorithm 5 Approximate Decomposition Algorithm

1: **decompose**$(R, \mathcal{X}, \mathbf{Y})$
2: **for** $\mathbf{X} \in \mathcal{X}$ **do**
3: $\quad R^{\mathbf{X}} \leftarrow \emptyset$
4: **end for**
5: **while** R is not empty **do**
6: $\quad \mathbf{X}, \mathbf{x} \leftarrow \text{argmin}_{\mathbf{X} \in \mathcal{X}, \mathbf{x} \in \pi_{\mathbf{X}} R} |\pi_{\mathbf{Y}} \sigma_{\mathbf{X}=\mathbf{x}} R|$
7: $\quad R^{\mathbf{X}} \leftarrow R^{\mathbf{X}} \cup \sigma_{\mathbf{X}=\mathbf{x}} R$
8: $\quad R \leftarrow R \setminus \sigma_{\mathbf{X}=\mathbf{x}} R$
9: **end while**
10: **return** $(R^{\mathbf{X}})_{\mathbf{X} \in \mathcal{X}}, \max_{\mathbf{X} \in \mathcal{X}} DC_{R^{\mathbf{X}}}(\mathbf{X}, \mathbf{Y})$

Theorem 4.4.1. *For a relation $R(\mathbf{Z})$ and subsets $\mathbf{Y} \subseteq \mathbf{Z}, \mathcal{X} \subseteq 2^{\mathbf{Y}}$, Algorithm 5 computes a partitioning $\bigcup_{\mathbf{X} \in \mathcal{X}} R^{\mathbf{X}} = R$ in time $O(|R|)$ (data complexity) such that*

$$DC_{R^{\mathbf{X}}}(\mathbf{X}, \mathbf{Y}, |\mathcal{X}|d)$$

holds for every $\mathbf{X} \in \mathcal{X}$ where $d = PC(\mathcal{X}, \mathbf{Y})$.

Proof Sketch. We start by providing intuition for the linear runtime. Each iteration of the while loop (line 4) places a set of tuples in a partition (line 6) and removes them from the original relation (line 7). Both of these operations can be done in constant time per tuple, so we simply need to show that we can identify the lowest degree attribute/value pair in constant time each iteration. This is done by creating a priority queue structure for each $\mathbf{X} \in \mathcal{X}$ where priority is equal to degree, and we begin by adding each tuple in R to each priority queue. Because the maximum degree is less than $|R|$, we can construct these queues in linear time using bucket sort. We will then decrement these queues by 1 each

time a tuple is removed from R. While arbitrarily changing an element's priority typically requires $O(log(|R|))$ in a priority queue, we are merely decrementing by 1 which is a local operation that can be done in constant time. So, construction and maintenance of these structures is linear w.r.t. data size. We can then use these queues to look up the lowest degree attribute/value pair in constant time.

Next, we prove the approximation guarantee by contradiction. If the algorithm produces a partition $R^{\mathbf{X}}$ with $DC_{R^{\mathbf{X}}}(\mathbf{X}, \mathbf{Y}) > |\mathcal{X}|d$, then there must be some value $\mathbf{x} \in \pi_{\mathbf{X}} R^{\mathbf{X}}$ where $|\pi_{\mathbf{Y}} \sigma_{\mathbf{X}=\mathbf{x}} R^{\mathbf{X}}| > |\mathcal{X}|d$. At the moment before this value was inserted into $R^{\mathbf{X}}$ and deleted from R, all attribute/value pairs must have had degree at least $|\mathcal{X}|d$ in the current state of R which we denote $R_{\mathcal{A}}$. Through some algebraic manipulation, we show that this implies $|R_{\mathcal{A}}| > \sum_{\mathbf{X} \in \mathcal{X}} |\pi_{\mathbf{X}} R_{\mathcal{A}}|d$. On the other hand, we know that $R_{\mathcal{A}}$ respects the original partition constraint because $R_{\mathcal{A}} \subseteq R$, and we show that this implies the converse $|R_{\mathcal{A}}| \leq \sum_{\mathbf{X} \in \mathcal{X}} |\pi_{\mathbf{X}} R_{\mathcal{A}}|d$. This is a contradiction, so our algorithm must not produce a poor approximation in the first place. □

Next, we describe an exact algorithm that requires quadratic time. Intuitively, Algorithm 6 also computes a decomposition of R in a greedy fashion by iteratively allocating (groups of) tuples $\mathbf{y}_0$ of $\pi_Y R$ to partitions $R^{\mathbf{X}}$, preferring allocations to partitions where the maximum over the relevant degree constraints, i.e., $\max_{\mathbf{X}} DC_{R^{\mathbf{X}}}(\mathbf{X}, \mathbf{Y})$, does not increase. However, decisions greedily made at the start may be sub-optimal and may not lead to a decomposition that minimizes $\max_{\mathbf{X}} DC_{R^{\mathbf{X}}}(\mathbf{X}, \mathbf{Y})$. To overcome this, instead of simply allocating $\mathbf{y}_0$ to a partition, the algorithm also checks whether it is possible to achieve a better overall decomposition by reallocating some other tuples in a cascading manner. To that end, we look for elements $\mathbf{y}_1 \in \pi_Y R^{\mathbf{X}_1}, \ldots, \mathbf{y}_m \in \pi_Y R^{\mathbf{X}_m}$ and a further $\mathbf{X}_{m+1}$ such that for all $\mathbf{y}_i, i = 1, \ldots, m$, the tuples matching $\mathbf{y}_i$ can be moved from $R^{\mathbf{X}_i}$ to $R^{\mathbf{X}_{i+1}}$ and the tuples matching $\mathbf{y}_0$ can be added to $R^{\mathbf{X}_1}$, all without increasing $\max_{\mathbf{X}} DC_{R^{\mathbf{X}}}(\mathbf{X}, \mathbf{Y})$. To achieve this, $\mathbf{y}_i$ is selected such that it matches $\mathbf{y}_{i-1}$ on the variables $\mathbf{X}_i$. Thus, for each $R^{\mathbf{X}_i}$, the value of $DC_{R^{\mathbf{X}_i}}(\mathbf{X}, \mathbf{Y})$ is the same before and after the update. There only has to be space for $\mathbf{y}_m$ in the final relation $R^{\mathbf{X}_{m+1}}$.

The sequence $(\mathbf{y}_1, \ldots, \mathbf{y}_m, \mathbf{X}_1, \ldots, \mathbf{X}_{m+1})$ constitutes an augmenting path which was

first introduced by [48] and used for matroids. Adapted to the present setting, we define an augmenting path as below. By slight abuse of notation, we write $\sigma_{\mathbf{X}=\mathbf{y}}R$ even though $\mathbf{y}$ fixes more variables than specified in $\mathbf{X}$. Naturally, this is meant to select those tuples of R that agree with $\mathbf{y}$ on the variables $\mathbf{X}$.

Definition 4.4.2. Let $(R^{\mathbf{X}})_{\mathbf{X}\in\mathcal{X}}$ be pairwise disjoint subsets of some relation $R(\mathbf{Z})$ with $\mathbf{Y} \subseteq \mathbf{Z}, \mathcal{X} \subseteq 2^{\mathbf{Y}}$, and let $\mathbf{y}_0 \in \pi_{\mathbf{Y}}R \setminus \pi_{\mathbf{Y}} \bigcup_{\mathbf{X}\in\mathcal{X}} R^{\mathbf{X}}$ be a new tuple. An augmenting path $(\mathbf{y}_1, \ldots, \mathbf{y}_m, \mathbf{X}_1, \ldots, \mathbf{X}_{m+1})$ satisfies the following properties:

1. For all $i \in \{1, \ldots, m+1\} : \mathbf{X}_i \in \mathcal{X}$.
2. For all $i \in \{1, \ldots, m\} : \mathbf{y}_i \in \pi_{\mathbf{Y}} R^{\mathbf{X}_i}$.
3. For all $i \in \{1, \ldots, m\} : \mathbf{y}_i$ agrees with $\mathbf{y}_{i-1}$ on $\mathbf{X}_i$.
4. $|\pi_{\mathbf{Y}}\sigma_{\mathbf{X}_{m+1}=\mathbf{y}_m} R^{\mathbf{X}_{m+1}}| < \max_{\mathbf{X}} DC_{R^{\mathbf{X}}}(\mathbf{X}, \mathbf{Y})$.

We omit the references to $(R^{\mathbf{X}})_{\mathbf{X}\in\mathcal{X}}$, $\mathbf{Y}$, and $\mathbf{y}_0$ when they are clear from the context.

The next theorem shows that Algorithm 6 correctly computes an optimal decomposition.

Theorem 4.4.3. *For a relation $R(\mathbf{Z})$ and subsets $\mathbf{Y} \subseteq \mathbf{Z}, \mathcal{X} \subseteq 2^{\mathbf{Y}}$, Algorithm 6 computes a partitioning $\bigcup_{\mathbf{X}\in\mathcal{X}} R^{\mathbf{X}} = R$ in $O(|R|^2)$ time (data complexity) such that*

$$DC_{R^{\mathbf{X}}}(\mathbf{X}, \mathbf{Y}, d)$$

holds for every $\mathbf{X} \in \mathcal{X}$ where $d = PC(\mathcal{X}, \mathbf{Y})$.

Proof Sketch. The intuition for the algorithm's quadratic runtime boils down to ensuring that the search for an augmenting path is linear in $|R|$. To show that this is the case, we model the search for an augmenting path as a breadth-first search over a bipartite graph whose nodes correspond to full tuples $\mathbf{y} \in \pi_{\mathbf{Y}}R$ and the partial tuples $\mathbf{x} \in \pi_{\mathbf{X}}R$ for all $\mathbf{X} \in \mathcal{X}$. An edge exists between a tuple $\mathbf{y}$ and a partial tuple $\mathbf{x}$ if they agree on their shared attributes. This search starts from the new tuple $\mathbf{y}_0$ and completes when it finds a tuple $\mathbf{y}_m$ that can be placed in one of the relations $R^{\mathbf{X}}$ without increasing its DC. The number

Algorithm 6 Exact Decomposition Algorithm

decompose$(R, \mathcal{X}, \mathbf{Y})$
$d \leftarrow 0$
for $\mathbf{X} \in \mathcal{X}$ **do**
  $R^{\mathbf{X}} \leftarrow \emptyset$
end for
for $\mathbf{y}_0 \in \pi_{\mathbf{Y}} R$ **do**
  if there exists a shortest augmenting path $(\mathbf{y}_1, \ldots, \mathbf{y}_m, \mathbf{X}_1, \ldots, \mathbf{X}_{m+1})$ **then**
    for $i = m, \ldots, 1$ **do**
      $R^{\mathbf{X}_{i+1}} \leftarrow R^{\mathbf{X}_{i+1}} \cup \sigma_{\mathbf{Y}=\mathbf{y}_i} R^{\mathbf{X}_i}$
      $R^{\mathbf{X}_i} \leftarrow R^{\mathbf{X}_i} \setminus \sigma_{\mathbf{Y}=\mathbf{y}_i} R^{\mathbf{X}_i}$
    end for
    $R^{\mathbf{X}_1} \leftarrow R^{\mathbf{X}_1} \cup \sigma_{\mathbf{Y}=\mathbf{y}_0} R$
  else
    $d \leftarrow d + 1$
    $R^{\mathbf{X}} \leftarrow R^{\mathbf{X}} \cup \sigma_{\mathbf{Y}=\mathbf{y}_0} R$ $\{\mathbf{X} \in \mathcal{X}$ can be selected arbitrarily here.$\}$
  end if
  $R \leftarrow R \setminus \sigma_{\mathbf{Y}=\mathbf{y}_0} R$
end for
return $(R^{\mathbf{X}})_{\mathbf{X} \in \mathcal{X}}, d$

of edges and vertices in this graph is linear in the input data, so the breadth-first search is linear as well.

We now provide intuition for the algorithm's correctness. Suppose that we are partway through the algorithm and are now at the iteration where we add $\mathbf{y}_0$ to a partition. Further, let $R_{\mathcal{A}}$ be the set of tuples which have been added so far, including $\mathbf{y}_0$. Because there exists an optimal partitioning of $R_{\mathcal{A}}$, we should be able to place $\mathbf{y}_0$ in the partition where it exists in the optimal partitioning. If we cannot, then a tuple in that partition with the same $\mathbf{X}$-value must not be in its optimal partition. We identify one of these tuples and move it to its optimal partition. We continue this process inductively of shifting one tuple at a time to its optimal partition until one of these shifts no longer violates the PC. The sequence of shifts that we have performed constitutes an augmenting path by definition, and its existence implies that we would not violate the PC in this iteration by incrementing d past it. This construction process must end in a finite number of moves because each step increases the number of tuples placed in their optimal partitioning. If all tuples are in their optimal partition, the final shift must not have violated the PC due to the optimal partition's definition. □

4.4.2 Lifting DC-Based Bounds and Algorithms to PCs

We now apply the developed decomposition algorithms to the general case and show how to use WCOJ algorithms as a black box to carry over guarantees for DCs to the general case of PCs. First, we extend the definition of an arbitrary cardinality bound CB defined over sets of DCs to sets of PCs.

Definition 4.4.4. Let CB be a cardinality bound. Then, we extend CB to PCs by setting

$$CB(Q, \mathbf{PC}) := \sum_{\mathbf{X}^1 \in \mathcal{X}_1, \cdots, \mathbf{X}^l \in \mathcal{X}_l} CB(Q, \{DC_{P_1}(\mathbf{X}^1, \mathbf{Y}_1, d_1), \ldots, DC_{P_l}(\mathbf{X}^l, \mathbf{Y}_l, d_l)\}),$$

where $\mathbf{PC} = \{PC_{P_i}(\mathcal{X}_i, \mathbf{Y}_i, d_i) \mid i\}$ is an arbitrary set of partition constraints.

Note that the bound in Definition 4.4.4 is well-defined: If $\mathbf{PC}$ is simply a set of DCs, the extended version of the cardinality bound CB coincides with its original definition. If $\mathbf{PC}$ is an arbitrary set of PCs, then it remains a valid bound on the size of the join.

Theorem 4.4.5. *Let $CB(Q, \mathbf{PC})$ be an extended cardinality bound. Then,*

$$|Q^I| \leq CB(Q, \mathbf{PC}) \quad \forall I \vDash \mathbf{PC}.$$

Proof. Let I be a database satisfying $\mathbf{PC} = \{PC_{P_i}(\mathcal{X}_i, \mathbf{Y}_i, d_i) \mid i = 1, \ldots, l\}$ and $Q \leftarrow R_1(\mathbf{Z}_1) \bowtie \cdots \bowtie R_k(\mathbf{Z}_k)$. Thus, for each P_i there is a partitioning $P_i = \bigcup_{j=1}^{|\mathcal{X}_i|} P_i^j$ with $DC_{P_i^j}(\mathbf{X}_i^j, \mathbf{Y}_i, d_i)$ and $\mathcal{X}_i = \{X_i^j | j = 1, \ldots, |\mathcal{X}_i|\}$. Now, let us now fix some $j_1 \in \{1, \ldots, |\mathcal{X}_1|\}, \ldots, j_l \in \{1, \ldots, |\mathcal{X}_l|\}$. Furthermore, for each $i \in \{1, \ldots, l\}$ let $s(i) \in \{1, \ldots, k\}$ be the relation $R_{s(i)} = P_i$. Set $R_s^{j_1 \cdots j_l} = \bigcap_{i:s=s(i)} P_i^{j_i}$ for all $s \in \{1, \ldots, k\}$. Then consider $Q^{j_1 \cdots j_l} = R_1^{j_1 \cdots j_l} \bowtie \cdots \bowtie R_k^{j_1 \cdots j_l}$. We claim two things:

1. $Q^{j_1 \cdots j_l}$ partitions Q^I, i.e., $Q^I = \bigcup_{j_1=1}^{|\mathcal{X}_1|} \cdots \bigcup_{j_l=1}^{|\mathcal{X}_l|} Q^{j_1 \cdots j_l}$, and

2. $|Q^{j_1 \cdots j_l}| \leq CB(Q, \{DC_{P_1}(\mathbf{X}_1^{j_1}, \mathbf{Y}_1, d_1), \ldots, DC_{P_l}(\mathbf{X}_l^{j_l}, \mathbf{Y}_l, d_l)\})$.

For the first bullet point, let $t \in Q^I$. Clearly, for each $i \in \{1, \ldots, l\}$ there exists exactly one j_i such that t agrees with an element in $P_i^{j_i}(\mathbf{Z}_{s(i)})$ on the variables $\mathbf{Z}_{s(i)}$. Thus, $t \in Q^{j_1 \cdots j_l}$.

For the second bullet point, consider the relations $P_i^{j_i}$. These are supersets of the relations $R_{s(i)}^{j_1 \cdots j_l}$ and, therefore, we can assert $DC_{R_{s(i)}^{j_1 \cdots j_l}}(\mathbf{X}_i^{j_i}, \mathbf{Y}_i, d_i)$. Viewed as a query, $Q^{j_1 \cdots j_l}$ has the same form as Q. Hence, $|Q^{j_1 \cdots j_l}| \leq CB(Q, \{DC_{P_1}(\mathbf{X}_1^{j_1}, \mathbf{Y}_1, d_1), \ldots, DC_{P_l}(\mathbf{X}_l^{j_l}, \mathbf{Y}_l, d_l)\})$. □

Crucially, this upper bound is not loose; it preserves the tightness of any underlying DC-based bound. Specifically, the extended version of the combinatorics bound CB_{Comb} is asymptotically close to the actual worst-case size of the join, with the constant depending on the query. For example, the extended version of combinatorics bound is $O(n)$ for the query $Q_{\hexagon}$ (see Section 4.3) when using all PCs while the combinatorics bound based on the DCs alone was $\Omega(n^{\frac{4}{3}})$.

Lemma 4.4.6. *Let $CB_{Comb}(Q, \mathbf{PC})$ be the extended version of the combinatorics bound CB_{Comb}. Then,*

$$CB_{Comb}(Q, \mathbf{PC}) = O(\max_{I \vDash \mathbf{PC}} |Q^I|).$$

Proof. Let $\mathbf{X}^1 \in \mathcal{X}_1, \ldots, \mathbf{X}^l \in \mathcal{X}_l$ be such that $CB_{Comb}(Q, \mathbf{DC})$ is maximized where $\mathbf{DC} := \{DC_{P_1}(\mathbf{X}^1, \mathbf{Y}_1, d_1), \ldots, DC_{P_l}(\mathbf{X}^l, \mathbf{Y}_l, d_l)\}$. Thus, there exists a database I such that $I \vDash \mathbf{DC}$ and $|Q^I| = CB_{Comb}(Q, \mathbf{DC})$. Furthermore, I must then also trivially satisfy $\mathbf{PC}$ by definition. For each PC on P_i the witnessing partitioning is simply $P_i = P_i \cup \emptyset \cup \cdots \cup \emptyset$. Therefore,

$$CB_{Comb}(Q, \mathbf{PC}) \leq |\mathcal{X}_1| \ldots |\mathcal{X}_l||Q^I| = O(\max_{D \vDash \mathbf{PC}} |Q^I|).$$

For the last equality, note that $|\mathcal{X}_1| \ldots |\mathcal{X}_l|$ is a query-dependent constant as we assume all PCs to be on different variables $\mathcal{X}_i$. □

Next, we show how algorithms that are worst-case optimal relative to a cardinality bound can likewise be adapted and become worst-case optimal relative to the extended bound. In this way, progress on WCOJ algorithms based on DCs immediately leads to improved algorithms that take advantage of PCs.

Theorem 4.4.7. *Given a cardinality bound $CB_{\mathcal{A}}$. If there exists a join enumeration algorithm $\mathcal{A}$ which is worst-case optimal relative to $CB_{\mathcal{A}}$, then there exists an algorithm $\mathcal{A}^*$ which is worst-case optimal relative to the extended version of the cardinality bound. I.e., for fixed query Q, $\mathcal{A}^*$ runs in time $O(|I| + CB_{\mathcal{A}}(Q, \mathbf{PC}))$ for arbitrary $\mathbf{PC}$ and database $I \vDash \mathbf{PC}$.*

Proof. Suppose we have a query $Q \leftarrow R_1(\mathbf{Z}_1) \bowtie \ldots R_k(\mathbf{Z}_k)$, a set of partition constraints $\mathbf{PC} = \{PC_{P_1}(\mathcal{X}_1, \mathbf{Y}_1, d_1), \ldots, PC_{P_l}(\mathcal{X}_l, \mathbf{Y}_l, d_l)\}$, and a database $I \vDash \mathbf{PC}$. We can follow the same idea already used in the proof of Proposition 4.4.5. Concretely, we partition each P_i into $(P_i^j)_{j=1,\ldots,|\mathcal{X}_i|}$. For this, we use Algorithm 5 and Theorem 4.4.1. Thus, we get $DC_{P_i^j}(\mathbf{X}_i^j, \mathbf{Y}_i, |\mathcal{X}_i|d_i)$. Let $\alpha := \max_i |\mathcal{X}_i|$.

We can now continue following the idea of proof of Proposition 4.4.5 up to the two claims where we only need the first. Now, instead of bounding the size of $Q^{j_1,\ldots,j_l}$, we now want to compute each subquery using $\mathcal{A}$. Thus, note that $DC_{R_{s(i)}^{j_1 \ldots j_l}}(\mathbf{X}_i^{j_i}, \mathbf{Y}_i, \alpha d_i)$. This implies that $\mathcal{A}$ runs in time $O(|I| + CB_{\mathcal{A}}(Q, \{DC_{P_i}(\mathbf{X}_i^{j_i}, \mathbf{Y}_i, \alpha d_i) \mid i\}))$. The constant α can be hidden in the O-notation while summing up the time required for each $Q^{j_1,\ldots,j_l}$ results in an overall runtime of $O(|I| + \sum_{\mathbf{X}^1 \in \mathcal{X}_1 \cdots \mathbf{X}^l \in \mathcal{X}_l} CB_{\mathcal{A}}(Q, \{DC_{P_i}(\mathbf{X}_i^{j_i}, \mathbf{Y}_i, d_i) \mid i\}) = O(|I| + CB_{\mathcal{A}}(Q, \mathbf{PC}))$. □

For example, PANDA is a WCOJ algorithm relative to the polymatroid bound, and we can use this approach to translate it to an optimal algorithm for the extended polymatroid bound. While it is an open problem to produce a WCOJ algorithm relative to $CB_{Comb}(Q, \mathbf{DC})$, any such algorithm now immediately results in a WCOJ algorithm relative to $CB_{Comb}(Q, \mathbf{PC})$. Combined with Proposition 4.4.6 this implies that such an algorithm then only takes time relative to the worst-case join size of instances that satisfy the same set of PCs.

Corollary 4.4.8. *If there exists a WCOJ algorithm relative to $CB_{Comb}(Q, \mathbf{DC})$, then there exists an WCOJ algorithm relative to $CB_{Comb}(Q, \mathbf{PC})$.*

4.5 Conclusions and Future Work

In this work, we introduced PCs as a generalization of DCs, uncovering a latent structure within relations and that is present in standard benchmarks. PCs enable a more refined approach to query processing, offering asymptotic improvements to both cardinality bounds and join algorithms. We presented algorithms to compute PCs and identify the corresponding partitioning that witness these constraints. To harness this structure, we then developed techniques to lift both cardinality bounds and WCOJ algorithms from the DC framework to the PC framework. Crucially, our use of DC-based bounds and algorithms as black boxes allows future advances in the DC setting to be seamlessly integrated into the PC framework.

On the practical side, future research should explore when and where it is beneficial to leverage the additional structure provided by PCs. In particular, finding ways to minimize the constant factor overhead by only considering a useful subset of PCs or sharing work across evaluations on different partitions could yield significant practical improvements in query performance. On the other hand, further theoretical work should try to incorporate additional statistics into this partitioning framework, e.g., l_p-norms of degree sequences. Ultimately, the goal of this line of work is to capture the inherent complexity of join instances through both the structure of the query and the data.

Chapter 5

SAFEBOUND

This chapter attempts to translate the degree sequence bound, presented in Ch. 3, from theory to practice. Theoretically, the degree sequence bound takes in a conjunctive query and a set of degree constraints, and it produces a tight upper bound on the query's output size. Translating this theory into a practical system that can handle real data and queries requires tackling two main challenges that were left unsolved in that chapter. First, no method was presented for compressing degree sequences, and a poor choice of compression can result in a bad space-accuracy tradeoff. Second, selection predicates (e.g. "CUSTOMER.AGE ¿ 35") were not addressed, and they present a major challenge in cardinality estimation because we do not have access to the data at runtime. This means that we need to collect additional statistics offline to estimate the impact of selection predicates.

To solve these challenges, we present SafeBound, the first practical system for generating cardinality bounds. Like any cardinality estimator, SafeBound has an offline and an online phase (Sec. 5.2.1). During the offline phase, it computes a set of summary structures that allow it to bound the degree sequence of relations after they are filtered by predicates. To support a wide array of predicates (equality, range, LIKE, conjunctions, and disjunctions), SafeBound adapts classical structures like histograms, most common value lists, and trigrams, altering each one to store degree sequences and to address the unique requirements of cardinality bounds. Because these degree sequences are too large to be stored explicitly, we develop an algorithm for compressing degree sequences. Any compression necessarily looses some precision, and no compression scheme is optimal for all queries. Thus, we describe a heuristic-based compression algorithm in Sec. 5.2.3. At the end of the offline phase of SafeBound we have a set of summary structures that produce compressed degree sequences. Next, during the online phase, SafeBound takes a query consisting of joins and predicates, and computes a guaranteed upper bound on its output cardinality, using the

compressed degree sequences. Following Alg. 3, SafeBound orients the query as a tree then computes the bound bottom-up in linear time w.r.t. the size of the relevant compressed degree sequences. In addition to these fundamental techniques, we describe several optimizations in Sec. 5.3. Finally, we evaluate SafeBound empirically in Sec. 5.4. Across four workloads, SafeBound achieves up to 80% lower end-to-end runtimes than PostgreSQL, and is on par or better than state of the art ML-based estimators and pessimistic cardinality estimators. Its performance gains come especially from the expensive queries, because its guarantees on the cardinality prevent the optimizer from making overly optimistic decisions. In the tradeoff between accuracy and build time, SafeBound aims for accuracy. This results in slower build times than Postgres although it is 2-20x faster to build than state of the art ML methods. SafeBound also saves up to 500x in query planning time, and uses up to 6.8x less space compared to state of the art cardinality estimation methods.
In summary, this chapter makes the following contributions:

- *SafeBound Design:* We describe the architecture of SafeBound, the first practical system for cardinality bounds, in Sec. 5.2.1.

- *Predicates:* We introduce a scheme for conditioning degree sequences on predicates (Sec. 5.2.2).

- *Compression Algorithm:* We present an algorithm for compressing a degree sequence with minimal loss (Sec. 5.2.3).

- *Optimizations:* We describe several optimizations in Sec. 5.3.

- *Experimental Evaluation:* We perform a thorough experimental evaluation on the JOB-Light, JOB-LightRanges, JOB-M, and STATS-CEB benchmarks demonstrating SafeBound's fast inference, low memory overhead, and nearly optimal workload runtimes across all benchmarks. Sec. 5.4

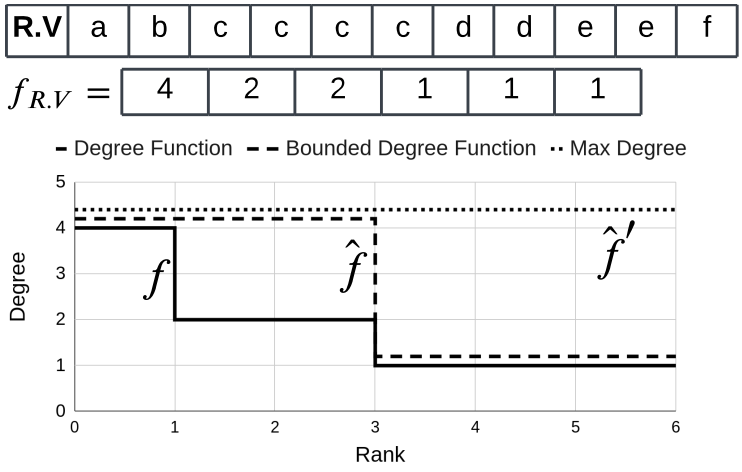


Figure 5.1: An example column, $R.V$, and its degree sequence f.

5.1 Abbreviated Description of the Degree Sequence Bound

For a formal description of the degree sequence bound, we refer the reader to Ch. 3 which includes the full details of the bound. However, this chapter does not require the full complexity presented there, so we provide an abbreviated description below which should be sufficient for understanding SafeBound.

5.1.1 The Degree Sequence

Consider a variable V (a.k.a. column) of a relation R with domain $\mathbb{D}_V$ and values $v \in \mathbb{D}_V$. The frequency of a particular value v in relation R is $|\sigma_{V=v}(R)|$. Ranking the values in descending order by frequency, we get a list of values $v^{(1)}, \ldots, v^{(|\mathbb{D}_V|)}$ whose frequencies, $f_{R.V,i} \stackrel{\text{def}}{=} |\sigma_{V=v^{(i)}}(R)|$, form the *degree sequence* of $R.V$: $f_{R.V,1} \geq f_{R.V,2} \geq \cdots$ The index i is called the *rank*. The *cumulative degree sequence*, CDS, is the running sum of the DS, i.e. $F_{R.V,i} \stackrel{\text{def}}{=} \sum_{j=1}^{i} f_{R.V,j}$. In the other direction, the DS can be defined as the *discrete derivative* of the CDS, $f_{R.V,i} = \Delta_i F_{R.V,i} = F_{R.V,i} - F_{R.V,i-1}$.

We often manipulate these sequences as piecewise functions and may use the following notation, $f_{R.V}(i) = f_{R.V,i}$ and $F_{R.V}(i) = F_{R.V,i}$, for the DS and CDS, respectively. The set of DSs for a relation R will be denoted by s_R, and the set of DSs for all relations by s;

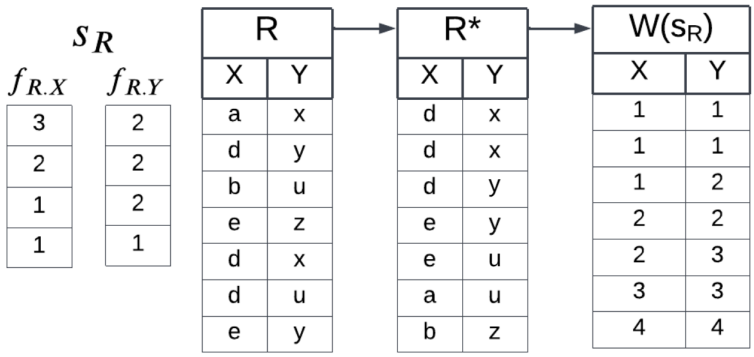


Figure 5.2: Example worst-case instance.

similarly, S_R and S stand for the CDSs of one relation, or all relations. Let $s, \hat{s}$ be two sets of statistics; we write $s \leq \hat{s}$ to mean that $f_{R.V}(i) \leq \hat{f}_{R.V}(i)$ for all relations R, attributes V, and ranks i. Given a set of statistics $\hat{s}$ and a database D, we say that D is *consistent* with $\hat{s}$ if $s(D) \leq \hat{s}$, where $s(D)$ are the degree sequences for the instance D. An example of these concepts can be seen in Figure 5.1. At the top are the actual values of a column $R.V$ of the database, and below is the degree sequence: the value 4 corresponds to c, c, c, c, the values $2, 2$ correspond to d, d and e, e, etc. The graph shows the degree sequence f as a solid line. In general, the degree sequence can be large. For the purpose of computing an upper bound of the size of the query, it suffices to compress the DS by upper bounding it using a piecewise constant function: two such functions are shown in the figure, $f \leq \hat{f} \leq \hat{f}'$.

5.1.2 The Degree Sequence Bound

The Worst-Case Instance: The DSB is defined in terms of a *worst-case relation*, $W(s_R)$, associated with the statistics s_R, and the *worst-case instance* of the database, $W(s)$, consisting of all relations $W(s_R)$. These relations are defined such that the size of any query's answer on $W(s)$ is an upper bound of its size on any database consistent with s; in this sense, $W(s)$ is the "worst" instance. Because degree sequences do not capture the correlation between columns within relations or between relations, the worst-case assumption is

that the frequency of values is perfectly correlated across columns both within and between relations: i.e., high frequency values occur in the same tuples within relations and join with high frequency values across relations.

We illustrate how to produce a worst-case relation $W(s_R)$ from a binary relation $R(X, Y)$ in Figure 5.2. While we are given only the statistics s_R, i.e. the two degree sequences $f_{R.X}, f_{R.Y}$, we also show an instance R, to help build some intuition into $W(s_R)$. First, we sort each column independently by frequency to produce R^*. This ensures that high-frequency values occur in the same tuples. Next, we relabel our join values in order of frequency to produce $W(s_R)$, i.e. $x^{(1)} = 1, x^{(2)} = 2, \ldots$. This ensures that, when we join, say, R and T, the high-frequency values in $W(s_R)$ will join with the high-frequency values in $W(s_T)$.

The Degree Sequence Bound: The DSB is the size of the query Q run on the worst-case instance $W(s)$, in other words, $|Q(W(s))|$. The following is shown in Ch. 3:

Theorem 5.1.1. *Suppose Q is a Berge-acyclic query, and let s be a set of degree sequences, one for each attribute of each relation. Then, the following is true,*

$$\forall D \models s \quad |Q(D)| \leq |Q(W(s))| \tag{5.1}$$

Further, $W(s) \models s$, which proves that the bound in (5.1) *is tight.*

Suppose that $\hat{s}$ is a compressed (lossy) representation of s, such that $\sum_{i=1}^{j} s_i \leq \sum_{i=1}^{j} \hat{s}\ \forall\ j$. For example, $\hat{s}$ may replace the degree sequences with piecewise constant functions with a small number of segments, as in Fig 5.1. Then $|Q(W(\hat{s}))|$ is still an upper bound for $Q(D)$, because $D \models s$ implies $D \models \hat{s}$, and we apply the theorem to $\hat{s}$.

5.2 SafeBound

In this section, we present SafeBound, the first practical system for cardinality bounding, which is based on several extensions of the results in Ch. 3. We start with an overview of SafeBound.

5.2.1 Overview

SafeBound has an offline and an online phase. During the *offline phase* the system computes the degree sequences of every join-able attribute of every relation (keys and foreign keys). In addition it also considers a variety of predicate types on each relation, and computes refined degree sequences conditioned on those predicates: this is described in Sec. 5.2.2. The last step of the offline phase consists of compressing the degree sequences; this is described in Sec. 5.2.3. Rather than directly compressing the degree sequence, it compresses the cumulative degree sequence which does not inflate the cardinality of the relation and the algorithm for doing this is described in 5.2.3. During the *online phase*, SafeBound receives a query, as defined in Sec. 5.1, and computes an upper bound on the query's output using the pre-computed compressed degree sequences. It does not apply the formula in Theorem 5.1.1 naively, but, instead, it implements a fast algorithm that runs in time proportional to the total size of the compressed sequences; this is described in Sec. 5.2.4.

Example 5.2.1. *We will refer to the following running example:*

$$Q \stackrel{def}{=} R(X, A, B) \wedge S(X, Y, C) \wedge T(Y)$$
$$\wedge (A < 5) \wedge (B = 2) \wedge (C \texttt{ LIKE '\%Abdul\%'})$$

During the offline phase (before the query arrives) SafeBound computes the degree sequences for $R.X, R.A, \ldots, T.Y$, as well as degree sequences conditioned on predicates, e.g. degree sequences of $R.X$ conditioned on range predicates applied to $R.A$. All these degree sequences are compressed and stored. During the online phase, SafeBound takes the query Q above, and uses all available degree sequences to compute an upper bound to the query's output.

We will use the following terminology. A column to which a predicate is applied is called a *filter column*; a column used in a join is called a *join column*. Note that a column can be both a filter column and a join column.

5.2.2 Conditioning on Predicates

Predicates in a query can significantly reduce the cardinality of the output. SafeBound accounds for predicates by computing additional degree sequences for each join column by

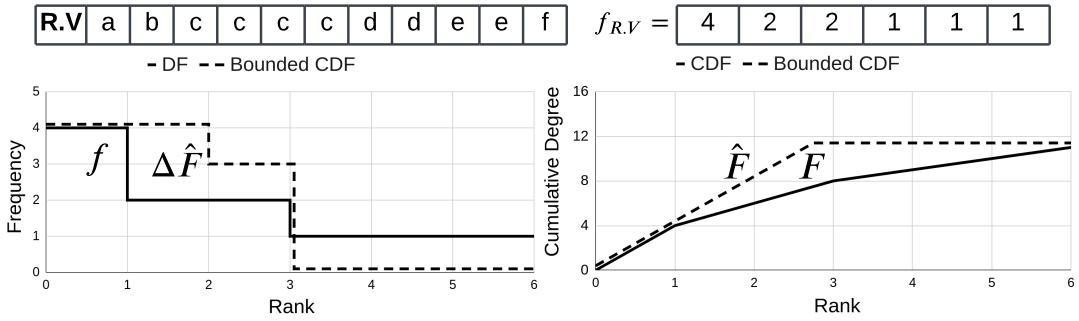


Figure 5.3: Compressed degree sequence example.

conditioning on predicates. SafeBound supports five types of predicates: equality, range, conjunctions, LIKE, and disjunctions. In this section we assume that all degree sequences are exact; once we replace them with compressed sequences in the next section, we will discuss some minor adjustments to the formulas introduced here.

Equality Predicates: The main idea is the following. Let V be a join column of a relation R, and consider a query that includes an equality predicate, $R.A = a_k$, for some constant $a_k \in \mathbb{D}_A$. If we ignore the predicate and compute the upper bound of the query without this predicate, then we massively overestimate the query's cardinality. There are two extreme ways to account for the equality predicate. At one extreme, we could compute a separate degree sequence, $f_{R.V|(A=a_\ell)}$, for each value $a_1, a_2, \ldots \in \mathbb{D}_A$ from the subset $\sigma_{A=a_\ell}(R)$. At query estimation time, we use the degree sequence $f_{R.V|(A=a_k)}$ instead of the unconditioned sequence $f_{R.V}$. But this approach is prohibitive, because it requires storing a large number of degree sequences. At the other extreme, we could store only the max of all these individual degree sequences,

$$f_{R.V|A} \stackrel{\text{def}}{=} \max_{\ell} f_{R.V|(A=a_\ell)} \tag{5.2}$$

and use this to estimate the query's upper bound. This uses very little space, only one DS which we use for every equality predicate on A, no matter the constant a_k. But this may lead to significant over-approximations of the upper bound. SafeBound adopts a compromise. For each attribute A, it computes a Most Common Value (MCV) list, computes separate degree sequences for V conditioned on each of these values, and computes one default DS,

given by formula (5.2), where a_ℓ ranges over non-MCV values. At estimation time, if a_k is in the MCV list then we use its own degree sequence, otherwise we use the default one. In our system, we chose between 1000 and 5000 values to include in every MCV list. We will slightly revisit Eq. (5.2) in the next section.

Range Predicates: To handle range predicates, SafeBound uses a data structure that builds on the idea of traditional histograms. The naive way to do this is to compute an equi-depth histogram over $R.A$ where each bucket stores the degree sequence of $R.V$ restricted to that bucket. However, a range predicate may overlap multiple buckets which would require us to return the summation of the degree sequences in those buckets. This summation artificially inflates the highest frequency values, increasing the DSB significantly. Instead, we build a hierarchy of equi-depth histograms with $2^k, 2^{k-1}, \ldots, 2^1$ buckets. At query time, we identify the smallest bucket which fully encapsulates the range query and return the degree sequence stored there. In our system, we typically used $k = 7$.

Conjunctions: Suppose that the query contains a conjunction of predicates on the same relation R: $P_1(R.A) \wedge P_2(R.B) \wedge \cdots$ For each predicate, we have computed a degree sequence conditioned on that predicate, $f_{R.V|A}, f_{R.V|B}, \ldots$ Then, we take their minimum:

$$f_{R.V|(A \wedge B \wedge \cdots)}(i) \stackrel{\text{def}}{=} \min(f_{R.V|A}(i), f_{R.V|B}(i), \ldots) \tag{5.3}$$

Referring to our running Example 5.2.1, there are two predicates on R. The degree sequence $f_{R.X}$ conditioned on the conjunction is $f(i) \stackrel{\text{def}}{=} \min(f_{R.X|(A<5)}(i), f_{R.X|(B=2)}(i))$.

LIKE Predicates: SafeBound converts a predicate $R.A$ `LIKE'%xyzu%'` into a conjunction of predicates on 3-grams. For every attribute $R.A$ of type `text`, SafeBound first computes an MCV list of 3-grams that occur in A and calculates the degree sequences of $R.V$ conditioned on all 3-grams that occur in the MCV. Separately, it calculates the degree sequence of $R.V$ conditioned on A not containing any 3-gram in the MCV. At query time, we split the text in the `LIKE` predicate into 3-grams, then compute the min of all degree sequences conditioned on each 3-gram that occurs in the MCV list. If none of the predicate's 3-grams appear in this list, we use the degree sequence conditioned on not containing common 3-grams. Referring to our running Example 5.2.1, the string `'%Abdul%'` is split into the 3-grams `Abd`, `bdu`, `dul`; for each 3-gram we retrieve the degree sequence $S.X$ conditioned on that 3-gram,

then compute their min (or take the default if none of them are in the MCV list); we apply the same process to compute the degree sequence of $S.Y$.

Disjunctions: Suppose a query has a disjunction of predicates over a relation R, $P_1(R.A_1) \vee \cdots \vee P_k(R.A_k)$. For each predicate we have the degree sequence of $R.V$ conditioned on it; then, we take their sum. For example, the `IN` predicate in SQL is treated as a disjunction: the degree sequence condition on of $R.A$ IN ['a','b','c'] is $f_{R.V|(A=a)} + f_{R.V|(A=b)} + f_{R.V|(A=c)}$.

Example 5.2.2. *The* `Title` *relation in the JOBLightRanges benchmark has 7 filter columns (*`episode_nr`*,* `season_nr`*,* `production_year`*,* `series_years`*,* `phonetic_code`*,* `series_years`*,* `imdb_index`*) and two join columns (*`id`*,* `kind_id`*). This results in seven histograms, MCV lists, and, for the string attributes, 3-gram lists which store 2 degree sequences per bin, MCV, and tri-gram, respectively. In total, there are* $18,522$ *degree sequences for the relation* `Title`*, each describing a subset of the table. This motivates our compression technique in Sec. 5.2.3, and the group compression in Sec. 5.3.1.*

Discussion: SafeBound computes DS's only for *join columns* (keys and foreign keys), each conditioned on every *filter column*. In theory this could lead to $O(n^2)$ DS's for a table with n attributes, but in practice, a typical table has $O(1)$ foreign keys, resulting in $O(n)$ DS's per table.

5.2.3 Compressing Degree Sequences

The degree sequence statistics, s, are as large as the database instance, D, hence they are impractical for cardinality estimation. SafeBound compresses each degree sequence using a piecewise constant function with a small number of segments. We denote by $\hat{s}$ the collection of compressed degree sequences. In Ch. 3 (Thm. 3.5.3, we proved that the true degree sequence of a relation can be replaced with a compressed version while still maintaing an upper bound. However, it required that the cumulative degree sequence (CDS) of the compressed version lies above the CDS of the true degree sequence. For convenience, we provide the notation again here,

Definition 5.2.3. Let $R.V$ be a column with degree sequence $f_{R.V}$. We say that $\hat{f}$ is *valid* for $f_{R.V}$ if (a) it is a degree sequence, meaning it is non-increasing, $\hat{f}(i-1) \geq \hat{f}(i)$, (b) its

CDS $\hat{F}(i) \stackrel{\text{def}}{=} \sum_{j \le i} \hat{f}(i)$ dominates the CDS of $R.V$: $F_{R,V}(i) \stackrel{\text{def}}{=} \sum_{j \le i} f_{R.V}(i)) \le \hat{F}(i)$, $\forall i$, (c) it preserves the cardinality $|R| = F_{R.V}(|\mathbb{D}_V|) = \hat{F}(|\mathbb{D}_V|)$.

To summarize, SafeBound compresses every DS $f_{R.V}$ into a valid DS $\hat{f}_{R.V}$. Since $\hat{f}_{R.V}$ no longer dominates $f_{R.V}$, we need to make some small adjustments to the way we compute conditioned degree sequences in Sec. 5.2.2, as follows. The max-degree sequence for the non-MCV values, Eq. (5.2), will be computed over CDS rather than DS, in other words we replace it with $\hat{F}_{R.V|A} \stackrel{\text{def}}{=} \max_\ell \hat{F}_{R.V|(A=a_\ell)}$: this, improves the bound. The min-degree sequence for a conjunction of predicates in Eq. (5.3) will also use the CDS, in other words it becomes $\hat{F}_{R.V|(A \wedge B \wedge \cdots)}(i) \stackrel{\text{def}}{=} \min(\hat{F}_{R.V|A}(i), \hat{F}_{R.V|B}(i), \ldots)$: this worsens the bound, but this is necessary to ensure correctness. All other computations remain unchanged, with each f replaced by $\hat{f}$. Next, we discuss how to compute a good valid compression for a given degree sequence.

Example 5.2.4. *In Fig. 5.3, we show how to compress the CDS of the column $R.V$. The degree sequence f on the left has the cumulative degree sequence F on the right. We upper bound the latter by $\hat{F}$, a piecewise linear function with two segments. Notice that the cardinality of the relation is preserved: $|R| = F(6) = \hat{F}(6) = 11$. The degree sequence $\hat{f} \stackrel{def}{=} \Delta\hat{F}$ associated to $\hat{F}$ no longer dominates the original f. Yet, Theorem 3.5.3 proves that we can still use $\hat{f}$ to compute an upper bound on the cardinality of a query.*

We now describe the compression algorithm of a degree sequence f to $\hat{f}$. Function approximations are defined by a model class (e.g. polynomial, sinusoidal, etc), a loss function (e.g. mean squared error), and an approximation algorithm (e.g. gradient descent, convex hull, etc). We have three requirements: (1) the approximation of the CDS must be an upper bound of the original CDS, i.e. $F \le \hat{F}$ (by Thm. 3.5.7), (2) the model class must be closed under both multiplication with the derivative and composition, and (3) the model class must be invertible. The last two requirements are needed by Algorithm 8, which is described in the next section. Given these requirements, SafeBound uses piecewise linear representation for $\hat{F}$, or, equivalently, piecewise constant for $\hat{f}$:

Definition 5.2.5. A function $f(x)$ is *k-piecewise linear* over the domain $(m_0, m_k]$ if there exist tuples $\{(m_0, m_1, a_1 x + b_1), \ldots,$

$(m_{k-1}, m_k, a_k x + b_k)\}$ such that $f(x) = a_i x + b_i \ \forall x \in (m_{i-1}, m_i]$
$\forall 1 \le i \le k$. We call the values, $m_0, \ldots, m_k$, *dividers* and call each interval, $(m_{\ell-1}, m_\ell]$, a *segment*. When $a_1 = \cdots = a_k = 0$, we say that f is *piecewise constant*.

Every piecewise linear function with k segments can be stored in $O(k)$ space. If $\hat{f}$ is piecewise constant, then $\hat{F}(i) \stackrel{\text{def}}{=} \sum_{j \le i} \hat{f}(j)$ is piecewise linear, and, conversely, if $\hat{F}$ is piecewise linear and continuous, then $\hat{f} \stackrel{\text{def}}{=} \Delta\hat{F}$ is piecewise constant. SafeBound compresses degree sequences as piecewise constant, or, equivalently compresses cumulative degree sequences as piecewise linear functions; the conversion from one to the other is done in time $O(k)$.

Degree sequences naturally compress very well. If $R.V$ is a key, then its degree sequence $f(1) = f(2) = \cdots = f(N) = 1$ compresses losslessly to a single segment, $k = 1$. Even if $R.V$ is not a key, its degree sequence can still be compressed losslessly:

Lemma 5.2.6. *Let f be the degree sequence of a column $R.V$, and suppose R has N tuples. Then f compresses losslessly to a piecewise constant function with $k \le \min(\sqrt{2N}, f(1))$ segments.*

Proof. We have $f(1) \ge f(2) \ge \cdots \ge f(d)$, where d is the number of distinct values in $R.V$. Assume w.l.o.g. that $f(d) > 0$ (otherwise, decrease d). Consider its natural dividers into k segments, $0 = i_0 < i_1 < \ldots < i_k \stackrel{\text{def}}{=} d$ such that $f(j)$ is constant for $i_{\ell-1} < j \le i_\ell$ and $f(i_1) > \cdots > f(i_k) > 0$. Since f takes integer values, it follows that $f(1) = f(i_1) \ge k$. On the other hand, $|R| = \sum_i f(i) \ge f(i_1) + \cdots + f(i_k) \ge k + (k-1) + \cdots + 1 + 0 = k(k+1)/2 \ge k^2/2$, which implies $k \le \sqrt{2N}$. This proves $k \le \min(\sqrt{2N}, f(1))$. □

SafeBound does not rely on the natural compression, but instead uses a more aggressive, lossy compression, with a much smaller number of segments than given by the lemma. Algorithm 7, called **ValidCompress**, takes as input a degree sequence $f_{R.V}$, and an accuracy parameter $c > 0$, and computes a valid compression using the following heuristic: if $SJ = \sum_{i=1}^{|\mathbb{D}_V|} f_{R.V}(i)^2$ is the exact Degree Sequence Bound of the self-join on the column $R.V$, then the algorithm ensures that no segment increases the DSB by more than $c \cdot SJ$.

We describe the algorithm and prove its correctness. It iterates through the degree sequence $f_{R.V}(i)$, $i = 1, 2, 3, \ldots$ and builds the segments of $\hat{F}_{R.V}$ one by one: $(m_0 \stackrel{\text{def}}{=}$

Algorithm 7 ValidCompress

Require: $f_{R.V}, c$ // $f_{R.V}$ = exact DS, c = accuracy parameter

1: $d = |\mathbb{D}_{R.V}|$ // the number of distinct values in $R.V$

2: $SJ = \sum_{i=1}^{d} (f_{R.V}(i)^2)$ // exact DSB of selfjoin

3: // initialize 1st segment $(m_0, m_1]$:

4: $k = 1; \epsilon_1 = 0; m_0 = m_1 = 0;\ a_1 = f_{R.V}(1); b_1 = 0$

5: **for** $i \in [1, \ldots, d]$ **do**

6: $\quad \epsilon_k = \epsilon_k + a_k^2 \cdot (f_{R.V}(i)/a_k) - f_{R.V}(i)^2$

7: $\quad$ **if** $\epsilon_k \geq c \cdot SJ$ **then**

8: $\quad\quad$ // DSB error too big?

9: $\quad\quad k = k + 1; \epsilon_k = 0$ // start new segment $(m_{k-1}, m_k]$

10: $\quad\quad m_k = m_{k-1}; a_k = f_{R.V}(i); b_k = b_{k-1} + a_{k-1}(m_{k-1} - m_{k-2})$

11: $\quad$ **end if**

12: $\quad m_k = m_k + f_{R.V}(i)/a_k$ // extend segment $(m_{k-1}, m_k]$

13: **end for**

14: $\hat{F}_{R.V} = \{(m_{l-1}, m_l, a_l(x - m_{l-1}) + b_l) \,|l = 1, k\} \cup \{(m_k, d, |R|)\}$

15: **return** $\hat{f}_{R.V} \stackrel{\text{def}}{=} \Delta \hat{F}_{R.V}$ // $k + 1$ segments

$0, m_1], (m_1, m_2],$

$\ldots, (m_{k-1}, m_k]$. Initially, $k = 1$, the first segment is empty $(0, 0]$, and the initial slope is $a_1 = f_{R.V}(1)$. The `for`-loop in lines 5-13 iterates over each rank $f_{R.V}(i)$ and does one of two things: it either extends the current segment $(m_{k-1}, m_k]$ by increasing m_k (line 12), or it increases k and starts a new empty segment (line 9), which is also immediately extended in line 12. The choice between these actions is dictated by our heuristics: ensure that each segment contributes at most $c \cdot SJ$ to the DSB. We prove that, regardless of the heuristic, the algorithm always computes a valid compression, by checking conditions (a), (b), (c) in Def. 5.2.3. The following invariant holds at the beginning of each iteration of the `for`-loop (line 5): if $\hat{F}_{R.V}$ denotes the current piecewise linear function, defined on $(0, m_k]$, then:

$$\hat{F}_{R.V}(m_k) = F_{R.V}(i)$$

This follows by induction on i. Before the first iteration, $i = 0$, $m_1 = 0$ and $\hat{F}_{R.V}(0) = F_{R.V}(0) = 0$. Consider the inductive step, from $i-1$ to i. On one had, the value of $F_{R.V}(i)$ grows by $f_{R.V}(i)$; on the other hand, m_k increases in line 12 and its current slope is a_k, hence the value $\hat{F}_{R.V}(m_k)$ will grow by exactly $a_k \cdot (f_{R.V}(i)/a_k) = f_{R.V}(i)$, proving the invariant. In particular, this implies that $\hat{F}_{R.V}(m_k)$ is always $\leq F_{R.V}(d) = |R|$, where $d = |\mathbb{D}_V|$: it justifies adding the a constant segment $(m_k, d, |R|)$ (line 14), and proves condition (c): cardinality is preserved. Since the slopes a_k defined in Line 10 are decreasing, condition (a) holds: $\Delta\hat{F}_{R.V}$ is decreasing. Finally, condition (b), $\hat{F}_{R.V}(i) \geq F_{R.V}(i)$, follows from the fact that during the `for`-loop $m_k \leq i$, since i always grows by 1, while m_k grows by $f_{R.V}(i)/a_k \leq 1$. Using the invariant and the fact that $\hat{F}_{R.V}$ is monotonically increasing, we obtain $\hat{F}_{R.V}(i) \geq \hat{F}_{R.V}(m_k) = F_{R.V}(i)$, proving (b). In summary:

Theorem 5.2.7. *Algorithm 7 computes a valid compression of $\hat{f}_{R.V}$ of $f_{R.V}$, with $k+1$ segments and a relative self-join error $\leq c \cdot k$.*

Our algorithm is loosely inspired by approximate convex hull algorithms such as the one used in [53], and it is similarly linear in time and space with respect to the degree sequence length. Further, calculating a DS from a column of length n requires $O(n \log(n))$ time and $O(n)$ space. In our implementation, we typically choose $c = .01$ which results in $k = 20 - 30$ segments for compressing the DS of a foreign key and < 10 segments for the DS conditioned on an element of the MCV list. Further, if the join column is a key, then it always compresses to a single segment.

Discussion We briefly justify our choice of heuristics over other possible choices. One choice would be to minimize the absolute distance between the true CDS and the approximation,
$\sum_{i=1}^{|\mathbb{D}_V|} |F_{R.V}(i) - \hat{F}_{R.V}(i)|$. However, this distance would treat errors on high frequency and low frequency values as equally undesirable when the high frequency values actually have a much larger impact on the final bound. This is due to high frequency values joining with high frequency values in the worst-case instance. Alternatively, one could choose some specific weighted distance to use for modeling all columns, $\sum_{i=1}^{|\mathbb{D}_V|} w_i |F_{R.V}(i) - \hat{F}_{R.V}(i)|$. However, because that optimal weighting will depend on the adjoining tables, choosing a

single weighting for all columns assumes that they will all have similarly distributed adjoining tables. For instance, this would imply that a column containing country IDs will join with the same columns as one that contains employee IDs. Our choice of the self-join error metric amounts to assuming that tables will join with similarly skewed tables. Future work may consider the skewness of adjoining tables in the database schema or a sample workload to create a more accurate metric.

5.2.4 Fast Computation of the Upper Bound

We finally turn to the online phase of SafeBound: given a query and the collection of compressed degree sequences, use the statistics $\hat{s}$ to compute the upper bound $|Q(W(\hat{s}))|$. Throughout this section, we assume that $\hat{s}$ are valid compressions (see Def. 5.2.3) and represented by piecewise constant functions; equivalently, their CDS $\hat{S}$ are piecewise linear functions. We assume that all predicates have been applied to the base tables, and $\hat{s}$ includes all conditional degree sequences needed for the predicates, as discussed in Sec. 5.2.2; in other words, we will assume w.l.o.g. that Q consists only of joins, and no predicates. Referring to the running Example 5.2.1, we assume that the query is $R'(X) \wedge S'(X,Y) \wedge T(Y)$, where the degree sequence of $R'.X$ is $\min(\hat{F}_{R.X|(A<5)}(i), \hat{F}_{R.X|(B=2)}(i))$ (see the discussion at the end of Sec. 5.2.3) and the DS for $S'.X$ and $S'.Y$ are given by conditioning on the predicate `LIKE` $'\%Abdul\%'$.

The naive computation requires materializing the worst case instance, W, and is totally impractical, since W is at least as large as the database instance, regardless of how well we compress the statistics $\hat{s}$. Instead, SafeBound implements Alg. 3, which avoids materializing W, but instead computes the bound directly, in time that depends only on the total size of all compressed degree sequences.

The starting observation is that Q is acyclic, and can be computed bottom-up, where at each tree node we join the current relation with its children and project out all attributes except the unique attribute needed by its parent. We write this plan as an alternation

between two kinds of operations, which we call α and β steps:

$$\alpha : \qquad A(X) = B_1(X) \wedge \ldots \wedge B_m(X) \tag{5.4}$$

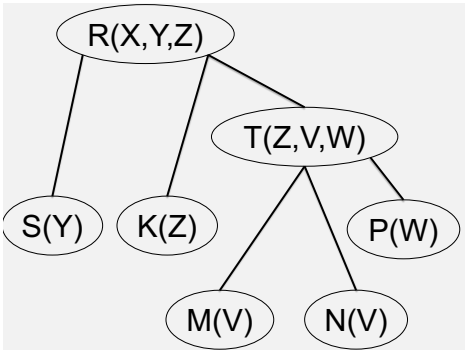
$$\beta : \qquad B(X_0) = R(X_0, X_1, \ldots, X_k) \wedge A_1(X_1) \wedge \ldots \wedge A_k(X_k) \tag{5.5}$$

An α-step intersects unary relations, while a β-step is a star-join followed by a projection on a single variable. Recall that all our queries have bag semantics, so this projection does not reduce the cardinality. The cardinality of Q is the cardinality of the last unary relation, corresponding to the root of the tree.

Example 5.2.8. *We briefly illustrate the α, β steps for the query Q:*

$$R(X,Y,Z) \wedge S(Y) \wedge K(Z) \wedge T(Z,V,W) \wedge M(V) \wedge N(V) \wedge P(W)$$

We only show the attributes used in joins: e.g. relation S may have attributes $S(Y, A, B, C, D, \ldots)$ but we only show the join attribute Y. Fig. 5.4 shows a tree decomposition for Q, and a plan consisting of two α and two β steps. The cardinality of the original query, Q, is the cardinality of the last unary relation, B'.



$$A(V) = M(V) \wedge N(V)$$
$$B(Z) = T(Z,V,W) \wedge A(V) \wedge P(W)$$
$$A'(Z) = K(Z) \wedge B(Z)$$
$$B'(X) = R(X,Y,Z) \wedge S(Y) \wedge A'(Z)$$

Figure 5.4: A tree decomposition for Q in Example 5.2.8

Algorithm 8 evaluates Q on the worst case instance $W(\Delta\hat{S})$ without materializing W. Since the algorithm uses piecewise linear CDS, the computation may lead to rounding errors causing a slight over-approximation of the uncompressed Degree Sequence Bound. For that reason, the bound computed by the algorithm is called the *Functional Degree Sequence*

Bound, FDSB. The key observation in the algorithm is that every unary relation $A(X)$ or $B(X_0)$ is also piecewise constant; for that reason we call them in the algorithm $\hat{f}_{A.X_0}(i)$ and $\hat{f}_{B.X}(i)$ respectively. To see this, consider an α-step: $A(X) = B_1(X) \wedge B_2(X) \wedge \cdots$ If each B_ℓ contains the values $1, 2, 3, \ldots$, then so will A, and the multiplicity of the value i is the product of the multiplicities in the B_ℓ's: this justifies line 4: $\hat{f}_{A.X}(i) = \hat{f}_{B_1.X}(i) * \hat{f}_{B_2.X}(i) * \cdots$ The product of piecewise constant functions is still piecewise constant, with a number of segments equal to the sum of all segments of the summands. Consider now a β-step, Eq. (5.5). The multiplicity of i in the output B is the product $\hat{f}_{R.X_0}(i) * \hat{f}_{A_1.X_1}(i_1) * \hat{f}_{A_2.X_2}(i_2) * \cdots$, where the ranks $i_1, i_2, \ldots$ need to be looked up using the expression $i_\ell = \hat{F}^{-1}_{R.X_\ell}(\hat{F}_{R.X_0}(i))$; this justifies line 9. Here, too, we observe that i_ℓ is given by a piecewise linear function in i, while $\hat{f}_{A_\ell.X_\ell}(i_\ell)$ is piecewise constant. In our implementation, both lines 4 and 9 of the algorithm compute a *representation* of the resulting piecewise constant functions. This representation consists of three vectors; slopes, intercepts, and right interval edges. Because of this representation, multiplication of functions only requires a single pass over these arrays while computing the inverse term, $\hat{F}^{-1}_{R.X_\ell}(\hat{F}_{R.X_0}(i))$, involves performing a binary search for each of the interior function's segments. Finally, the algorithm returns the cardinality of the last unary relation at the root. The following theorem from Ch. 3 also applies to our setting:

Theorem 5.2.9. *Let $\hat{S}$ consists of piecewise linear CDS, and let K be the total number of segments occurring in all CDS. Then,*

1. $|Q(W(\Delta\hat{S}))| \leq FDSB(Q, \hat{S})$.

2. *$FDSB(Q, \hat{S})$ can be computed in time $O(K \log(K))$.*

The first item asserts that FDSB is a correct upper bound. The second item shows that we can compute it in log-linear time relative to the size of the compressed representations. We usually have $10 - 30$ segments per degree sequence, thus a query with 5 joins has at most $K = 300$. This results in very fast inference as shown in Section 5.4.

Algorithm 8 Algorithm for FDSB$(Q, \hat{S})$.

Require: A query plan for Q consisting of α, β steps.

for each step **do**
  if α-step **then**
    // $A(X) = B_1(X) \wedge \ldots \wedge B_m(X)$
    $\hat{f}_{A.X}(i) = \prod_{\ell=1}^{m} \hat{f}_{B_\ell.X}(i)$
    // Note: $\hat{f}_{A.X}$ is a piecewise constant function
  end if
  if β-step **then**
    // $B(X_0) = R(X_0, X_1, \ldots, X_k) \wedge A_1(X_1) \wedge \ldots \wedge A_k(X_k)$
    $\hat{f}_{B.X_0}(i) = \hat{f}_{R.X_0}(i) \prod_{\ell=1}^{k} \hat{f}_{A_\ell.X_\ell}\left(\hat{F}^{-1}_{R.X_\ell}(\hat{F}_{R.X_0}(i))\right)$
    // Note: $\hat{f}_{B.X_0}$ is a piecewise constant function
  end if
end for
return $\sum_i \hat{f}_{B_{\text{root}}.X}(i)$

5.2.5 Discussion: More Complex Join Graphs

Cyclic Queries: If Q is a cyclic query, then we compute its upper bound as the minimum of the DSBs of all its spanning trees (which are acyclic). For example, if Q is $R(X,Y)S(Y,Z)T(Z,X)$, there are three spanning trees $R(X,Y)S(Y,Z)$, and $R(X,Y)T(Z,X)$, and $S(Y,Z)T(Z,X)$, and we return the minimum of their DSB.

Multi-Column Joins: We provide two methods of handling multi-column joins. (1) Treat multiple columns as a single column at construction time and calculate its CDS just like for a single column. (2) Alternatively, we note that the CDS for any single column is an upper bound on the CDS of a set of columns. Therefore, we can always upper bound the CDS of the set of columns by taking the minimum of the CDS of each of its columns.

Undeclared Join Columns: We provide a fallback mechanism to handle queries with joins on columns that are not in the declared join column set. During offline construction, we keep a compressed CDS for every column in the relation without conditioning on any predicates.

This incurs minimal overhead as it only calculates one CDS per column. At query-time, we calculate a CDS for any declared join column and use it to derive an upper bound on the filtered relation's cardinality. We then truncate the undeclared join column's CDS to match this single-table cardinality bound and use this to approximate the filtered CDS of the undeclared join column. This allows us to adapt the bound to the lower cardinality induced by any predicates on the relation. However, the resulting CDS is likely overly skewed because it assumes that the predicate retains precisely the high degree values.

5.3 Optimizations

Here, we present optimizations that reduce the space consumption and increase the accuracy of the vanilla design.

5.3.1 Compressing a Set of CDS

To reduce the memory overhead of storing a CDS set for every bin, value, and N-gram, we go beyond compressing a single CDS and consider compressing groups of CDS sets together. Specifically, we divide the CDS sets into groups of "similar" functions and replace each group with the point-wise maximum of those sets.

To motivate this, we first present a breakdown of the memory cost of SafeBound's statistics. Let b be the granularity, e.g. the number of buckets in the histogram, and let k be the number of segments in each CDS. Further, let $|V_J|$ and $|V_F|$ be the number of join and filter columns, respectively. The memory footprint without performing any grouping compression is $O(b|V_F| + bk|V_F||V_j|)$ where the first term is the bucket bounds or values in the MCV list while the second term is the cost of storing the CDS sets. Now, consider dividing the CDS sets into M groups and storing just the maximum over each group. The memory footprint then becomes $O(b|V_F| + Mk|V_F||V_j|)$ which allows us to decouple the granularity of our statistics, b, from the accuracy of our approximations, Mk. This is crucial for workloads which feature highly selective predicates because it allow us to keep more fine-grained histogram buckets, MCV lists, and N-grams.

As an example, consider a range predicate $.1 \leq R.A < .2$. We may only have the memory to store buckets of width 1 if we store every bucket's CDS exactly. However, if we cluster

and compress our CDS sets with an average cluster size of 10, we may able to have buckets of width .1 which encapsulate the query much tighter while only incurring a 40% relative approximation error. The relative approximation error in this case is far outweighed by the improved granularity of our statistics.

Choosing a Distance Metric: The first step to clustering is choosing a distance metric for the problem. The perfect distance metric for this problem is the average error incurred on the workload when the two functions are replaced with their maximum. However, we don't have access to the workload when clustering, so we instead use the same assumption that we used in Sec. 5.2.3, that the workload consists of self-joins. Therefore, our distance metric becomes the self-join error, i.e.

$$d(F_1, F_2) = \frac{\sum_{i=1}^{|\mathbb{D}_V|} \Delta \max(F_1(i), F_2(i))^2}{\sum_{i=1}^{|\mathbb{D}_V|} f_1(i)^2} + \frac{\sum_{i=1}^{|\mathbb{D}_V|} \Delta \max(F_1(i), F_2(i)^2}{\sum_{i=1}^{|\mathbb{D}_V|} f_2(i)^2}$$

.

Choosing a Clustering Algorithm: Given this distance metric, we need to choose a clustering algorithm, and we choose *complete-linkage clustering* [107]. This method of hierarchical clustering defines the distance between clusters as the maximum distance between points in each cluster. As opposed to other clustering methods such as single-linkage clustering, it produces tighter clusters and avoids long "chain" clusters which contain highly dissimilar points. This results in clusters of functions which are well approximated by their point-wise maximums.

5.3.2 Pre-Computing Primary Key Joins

Predicates Induce Cross-Join Correlation: As described in Section 5.1.2, the FDSB makes worst-case assumptions about the correlation of columns in joining tables. This assumption is fundamental to computing an upper bound. However, particularly in the presence of predicates, these assumptions may not hold, causing SafeBound to overestimate the query size.

For example, consider the tables `MovieKeywords` and `Keywords` from the JOB Benchmark. The former is a fact table with two foreign key columns, `MovieId` and `KeywordId`, that associate movies with keywords. The latter is a much smaller dimension table with

a primary key column `KeywordId` and a filter column `Keyword`, which provides human-readable descriptions of these keywords, e.g. 'character-name-in-title' or 'pg-13'. A natural query would join them with an equality predicate on the `Keyword` column to find movies with a particular keyword. A naive version of SafeBound would assume that the selected keyword corresponds to the most frequent value of `KeywordId` in the `MovieKeywords` table. If the queried keyword actually occurs infrequently in `MovieKeywords`, this could introduce a massive error in the final estimation.

Handling Predicate-Induced Correlation: To avoid this issue, SafeBound precomputes PK-FK joins and stores statistics about the filter columns of the PK relations. In our example, this would mean joining `MovieKeywords` and `Keywords` then generating statistics on the resulting `keyword` column in `MovieKeywords`. When an equality predicate is applied to the `keyword` column on the `Keywords` table, SafeBound applies this predicate to the `MovieKeywords` table as well, allowing it to directly estimate the CDS set given the predicate without resorting to worst-case assumptions.

Fortunately, the PK-FK join size is bounded by the size of the FK table, so this precomputation is tractable. While this does not capture all correlations, it does enable accurate estimation for the ubiquitous fact/dimension table design where predicates are applied to dimension tables then propagated to fact tables via PK-FK joins.

5.3.3 Bloom Filters

An important source of overhead in SafeBound's data structures are the most common values lists (MCV lists) that it keeps for handling equality predicates. Because values can have an unbounded size, storing a naive MCV list can result in significant memory and lookup overhead. To avoid this overhead, we instead represent our MCV lists as a set of *Bloom filters*. A Bloom filter is an approximate data structure, which answers the question "is x an element of the set S?" while allowing some false positives and no false negatives. In exchange for approximation, Bloom filters provide a compressed memory footprint ($\approx$ 12

bits/value) and fast, constant lookup[1].

Because Bloom filters only return a positive/negative and SafeBound needs to connect values to their CDS group, we can't represent the whole MCV list in one filter. Instead, we allocate a filter for each CDS group and insert all values whose CDSs are in that group into its filter. At query time, SafeBound then checks for membership in every group's filter and takes the maximum over all CDS sets whose filter return positive.

5.4 Evaluation

In this section we present an empirical evaluation of SafeBound. We addressed the following questions. How well does SafeBound perform in end-to-end workloads (Sec. 5.4.1)? How does its memory footprint and inference time compare to existing methods (Sec. 5.4.2)? How does SafeBound affect DBMS robustness, e.g. performance regressions when new indices are added (Sec. 5.4.3)? We also conducted several micro-benchmarks in Sec. 5.4.4, and explored how SafeBound scaled in Sec. 5.4.5.

Metrics We used the following metrics in our evaluation. (1) Plan Quality (Workload Runtime): Following recent work on benchmarking cardinality estimators [72] we measure the end-to-end runtime of a query workload in Postgres where we injected alternate cardinality estimators into the optimizer. We run each workload and method five times from a cold cache and present the average relative to the baseline of inserting the true cardinality estimates. (2) Memory Footprint: We compare the size of the stored statistics file on disk, and for Postgres we calculate the size of the `pg_statistic` and `pg_statistic_extended` catalog tables. We do not report memory statistics for PessEst as it does not pre-compute statistics. (3) Planning Time: We further consider the planning time for each method. This includes the inference time required to get estimates for every sub-query as well as Postgres' optimization time given injected estimates. (4) Relative Error: Lastly, we present the relative error of each method as Error = (Estimate/True Cardinality). We prefer this metric to q-error as it retains information about whether a method overestimates or underestimates.

[1]There are many variants on Bloom filters which would work equally well here, e.g. Cuckoo filters, quotient filters, and XOR filters.

Datasets We use two datasets, *IMDB* and *Stats*. For *IMDB* we consider three different query workloads from previous work [155, 94, 77][2]: *JOB-Light* consists of 70 queries on a subset of 6 tables in IMDB with $2-5$ PK-FK joins and $1-4$ filter predicates on numeric columns. *JOB-LightRanges* operates on the same table subset as JOB-Light, but it has 1000 queries, includes additional columns, and predicates over string columns. And *JOB-M* is a modified version of the original JOB benchmark; it is the most complex benchmark considered, with 113 queries over 16 tables, and includes significantly more complicated expressions such as `IN` and `LIKE` predicates. The *Stats* dataset is built over a Statistics StackOverflow, and consists of a workload with 146 queries spanning 8 tables. While restricted to numeric columns, it has $2-16$ predicates and joins $2-8$ tables per query making it is considered to be a challenging benchmark for cardinality estimation [72]. It has a complicated schema with cyclic primary key/foreign key relationships.

Compared Systems We compared SafeBound against the following systems. (1) Postgres: As a baseline, we compare against the built-in cardinality estimator for Postgres v13. This system uses System-R style estimation combined with years of tuning and carefully chosen magic constants. It stores 1D histograms, most common value lists, and distinct counts for each attribute in a relation. (2) Postgres 2D: We make use of Postgres' extended statistics, which allows the user to keep statistics on pairs of columns. We instruct the system to store statistics for every pair of filter columns. (3) Postgres PK: SafeBound precomputes statistics on key, foreign-key joins, and so do BayesCard, and NeuroCard; PessEst computes PK-FK joins at query time when needed [30, Sec.3.3]. To understand the effect of these computations, we measured how much such precomputations could help Postgres. We pre-computed and materialized the PK-FK joins, replaced the FK tables with this join (extending them with additional columns from the PK tables), and computed statistics on these tables. We also adjusted the queries accordingly. For example, consider the query $Q(X, A, B) = R(X, A)S(\underline{A}, B, Y)T(\underline{B}, Z) \wedge S.Y < 10 \wedge T.Z > 5$ where $S.A$, $T.B$ are PKs. We calculate the PK-FK join results $R'(X, A, Y')$ and $S'(A, B, Y, Z')$ and adjust the query

[2]We do not use the original JOB benchmark because it contains negation predicates which are not supported by SafeBound, NeuroCard, and BayesCard.

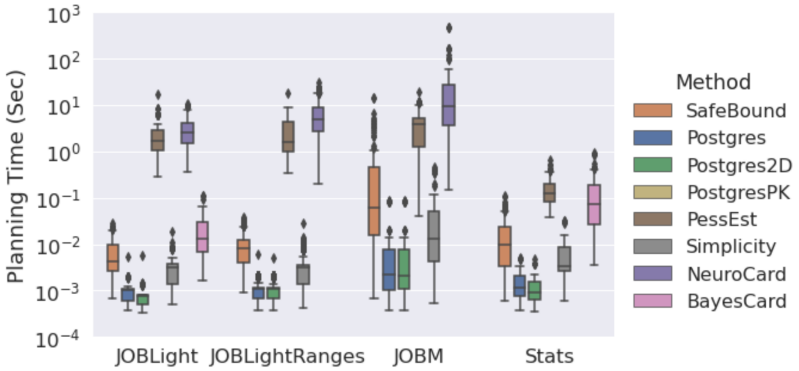
to $Q'(X, A, B) = R'(X, A, Y')S'(\underline{A}, B, Y, Z')T(\underline{B}, Z) \wedge R'.Y' < 10 \wedge S'.Z' > 5$. We call this modifed system PostgresPK. Notice that this mirrors our method by propagating statistics across PK-FK joins, without modifying the query's join graph.

(4) BayesCard: This is an ML method that uses ensembles of Bayesian Networks trained on subsets of the join schema to produce cardinality estimates [154]. Recent work has shown that it matches previous ML methods in accuracy while being faster and more compact [72]. (5) NeuroCard: this is an ML method that builds an autoregressive model over a sample of the full outer join of the schema [156]. (6) PessEst: The main prior work on cardinality bounding [30]. It refines a subset of the Polymatroid Bound using a hash partitioning scheme; we use 4096 hash partitions. However, this method requires scans of the base table to estimate queries with predicates. (7) Simplicity: a cardinality estimator which uses single-table cardinalities and max degrees of join columns [76]. In order to improve the max degree in the presence of predicates, Simplicity relies on samples [76] or on estimates derived from Postgres, which are no longer leading to *guaranteed* upper bounds. In the original prototype and our implementation, the single-table estimates are derived from Postgres although more complicated sampling mechanisms are proposed in the paper. Similarly, we do not consider their greedy join ordering algorithm, instead focusing solely on the cardinality estimator.

Experimental Setup In our experiments, we use an instance of Postgres v13 and input cardinality estimates using the `pg_hint_plan` extension. We adjust the default settings of Postgres per the recommendation of [100] setting the shared memory to 4GB, worker memory to 2GB, implicit OS cache size to 32 GB, and max parallel workers to 6. Additionally, we enable indices on primary and foreign keys. We run all experiments on an AWS EC2 instance (m5.8xlarge) with 32 vCPUs and 128 GB of memory.

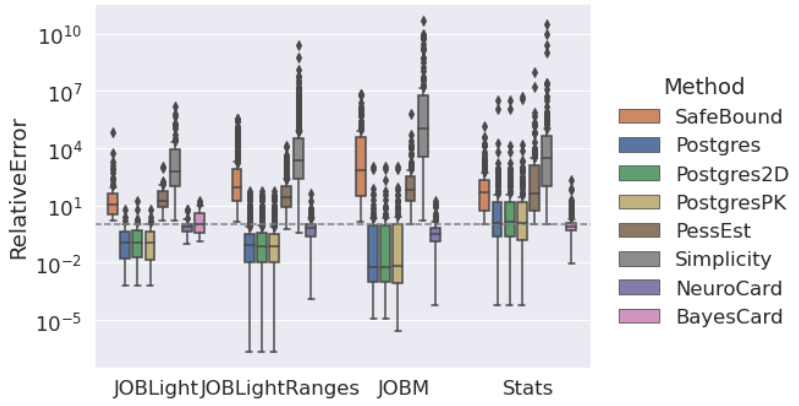
5.4.1 End-to-End Performance

Nearly optimal workload runtimes across all benchmarks. We show the workload runtimes across a variety of benchmarks and cardinality estimation methods in Figure 5.5b. Across all four benchmarks, plans generated using SafeBound's estimates achieve workload runtimes equivalent to those generated with the true cardinalities. As found in previous



(a) SafeBound achieves **3x-500x** faster median planning time than PessEst and ML-based methods across all benchmarks.

(b) Total workload runtimes relative to runtimes achieved with perfect cardinality estimates. SafeBound results in nearly optimal overall runtimes.



(c) Relative errors for cardinality estimates. SafeBound's bounds have similar errors to traditional estimates while never underestimating cardinalities.

Figure 5.5: Workload runtimes, planning time, and estimation error

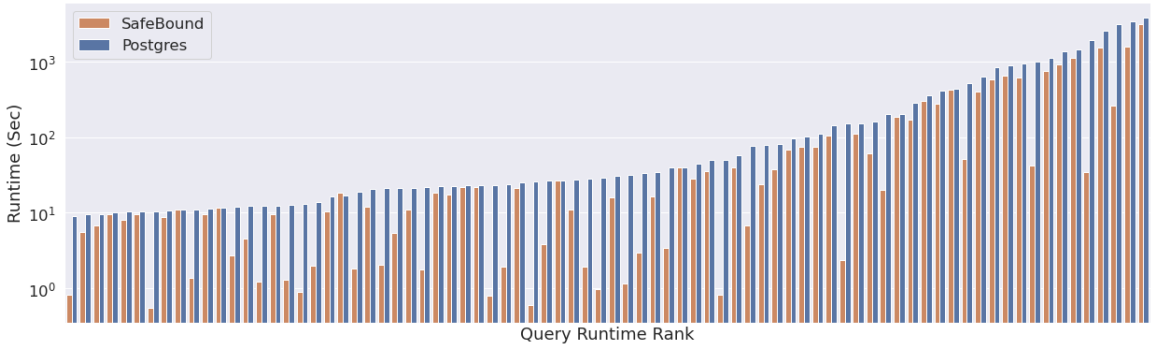


Figure 5.6: The runtime of the 80 longest-running queries across all benchmarks.

work, using true cardinality does not always lead to optimal plans due to imperfect cost modeling [72]. SafeBound achieves $20\% - 85\%$ lower runtimes than Postgres on all benchmarks. BayesCard and PessEst perform similarly to SafeBound on all benchmarks while NeuroCard has $20 - 30\%$ worse performance on both JOBLight and JOBM. Bayescard does not support the string predicates of JOB-LightRanges or JOB-M, and NeuroCard does not support the cyclic schema of the Stats benchmark. All pessimistic systems achieve good performance on the JOB benchmarks, pointing to the utility of even fairly loose cardinality bounds. However, Simplicity results in a poor join ordering for query 132 of the Stats benchmark, resulting in a 1500x slowdown.

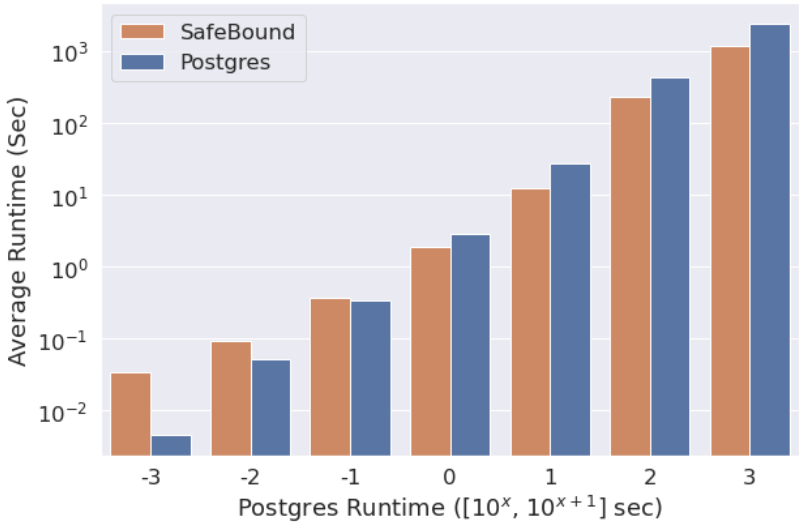
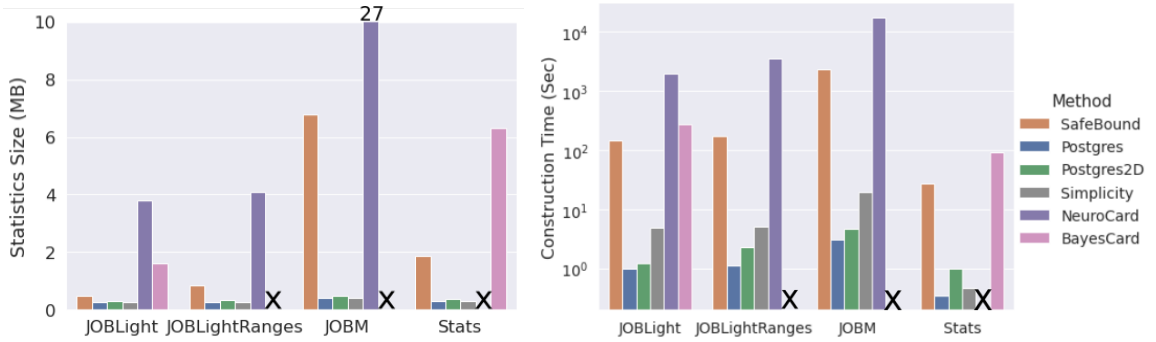


Figure 5.7: The average runtime of queries binned by their runtime using Postgres' estimates.

Efficient plans for the queries that matter. To provide context for SafeBound's performance, we examine the runtime of the longest running queries in Figure 5.6. The queries that make up the bulk of the runtime can often be sped up significantly (up to $60x$) by using SafeBound instead of Postgres for cardinality estimates.

Figure 5.7 buckets the queries across all workloads by runtime and shows the average runtime using SafeBound and Postgres's estimates. Here, we can see SafeBound generally achieves a significant speedup for queries that take over a second. We see these speedups because cardinality bounds encourage the query optimizer to make conservative decisions (e.g. choosing hash joins over nested loop joins) which tend to be the correct decisions for queries with long runtimes or large outputs. For the fastest queries, SafeBound often results in slower execution as it discourages the optimizer from choosing high-risk high-reward plans.

Estimation Errors. In Figure 5.5c, we show the relative estimation error on full queries for each of the benchmarks and methods. SafeBound has a similar range of errors as Postgres, but guarantees that it never underestimates: its estimates in the figure always lie above the



(a) Statistics size comparison.

(b) Construction time comparison.

Figure 5.8: Statistics size and Construction Time

center line. Traditional estimators frequently underestimate by 10^3 or more which is detrimental to query optimization. Notably, additional optimisations such as Postgres2D and PostgresPK do not significantly alter the estimates. This implies that the errors primarily stem from the fundamental independence assumptions rather than a lack of statistics. ML-based methods produce accurate estimates, but still lack guarantees; NeuroCard is prone to significant underestimates. The Simplicity system overestimates significantly due to its reliance on the max degree without conditioning on predicates. Moreover, as discussed, its "upper bound" is not guaranteed: it returns a wrong upper bound on two of the queries of JOBLightRanges. Because it handles predicates by scanning the table at estimation time, PessEst has good estimates for queries with many predicates or with challenging predicates such as JOB-LightRanges and JOB-M. We provide a more detailed breakdown of estimation error by number of tables in the appendix.

5.4.2 Planning Time, Memory, and Build Time

Figure 5.5a reports the planning times (Postgres' optimization and the cost estimation time for all sub-queries) for all systems and benchmarks. Postgres' efficient C implementation and use of dynamic programming in estimating sub-queries results in the fastest planning time. The Simplicity system achieves good planning times thanks to its straightforward

bound calculation and reliance on Postgres's fast single-table estimates. PessEst requires scanning the base tables at runtime when predicates are applied, resulting in 12x-420x slower inference. The ML methods, particularly Neurocard, perform inference on complex black-box models which leads to poor inference latency. SafeBound implements Algorithm 13, which runs in log-linear time in the size of the compressed degree sequences. This results in much faster planning times than PessEst, BayesCard, and NeuroCard across all benchmarks.

Next, we turn to the memory footprint, which we report in Figure 5.8a. Safebound's simple statistics and compression techniques allow it to achieve a compact memory footprint close to traditional methods like Postgres. For instance, group compression results in 7.3-43x fewer degree sequences being stored across benchmarks. This results in statistics that are only 200KB larger than Postgres' for the JOB-Light benchmark and over 3x smaller for all benchmarks than BayesCard and NeuroCard which rely on complex black box models. Simplicity relies on the statistics stored in Postgres and the max degree of each join column, resulting in a small memory footprint. PessEst does not operate on pre-computed statistics, so its space and build time is not reported.

Finally, we considered the build time. SafeBound has 2-3.5x and 8-20x faster build time than BayesCard and NeuroCard, respectively. We note that SafeBound's build process (Algorithm 7) is a series of aggregations over the base data to create histograms and MCV lists, resulting in relatively fast construction despite building from the full dataset rather than a sample. Traditional estimators remain the fastest as they perform similar aggregations to SafeBound but on a small sample of the data. The string predicates of JOB-M result in longer build times for SafeBound and NeuroCard because they require the calculation of tri-grams and factorized columns, respectively. Postgres(2D), on the other hand, does not keep statistics for LIKE predicates (instead handling them via a magic constant), so the additional complexity does not affect their time.

5.4.3 Robustness Against Regressions

When attempting to tune database instances, users often face inexplicable performance regressions. Creating an index on a join column is reasonably assumed by users to improve

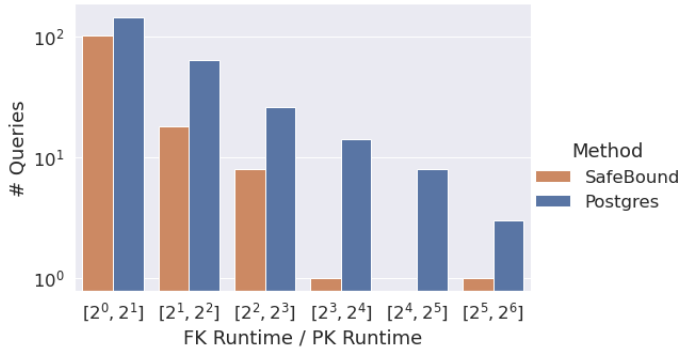


Figure 5.9: The frequency and magnitude of FK index performance regressions.

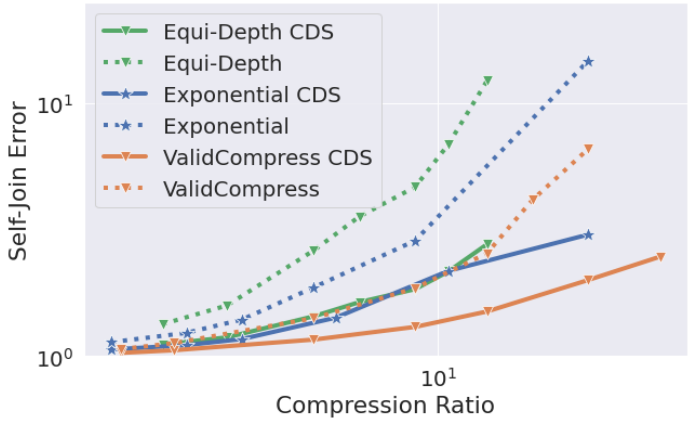


Figure 5.10: CDS modeling micro-benchmark.

query performance, but will frequently cause queries to run significantly slower. This is primarily due to the query optimizer receiving cardinality underestimates and optimistically making use of the index for a query where a hash or merge join is faster. In figure 5.9, we show the frequency and severity of regressions across all benchmarks when Postgres' internal estimates are used vs SafeBounds cardinality bounds. While some regressions still occur due to issues in cost modeling, SafeBound produces half as many performance regressions across all benchmarks, 129 to Postgres' 259, and they are half as severe on average, 1.7x to Postgres' 3.3x.

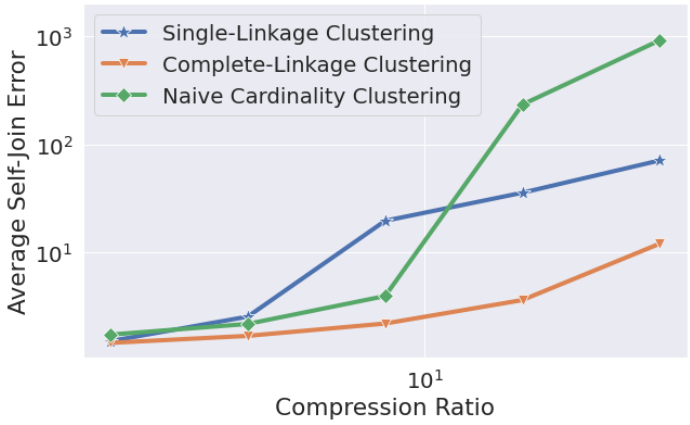


Figure 5.11: CDS clustering microbenchmark.

5.4.4 Micro-Benchmarks

In the following experiments, we evaluate how SafeBound's components perform individually as compared to alternatives. To do this, we calculate the error on self-join queries with equality predicates, specifically the `MovieCompanies` relation joining with itself on `MovieId` with and without an equality predicate on `ProductionYear`.

Modeling the CDS rather than the DS reduces error by up to 20x. By avoiding artificial inflation of the relations' cardinality caused by modeling the DS directly, SafeBound achieves significantly more accurate estimates. This can be seen on Figure 5.10 which shows the accuracy of various approximation methods for modeling the full CDS/DS of `MovieCompanies.MovieId` versus the compression ratio (# distinct frequencies/ # segments). The different colors correspond to different ways of choosing the segment boundaries for the approximation while the solid vs dashed lines correspond to whether the method models the CDS or the DS, respectively. As we would expect, every approximation method has lower error when applied to the CDS rather than the DS.

The ValidApprox algorithm efficiently models the CDS. Looking again at Figure 5.10, we can compare the solid lines to get a sense for how different segmentation strategies affect the accuracy/compression trade-off. We compare against a couple of reasonable baselines, 1) an equi-depth strategy which segments the degree sequence into equal-cardinality

segments 2) an exponential strategy which uses a geometric sequence for segment boundaries. Our two-pass algorithms outperform both baselines because it adjusts the size of buckets based on the skewness of the underlying degree sequence, taking into account both the importance of high frequency items and the long tail of the distribution.

Complete-linkage clustering of CDSs provides low error at high compression ratios. The experiment in Figure 5.11 shows the effect of different clustering techniques. In this experiment, we joined the `MovieCompanies` relation with the `Title` relation according to their FK/PK relationship then calculate an MCV list for the `ProductionYear` attribute as described in Sec. 5.2.2. This results in 132 CDS that we use to test various clustering methods, which cluster them into between 4 and 64 groups, and we represent each cluster with the maximum as in Sec. 5.3.1. The error metric is then the average relative self-join error over all the original CDS when they are approximated using their cluster's maximum. In this case, compression ratio is defined as the number of original groups, 132, divided by the number of clusters after compression.

We compare SafeBound's method, *complete-linkage clustering*, with *single-linkage clustering* and the naive method of grouping the functions into equal sized clusters by cardinality. Across these methods, complete-linkage clustering results in lower error for all compression ratios. This is because naive clustering doesn't take into account the shape of the CDS and doesn't adaptively choose cluster sizes, and single-linkage clustering often produces long chaining clusters where one CDS dominates the maximum.

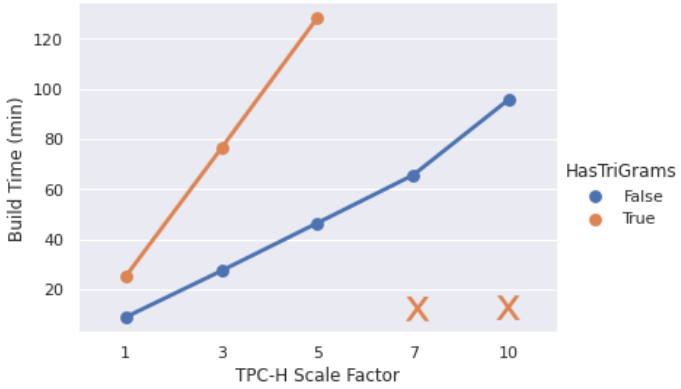


Figure 5.12: Construction scaling experiment.

5.4.5 Scalability

To test how SafeBound scales to both more complex schemas and larger datasets, we experiment with the TPC-H benchmark at various scaling factors in Fig 5.12. [3] This benchmark has 14 join columns, 46 filter columns, and 9 PK-FK relationships over 8 tables, and we vary the scale factor from 1GB to 10GB. We further build two versions of SafeBound: one where it constructs tri-gram statistics for LIKE predicates and one where it does not. This experiment shows that the construction time increases linearly in the size of the data. However, it also points to the inefficiency of the current python implementation in two ways. It runs out of memory when computing tri-grams for higher scale factors (as denoted by the X marks), and the build process is slower than expected given the simple underlying operations. This is in line with recent research which has shown an average of 29x worse performance when using CPython rather than C++ for a variety of applications due to dynamic type checking, interpreter overhead, and the global interpreter lock [102].

[3] We do not include this benchmark in our other experiments as its lack of skew, independence between columns, and independence across tables poorly represents real world distributions as discussed in [100].

5.5 Conclusion & limitations

This chapter presented SafeBound, a first practical system for computing *guaranteed* cardinality upper bounds: previous upper bound systems either require significant query time computation (PessEst), or do not provide *guaranteed* bounds (Simplicity). SafeBound achieves up to 80% lower end-to-end runtimes than PostgreSQL across workloads, and is on par or better than state of the art ML-based estimators and pessimistic cardinality estimators. Yet, the current SafeBound prototype has a few limitations, which we discuss here.

Build Time: Any upper bound system must, at some point, read all rows in the database. This places a hard lower bound on the build time for cardinality bounding systems. Therefore, future work to reduce this expense will need to exploit parallelism or take advantage of times when the system is already scanning the data (e.g. during bulk loading or query execution).

Handling Updates: A challenge that we leave open is handling updates without recomputing the degree sequences. We note here that a degree sequences is essentially a `group-by/count/order-by` query and could benefit from IVM techniques [96].

Multiple Predicates Per Table: SafeBound handles the existence of multiple predicates on a single relation by taking the minimum of their induced CDS'. However, this results in an estimated cardinality equal to the selectivity of the most selective predicate. This could be inaccurate when faced with multiple moderately selective predicates which are jointly highly selective. Future work should consider ways of keeping lightweight statistics on combinations of filter columns in order to tackle this problem.

Chapter 6

COLOR

In this chapter, we propose a new approach to both cardinality bounding and cardinality estimation based on graph compression, called COLOR. By addressing estimation as well, this chapter departs slightly from the previous ones which had focused solely on deterministic upper bounds. However, the majority of the techniques presented here will be applicable to both goals. Further, we focus specifically on *subgraph matching* in this chapter. Subgraph matching is the core operation of queries over graphs where instances of a query graph pattern, Q, are found within a larger data graph, G. Formally, subgraph matching is equivalent to conjunctive queries restricted to binary relations. This is the main primitive in graph query languages like GQL, SQL/PGQ, and SPARQL which are implemented by graph database management systems such as Neo4J, TigerGraph, Virtuoso, QLever, and Amazon Neptune [109, 45, 51, 14, 17]. On critical workloads like financial fraud detection, subgraph matching is a part of virtually all analysis pipelines [128]. For example, money laundering often manifests as cycles of transactions in financial networks [121, 69].

By taking advantage of recent advances in lossy graph representations such as quasi-stable coloring [85], COLOR approximately captures the topology of the graph in a small summary. It then directly estimate cardinalities on the summary without needing to access the data graph. The key idea is to color the graph G such that nodes of the same color have a similar number of edges to each other color. This mitigates the effect of the uniformity and independence assumptions. In Figure 6.1 for example, the coloring assigns the c, d, e nodes to the same green color. This is because green nodes have a similar number of incoming edges from blue and orange nodes, and no out edges. Meanwhile, a, b are assigned to the blue color as they have a similar number of outgoing edges to green nodes and none to orange nodes. This helps mitigate the uniformity assumption because, within nodes of a fixed color, the edge distribution is nearly uniform. Further, because high and low degree

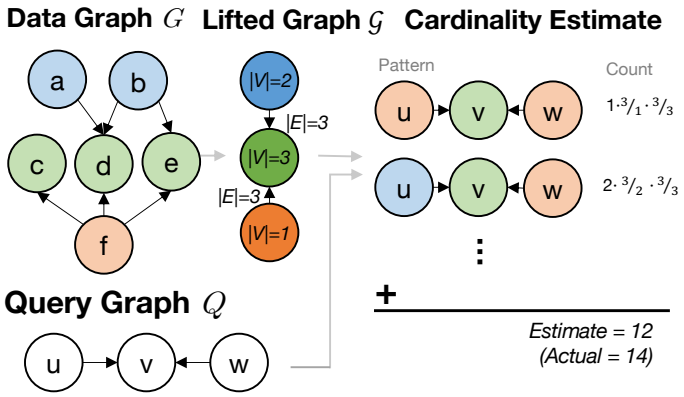


Figure 6.1: Lifted counting example.

nodes tend to be placed in different colors, correlations in the connections between them can be identified, mitigating the independence assumption.

Real world graphs are more complex, but it turns out that our approach can meaningfully capture their topology with a small number of colors, typically just 32 in our experiments. Figure 6.2 shows the maximum difference in edge counts from nodes in one color to nodes in another, averaged over all pairs of colors for four of our benchmark datasets. Lower values indicate a better coloring and smaller differences between the two most different nodes in each color. A handful of colors is sufficient to capture most of the graph topology and reduce non-uniformity by 1-2 orders of magnitude.

With these colorings in mind, we return to the example in Figure 6.1. During the offline phase, our method takes the data graph, G, and produces a compact summary, $\mathcal{G}$ called a *lifted graph*, with one super-node for each color (Sec. 6.2). We keep statistics on the number and kinds of edges which pass between these colors. During the online phase, the cardinality estimate is computed on the lifted graph by performing a weighted version of subgraph counting which we call *lifted subgraph counting* (Sec. 6.3). To extend this to cyclic queries, which occur frequently in graph databases, we also propose a technique based on a novel statistic called the *path closure probability* (Sec. 6.4).

To enable efficient inference on a more detailed, and therefore accurate, lifted graph,

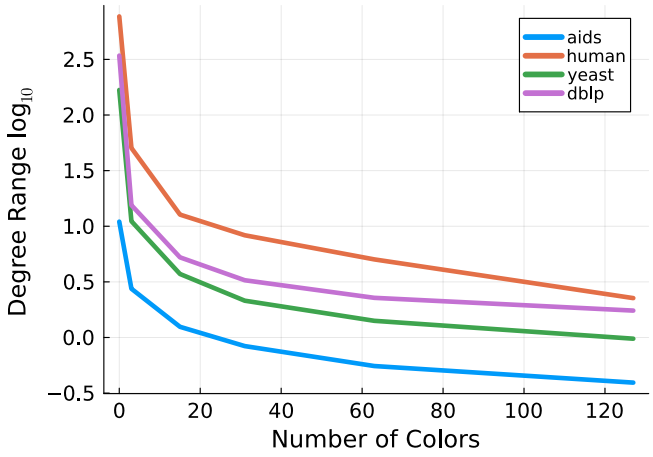


Figure 6.2: Accuracy of coloring as the number of colors increases

we propose three critical optimizations. Tree-decompositions and partial aggregation, introduced in Sec. 6.6.1, reduce the inference latency by over **100**×. Importance sampling and Thompson-Horowitz estimation over the lifted graph, the subject of Sec. 6.6.2, ensure a linear latency w.r.t. the size of the query while maintaining a **6x** lower error than a naive sampling approach. Lastly, we demonstrate how to maintain the lifted graph under updates in Sec. 6.6.3, reducing the need to rebuild the summary by providing reasonable estimates even when over 1/2 of the graph is updated.

In summary, we make the following contributions in this chapter:

- Develop the COLOR framework for producing lifted graph summaries from colorings (Sec. 6.2) and evaluate six possible coloring schemes (Sec. 6.5).

- Define a general formula for performing inference over a lifted graph, show its optimality for acyclic queries (Sec. 6.3), and extend it to cyclic ones (Sec. 6.4).

- Develop optimizations that allow for efficient and accurate inference and robust handling of updates (Sec. 6.6). These optimizations are: tree-decomposition, importance sampling, and Thompson-Horowitz estimation.

- Empirically validate COLOR's superior performance on eight standard benchmark datasets and against nine comparison methods (Sec. 6.7).

6.1 Problem Setting

Property Graphs The data model that we use for this work is property graphs. These are directed graphs where each edge and vertex is associated with a set of attributes. These attributes can be simple labels (e.g. : $friendOf$) or key-value pairs (e.g. $Age = 72$). This is the most general data model for graphs; it matches the model of GQL, Cypher, and GraphQL, and it captures the RDF model [56, 74, 9].

Formally, we define property graphs as follows:

Definition 6.1.1. A property graph, $G(V, E, \lambda, \chi)$, is a directed graph with vertices V, edges E, attribute domain $\mathbb{A}$, and two annotation functions,

- λ : $V \to 2^{\mathbb{A}}$ which maps a vertex to a set of attributes [1]
- χ : $E \to 2^{\mathbb{A}}$ which maps an edge to a set of attributes

Subgraph Counting The goal of subgraph counting is to find the number of occurrences of a query graph Q in a larger data graph G. On a property-less graph, an occurrence is defined as any mapping from the vertices in Q to the vertices in G such that all edges in Q are mapped to edges in G, i.e. a homomorphism from Q to G. To account for properties, each vertex and edge of the query graph is associated with a predicate, P, that returns true or false based on the attributes (e.g. "*hasLabel:friendOf*" or "*age¿60*"). Formally, this is defined as follows,

Definition 6.1.2. For a property-less query graph Q and data graph G, we define the set of subgraph matches as,

$$\hom(Q, G) = \{\pi : V_Q \to V_G \mid \pi(E_Q) \subseteq E_G\}$$

---

[1] 2^X denotes the set of subsets of X.

Each match, π, is a function from V_Q to V_G. When Q and G are property graphs, we add the natural conditions for each π,

$$P_v(\pi(v)) = \mathbf{1} \quad \forall v \in V_Q \qquad P_e(\pi(e)) = \mathbf{1} \quad \forall e \in E_Q \tag{6.1}$$

The subgraph count is then $|\hom(Q, G)|$.

This work studies the problem of *cardinality estimation*. Recent work in both graph and relational databases has demonstrated the importance of cardinality estimation for producing efficient query plans [117, 99]. This problem consists of two phases: 1) a preprocessing phase where the statistics, denoted $\mathbf{s}$, are computed and 2) an online phase where query graphs come in and approximate subgraph counts are returned. Formally, we can view it as follows,

Definition 6.1.3. A cardinality estimation method $\mathcal{E}$ consists of two algorithms: 1) computing statistics during the preprocessing phase, $\mathbf{s} \stackrel{\text{def}}{=} \mathcal{E}_{pre}(G)$ and 2) estimating the cardinality during the online phase, $c \stackrel{\text{def}}{=} \mathcal{E}_{on}(Q, \mathbf{s}) \in \mathbb{R}_+$. The goal is to produce an estimate where $c \approx |\hom(Q, G)|$.

The primary metrics for these algorithms are: 1) the accuracy of $c \approx |\hom(Q, G)|$ 2) the latency of $\mathcal{E}_{on}$ and 3) the size of $\mathbf{s}$.

Traditional Estimators The classic System R approach to cardinality estimation in relational databases combines the number tuples in the joining relations, the number of unique values in the joining columns, and assumptions (uniformity, independence, preservation of values) to produce a basic cardinality estimate [99, 68]. In the graph setting, this estimation method looks like the following,

Definition 6.1.4. Given a query graph $Q(V_Q, E_Q)$ and data graph $G(V, E)$, the traditional estimation method is,

1. $\mathcal{E}^{trad}_{pre}(G) = (|V|, |E|)$ (number of vertices and of edges)

2. $\mathcal{E}^{trad}_{on}(Q, \mathbf{s}) = \prod_{v \in V_Q} |V| \cdot \prod_{e \in E_Q} \frac{|E|}{|V|^2}$

Intuitively, the estimation formula calculates the number of possible embeddings of the query graph in the data graph, $\prod_{v \in V_Q} |V|$, then scales this by the probability of any embedding having the correct set of edges, $\prod_{e \in E_Q} \frac{|E|}{|V|^2}$. In effect, this estimation procedure assumes that the data graph is distributed like an Erdos-Renyi random graph with $|E|$ edges and $|V|$ vertices and produces an accurate estimate given this assumption. However, the structure of most real world graphs is much more complex. This results in very different subgraph counts from those on Erdos-Renyi random graphs and motivates the use of more complex estimators.

Example 6.1.5. *Recall that the standard estimate of a join* $Q(x, y, z) = R(x, y) \wedge S(y, z)$ *is* $\frac{|R| \cdot |S|}{\max(|R.y|, |S.y|)}$. *When both* R, S *are the edge relation* E, *then* $R.y = S.y = V$ *(assuming no isolated vertices) and the traditional estimator becomes* $\frac{|E| \cdot |E|}{|V|}$, *which is the same as the formula above,* $|V|^3 \frac{|E|^2}{|V|^4}$.

6.2 Colorings & Lifted Graphs

Colorings and lifted graphs are the core of our framework, so we start by formally defining them here. For clarity of presentation, we will begin by ignoring predicates and reintroduce them later. Given a graph $G(V, E)$, a *coloring* σ is a function from V to C where C is a small set of colors, $|C| \ll |V|$. Under a *quasi-stable* coloring [85], two vertices in the same color will have similar distributions of outgoing edges to different colors, i.e. any two red vertices should have nearly the same number of edges to blue vertices. Formally,

Definition 6.2.1. A coloring, σ, is quasi-stable if the following properties hold for all pairs of vertices v_1, v_2. If $\sigma(v_1) = \sigma(v_2)$, then:

$$\forall\, c \in C : |\{(v_1, v) \in E | \sigma(v) = c\}| \approx |\{(v_2, v) \in E | \sigma(v) = c\}| \tag{6.2}$$

In English, σ is quasi-stable if for any two colors c_0, c, any two vertices v_1, v_2 colored c_0 have approximately the same number of neighbors colored c. If we replace $\approx$ with $=$ in (6.2), then σ is called a *stable coloring*. Stable colorings are commonly used in graph isomorphism algorithms, and have elegant theoretical properties [62, 66]. However, they are unsuitable for our purpose, because stable colorings of real-world graphs require a very large

number of colors [85]. In fact, in a random graph, every vertex has a distinct color with high probability [64]. Instead, we relax equality $=$ to approximation $\approx$ in Definition 6.2.1 in exchange for using a much smaller number of colors. To do this, we apply a variety of coloring algorithms (Sec. 6.5) which produce a dramatic reduction in the number of colors with only a small relaxation of $=$ to $\approx$. We demonstrate this in Fig. 6.2. This graph shows the average range of degrees from nodes in one color to nodes in another as the number of colors varies. Across all graphs, a coloring with just 32 colors (using the Quasi-Stable coloring method from [85]) lowers the average degree range by over 2 orders of magnitude as compared to the initial graph.

For any color $c \in C$, we denote the set of vertices colored c by $V_c \stackrel{\text{def}}{=} \{v \in V \mid \sigma(v) = c\}$. For any two colors c_1, c_2 we denote the set of edges between them by $E_{c_1c_2} \stackrel{\text{def}}{=} E \cap (V_{c_1} \times V_{c_2})$. The *lifted graph* consists of a quasi-stable coloring, together with statistics for each pair of colors:

Definition 6.2.2. Fix a directed graph $G = (V, E)$, and a coloring σ with colors C. A *lifted graph* is a triple, $\mathcal{G} = (F, \psi, \tau)$, consisting of the following parts.

- $F = (V_F, E_F)$ is a graph where $V_F = C$ and $E_F = \{(c_1, c_2) \mid E_{c_1c_2} \neq \emptyset\}$.
- $\psi : C \to \mathbb{N}$ where $\psi(c) = |V_c|$
- $\tau : E_{c_1c_2} \to \mathbb{R}_+$ maps edges of the graph to statistics about the edges in $E_{c_1c_2}$

We consider the following choices for the edge statistics function τ:

$$\begin{aligned}
\tau_{\min}(c_1, c_2) &= \min\{|\{(v_1, v_2) \mid v_2 \in V_{c_2}\}| \mid v_1 \in V_{c_1}\} \\
\tau_{\text{avg}}(c_1, c_2) &= \frac{|E_{c_1c_2}|}{|V_{c_1}|} \\
\tau_{\max}(c_1, c_2) &= \max\{|\{(v_1, v_2) \mid v_2 \in V_{c_2}\}| \mid v_1 \in V_{c_1}\}
\end{aligned}$$

They represent the minimum degree, the average degree, and the maximum degree of a vertex in c_1 to vertices in c_2 respectively. In the following sections, it will be sufficient to assume that τ means τ_{avg}, unless otherwise noted.

Thus, ψ is a function that returns the number of vertices with the color c, and τ is a function that returns statistics about the set of edges between a pair of colors. The lifted graph forms a fuzzy compression of the data graph, and is computed offline, during preprocessing. In our approach, this is the statistic, $\mathbf{s} = \mathcal{E}_{pre}$, as defined in Def. 6.1.3.

Example 6.2.3. *Consider again the example data graph and coloring in Figure 6.1. First, the data graph G is colored with a quasi-stable coloring. This produces three different colors, which reflect that topologically there are three different "kinds" of vertices in the data graph. In particular, orange vertices have an 3 out-degree, no in-degree; the green vertices no out-degree, 2 out-degree; and blue 1.5 average out-degree, no in-degree. Due to the arrangement of these edges, we can produce a "good" coloring where vertices a and b are assigned to the blue partition. Further, we can produce the lifted graph $\mathcal{G}$ by choosing the degree sum as our edge statistic. In this figure, the vertex labels are the partition cardinalities, i.e. the values of ψ. The label edges represent the sum of edges between two colors: for example, green vertices have a total of 3 edges into the orange vertices, i.e. the values of τ_Σ. Note that this lifted graph closely captures the distribution of edges in the data graph while being half the size.*

Lifted Property Graphs To account for attributes and predicates in the lifted graph, we adjust the definition of ψ and τ to accept predicates in addition to colors. Suppose that a query has a vertex predicate P, then we define $V_{c,P}$ as the set of data vertices in the color c which pass the predicate P. Similarly, E_{c_1,c_2,P_e,P_v} is the set of edges starting in color c_1, matching predicate P_e, and landing on a node in color c_2 which matches P_v. With this, we can then redefine ψ and τ_{avg},

$$\psi(c, P) = |V_{c,p}| \qquad \tau_{avg}(c_1, c_2, P_e, P_v) = \frac{|E_{c_1,c_2,P_e,P_v}|}{|V_{c_1}|}$$

We allow these functions to be exact or approximate in order to accommodate more complex predicates. If the predicates are all of the form $hasLabel : X$ and there are few edge label/vertex label combinations, then this can be calculated and stored explicitly. However, predicates like range or LIKE benefit from approximating $|E_{c_1,c_2,P_e,P_v}|$ using standard

Table 6.1: Notation Dictionary

Symbol	Meaning
$G(V, E, \lambda, \chi)$	Property graph with vertices V, edges E, vertex attributes λ, and edge attributes χ.
$\mathcal{G}(F, \psi, \tau)$	Lifted graph with color graph F, color cardinalities ψ, and color edge statistics τ.
$W(\pi, Q, \mathcal{G})$	Estimate of the subgraph count for query Q with coloring π based on lifted graph $\mathcal{G}$.
$\Phi(Q, \mathcal{G})$	Estimate of the subgraph count for query Q based on lifted graph $\mathcal{G}$.
$\gamma(C_1, C_2, D)$	Probability of a path closing a cycle from C_1 to C_2 with directionality D

techniques like histograms or n-grams. This can then be extended to t_{min} and t_{max} using techniques similar to those in [43].

6.3 Lifted Subgraph Counting

We have described the lifted graph, a small weighted graph which approximately captures the topology of the data graph. Next, we show how to use the lifted graph for cardinality estimation, which we call the *lifted subgraph counting problem*,

Definition 6.3.1. Fix a lifted graph $\mathcal{G} = (F, \psi, \tau)$ and a query graph Q. An estimation procedure is a function

$$W : \hom(Q, F) \times Q \times \mathcal{G} \to \mathbb{R}_+$$

The lifted subgraph count is,

$$\Phi(Q, \mathcal{G}) = \sum_{\pi \in \hom(Q,F)} W(\pi, Q, \mathcal{G}) \tag{6.3}$$

A homomorphism $\pi : Q \to F$ associates to each query vertex x a color $\pi(x) \in C$; we will also call π a *coloring* of the query Q. The estimator $W(\pi, Q, \mathcal{G})$ approximates the number of outputs of the query with coloring π. Note, we generally drop Q and $\mathcal{G}$ when obvious from context. The total estimate, $\Phi(Q, \mathcal{G})$ is simply the sum over all colorings π. To see the intuition, recall that errors in traditional cardinality estimation come from correlation and skew. For example, the former could mean that high degree vertices are more likely to be connected to high degree vertices (or vice versa), while the latter means that the degrees of vertices have a wide range. By grouping vertices into colors based on their local topology (their degrees, their neighbors' degrees, etc), and fixing a particular coloring π of the query vertices, we reduce the variance of the estimate $W(\pi)$, leading to a reduced error overall. In the rest of this section we define the estimate function W assuming that the query graph is acyclic. We will extend it to arbitrary query graphs in Sec. 6.4.

6.3.1 Acyclic Query Graphs

For acyclic queries, we use the following function W:

Definition 6.3.2 (Lifted Estimator for Acyclic Queries)**.** Let $Q = (V_Q, E_Q)$ be an acyclic query graph, and let $x_1, \ldots, x_{|V_Q|}$ be a topological ordering of its vertices: in other words, every vertex x_j is connected to some x_i for $i < j$. For any homomorphism $\pi : Q \to F$, we define:

$$W(\pi) \stackrel{\text{def}}{=} \psi(\pi(x_1), P_{x_1}) \prod_{(x_i,x_j) \in E_Q} \tau(\pi(x_i), \pi(x_j), P_{x_j}, P_{(x_i,x_j)}) \tag{6.4}$$

We adopt the convention that if x_i and x_j are connected by a reverse edge, i.e. E_Q contains (x_j, x_i) rather than (x_i, x_j), then this is reflected in the query graph with a predicate, $dir = \leftarrow$, on the edge. Intuitively, we process the edges in the topological order, so we always multiply with the in/outdegree of the color assigned to the topologically earlier vertex. We will choose the topological order such as to minimize the time needed to compute $\hom(Q, G_F)$, see Sec. 6.6.

Example 6.3.3. *Here we show that the lifted graph estimator generalizes the traditional estimator in Def. 6.1.4. We illustrate this with a graph $G = (V, E)$ and the 2-edge query $Q(x, y, z) = E(x, y) \wedge E(y, z)$ from Example 6.1.5. Assume that we use a lifted graph, $\mathcal{G}(F, \psi, \tau)$, consisting of a single color c and a single edge: $F = (\{c\}, \{(c, c)\})$. Then the statistics are $\psi(c) = |V|$ and $\tau(c, c) = \frac{|E|}{|V|}$ (recall that we assumed τ refers to τ_{avg}), there is a single homomorphism $\pi : G \to F$, and our estimate is $W(\pi) = \psi(c)\,(\tau(c, c))^2 = |V|\frac{|E|^2}{|V|^2} = \frac{|E|^2}{|V|}$. This is the same as the traditional estimator in Example 6.1.5.*

Example 6.3.4. *We illustrate now how a better designed lifted graph can lead to an improved estimator. Assume that the data graph is the disjoint union of two graphs, $G = G_1 \cup G_2$, where $G_1(V_1, E_1)$ is a 2-regular graph on 10000 vertices, and $G_2(V_2, E_2)$ is a clique of size 100. Suppose Q is a path of length k. Since the average degree in G is ≈ 4, a traditional estimate for Q is $10100 \cdot 4^k$, which vastly underestimates the true count, because it doesn't capture the skew and correlation introduced by the clique sub-graph. Suppose we pre-compute a lifted graph consisting of two colors: green contains all vertices V_1 and red color contains all vertices V_2. There are only two edges (green, green) and (red, red), and therefore only two colorings π of the query Q. We compute W separately for each of the two colorings, then return their sum, $100 \cdot 99^k + 10000 \cdot 2^k$, which, for our simple data graph, is an exact count.*

6.3.2 Special Case: Stable Colorings

As a theoretical justification of our method, we prove that if the lifted graph is a *stable* coloring (meaning: $\approx$ is $=$ in Def. 6.2.1), then our estimate for acyclic queries is exact, although we defer the formal proof to App. C.1.

Theorem 6.3.5. *Let $\mathcal{G}$ be a lifted graph defined by a stable coloring σ. Then $\tau_{min} = \tau_{avg} = \tau_{max}$, and, for any acyclic query Q, the lifted graph estimator is exact:*

$$|\hom(Q, G)| = \Phi(Q, \mathcal{G}) \tag{6.5}$$

This theorem states that stable colorings are a perfect statistic for cardinality estimation of acyclic query graphs. However, we cannot use them in practice, because the number of

stable colors needed to represent real graphs is close to the number of vertices in the data graph [106, 85]. Fortunately, we can prove an additional corollary for quasi-stable colorings:

Corollary 6.3.6. *Let ϵ be the approximation error of the lifted graph $\mathcal{G}$ defined as,*

$$\epsilon = \max_{c_1,c_2 \in C} \frac{\tau_{\max}(c_1, c_2)}{\tau_{\min}(c_1, c_2)} \tag{6.6}$$

Then, we can bound the error of our subgraph estimator as,

$$max(\frac{\Phi(Q, \mathcal{G})}{|hom(Q, G)|}, \frac{|hom(Q, G)|}{\Phi(Q, \mathcal{G})}) \leq \epsilon^{(|V_Q|-1)} \tag{6.7}$$

Intuitively, as colorings approach stability, the estimate converges to the true subgraph count.

6.4 Handling Cycles

Cyclic queries are a fundamentally different challenge for cardinality estimators because they allow complex dependencies between vertices within the query graph. Practically, they require estimating the probability that an edge between two vertices exists, conditioned on the fact the these vertices are already connected in the query graph. This section outlines new techniques to estimate this *cycle closure probability*.

6.4.1 Cycle Closure Probability

To ground this discussion, we begin by defining the probability space and random variables. The former is defined by a uniform random selection of $|V_Q|$ vertices from V_G with replacement. This is equivalent to a random mapping from V_G to V_Q. The set of vertices selected by this process is denoted with the random variables $V_1, \ldots, V_{|V_Q|}$. Further, each edge of the query graph, $(v_i, v_j) = e_i \in E_Q$, is associated with a binary random variable E_i which is true when $(V_i, V_j) \in E_G$. In other words, E_i is true iff the data vertices mapped to that edge of the query have an edge between them.

As a basic example, we can calculate the unconditioned probability of E_i for any edge e_i as follows,

$$P(E_i) = \frac{|E_G|}{|V_G|^2}$$

Further, we can express the cardinality of an arbitrary query as,

$$|Hom(Q,G)| = |V_G|^{|V_Q|} \cdot P(\cap_{e_i \in E_Q} E_i)$$

With conditional probability, we can expand the probability as,

$$P(\cap_{e_i \in E_Q} E_i) = \prod_{e_i \in E_Q} P(E_i | E_1, \ldots, E_{i-1})$$

When the endpoints of an edge are contained within the previous edges, $e_i \in \cap_{j=1}^{i-1} e_j$, the probability within the product is a cycle closure probability. It is this probability which we try to estimate in this section, and the crucial challenge is estimating the effect of the previous edges, $E_1, \ldots, E_{i-1}$.

The naive solution is to consider all patterns of a fixed size (i.e. the pattern induced by $E_1, \ldots, E_{i-1}$) and calculate the probability of an edge occurring between two nodes of that pattern in the data graph. However, the number of patterns increases super-exponentially in their size due to the choices of basic graph pattern, edge direction, and predicates, so this approach is infeasible for all but the smallest queries. Our approach attempts to tackle this by considering a smaller set of patterns and composing them smartly.

6.4.2 Path Closure Probability

We introduce *path closure probabilities* which represent the probability that a path in the data graph is "closed", i.e. there is an edge from the starting vertex to the ending vertex. To limit the number of probabilities that we store, paths are grouped by their directionality, e.g. $D = \{\leftarrow, \rightarrow\}^k$, and by the color of their starting and ending vertices. We denote this probability as,

Definition 6.4.1. Let $\mathcal{P}_{c_1,c_2,D}(G)$ be the set of paths in the data graph G with directions matching D and starting/ending color c_1/c_2. Let $\mathcal{C}_{c_1,c_2,D}(G)$ be the subset of paths in $\mathcal{P}_{c_1,c_2,D}(G)$ which are closed. The path closure probability is then,

$$\gamma(c_1, c_2, D) = \frac{|\mathcal{C}_{c_1,c_2,D}(G)|}{|\mathcal{P}_{c_1,c_2,D}(G)|}$$

Given this statistic, we can define an expression for the cycle closure probability, $P(E_i|E_1,\ldots,E_{i-1})$. Let $\mathcal{S}_{i-1}$ be the set of simple paths in $E_1,\ldots,E_{i-1}$ which start at the source of E_i and end at its destination. Note that the closure of any path within S implies that E_i is true. Based on this, we treat the closure of each of these paths as an independent event (a conservative assumption), and calculate the cycle closure probability as,

$$P(E_i|E_1,\ldots,E_{i-1}) = 1 - \prod_{(c_1,c_2,D)\in\mathcal{S}_{i-1}} (1-\gamma(c_1,c_2,D))$$

We can now explain how we extend the definition of the acyclic estimator (6.4) to handle arbitrary query graphs. Fix a query graph $Q=(V_Q,E_Q)$, and consider a topological edge ordering, $e_1,\ldots,e_{|E_Q|}$, which means that every edge e_j has a vertex in common with some previous edge e_i, $i<j$. This ordering defines a spanning tree T, consisting of the subset of edges that introduce a new vertex, i.e. $T=\{e_j \mid e_j \not\subseteq \bigcup_{i<j} e_i\}$. If $e_i=(x,y)$ is not a tree edge, then both vertices x,y are already connected in the subgraph consisting of $e_1,\ldots,e_{i-1}$. The modified definition of the estimator (6.4) is:

$$W(\pi) = \psi(\pi(x_1,\lambda_Q(x_1))) \prod_{e_i\in E_Q} \omega(e_i) \tag{6.8}$$

$$\omega(e_i) = \begin{cases} \tau(\pi(x),\pi(y),\lambda_Q(x),\chi_Q(x,y))) & \text{if } e=(x,y)\in T \\ 1-\prod_{(c_1,c_2,D)\in\mathcal{S}_{i-1}}(1-\gamma(c_1,c_2,D)) & \text{if } e=(x,y)\notin T \end{cases} \tag{6.9}$$

To keep the construction of our statistics tractable, we do not calculate γ exactly. Instead, we sample paths from the lifted graph and use these to calculate probabilities. As a default, we use $100,000$ sampled paths when calculating these statistics in our experiments. If a particular color combination doesn't occur in our samples, we fall back to the probability just conditioned on the sequence of directions.

6.5 Alternate Coloring Methods

Because a coloring can be any mapping from vertices to colors, there is a wide design space of algorithms for creating colorings. The goal of this section is to find colorings which facilitate improved cardinality estimations. In this work, we focus on divisive colorings where all vertices begin in the same color and then the following steps proceed iteratively:

1) identify a color, c, to split into two colors 2) for each vertex in c, determine whether it should stay in c or join the new color. The benefit of this approach is that arbitrary coloring methods can be composed, allowing for more robustness and accuracy.

Quasi-Stable Coloring [85] As explained earlier, this is a generalization of the traditional color-refinement algorithm. Rather than producing a stable coloring, this algorithm softens the requirements and instead requires vertices in each color to have a "similar" number of edges to each other color. At each iteration, it selects the color with the widest range of degrees w.r.t. another color and splits it into two colors with more uniform counts relative to the other color.

Degree Coloring This coloring simply separates vertices into colors based on their overall degree. The intuition is that vertices with high degree are generally occupying similar positions in the data graph and vice versa with low degree vertices. It begins by selecting the color with the largest range of degrees to split, then separates vertices into two colors depending on whether they are above or below the average degree.

Neighbor Label Coloring An alternative to quasi-stable coloring, this method attempts to reduce the variance introduced by label predicates (e.g. "hasLabel:X") by grouping vertices based on their neighbors' label attributes. First, it selects the color whose vertices have the widest range of degrees w.r.t. the vertex labels of their neighbors. It then splits it into two colors which have more uniform connections to vertices with each vertex label.

Vertex Label Coloring A more direct version of the previous approach, this coloring also aims to reduce the effect of label predicates. This time it aims to make the distribution of vertex labels within colors more uniform. To do this, it first identifies the color, c_1, with the most even distribution of a particular label, weighted by size. The nodes in c_1 which have that label attribute are then put in a new color and the ones which do not remain in c_1.

Mixture Coloring The previous colorings generally target a particular source of variance related to either topology or attribute distributions, and they divide the color where this kind of variance appears most strongly. So, it makes sense to layer these colorings in order to jointly manage these different concerns, and we call this a mixture coloring. In the experiments (Fig. 6.9), we show that this is the most accurate coloring across a range of workloads.

Hash Coloring For completeness, we consider the naive hash coloring which uniformly randomly sorts vertices into colors. This corresponds to the partitioning used by [30] to tighten their cardinality bounds. This method is convenient because construction is linear in the size of $|G|$, and it does not require coordination in a distributed setting. However, it offers limited improvement to the estimator because it doesn't take the specific topology or attributes of the graph into account.

6.6 Optimization

6.6.1 Partial Aggregation

Naively, the runtime of inference on the lifted graph is exponential in the size of the query graph, making it intractable for even moderately sized query graphs. The lifted graph is dense so the size of $Hom(Q, F)$ is approximately $|C|^{|Q|}$. Fortunately, we can rephrase Eq. 6.3 to drastically reduce this runtime via aggregate push-down. To do this, we express the set of lifted graph matches, $\hom(Q, F)$, as $(c_1, \ldots, c_{|Q|}) \in C^{|Q|}$. We then use the definition of W from 6.8 and, as before, we assume a topological ordering on the edges, $e_1, \ldots, e_{|E|}$, and vertices, $v_1, \ldots, v_{|Q|}$.

$$\Phi(Q, F, W_{\psi,\tau}) = \sum_{c_1,\ldots,c_{|Q|} \in C} \psi(c_1) \prod_{i=1}^{|E_Q|} \omega(\pi_{c_1,\ldots,c_{|Q|}}, e_i) \tag{6.10}$$

At this point, we can identify this as a *Functional Aggregate Query* [88] and apply the techniques there to solve it efficiently.[2] As an example, suppose that the query graph Q is a

[2]Note, this is closely related to the variable elimination algorithm for probabilistic graphical models as well as tensor contraction algorithms.

Algorithm 9 Optimized Inference Algorithm

Require: $F_{G,\sigma,l}(G_F(C, E_V), \psi, \tau)$, $Q(V_Q, E_Q)$, $v_1, \ldots, v_{|V_Q|}$ // Lifted Graph, Query Graph, Vertex Order

$PC = \{(\{\}, 1)\}$ // Partial Colorings
$V_F = \{\}$
for $i \in [1, \ldots, |Q|]$ **do**
  $PC' = \{\}$
  $E_i = \{e \in E_Q \mid v_i \in e\}$
  for $\pi, w \in PC$ **do**
    for $c \in C$ **do**
      $\pi' = \pi \cup (v_i \to c)$
      $w' = w \cdot \prod_{e \in E_i} \omega(\pi', e)$ ← Estimator Sec. 6.4.1
      $PC' = PC' \cup \{\pi', w'\}$
    end for
  end for
  $PC = PC'$
  $V_S = \{v \in V | (v, v_j), (v_j, v) \notin E_Q \; \forall \; j > i\} \setminus V_F$
  $V_F = V_F \bigcup V_S$
  $PC = \sum_{V_S} PC$ ← Partial Aggregation Sec. 6.6.1
  $PC = Sample(PC)$ ← Sampling Sec. 6.6.2
end for
return PC

line graph $v_1 \to v_2 \to v_3$. The naive expression would be the following $O(|C|^3)$ expression,

$$\Phi(Q, F, W_{\psi,\tau}) = \sum_{c_1,c_2,c_3 \in C} \psi(c_1)\tau((c_1, c_2))\tau((c_2, c_3))$$

However, by pushing down the summation over c_1, we can produce an expression which requires $O(|C|^2)$ time to evaluate.

$$f(c_2) = \sum_{c_1 \in C} \psi(c_1)\tau((c_1, c_2))$$
$$\Phi(Q, F, W_{\psi,\tau}) = \sum_{c_2,c_3 \in C} \tau((c_2, c_3))f(c_2)$$

This version materializes a a vector of intermediate values and then uses that vector in the second line. Doing this allows us to avoid performing an unnecessary summation over 3 variables at once.

More generally, we can apply this strategy by choosing a variable order and at each step summing out the next variable in the order. The efficacy depends on the maximum number of variables present in any intermediate product. If an intermediate product involves k variables, then we need to compute a relation of size $|C|^k$ which dominates the runtime. We defer the details of the proof to App. C.2, but this intuition can be formalized as follows using the theory of tree decompositions and treewidth,

Theorem 6.6.1. *Given a query graph, Q, a lifted graph, F, and a decomposable estimator W, $\Phi(Q, F, W_{\psi,\tau})$ can be computed in time $O(|C|^{tw(Q)+1})$ where $tw(Q)$ is the treewidth of Q.*

Of course, this relies on finding a good ordering of the query vertices, and finding the optimal one is naively NP-Hard with respect to the size of Q. Fortunately, there are very effective heuristics for identifying good tree decompositions, and we apply these in our system, using the min-fill heuristic [126]. To accommodate our sampling techniques, we restrict these tree decompositions to path decompositions and get results relative to the pathwidth.

Table 6.2: Estimator Failure Rates per dataset.

Dataset\Method	cs	wj	jsub	impr	cset	alley	alleyTP	LSS	BSK+	sumrdf	Color
human	0.67	**0.00**	0.22	0.63	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
aids	0.69	0.07	0.14	0.28	**0.00**	0.04	0.01	**0.00**	0.02	0.39	**0.00**
lubm80	0.83	0.17	0.67	0.67	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
yeast	1.00	0.97	0.97	0.11	**0.00**	0.63	0.60	**0.00**	0.63	0.88	**0.00**
dblp	1.00	0.99	0.94	0.15	**0.00**	0.14	0.14	**0.00**	0.70	0.85	**0.00**
youtube	0.99	0.93	1.00	0.22	**0.00**	0.10	0.05	**0.00**	0.63	0.78	**0.00**
eu2005	0.95	0.90	0.91	0.55	**0.00**	**0.00**	**0.00**	**0.00**	0.22	0.44	**0.00**
patents	0.98	0.88	0.98	0.08	**0.00**	0.13	0.13	**NA**	0.67	0.79	**0.00**

6.6.2 Sampling Techniques

In real systems, cardinality estimation needs to be extremely fast and consistent because its overhead is seen by every query. While partial aggregation speeds up inference for simple query graphs with low treewidth, larger and denser query graphs still pose a problem. To avoid this, we use a sampling procedure that ensures linear inference w.r.t query size (see Fig. 6.13). This is similar to a weighted version of the WanderJoin algorithm from [101]. We apply a Thompson-Horowitz estimator to random paths within the lifted graph. However, we adjust the method in two important ways: 1) we incorporate the sampling into the aggregation framework from Sec. 6.6.1 2) we apply importance sampling to account for the fact that different paths within the lifted graphs contribute more or less to the cardinality estimate.

Sampling During Aggregation At each step, we process a vertex of the query graph and materialize an intermediate result consisting of partial colorings and their weights. After materialization, we apply sampling in order to reduce the amount of partial colorings that we extend in the next step. By doing this at each step, we can maintain a constant number of partial colorings at all times and ensure a linear runtime w.r.t. the size of the query

graph. Because we still apply aggregation, we reduce the amount of sampling required.

Importance Sampling This is a classical technique for approximating the value of an integral, and we adapt it here by noting that our summation in 6.10 is a discrete integral over a product. The core idea is to sample points of the integrand which contribute more heavily to the result with higher probability in order to reduce the variance. Because determining the contribution of a partial coloring to the final result is challenging, we approximate this contribution via its partial count. This assumes that a high partial count leads to a high contribution to the final sum. To keep our estimator unbiased, we apply Thompson-Horowitz estimation and scale each sampled partial count by the inverse of its selection probability. Finally, we scale the total sampled weight to set it to the total weight prior to sampling.

6.6.3 Handling Updates

Updates pose a challenge to summary-based estimators because the statistics which they collect become stale over time as updates are applied to the database. Traditional estimators simply rebuild the summary intermittently[68]. This leads to severe decreases in accuracy under even modest updates because the estimator is entirely "blind" to them. On the other hand, recalculating the summary before each query is very costly. In this section, we demonstrate how COLOR supports a middle ground approach that applies fast, basic updates to the lifted graph, allowing it to maximize the time between full rebuilds.

First, we formally define updates in our setting,

Definition 6.6.2. Given a data graph G, an update θ can either add an edge between existing vertices or a new vertex with attributes A:

- $\theta_V = (v, A)$ where $v \in G_V, A \in \mathbb{A}$
- $\theta_E = (v_1, v_2, A)$ where $v_1 \in G_V, v_2 \in G_V, A \in \mathbb{A}$

This definition allows for adding a single edge or vertex to the graph at a time. We then define a summary update function to incorporate these updates without accessing G.

Definition 6.6.3. Given a data graph G, a lifted graph $F = (G_F, \psi, \tau)$, and update θ, we define the summary update function as follows where F' is the updated lifted graph,

$$F' = \delta(F, \theta)$$

Depending on the type of θ, the functionality of δ can change:

$$\delta = \begin{cases} \delta_V, & \theta \in \theta_V \\ \delta_E, & \theta \in \theta_E \end{cases}$$

Depending on the estimator, the correct definition of δ will change. Here, we focus on the average degree estimator.

Vertex Updates The vertex update function δ_V affects the stored edge statistics τ and color sizes ψ in the lifted graph. Because a new vertex has no edges, we have no knowledge about which color it should be placed once its edges are added. Conservatively, we simply add it to the largest existing color which dilutes the impact on the average degree. In this way, we preserve the high quality information in other colors while the largest color gracefully degrades to a traditional estimator as in Def. 6.1.4. After choosing the color for the new vertex, we adjust $\psi(c)$ by incrementing its value by one, and we scale down τ to account for the new vertex.

Edge Updates Given an update $\theta_E = (v_1, v_2, A)$, the edge update function δ_E marginally increases τ for the combination of attributes and colors in the edge update. However, the edge update doesn't directly contain the colors of v_1 and v_2 or the attributes of v_2. To retrieve the colors, c_1 and c_2, associated with v_1 and v_2, we look up their values in π which we store compactly (and approximately) as a series of cuckoo filters. To calculate the attributes of v_2, we leverage statistics about the attribute distribution in c_2 to sample a set of attributes.

Path Closure Probabilities The lifted graph contains statistics about the cycle-closing probability for existing nodes and edges, but additions to the graph change this probability. To account for this, we make an adjustment to the ω°_{CCP} function. We calculate the

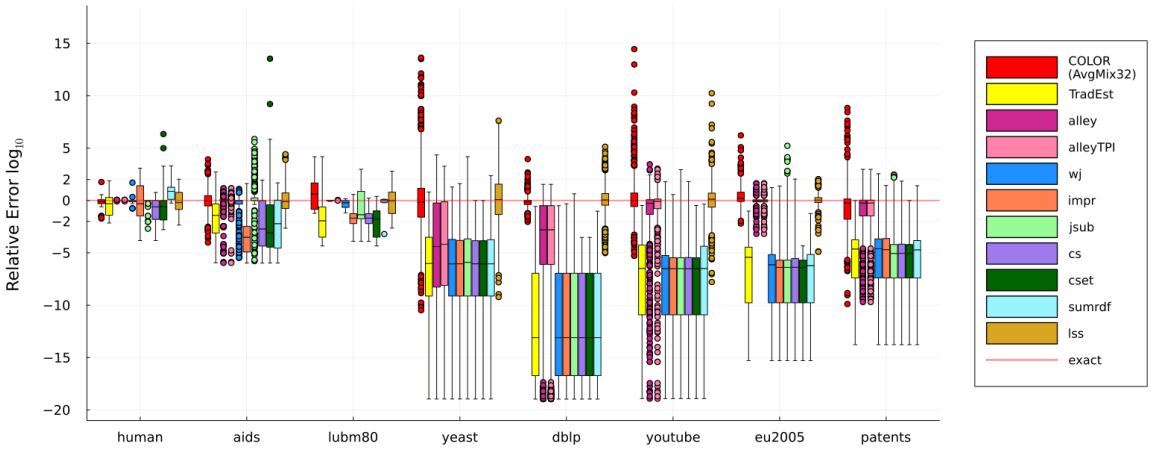


Figure 6.3: Relative Error by Estimator

probability that either the path was originally closed, $\gamma(c_1, c_2, D)$, or is closed by an update edge. For a set of edge updates, $\mathbb{S}_{\theta E}$:

$$\gamma'(c_1, c_2, D) = 1 - (1 - \gamma(c_1, c_2, D))(1 - \frac{|\mathbb{S}_{\theta E}|}{|V_F|^2})$$

Deletions To handle deletions, COLOR simply performs the inverse of the update logic.

6.7 Evaluation

In this section, we provide a detailed experimental analysis of our framework on eight benchmark datasets and against nine competitive baselines. Compared to existing methods, COLOR exhibits competitive accuracy, speed, and scalability. We also demonstrate the significance of our performance optimizations for building and maintaining graph summaries. Overall, we show that COLOR:

1. Never experiences estimation failure on any query across all workloads.

2. Has a median error $<$ **10** across all workloads and is up to $\mathbf{10^3}\times$ more accurate than competing methods.

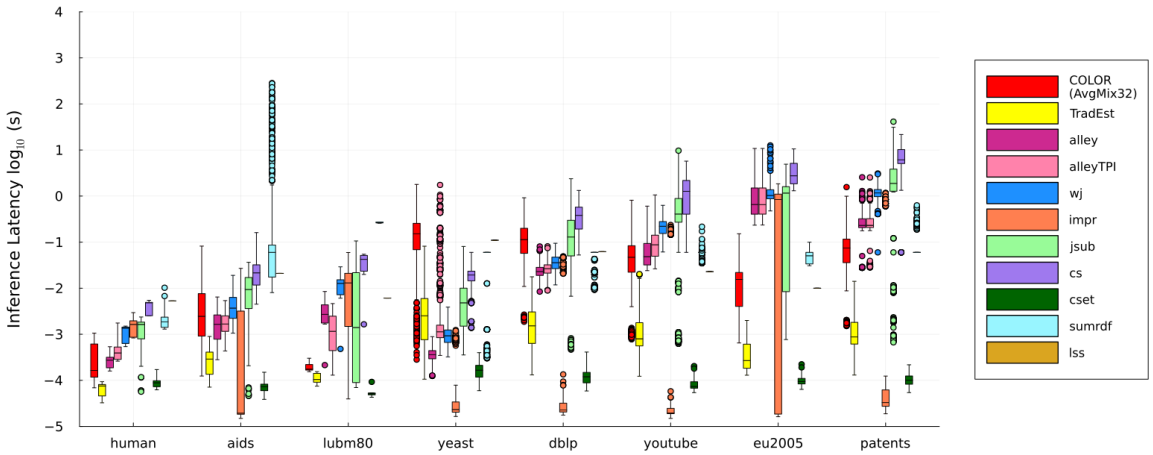


Figure 6.4: Inference Time by Estimator

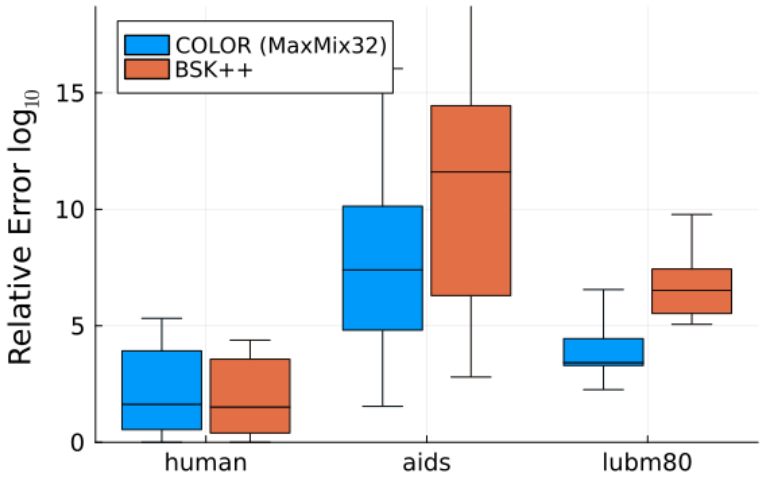


Figure 6.5: Relative Error by Cardinality Bound Method

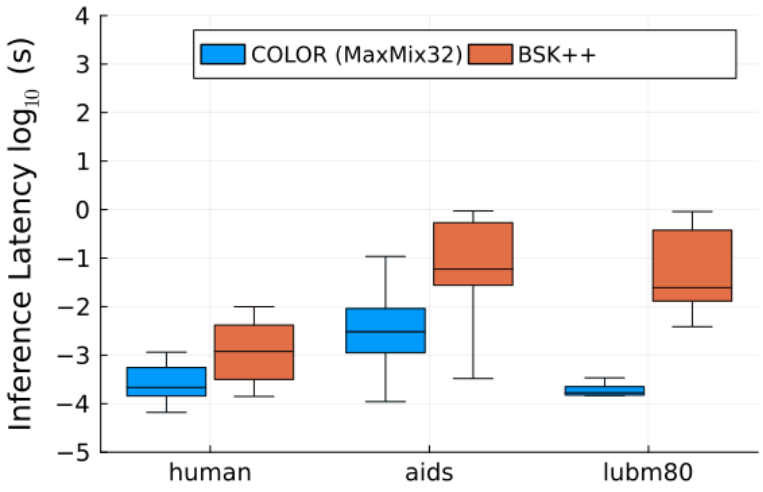


Figure 6.6: Inference Time by Cardinality Bound Method

3. Produced up to $\mathbf{10^7}\times$ tighter bounds when using τ_{max} than BoundSketch while being many orders of magnitude faster.

4. Requires up to $\mathbf{80 - 800}\times$ less space than competing methods and is up to $\mathbf{10 - 100}\times$ faster to construct.

5. Handles updates gracefully with only $\mathbf{3}\times$ worse median error when half the graph is updated.

Datasets & Workload. We consider datasets from [116] and [138] for our analysis. These datasets come from a variety of domains. Those from [138] are undirected graphs whose queries are larger and vary in density. Those from [116] are directed graphs whose queries are smaller and vary in shape. We adapt undirected workloads to directed methods by including reverse edges in the data graph but not in the query graphs. Both of these sources include label predicates in their queries with the former using both edge and vertex labels and the latter using only vertex labels. Table 6.3 shows their different characteristics where $|\ell_V|$ and $|\ell_E|$ are the number of edge and vertex labels.

Table 6.3: Experimental Datasets.

Dataset	—**V**—	—**E**—	—ℓ_V—	—ℓ_E—
human	4674	86282	89	1
aids	254000	548000	50	4
lubm80	2.6M	12.3M	35	35
yeast	3112	12519	71	1
dblp	317080	1M	15	1
youtube	1.1M	3M	25	1
eu2005	862664	16M	40	1
patents	3.8M	16.5M	20	1

Comparison Methods. For comparison, we use a superset of the methods considered in [116] and additionally apply them to the larger, more complex datasets from [138]. These methods include: 1) Correlated Sampling (CS) [143] 2) Characteristic Sets (CSet)[111] 3) Wander Join (WJ) [101] 4) Alley (alley) and alleyTPI [93] 5) Join Sampling with Upper Bounds (JSUB), an adaptation of [161] 6) Bound Sketch (BSK) [30] which corresponds to a hash-based coloring and using the max degree estimator. We further apply our partial aggregation optimization, and we call this improved version BSK++. 7) IMPR [33] and 8) SumRDF [135] 9) Learned Sketch for Subgraph Counting (LSS) [160], a query-driven deep learning approach. Due to hardware compatibility issues, this method was run on a different machine than the others with an Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz CPU. This machine had 32GB of memory, resulting in an OOM error for the patents dataset which we treat as NA rather than estimator failure. 10) We also include a traditional independence-based estimator (IndEst) corresponding to Def. 6.1.4. For the sampling based estimators, we apply the default sampling ratios from [116] and [93] (i.e. .03 for all methods except for Alley which uses .001). AlleyTPI uses a maximum pattern length (MAX_L) and a maximum number of stored label groups (NUM_GROUPS) when building its index, with default values of MAX_L=4 or MAX_L=5 depending on the dataset, and NUM_GROUPS=32. To prevent excessive index build times on AlleyTPI ($>$ 12 hours) for eu2005, dblp, and patents, we used

MAX_L=4. We decreased NUM_GROUPS to 16 for eu2005 and dblp and 8 for patents. We adjusted NUM_GROUPS more than MAX_L because small adjustments to the maximum pattern length greatly decrease the domain of patterns stored by the index [93]. As in the original work, we get estimates for LSS via a 5-fold cross-validation where 4/5ths of the queries are trained on and 1/5 is estimated during each fold. Lastly, recently, there has been work on the isomorphism variant of subgraph cardinality estimation from both the deep learning and sampling perspective[133, 145]. Unfortunately, this difference in problem setting makes them incomparable with the other methods considered here, so we exclude them from our evaluation.

Additionally, we experiment with several instantiations of our framework which use the following naming convention; first, we note the kind of degree statistic (Min/Avg/Max), then we describe the coloring scheme, e.g. $Q64$ as 64 colors from the quasi-stable coloring method. The mixed coloring scheme, $Mix32$, that we use as the default involves 8 divisions from degree coloring, quasi-stable coloring, neighbor labels coloring, and node label coloring, in that order. Unless otherwise noted, we use 500 samples during inference and keep track of cycle probabilities for cycles up to length 6.

Experimental Setup. To reduce noise, we repeat all inference results 3 times and report the median inference time. We do not do this for our cardinality estimates because this would unfairly reduce the impact of our sampling approach. These experiments are run on a server with an Intel(R) Xeon(R) CPU E7-4890 v2 @ 2.80GHz CPU, and all summary building and inference is done using a single thread, unless stated otherwise. The reference implementation is available at: `https://github.com/uwdb/color`

6.7.1 Estimator Failure

There are two ways that an estimator can fail to provide meaningful results: 1) time outs, which we define as taking longer than 1 minute to report a result 2) sampling failure, not finding any qualifying samples. In Table 6.2, we show the proportion of queries that result in estimation failure for each dataset and technique. Simpler sampling-based methods (CS, WJ, JSUB, IMPR) face estimation failure even on the smaller query workloads and fail to

find any samples most queries on the larger workloads. Alley achieves much higher success rates due to its sophisticated sampling approach but still fails a significant portion of the time on four datasets. Summary-based methods (BSK++ and SumRDF), on the other hand, time out on over half of queries for four datasets. In contrast, because our approach applies sampling to a highly dense lifted graph, we never experience sampling failure on any query, across all workloads. As the data graph contains hundreds of thousands of edges or more, which are mapped into a lifted graph with at most 32 nodes (colors), the probability that any two colors have an edge connecting them is high.

6.7.2 Accuracy

In Fig. 6.3, we show the relative error of various methods and workloads[3]. Relative error is defined as the ratio of the estimate to the true cardinality. Scores greater than one indicate an overestimate, those under an underestimate. Across workloads, our method, AvgMix32, is unbiased and scales well to larger, more cyclic workloads (yeast, dblp, youtube, eu2005, patents). It even achieves a median error of less than 2 on human, aids, dblp, and eu2005. We also reproduce the high accuracy of WanderJoin and Alley on the G-Care datasets. However, we find that all methods from [116] fail to scale to the larger more cyclic workloads. In particular, we reproduce the finding in [93] that WanderJoin, IMPR, and JSUB overwhelmingly fail to find a positive sample in a reasonable time on these datasets, and that LSS achieves reasonable accuracy on a variety of workloads. Further, SumRDF times out on all larger queries.

Cardinality Bounds When comparing the cardinality bounding methods, BSK and MaxQ64, we find that applying a mixture of coloring methods rather than hash coloring produces up to 10^6 times lower error. Notably, because we apply sampling to this method, we guarantee a linear runtime for all queries in exchange for a less principled cardinality bound. However, across all workloads, this never results in significant underestimation.

[3]In these graphs, outliers (¿2 std. deviations) are shown as points. The inner box shows quartiles, and the whiskers are the max/min non-outlier values. Further, sample failure is treated as an estimate of 1.

Coloring Methods In Fig. 6.9, we examine the effect of choosing different colorings on the accuracy of the average degree estimator. The hash coloring performs the worst across all benchmarks which is expected because it does not take the labels or graph topology into account. On the other hand, the quasi-stable coloring algorithm from [85] works quite well on most datasets with the exception of dblp because it does not account for the distribution of labels. Overall, the mixed coloring performs well across datasets because it can supplement the topological colorings with a label-based colorings, accounting for both sources of error.

In this figure, we also vary the number of colors that we use for the lifted graph. Interestingly, increasing the number of colors kept does not straightforwardly improve accuracy. This is because we always use 500 samples during inference. As the lifted graph gets larger, the sampling procedure has a larger impact. Given this, we find that the optimal coloring uses either 32 or 64 colors.

6.7.3 Inference Latency

Fig. 6.4 shows the distribution of inference latencies for each method across workloads. We can see that the inference latency of COLOR lies in the middle of the competing methods across workloads. On the smaller, less cyclic queries of human, it achieves a median latency of around 10^{-4} seconds due to partial aggregation, and on the more complex queries of patents it has a median latency of $\sim$ .05 seconds via sampling. Similar to prior work, we select a one minute timeout because cardinality estimators may be called many times during the query planning phase [146, 93, 160, 117].

Compared to other graph summarization methods, SumRDF and BSK++, our framework scales far better to larger queries. The former methods timeout on queries of even moderate size as they consider the exponential number of potential colorings of the query. This occurs even when using partial aggregation (as in BSK++), demonstrating the necessity of sampling to achieve consistent latencies.

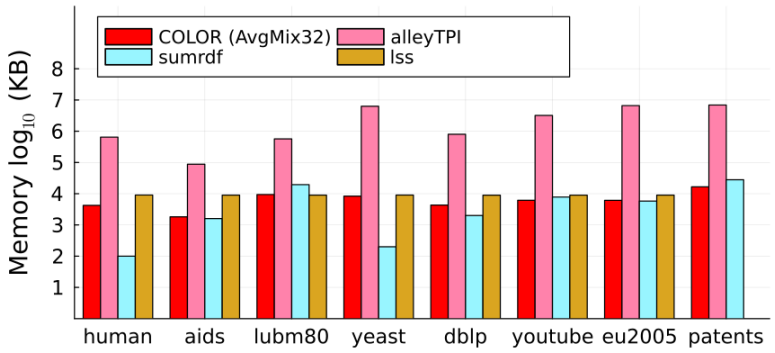


Figure 6.7: Statistics Size

6.7.4 Statistics Size & Build Time

Graph summarization approaches allow for a smooth tradeoff between accuracy and size/build time; a more granular summary of the graph will take more space but more accurately capture the structure of the data graph. Even a compact summary ($<$ 20MB) can be highly accurate as shown by Fig. 6.7. This comes from the fact that we store our summary sparsely. If two colors do not share an edge with a particular attribute, then we don't explicitly store any degree statistic about this combination. A better coloring can actually result in a more compact summary because many of these combinations won't occur if the nodes have been properly partitioned. On Human and Lubm80, AlleyTPI's pattern index requires 600 and 300 MB, respectively. LSS takes a considerable time to train across all datasets. This is the average training time over all folds of the cross validation, and, crucially, it does not include the time necessary to collect the training data (i.e. to run the queries and calculate a true cardinality).

With respect to build time, our summary construction scales linearly in the size of the data graph. For smaller graphs like human, aids, and yeast, summary construction takes less than 20 seconds, and it smoothly increases as the data graph gets larger. In Fig. 6.10, we confirm this by generating erdos-renyi graphs of varying sizes and recording the average time to build the lifted graph over 20 trials.

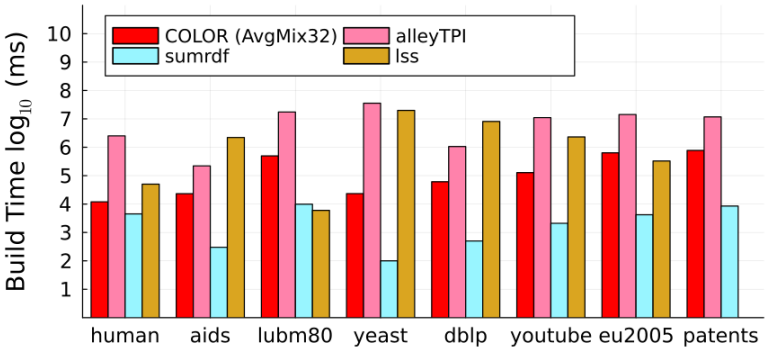


Figure 6.8: Build Time

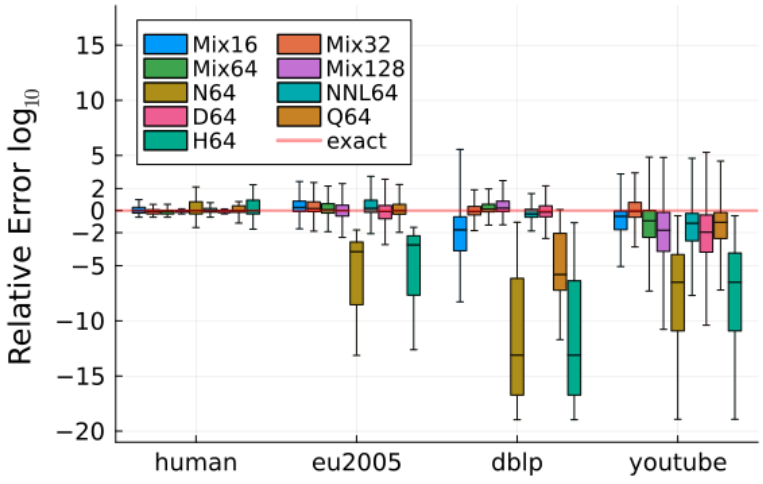


Figure 6.9: Relative Error by Coloring Method

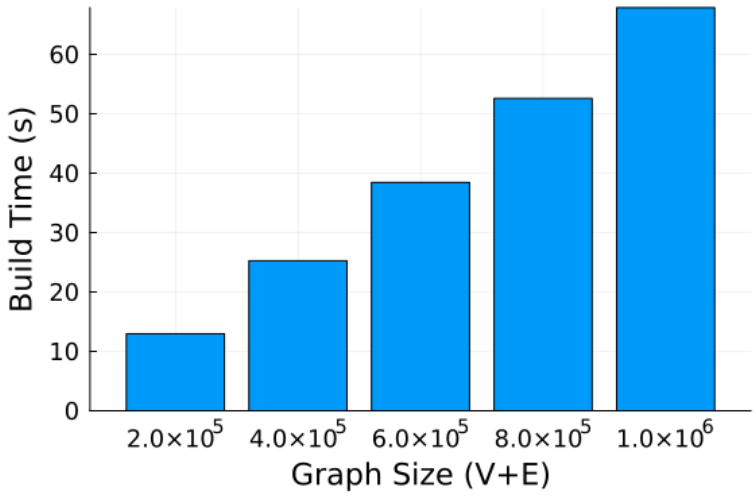


Figure 6.10: Construction Scaling

6.7.5 Updates

In Fig. 6.11, we evaluate the effectiveness of our method for handling updates (Sec. 6.6.3). To do this, we randomly partition the edges of the human dataset into an initial data graph and an ensuing set of updates, and we construct a lifted graph using the former then update it by adding one edge/vertex at a time based on the latter. Intuitively, when more of the graph is provided at the beginning, the lifted graph will be more accurate because it can take advantage of that knowledge when coloring the graph. We find that, as more of the lifted graph is updated, the accuracy remains very consistent and degrades to the traditional independence estimator. So, a full rebuilding of the lifted graph can occur very infrequently and be amortized over many updates. Due to the small size of the lifted graph, each update is very fast. The latency for vertex updates was roughly 0.4 ms while for edge updates it was roughly 0.1 ms.

6.7.6 Micro-Benchmarks

Path Closure Probabilities To show the importance of tracking cycle probabilities, we show the relative error as we vary the size of cycles whose probabilities we track in Fig.

6.12. At size 1, we do not track any cycle closure probabilities. So, when we close a cycle in the query graph, we scale down the estimate based on the uniform probability of an edge existing, i.e. $|E|/|V|^2$. When we begin to store larger cycle sizes, we quickly see the error decrease, and underestimation, in particular, is significantly reduced. These results validate the necessity of handling cycle closure with a more complex method than the standard independence estimator.

Partial Aggregation Fig. 6.13 demonstrates the effects of partial agg. and sampling. Using the Youtube dataset, we study the effect of partial agg. and sampling on inference latency across queries with different pathwidths. Recall that pathwidth is a measure of cyclicity where pathwidth 1 is acyclic and higher pathwidth queries are denser. Without partial agg., estimation times out (¿1 min) when pathwidth exceeds 2. Higher pathwidth queries are larger and the naive approach is exponential w.r.t. the query size. Using only partial agg., estimation times out when pathwidths exceed 4. Lastly, when sampling is applied, the inference latency becomes linear in the size of the query and unrelated to the pathwidth. The results demonstrate that partial agg. achieves speedups without affecting accuracy and sampling can achieve consistent, fast inference.

Inference Sampling In Fig. 6.14, we vary sample size to demonstrate the efficacy of our sampling method. Using a very small sample results in significant underestimation, but even a moderate number of samples quickly converges to the accuracy of a large number of samples. Further, this figure shows that importance sampling speeds up this convergence significantly over a naive uniform sample. For example, the performance at 250 samples for importance sampling is roughly equal to the accuracy of uniform sampling at 1000 samples.

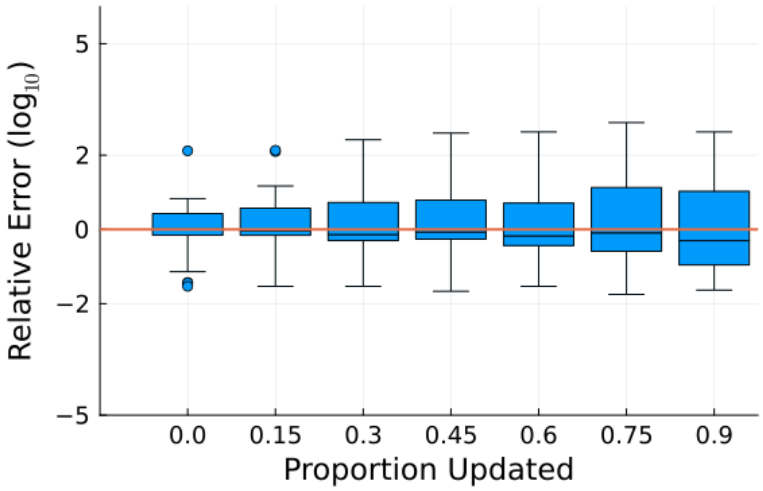


Figure 6.11: Relative Error vs Proportion Updated (Human)

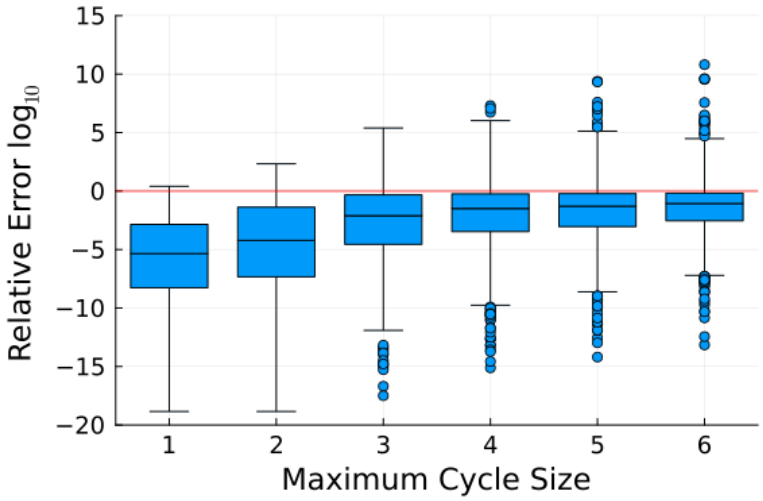


Figure 6.12: Relative Error vs Max Cycle Length Stored (Youtube)

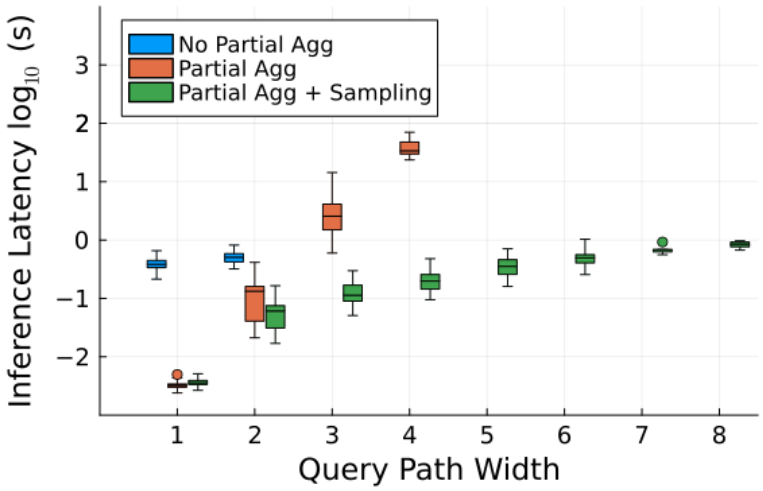


Figure 6.13: Inference Time vs Query Pathwidth (Youtube)

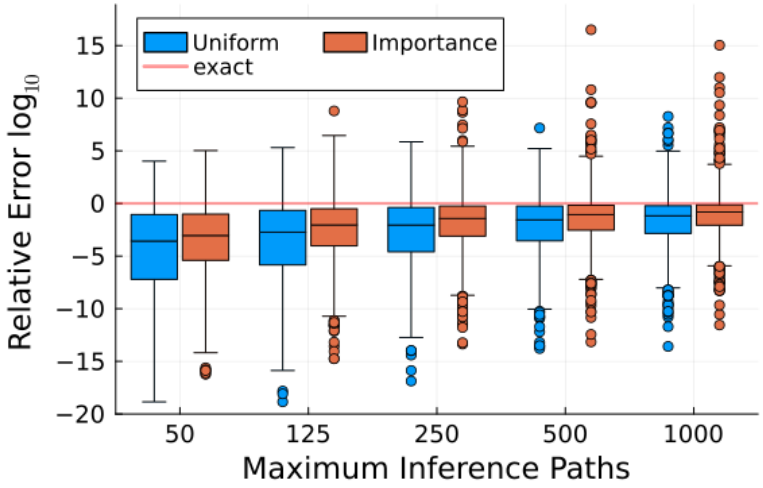


Figure 6.14: Relative Error vs Samples (Youtube)

6.8 Conclusion

This chapter presented COLOR, a framework for producing lifted graph summaries from colorings. We defined inference over the lifted graph for acyclic and cyclic queries, developing optimizations to accelerate estimation. We empirically validated COLOR's superior performance on eight benchmarks and compared to state-of-the-art methods. COLOR is up to $10^3\times$ more accurate than the baselines and stands out for never experiencing estimation failure. Further, it gracefully handles updates, degrading to only $3\times$ worse error when half of the data is replaced.

Chapter 7

GALLEY

This chapter builds on the data-aware complexity analysis techniques presented in the previous chapters and applies them to perform data-aware program optimization for sparse tensor programs. Unlike databases, existing systems for executing sparse tensor programs have lacked a deep optimization framework and instead relied on the user to make smart performance decisions[1, 118, 73]. The key insight in this work is that prior efforts at optimizing them struggled due to their data-dependent nature; the optimal implementation of a sparse tensor program depends on the specific sparsity structure of the input data. To optimize these programs, we need to apply data-aware optimization techniques and analyze the size of intermediates.

For a deep background on sparse tensor programs and earlier efforts at optimizing them, we direct the reader to Sec. 2.2. However, we provide a brief refresher here for convenience. Traditional tensor processing frameworks are collections of hand-optimized functions over *dense* tensors [1, 118, 73, 8]. To take advantage of sparsity, these frameworks need to provide implementations for every combination of input tensors' formats, resulting in spotty coverage for operations over sparse data [82]. Sparse tensor compilers (STCs) have been developed to automatically produce these implementations [95, 22, 3, 132, 80]. However, these compilers expose even more performance decisions than traditional frameworks, and they similarly lack automatic optimization capabilities.[1]

Example 7.0.1. *Consider Fig. 7.1 which implements logistic regression inference in the language of Finch, an STC[3]. Here, the user must choose the output format for the intermediate* `R` *(line 3). In this case, she chose a* `Dense` *rather than a* `Sparse` *format, which*

[1]Some systems separate declarative and imperative concerns with a scheduling language. However, the user still controls both aspects. For a more detailed description of the prototypical STC, we direct the reader to [95].

```
 0. # Manually specified format for input tensors
 1. FUNC log_regression(X::Dense(Sparse()), θ::Dense())
 2.     # Manually defined intermediate format
 3.     R = Dense()
 4.     # Manually defined loop order
 5.     FOR i=_
 6.         FOR j=_
 7.             # Manually defined iteration algorithm
 8.             R[i] += X[i::iter,j::iter]*θ[j::lookup]
 9.         END
10.     END
11.     P = Dense()
12.     FOR i=_
13.         P[i] = σ(R[i::iter])
14.     END
15. END
```

Figure 7.1: Logistic regression implemented in the language of a sparse tensor compiler.

would be $\approx 10\times$ *slower. Then, the user chooses the loop order (lines 5-6). In this case, she chose i-then-j, which is asymptotically faster than j-then-i because each out-of-order access to* X *requires a full scan of the tensor. Finally, the user picks a merge algorithm for each loop that describes how to iterate through the non-zero indices (line 8). Here,* X *is* `iterated` *through, and each non-zero* j *is* `looked up` *in* θ*. If she chose to* `iterate` *through* θ*, each inner loop would scan the entire vector. Even for a simple kernel, these decisions represent a minefield of potential slowdowns.*

In this chapter, we propose *Galley*, a system for declarative sparse tensor programming. Galley makes algorithmic decisions on the users' behalf, freeing them to focus on the high-level semantics of their program without sacrificing computational efficiency. It accepts input programs written in a declarative language, equivalent to the core of the NumPy API, and automatically produces an optimized STC implementation using the Finch com-

piler [73, 3]. To do so, it first restructures the program into a sequence of aggregation steps, minimizing total computation and materialization costs (Sec. 7.3). It then optimizes each step by selecting the loop order, the optimal formats for all intermediate tensors, and the merge algorithm for each loop (Sec. 7.4). These decisions are all guided by a system for estimating sparsity via statistics on the input tensors (Sec. 7.5). *Galley builds on fundamental principles from cost-based query optimization while developing new techniques that are specific to producing optimized code for sparse tensor compilers.*

Designing Galley required overcoming three key challenges. First, the high-level optimization requires a complex rewriting of the original program which must respect the algebraic properties of the program. We addressed this by introducing a novel extension of the FAQ framework that can handle *arbitrary sparse tensor programs* [88]. Second, STCs provide a vast design space for kernel implementations which makes the per-aggregate optimization challenging. Galley's physical optimizer searches through this space efficiently by separating concerns (loop order, output format, and intersection algorithm) and applying branch-and-bound optimization. Lastly, the computational cost of a sparse tensor program depends on the data distribution of the input data which complicates the optimization process. Galley produces these data-dependent cost estimates by leveraging the similarity of sparsity estimation and relational cardinality estimation. By overcoming these challenges, we have attempted to design Galley for a broad set of use cases ranging from sparse ML to graph algorithms and scientific simulations. To this end, we have incorporated Galley into the PyData/Sparse library which implements the full NumPy API for sparse arrays [2, 73].

Example 7.0.2. *Let A, B, and C be sparse matrices, and suppose that you want to compute the matrix chain ABC. Because they do not consider the sparsity of the inputs, traditional systems will always perform this in the order $(AB)C$ where the intermediate, AB, is stored as a sparse matrix. When given this problem, Galley will optimize at runtime for the input's sparsities. This allows it to consider plans that are only efficient for specific inputs. For example, it may: 1) re-order the operations to perform BC before multiplying with A 2) store the intermediate as a dense matrix 3) transpose B to iterate over the shared dimension first. In Sec. 7.6, we show that this can provide a* ***10x*** *speedup over state-of-the-art tensor*

frameworks for this example.

Contributions We claim the following contributions:

- We present Galley, a system for declarative sparse tensor programming (Sec.7.2). Galley is the first system to perform cost based lowering of sparse tensor algebra to the imperative language of sparse tensor compilers, and the first to optimize arbitrary operators beyond $\sum$ and $*$.

- Galley supports a *highly expressive language* for sparse tensor algebra with arbitrary algebraic operators, aggregates within expressions, and multiple outputs (Sec.7.2).

- Galley performs *cost-based logical optimization* with a novel extension of the variable elimination framework to handle arbitrary aggregations and pointwise operators (Sec.7.3). Galley performs *cost-based physical optimization* to determine loop orders, tensor formats, and merge algorithms (Sec.7.4).

- We propose a *minimal interface for sparsity estimation* to guide optimizations and implement two estimators (Sec.7.5).

- We evaluate Galley and show that it is **1-300x** faster than hand-optimized kernels for mixed dense-sparse workloads and **.25-100x** faster than a SOTA database for highly sparse workloads (Sec.7.6).

- We have implemented Galley as part of the PyData/Sparse sparse array project and the Finch tensor compiler[120, 3].

7.1 Background & Connection to RA

7.1.1 Tensor Index Notation

Input to Galley is written in an extended version of Einstein Summation (Einsum) notation that we call *tensor index notation*[6]. Traditional Einsum notation permits a single summation wrapped around a multiplication. For instance, you can describe triangle counting

in a graph with adjacency matrix E_{ij} using the following statement:

$$t = \sum_{ijk} E_{ij} E_{jk} E_{ik}$$

To capture the diverse workloads of tensor programming, we additionally allow the use of arbitrary functions for both aggregates and pointwise operations, nesting aggregates and pointwise operations, and defining multiple outputs. For example, a user could perform logistic regression to predict entities that might be laundering money. Then, they could filter this set based on whether the entities occur in a triangle in the transactions graph. This is represented by $\max_{jk}(E_{ij}E_{jk}E_{ik})$, which is 1 if i occurs in at least one triangle and 0. This can be written in tensor index notation as:

$$L_i = \sigma(\sum_j X_{ij}\theta_j) > .5$$

$$V_i = L_i \cdot \max_{jk}(E_{ij}E_{jk}E_{ik})$$

Tensor compilers like Halide, TACO, and Finch each build off of similar core notations, adding additional structures like FOR-loops to let users specify algorithmic choices [123, 95, 3]. Crucially, the vast majority of operations in array programming frameworks like NumPy can be expressed as operations in tensor index notation. Therefore, though we focus here on this notation, traditional tensor workflows can be captured and optimized in this framework.

7.1.2 Sparse Tensor Compilers

Over the last decade, compiler researchers have developed a series of sparse tensor compilers and shown that they produce highly efficient code for sparse tensor computations[95, 3]. We use this work as our execution engine, so we briefly explain its important concepts below.

Tensor Formats There are many different ways to represent sparse tensors, and the optimal approach depends on the data distribution and the workload. Work in this space has converged on the *fibertree* abstraction for describing the space of formats [95, 139]. In this formalism, a tensor format is a nested data structure resembling the one in Fig. 7.3.

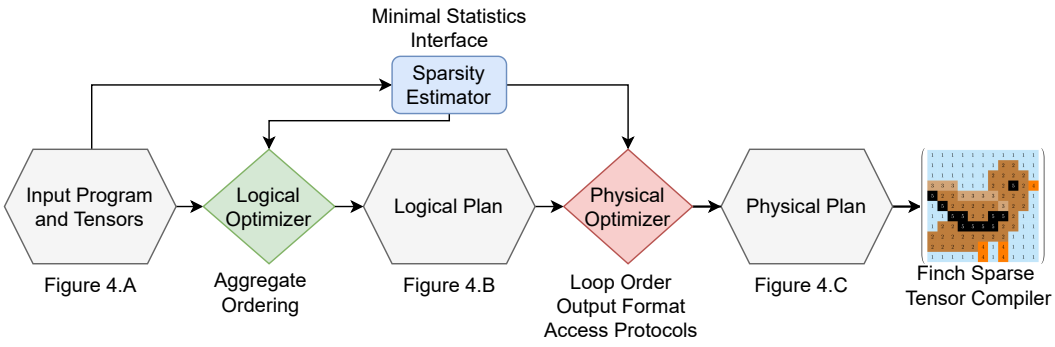


Figure 7.2: Galley overview.

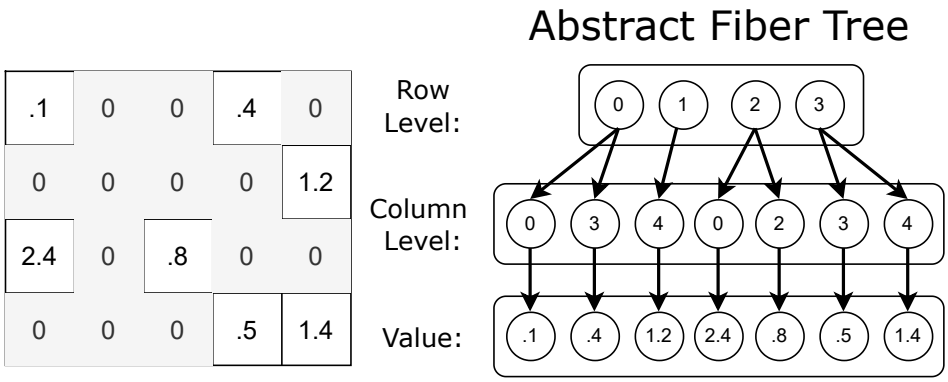


Figure 7.3: Fibertree format abstraction.

Each layer stores the non-fill (e.g., non-zero) indices in a particular dimension, conditioned on earlier dimensions, and pointers to the next dimension's non-fill indices. These layers can be represented in any format that enables iteration and lookup.

In this work, we consider sorted lists, hash tables, bytemaps, and dense vectors, which each perform differently in terms of iteration, lookup, and memory footprint. For example, the compressed sparse row (CSR) is a common format for sparse matrices. It stores the row dimension as a dense vector, where each entry points to the set of non-zero columns for that particular row. This set of non-zero columns is then stored in a sorted list, i.e., in

```
A. Input Program
Plan := Query...    Query := (Name, Expr)
Agg := (Op, Idx..., Expr)   Map := (Op, Expr...)
Expr := Agg | Map | Input | Alias
Input := Tensor[Idx...]  Alias := Name[Idx...]
B. Logical Plan
Plan := Query...   Query := (Name, Agg)
Agg := (Op, Idx..., Expr)   Map := (Op, Expr...)
Expr := Map | Input | Alias
Input := Tensor[Idx...]  Alias := Name[Idx...]
C. Physical Plan
Plan := Query...   Query := (Name, Mat, Idx...)
Mat := (Format..., Idx..., Agg)
Agg := (Op, Idx..., Expr)   Map := (Op, Expr...)
Expr := Map | Input | Alias
Input := Tensor[PIdx...]  Alias := Name[PIdx...]
PIdx := Idx::Protocol
```

Figure 7.4: Query plan dialects.

a compressed sparse format. Importantly, this abstraction requires tensors to be accessed in the order in which they are stored (e.g., row-then-column in the case of CSR), which restricts the set of valid loop orders, as we describe next.

Loop Execution Model The input to a Sparse Tensor Compiler is a high-level domain specific language (DSL); it consists of for-loops, in-place aggregates (e.g., $+=$), and arithmetic over indexed tensors (e.g., $A[i,j]*B[j,k]$). Crucially, the for-loops in these expressions are not executed in a dense manner. Instead, these compilers analyze the input formats

and the algebraic properties of the expression to determine which index combinations will produce non-fill entries. In Fig. 7.1, because 0 is the annihilator of multiplication (i.e., $x * 0 = 0$), only the values of i that map to non-zero entries in X *and* θ are processed. All other index values will return a zero. So, the outer loop is compiled to an iteration over the intersection of the non-zero i indices in X and θ; Fig. 7.3 shows how this is simply co-iteration over the top levels of their formats. The inner loop then iterates over the j indices that are non-zero in $X[i, \_]$, i.e., the non-zero columns that occur in each row.

Merge Algorithms Once the compiler has determined which tensors' non-zero indices must be merged to iterate over a particular index, it can apply several algorithms. All formats enable both ordered iteration and lookup operations; therefore, one algorithm iterates through the indices of all inputs, similar to a merge join, which is highly efficient per operation. However, this algorithm is linear in the total size of all inputs even if one is much smaller than the others. Another method is to iterate through a single input's level and lookup that index in the others. In this work, we take the latter approach, as described in Sec. 7.4.3. We refer to the mode of an individual tensor (such as "iterate" or "lookup") as an *access protocol* and the overall strategy as a *merge algorithm* [4].

7.1.3 The Equivalence of Positive Relational Algebra and Sparse Tensor Algebra

Relational algebra (RA) and sparse tensor algebra (STA) are deeply related from a theoretical perspective. In this section, we outline these similarities and prove a form of equivalence between the positive fragment of RA and STA. To do so, we need to define annihilation as an algebraic property.

Definition 7.1.1. Given a function $f(x_1, \ldots, x_k)$ and a constant c, c is an annihilator of f iff

$$\exists i \; x_i = c \implies f(x_1, \ldots, x_k) = 0 \tag{7.1}$$

Further, we need a bridge from tensors to relations which we denote the sparsity structure and define as follows.

Definition 7.1.2. The sparsity structure of a d-dimensional sparse tensor T is the following d-arity relation where n_j is the size of the jth dimension in T,

$$S_T(I_1, \ldots, I_d) = \{(i_1, \ldots, i_d) \in \mathbb{N}^d \mid T_{i_1,\ldots,i_d} \neq 0,\ i_j < n_j \ \forall\ j\} \tag{7.2}$$

Lastly, we need to a slightly more precise definition of the sparsity structure of STA expressions. Sparse tensor compilers traditionally distinguish between *implicit* zeros and *explicit* zeros. The former occur either in materialized tensors or due to the annihilation property, and the compiler will take advantage of them during computation and to save space when materializing. The latter occur due to interaction of two non-zero values, e.g. $1 - 1 = 0$. Explicit zeros are typically handled the same as any non-zero value by sparse tensor compilers for the sake of computation and materialization. Because we use the sparsity structure to reason about performance, we define the sparsity structure of an STA expression as specifically being the set of implicit zeros not including the explicit ones.

Given those definitions, the following two theorems show how to translate from positive RA to STA and back modulo the non-zero values.

Theorem 7.1.3. *Given an STA expression Φ, you can define a positive RA expression Ψ equal to S_Φ.*

Proof. We begin by viewing the sparse tensor expression as a tree where each inner node is a function, f, or aggregate, τ, and each leaf is an indexed tensor T_{I_T}. Then, we proceed inductively up the tree producing RA expressions for each sub-tree which equal that sub-tree's sparsity structure. The base case of our induction is the leaf nodes whose equivalent RA algebra expression is simply the sparsity structure of the tensor. Next consider the case of an aggregate node $\tau_J(\chi)$ where χ has the free indices I. By assumption, χ has an equivalent RA expression Ψ_χ, so we can define Ψ_τ as

$$\Psi_\tau = \pi_{I \setminus J}(\Psi_\chi)$$

When we encounter a function $f(\chi_1, \ldots, \chi_k)$, we need to consider two cases; 1) 0 is an annihilator of f and 2) 0 is not an annihilator of f. In the former, we can simply join the RA expressions of the children,

$$\Psi_f = \bowtie (\Psi_{\chi_1}, \ldots, \Psi_{\chi_k})$$

For the latter, we have to deal with the fact that unions in relational algebra occur over children with the same attributes. This is not necessarily true in sparse tensor algebra, e.g. $A_i + B_j$ is a valid expression. Fortunately, in tensor algebra every dimension has a bounded domain $D(J) = \{0, \ldots, n_j\}$. Taking advantage of this, we define Ψ_f as,

$$\Psi_f = \bigcup_i (D(I_1) \bowtie \ldots \bowtie D(I_d) \bowtie \Psi_{\chi_i})$$

Because we can produce a valid RA expression for the sparsity structure of each sub-tree in this way, we can produce it for the root as well. □

Theorem 7.1.4. *Given a positive RA expression Ψ, you can define an STA expression Φ such that S_Φ is equal to Ψ.*

Proof. As before, we proceed inductively bottom-up on the RA tree beginning with the relations at the leaves. Without loss of generality, we assume that the attributes in Ψ have a finite active domain contained in the natural numbers. Given this, we can transform every d-ary relation of Ψ into a d-dimensional boolean tensor whose entries are 1 if there is a corresponding tuple in the relation and 0 otherwise. Computing the sparsity structure of this tensor reproduces the original relation. If we encounter a union operator $\cup(\chi_1, \ldots, \chi_k)$, then the equivalent STA sub-tree is:

$$\Phi_\cup = \Phi_{\chi_1} + \cdots + \Phi_{\chi_k}$$

If we encounter a join operator$\bowtie (\chi_1, \ldots, \chi_k)$, then we insert a boolean AND function over the children's equivalent STA sub-trees:

$$\Phi_\bowtie = \Phi_{\chi_1} * \cdots * \Phi_{\chi_k}$$

Lastly, if we encounter a projection $\pi_I(\chi)$ where χ has the attributes J, then we insert an OR aggregate over χ's STA sub-tree:

$$\Phi_{\pi_I} = \bigvee_{J \setminus I} \Phi_\chi$$

□

7.2 Galley Overview

We now provide a high-level view of Galley. We show how it transforms an input program to a logical plan then to a physical plan that is executed by an STC, as illustrated in Fig. 7.2. These steps are each represented by a dialect of our query plan language, whose grammar is defined in Fig. 7.4. In the following discussion, we use this grammar as a guide to show how our example program, i.e., logistic regression, would be transformed through these steps.

7.2.1 Input Program Space

The input program dialect is equivalent to the tensor index notation defined in Sec. 7.1.1. Pointwise functions such as $A_{ij} * B_{jk}$ are represented with `Map`. Aggregates such as $\sum_i$ are denoted by `Agg`. Each assignment is a `Query`, and previous assignments are referenced with an `Alias`. Crucially, the `Op` terminal used in both `Map` and `Agg` can be any user defined function (e.g. $f(x, y) = sin(1 + x * y)$) as long as it accepts the correct number of arguments (i.e. the number of expressions in the `Map` and two arguments in `Agg`). Galley takes advantage of properties of these functions during optimization, specifically distributivity, commutativity, associativity, identity, idempotency, and the existence of an annihilator. Further, users can declare these properties to Galley at runtime. This extensibility is a benefit of Galley's formal framework. Lastly, `Idx`s are named symbols (e.g. i, j), and `Tensor`s are memory-resident input tensors. Our logistic regression example from Fig. 7.1 is defined in this dialect as

`Query(P, Map(`σ`, Agg(+, j, Map(*, X[i,j],` θ`[j]))))`

Note that this notation is compatible with array APIs like Numpy that do not have named indices. Operations like 'matmul' can be automatically mapped into this language by generating index names for inputs on the fly and renaming whenever operations imply equality between indices.

7.2.2 Logical Plan

The first task in our optimization pipeline, handled by the logical optimizer, breaks down the input program into a sequence of simple aggregates. This is enforced by converting the input program (7.4.A) to a logical plan (7.4.B). This dialect is a restriction of the input dialect, where each query contains a single aggregate statement that wraps an arbitrary combination of `Map`, `Input`, and `Alias` statements. Intuitively, each logical query corresponds to a single STC kernel that produces a single intermediate tensor, but it does not specify details like loop orders and output formats. To perform this conversion soundly, each input query must correspond to a logical query, which produces a semantically equivalent output. To do this efficiently, Galley must minimize the total cost of all queries in the logical plan.

Our logistic regression program above is not a valid logical plan because the outer expression is a pointwise function not an aggregate. However, it can be translated into the following logical plan

```
Query(R, Agg(+, j, Map(*, X[i,j], θ[j])))
Query(P, Agg(no-op, Map(σ, R[i])))
```

In this plan, the first query isolates the sum over the j index, while the second query performs the remaining sigmoid operation on the result. Note that the latter query uses a no-op aggregate to represent an element-wise operation while conforming to the logical dialect.

7.2.3 Physical Plan

Given the logical plan, Galley's physical optimizer determines the implementation details needed to convert each logical query to an STC kernel. Specifically, it defines the loop order of each compiled kernel, the format of each output, and the merge algorithm for each index. As above, this is expressed by converting the logical plan to a physical plan described in the most constrained dialect. To avoid out-of-order accesses, we require that the index order of inputs and aliases are concordant with the loop order, so the physical optimizer may insert additional queries to transpose inputs. Therefore, each logical query corresponds to *one or more* physical queries.

Using this language, we can precisely express the program from Fig. 7.1 as follows, where `it` means iterate and `lu` means lookup.

```
Query(R,Mat(dense,i,Agg(+,j,Map(*, X[i::it,j::it],
                                          θ[j::lu]))), i, j)
Query(P,Mat(dense,i, Map(σ, P1[i::it])), i)
```

The first query computes the sum by iterating over the valid i indices for `X`, iterating over the j indices in the intersection of `X[i,_]` and θ, and materializing (hence `Mat`) their product in a dense vector over the i indices. The second query runs over this output and applies the sigmoid function, returning the result as a dense vector.

7.2.4 Execution

Once Galley has generated a physical plan, the execution is very simple. For each physical query, it first translates the expression into an STC kernel definition and calls the STC to compile it. Then, Galley injects the tensors referenced by inputs and aliases and executes the kernel, storing the resulting tensor in a dictionary by name. After all queries have been computed, it returns the tensors requested in the input program by looking them up in this dictionary.

7.3 Logical Optimizer

Given the plan dialects above, we now describe the logical optimizer, which receives an input program (Dialect 7.4.A) and outputs a semantically equivalent *logical plan* (Dialect 7.4.B). Specifically, the logical optimizer converts each query in the input program to a sequence of logical queries, where the last query produces the same output as the input query. There are many valid plans, and the optimizer searches this space to identify the cheapest one. We now briefly define "cheapest" in this context before outlining the complex space of logical plans that are considered. Finally, we explain the algorithms that we use to perform this search.

7.3.1 Normalization & Pointwise Distributivity

The first step in logical optimization is to normalize the input program with a few simple rules that we apply exhaustively: (1) merge nested `Map` operators, (2) merge nested `Agg` operators, (3) lift `Agg` operators above `Map` operators, when possible, and (4) rename indices to ensure uniqueness. Applying these rules compresses the input program and makes our reasoning simpler in later steps by ensuring that operator boundaries are semantically meaningful.

Next, we consider whether to distribute pointwise expressions. Doing so may or may not yield a better plan because it both makes operations more sparse and produces larger expressions.

Example 7.3.1. *Consider the following expression which computes the loss function for the alternating least squares (ALS) algorithm and its distributed form:*

$$\sum_{ij}(X_{ij} - U_i V_j)^2 = \sum_{ij} X_{ij}^2 - 2\sum_{ij} X_{ij} U_i V_j + \sum_i U_i^2 \sum_j V_j^2$$

If all inputs are dense, the non-distributed form is more efficient because it results in fewer terms and has the same computational cost per term. However, if X_{ij} is sparse and U_i, V_j are dense, then the distributed form is more efficient because all terms can be computed in time linear w.r.t. the sparsity of X_{ij}. Note that the squaring operation here is a pointwise function, not a matrix multiplication.

To take advantage of this potentially asymptotic performance improvement, Galley performs a greedy search for the optimally distributed expression. At each step, it considers all single applications of distributivity in the expression. It then runs variable elimination for each (described later in this section) and computes the cost an optimal logical plan. If applying distributivity improved on the cost of the original expression, it continues. If not, it returns the optimal logical plan discovered so far. Lastly, we additionally consider the expression derived from applying distributivity exhaustively.

7.3.2 Cost Model

Overall, Galley's logical optimizer attempts to minimize the time required to execute the logical program. Because logical queries do not correspond to concrete implementations, our logical cost model aims to approximate this time without reference to the particular implementation that the physical optimizer will eventually decide on. This approximation considers two factors: (1) the number of non-fill entries in the output tensor and (2) the amount of computation (i.e., the number of FLOPs) needed to produce the output. The former corresponds to the size of the tensor represented by `Agg`, $nnz(\texttt{Agg})$, and the latter corresponds to the tensor size represented by the `MapExpr` within, $nnz(\texttt{MapExpr})$. We assume that the inputs are in memory; hence, there is no cost for reading inputs from disk. We then perform a simple regression to associate each cost with a constant, and we add them to produce our overall cost, c, as follows:

$$cost \approx a * nnz(\texttt{Agg}) + b * nnz(\texttt{MapExpr})$$

To estimate $nnz(\texttt{Agg})$ and $nnz(\texttt{MapExpr})$, we use the sparsity estimation framework described in Sec. 7.5.

7.3.3 Variable Elimination

The core of our logical optimizer is an extension of the *variable elimination* (VE) (eq. FAQ) framework [35, 88]. In its original context, this algorithm described a means of marginalization for probabilistic models by removing one variable at a time. When applied to our setting, it allows us to define the logical plan for an input query via an order on the indices being aggregated over, i.e., an *elimination order*. If we are given this order, we can construct a valid logical plan by iterating through the elimination order one index at a time in order to (1) identify the minimal sub-expression needed to aggregate over it, (2) create a new logical query representing the result of that sub-expression, and (3) replace it in the original query with an alias to the result. At the end of this process, the remaining query no longer requires any aggregation and therefore is itself a logical query.

Example 7.3.2. *Consider optimizing the following matrix chain multiplication:*

$$E_{im} = \sum_{jkl} A_{ij} B_{jk} C_{kl} D_{lm}$$

The elimination order jkl corresponds to a left-to-right multiplication strategy because eliminating j from the expression first requires performing the matrix multiplication between A and B. Eliminating k then requires multiplying that intermediate result with C, and so on. Concretely, this produces the following sequence of logical queries:

```
Query(I1, Agg(+, j, Map(*, A[i,j], B[j,k])))
Query(I2, Agg(+, k, Map(*, I1[i,k], C[k,l])))
Query(E, Agg(+, l, Map(*, I2[i,l], D[l,m])))
```

Similarly, the elimination order lkj corresponds to a right-to-left strategy, and the order klj to a middle-first strategy.

Unlike traditional VE for sum-product queries, we support complex trees of pointwise operators and aggregates. This makes identifying minimal sub-queries challenging since we must carefully examine the expression's algebraic properties. Given a strategy for this, the core problem of optimizing VE is to search the space of elimination orders for the most efficient one. In the worst case, this takes exponential time w.r.t. the number of indices being aggregated over. In the following sections, we describe how we identify minimal sub-queries and our search algorithm for finding the optimal elimination order.

7.3.4 Identifying Minimal Sub-Expressions

We now explain how to identify the minimal sub-expressions (MSEs) needed to eliminate an index. In sum-product expressions, the MSE is simply the product of the tensors that are indexed by it. For more complex input programs, we show that identifying MSEs corresponds to a careful traversal down the *annotated expression tree*, examining the algebraic properties of each operation to determine how to proceed.

Annotated Expression Tree. The annotated expression tree (AET) is constructed by examining the nested structure of `Agg`, `Map`, `Input`, and `Alias` nodes in the input query. To do this, Galley first removes all `Agg` nodes and annotates their inner expressions with

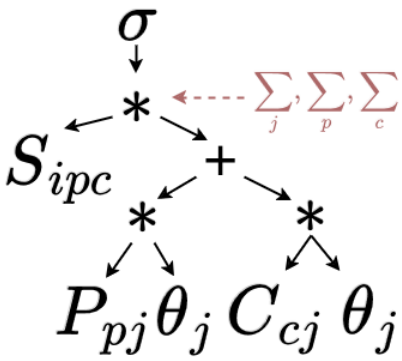


Figure 7.5: Annotated expression tree for $\sigma(\sum_{jpc} S_{ipc}(P_{pj}\theta_j + C_{cj}\theta_j))$

`(Idx, Op)`. It then replaces all `Map` nodes with their operator to get the final tree, where every internal node is a pointwise function and every leaf is either an `Input` or an `Alias`.

Example 7.3.3. *Fig. 7.5 shows the annotated expression tree for logistic regression where the input matrix is defined by a join-like expression* $X_{ij} = S_{ipc}(P_{pj} + C_{cj})$*. Further, Galley has pushed down* θ_j *into this expression. The sigmoid function is the outermost layer of the expression, so it appears at the top of the tree. The summations all occur just inside the sigmoid function, so they annotate the top multiplication operator.*

Given the AET, Galley identifies an index's MSEs by starting at the node where it is annotated and traversing downwards according to the algebraic properties of each internal node. We now describe the traversal rules for functions that are distributive, non-distributive, and commutative with respect to the aggregation operator.

Distributive Functions. When we reach a function that distributes over the aggregate (e.g., $*$ and $\sum$), we examine how many of the children, subtrees of the AET, contain the current index. If one child contains the index, we traverse down that child's branch, i.e., we factor the other children out of the aggregate. If multiple children contain the index, we wrap the sub-tree rooted at that node in the aggregate and return it as our MSE. If the function is commutative and associative, we only include the children that contain the index.

Commutative, Identical Functions. When the node's function is the same as the

aggregate function and is commutative, we can push the aggregate down to each child independently. For example, we can transform the expression $\sum_i A_i + B_i$ into $\sum_i A_i + \sum_i B_i$. For all children that contain the index, we add the result of traversing down its branch to the list of MSEs and replace it with an alias to the result. If a child does not contain the index, then we need to account for the repeated application of the aggregate function. For example, $\sum_i B = N_i * B$ where N_i is the size of the i dimension.

Blocking Functions. A function that does not distribute or commute with our aggregate function is called a *blocking function*. When we reach a blocking function in our traversal, we simply wrap it in our aggregate and return the sub-tree as an MSE. For example, the expression $\sum_j \sqrt{A_{ij} B_{jk}}$ cannot be rewritten as $\sqrt{\sum_j A_{ij} \sum_j B_{jk}}$ because $\sqrt{}$ is a blocking function.

Discussion. Galley builds upon and extends the theoretical FAQ framework for optimizing conjunctive queries with aggregation[88]. This framework explored the optimization of queries with the following form, where each $\bigoplus^{(i)}$ is either equal to or forms a semi-ring with $\otimes$:

$$\bigoplus_{v_1}^{(1)} \cdots \bigoplus_{v_k}^{(k)} F^1_{V_1} \otimes \cdots \otimes F^k_{V_k}$$

Similarly to Galley, the FAQ paper described the optimization problem as selecting an optimal elimination order over the aggregated variables. Though this framework captures many important problems, it lacks the flexibility needed to support a general tensor processing system. Consider a slightly modified version of the SDDMM kernel:

$$\sum_j A_{ik}(B_{ij} + C_{jk})$$

This expression is not an FAQ query because it mixes addition and multiplication in the pointwise expression. Galley extends this framework to accommodate arbitrary pointwise expressions and placement of aggregates within expressions.

7.3.5 Restricted Elimination Orders

Depending on the program structure, the order in which indices can be eliminated might be restricted. This could be due to *non-commutative aggregates* or *aggregate placement*. The

former is when an aggregate wraps another aggregate that it does not commute with. For example, given $\max_i \sum_j A_{ij}$, we must perform the summation before handling the maximum because max and $\sum$ do not commute. The latter is when an aggregate wraps another aggregate but cannot reach it via the traversal described above, e.g., $\sum_i \sqrt{\sum_j A_{ij}}$; in this case, the inner aggregate must be performed first. Collectively, these restrictions form a partial ordering on the index variables that must be respected when we enumerate elimination orders.

7.3.6 Search Algorithms

With the VE approach, we have simplified the complicated issue of high-level optimization to the discrete problem of choosing an optimal order on the aggregated index variables. We start by revisiting our example from Fig. 7.5. The input query is the following,

```
Query(X, Map(σ,
            Agg(+,p,c,j,
                Map(*,S[i,p,c],
                    Map(+,
                        Map(*, P[p,j], θ[j]),
                        Map(*, C[c,j], θ[j])))))
```

The elimination order for this expression is an ordering of the indices $\{p, c, j\}$. Galley's logical optimizer searches through these possible orders to find the most efficient one. In this case, it would choose $[j, p, c]$, resulting in the following logical plan,

```
Query(A1, Agg(+, j, Map(*, P[p,j], θ[j])))
Query(A2, Agg(+, j, Map(*, C[c,j], θ[j])))
Query(A3, Agg(+, p, c, Map(*, S[i,p,c],
                           Map(+, A1[p], A2[c]))))
Query(X, Map(σ, A3[i]))
```

We now present two algorithms to search for that optimal order using the tools described above.

Greedy. The greedy approach chooses the cheapest index to aggregate at each step by finding the minimal sub-query for each index and computing its cost. The cheapest index's minimal sub-query is removed from the expression, appended to the logical plan, and replaced with an alias to the result. This continues until no aggregates remain in the expression.

Branch-and-Bound. The branch-and-bound approach computes the optimal variable order and occurs in two steps. The first step uses the greedy algorithm to produce an upper bound on the cost of the overall plan; the second performs a dynamic programming algorithm. In the dynamic programming step, the keys of the memo table are unordered sets of indices, and the values are tuples containing a partial elimination order, residual query, and cost. The algorithm initializes the table with the empty set and a cost of zero. At each step, it iterates through table entries and attempts to aggregate out another index. It then uses the cost bound from the first step to prune entries from the memo table whose cost exceeds the bound; doing so is valid because costs monotonically increase as more indices are added to the set. At the end of this step, the algorithm returns the index order associated with the full set of indices.

7.4 Physical Optimizer

Each query in the logical dialect roughly corresponds to a single loop nest and materialized intermediate. However, several decisions remain about *how* the kernel is computed, including: (1) the loop order over the indices, (2) the format of the result, and (3) the merge algorithm for each index. The physical optimizer makes these decisions.

7.4.1 Loop Order

The loop order determines that inputs are accessed. An good loop order prunes the iteration space due to early intersection of sparse inputs. Intuitively, this is similar to selecting a variable order for a worst-case optimal join algorithm. Galley's physical optimizer searches the space of loop orders to find one with the minimum cost, defined below.

Cost Model The cost of a loop order is composed of each loop's number of iterations and the cost of transposing inputs to make them concordant with the loop order.

Example 7.4.1. *Consider matrix chain multiplication over three sparse matrices, A, B, and C, where*

$$D[il] = \sum_{jk} A[ij] * B[jk] * C[kl] \tag{7.3}$$

Suppose that A has only a single non-zero entry and that B and C have 5 non-zero entries per column and per row. In this case, the loop order $ijkl$ is significantly more efficient than $lkji$. In the former, the first two loops, over i and j, incur only a single iteration because they are bounded by the size of A. The third and fourth incur 5 and 5^2 iterations, respectively, because there are only 5 non-zero k's per j in B and 5 non-zero l's per k in C. In the latter, the first two loops iterate over the full matrix C despite most of those iterations not leading to useful computation.

Formally, let Q be the pointwise expression in our kernel, and let $Q_{(i_1,\ldots,i_k)}$ be the restriction of that expression to just the index variables $i_1, \ldots, i_k$. Let $\mathbf{A}^{(i_1,\ldots,i_k)}$ be the input tensors that are not concordant with $i_1, \ldots, i_k$. Then, we can define the cost of a loop order as follows,

$$cost(Q, (i_1, \ldots, i_k)) \approx \sum_{j=1}^{k} nnz(Q^{(i_1,\ldots,i_j)}) + \sum_{A \in \mathbf{A}^{(i_1,\ldots,i_k)}} |A|$$

In practice, we further refine this model to take into account the number and kind of tensor accesses at each loop.

Optimization Algorithm. To optimize the loop order, we combine this cost model with a branch-and-bound, dynamic programming algorithm. In the first pass, the optimization algorithm selects the cheapest loop index at each step until reaching a full loop order. This produces an upper bound on the optimal execution cost, which the algorithm uses to prune loop orders in the second step. This step applies a dynamic programming algorithm. Taking inspiration from Selinger's algorithm for join ordering, each key in the DP table is a set of index variables and a set of inputs. The former represents the loops that have been iterated so far, and the latter represents a set of inputs that must be transposed.

7.4.2 Intermediate Formats

Once the loop order has been determined, the physical optimizer selects the optimal format for each query's output. First, Galley sets the order of the indices to be concordant with either the loop order of the kernel where it will be consumed or the order requested by the user. Then, it selects a format for each index (e.g., dense vector, hash table, etc.). Two factors affect this decision: (1) the kind of writes being performed (sequential vs random) and (2) the sparsity of the tensor at this index. The former is important because many formats (e.g., sorted list formats) only allow sequential construction. These formats can only be used if the output indices form a prefix of the loop order.

When considering sparsity, Galley balances the fact that dense formats have better baseline efficiency, while sparse formats are asymptotically more efficient for highly sparse outputs. To describe this trade-off, we hand selected sparsity cutoffs between fully sparse, bytemap, and fully dense formats. To determine a particular output index's format, the physical optimizer first determines the sparsity at this index level and uses our cutoffs to determine which category of formats to consider. Then, it checks whether sequential or random writes are being performed and selects the most efficient format that supports the write pattern.

7.4.3 Merge Algorithms

The final decision the physical optimizer makes concerns the algorithm it will use to perform each loop's intersection. While there are more complex strategies, we adopt instead a minimal approach and select a single input to iterate over for each loop. The physical optimizer then probes into the remaining inputs. It makes this selection by estimating the number of non-zero indices that each input has, conditioned on the indices in the outer loops. This resembles the approach taken in [148] for optimizing WCOJ.

7.4.4 Common Sub-Expression Elimination

Galley takes a straightforward approach to avoiding redundant computation. Once a physical plan has been generated, the right hand side of each physical query is canonicalized and

hashed. When two physical queries result in the same hash, the latter query is removed from the plan and all references to it are replaced with a reference to the result of the former. This is helpful for caching small computations like transpositions, but it is also useful for reducing the overhead of applying distributivity which often results in duplicate sub-expressions.

7.5 Sparsity Estimation

We now describe how Galley performs the sparsity estimation that guides our logical and physical optimizers. First, we explore the subtle correspondence between sparsity and cardinality estimation. We then present a minimal interface for sparsity estimation inspired by this correspondence, after which we examine two implementations of this framework, i.e., the uniform estimator and the chain bound.

7.5.1 Sparsity and Cardinality Estimation

Sparsity estimation is highly related to cardinality estimation in databases. However, translating methods for the latter to the former requires analyzing the algebraic properties of our tensor programs. For example, let A_{ij} and B_{jk} be sparse matrices with a fill value of 0, and let $R_A(I, J)$ and $R_B(J, K)$ be relations that store the indices of their non-zero entries. Assume we are performing the following,

$$C_{ijk} = A_{ij}B_{jk}$$

In this case, the number of non-zero values in C is precisely equal to the size of the conjunctive query

$$nnz(C) = |R_A(I, J) \bowtie R_B(J, K)|$$

The correspondence results from the fact that 0 is the annihilator of multiplication (i.e., $x * 0 = 0 \forall x$), so any non-zero entry ijk in the output must correspond to a non-zero ij in A *and* a non-zero jk in B. Consider the following instead:

$$C_{ijk} = A_{ij} + B_{jk}$$

In this case, a nonzero ijk in the output can result from a non-zero ij in A *or* a non-zero jk in B. In traditional relational algebra, where relations are over infinite domains, this kind of disjunction would result in an infinite relation. However, tensors have finite dimensions, so we can introduce relations that represent the finite domains of each index, e.g., $D_i = \{1, ..., n_i\}$. This lets us represent the index relation of the output as

$$nnz(C) = |(R_A(I, J) \bowtie D_k(K)) \cup (D_i(I) \bowtie R_B(J, K))|$$

Finally, we can translate aggregations to the tensor setting as projection operations. Given the statement

$$C_{ik} = \sum_j A_{ijk}$$

we can express the non-zeros entries of C as

$$nnz(C) = |\pi_{I,K}(R_A(I, J, K))|$$

7.5.2 The Sparsity Statistics Interface

We use our statistics interface to annotate every node of the AST with statistics. Surprisingly, to support sparsity estimation over arbitrary tensor algebra expressions, we only need a few core functions: (1) a constructor, which produces statistics from a materialized tensor for `Input` and `Alias` nodes, (2-3) a function for (non) annihilating `Map` nodes (i.e., those whose children's fill values are the annihilator of its pointwise function), which merges the children's statistics, (4) a function for `Agg`, which adjusts the input's statistics to reflect an aggregation over some set of indices, and (5) an estimation procedure, which estimates the sparsity of a tensor based on its statistics.

7.5.3 Supported Sparsity Estimators

Uniform Estimator

The simplest statistic that can be kept about a tensor is the number of non-fill (e.g., non-zero) entries. The uniform estimator uses only this statistic and assumes these entries are

uniformly distributed across the dimension space. This corresponds to System-R's cardinality estimator with the added assumption that the active domain is the whole dimension for each index [131].

Constructor. This function simply counts the non-fill values in the tensor, $nnz(A)$, and notes the dimension sizes $n_{i_1}, \ldots, n_{i_k}$.

Map (Annihilating). To handle an annihilating pointwise operation, this function calculates the probability that a point in the output was non-fill in all inputs, then multiplies this by the dimension space of the output. For a set of inputs $A^{(1)}_{I_1} \ldots A^{(l)}_{I_l}$ and output C_{I_C}, where each I_j is a set of indices, this probability is

$$nnz(C) \approx \left(\prod_{i \in I_C} n_i \right) \cdot \left(\prod_j \frac{nnz(A_j)}{\prod_{i \in I_j} n_i} \right)$$

Map (Non-Annihilating). To handle an non-annihilating pointwise operation, this function calculates the probability that an entry in the output was *fill* in all inputs. Then, it takes the compliment to get the probability that it was non-fill in all inputs and multiplies this by the output dimension space. Using the preceding notation:

$$nnz(C) \approx \left(\prod_{i \in I_C} n_i \right) \cdot \left(1 - \prod_j \left(1 - \frac{nnz(A_j)}{\prod_{i \in I_j} n_i} \right) \right)$$

Aggregate. Given an input tensor A_I to aggregate over the indices I', this function computes the probability that an output entry is non-fill by calculating the probability that at least one entry in the subspace of the input tensor was not fill:

$$nnz(C) \approx \left(\prod_{i \in I \setminus I'} n_i \right) \cdot \left(1 - \left(1 - \frac{nnz(A_I)}{\prod_{i \in I} n_i} \right)^{\prod_{i \in I'} n_i} \right).$$

Estimate. The estimation function simply returns the current tensor's stored cardinality statistic.

Example 7.5.1. *Suppose A_{ij} and B_{jk} are $100x100$ sparse matrices with $nnz(A) = 1000$, $nnz(B) = 200$, and we want to estimate $nnz(\sum_j A_{ij}B_{jk})$. We first compute $nnz(A_{ij}B_{jk})$ as $100^3 * \frac{1000}{100^2} * \frac{200}{100^2} = 2000$. The fractions are the probability that i, j or j, k was zero in*

A or B, respectively. Next, we factor in the aggregation over j to get $nnz(\sum_j A_{ij}B_{jk}) = 100^2 * (1 - (1 - \frac{nnz(A_{ij}B_{jk})}{100^3})^{100}) \approx 1800$. *Here, the expression* $(1 - \frac{nnz(A_{ij}B_{jk})}{100^3})^{100}$ *is the probability that all entries were zero for a particular* i, k *pair.*

Degree Statistics and the Chain Bound

Galley stores degree statistics by default uses them to compute upper bounds on tensors' sparsities. A degree statistic, denoted as $D_A(X|Y)$, stores the maximum number of non-fill entries in the X dimensions conditioned on the Y dimensions for a tensor A. For example, given a matrix A_{ij}, $D_A(i|j)$ is the maximum number of non-fill entries per column, and $D_A(ij|\emptyset)$ is the total number of non-fill entries in the matrix. This approach follows work in cardinality bounding that has been shown to produce efficient query plans in the relational setting [43, 76, 30].

Constructor. This function first computes the boolean tensor representing the input's sparsity pattern. Then, to calculate each degree statistic, it sums over the X dimensions and takes the maximum over the Y dimensions. The set of degree statistics for a tensor A_I is denoted $\mathcal{D}_{A_I}$.

Map (Annihilating). Annihilating map operations can only reduce the degree for any X, Y pair. Therefore, every input's degree statistics are also valid for the output. If the inputs are $A^(1)_{I_1}, ..., A^(k)_{I_k}$, then the output's statistics are,

$$\mathcal{D}_C = \bigcup_j \mathcal{D}_{A^{(j)}_{I_j}}$$

Map (Non-Annihilating). In this case, Galley extends the degree constraints from each input to cover the full set of indices. Then, it computes degree statistics about the output, C, from the inputs $A^{(1)}_{I_1}, \ldots, A^{(k)}_{I_k}$ by addition:

$$D_C(X|Y) = \sum_j D_{A^{(j)}}(X|Y)$$

Estimator. This function calculates an upper bound (eq. performs sparsity estimation) using the breadth-first search approach described in [31]. Intuitively, each set of indices forms a node in the graph, and each degree constraint is a weighted edge from Y to X. Its

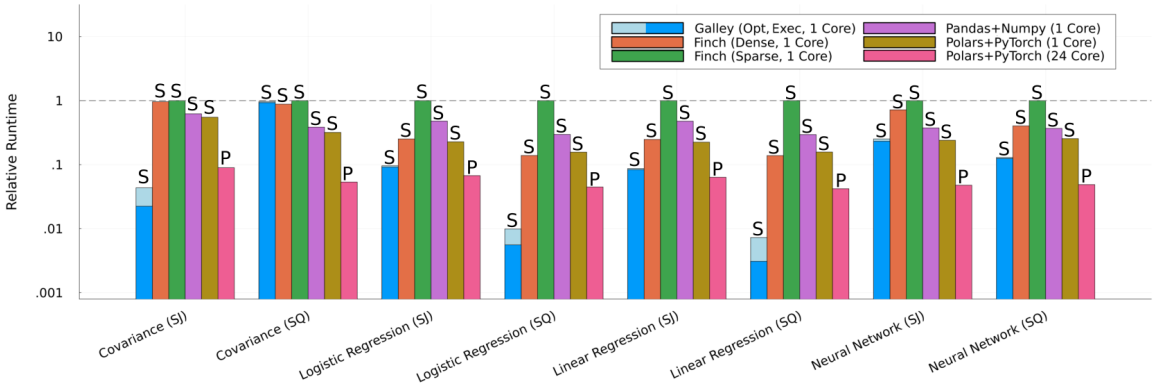


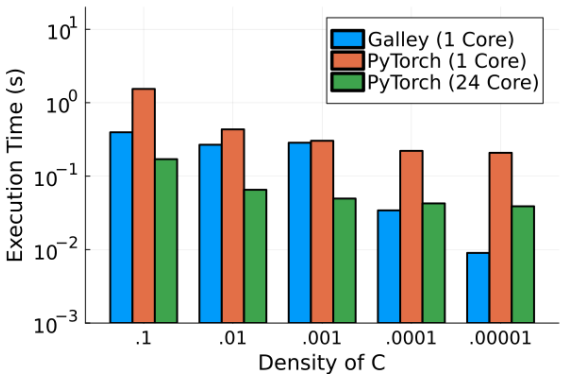
Figure 7.6: ML Inference Over Joins

search begins with the empty set; it then uses a breadth-first search to find the shortest weighted path to the full set of indices I. The product of the weights along this path bounds the number of non-zeros in the result.

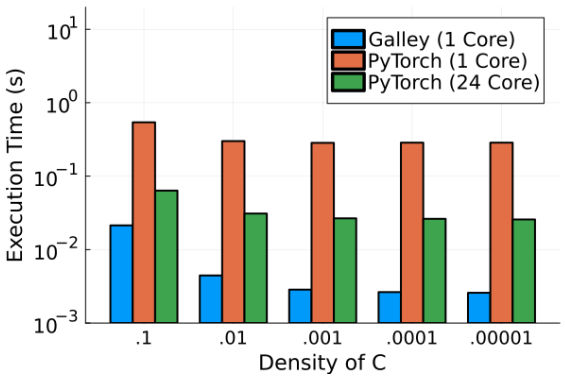
Example 7.5.2. *Suppose* A_{ij} *and* B_{jk} *are* $100x100$ *sparse matrices with* $D_A(ij|\emptyset) = 1000, D_A(i|j) = 10, D_B(jk|\emptyset) = 200$*, and we want to bound* $nnz(\sum_j A_{ij}B_{jk})$*. Because multiplication is an annihilating operation in this case, the degree constraints of* $\sum_j A_{ij}B_{jk}$ *are simply the union of the constraints for* A *and* B*. To get a bound, we start by conditioning on the empty set and try to reach the output's index set,* i, k*, via the constraints, e.g.* $D_B(jk|\emptyset) * D_A(i|j) = 2000$*.*

7.6 Experimental Evaluation

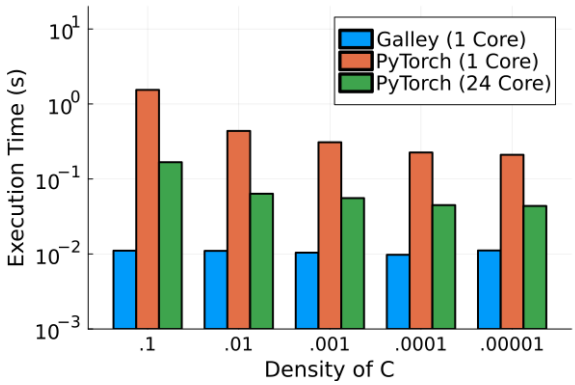
In this section, we evaluate the effectiveness of our optimizer on a variety of workloads: (1) ML algorithms over joins, (2) unstructured sparse linear algebra (3) subgraph counting, and (4) breadth-first search. We choose those workloads because they exercise different aspects of our optimizer on real-world use-cases: ML algorithms over joins require careful logical optimizations over programs with mixtures of dense and sparse inputs and non-linear operators; core linear algebra expressions demonstrate the broad utility of Galley; subgraph counting requires both logical and physical optimization of complex sum-product expressions over highly sparse inputs and demonstrates Galley's advantage over a relational engine even for very sparse workloads; breadth-first search requires careful selection of tensor formats



(a) Matrix Chain Multiplication (ABC)



(b) Elementwise Matrix Multiplication (A*B*C)



(c) Sum of Matrix Chain (SUM(ABC))

Figure 7.7: Linear Algebra Kernels

Table 7.1: Experimental Dataset Sizes

Dataset	**Size**
TPCH (SF .25 - SF 5.0)	.3-6 GB
aids	11 MB
human	1.5 MB
yeast	1.2 MB
dblp	21 MB
youtube	63 MB
Epinions	5.1 MB
Kron	34 MB
LiveJournal	.5 GB
Orkut	1.7 GB
RoadNet	41 MB

over the course of the computation, showing the benefit of physical optimization for even simple computations. Compared to hand-optimized solutions and alternative approaches, Galley is highly computationally efficient while requiring only a concise, declarative input program from the user. Overall, we show that Galley:

- Performs logical optimizations resulting in **1-300**× faster execution for ML algorithms over joins compared to hand-optimized and Pandas implementations and **.5-20**× faster runtime when including optimization.

- Optimizes in a mean time of at most **0.1** seconds for all subgraph counting workloads, with **5-20**× faster median execution than DuckDB.

- Selects optimal formats for intermediates, outperforming both fully dense and sparse formats for 4/5 graphs in a BFS application.

Experiment Setup These experiments are run on a server with an AMD EPYC 7443P Processor and 256 GB of memory. We implemented Galley in the programming language

Julia, and the code is available at `https://anonymous.4open.science/r/Galley-21BF/`. We used the sparse tensor compiler Finch[2] for execution, and all experiments are executed using a single thread. Unless otherwise stated, Galley uses the chain bound described in Sec. 7.5.3 for sparsity estimation. Experiments for all methods are run five times, and the mean execution time is reported. We perform all experiments on a warm cache, and we separately report the compilation and optimization times.

7.6.1 Machine Learning Over Joins

To explore end-to-end program optimization, we experiment with simple ML algorithms over joins. This represents a typical machine learning use case where the feature matrix is constructed from a variety of tables stored within an enterprise database that are joined together before training. Prior work has shown that co-optimizing these mixed RA/LA problems can yield significant benefits [129, 98, 32, 115, 61]. For this, we consider two join queries over the TPC-H benchmark: a star join and a self join, at scale factor 5 and .25, respectively. The star join is expressed as follows, where L, S, P, O, and C are tensors representing the line items, suppliers, parts, orders, and customers tables, respectively:

$$X_{ij} = \sum_{spoc} L_{ispoc}(S_{sj} + P_{pj} + O_{oj} + C_{cj})$$

The non-zero values in S, P, O and C are disjoint along the j axis, so the addition in this expression serves to concatenate features from each source, resulting in 139 features after one-hot encoding categorical features. The self-join query compares line items for the same part based on part and supplier features. In this case, the feature data is a 3D tensor because the data points are keyed by pairs of line items:

$$X_{i_1 i_2 j} = \sum_{s_1 s_2 p} L_{i_1 s_1 p} L_{i_2 s_2 p}(S_{s_1 j} + S_{s_2 j} + P_{pj})$$

We consider a range of ML algorithms: (1) linear regression inference, (2) logistic regression inference, (3) covariance matrix calculation, and (4) neural network inference. We implement two versions of each of these using the Finch compiler. The dense version uses a dense

[2] `https://github.com/FinchTensor/Finch.jl`

Table 7.2: Total Subgraph Counting Execution Time (S)

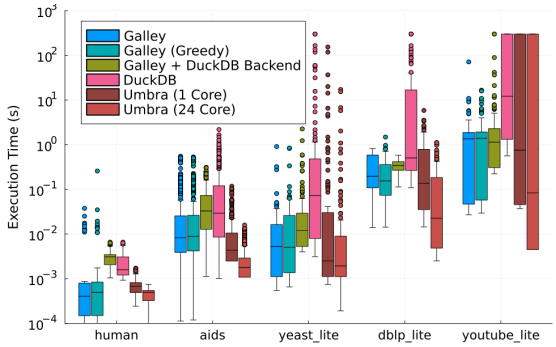
Workload	Galley (Greedy)	Galley+DuckDB	DuckDB	Umbra 1 (24)
human	.17 (.43)	.156	.12	.04 (.02)
aids	32.1 (29.43)	43.32	78.16	7.47 (1.89)
yeast_lite	2.96 (3.85)	8.91	1633	367.28 (51.56)
dblp_lite	32.31 (30.44)	39.31	3294	75.51 (18.23)
youtube_lite	240.24 (219.47)	1591.27	17203	14208 (13866)

feature matrix, and the sparse version uses CSR matrix to compress the one-hot encoding. We also implemented two standard baselines; 1) using Pandas for the joins and Numpy for the linear algebra 2) using Polars for the joins and PyTorch for the linear algebra. The latter supports parallelism, so we've included parallel results for it as well (marking the parallel bars with P and serial bars with S).

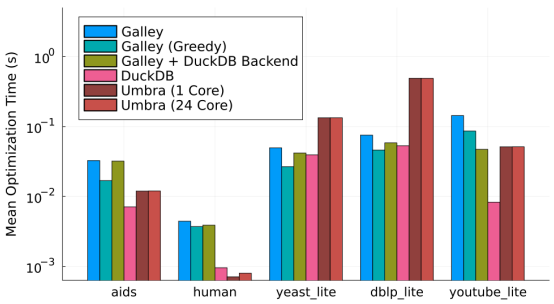
These algorithms stress the ability of Galley to handle complex operators and combinations of sparse and dense inputs. The definitions of the feature tensors combine pointwise multiplication and addition, and algorithms like logistic regression and neural networks wrap these definitions in non-linear operators (e.g. relu and sigmoid) and aggregates. Further, while the line item tensor is highly sparse, both the feature and parameter tensors are moderately to fully dense.

Execution Time. Fig. 7.6 shows that the execution time of Galley's optimized programs is $.5 - 300\times$ faster than the sparse Finch implementation. For the regression/neural network problems, this stems from pushing the multiplication with the parameter vector/-matrix down to the feature matrices. For the covariance calculation over the self join, Galley fully distributes the multiplication over the addition then aggregates away the sparse i_1, i_2 dimensions. This produces small, dense intermediates which can be used to calculate the covariance efficiently.

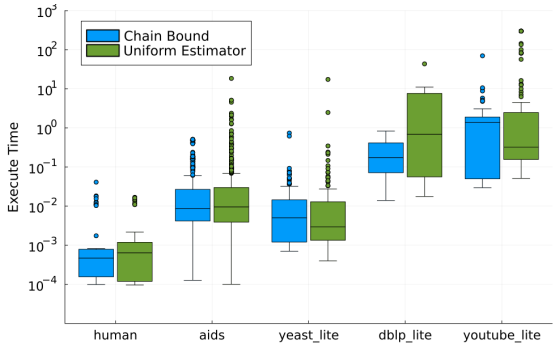
Optimization Time. Fig. 7.6 also shows that Galley's optimizer has a reasonable overhead in this setting. Concretely, optimization takes $.5 - 5.0$ seconds across all workloads.



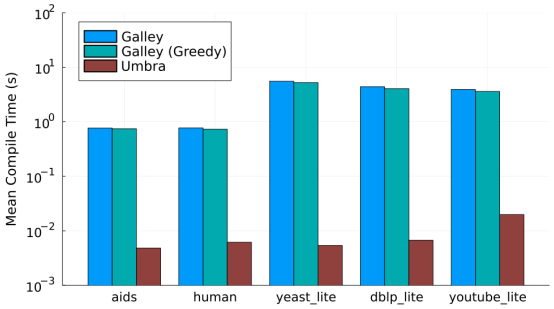
(a) Subgraph Counting Execution Time



(b) Subgraph Counting Optimization Time



(c) Sparsity Estimator Comparison



(d) Subgraph Counting Compilation Time

Figure 7.8: Subgraph Counting Experiments

7.6.2 Linear Algebra Kernels

In Fig. 7.7, we show how Galley can provide significant benefits even for simple workloads without structured data. In these experiments, A, B, and C are 2000×2000 uniformly sparse matrices. A and B have a density of .1, and the density of C varies on the X axis. In the first experiment, Fig. 7.7a, Galley improves on PyTorch's execution in two ways: 1) when the matrices are less sparse, it chooses fully dense formats to store then intermediates and outputs 2) when C is heavily sparse, it uses a right-to-left execution strategy for the chain, i.e. $(A(BC))$. In the second experiment, Fig. 7.7b, Galley is able to fuse the computation when PyTorch is unable to, removing an intermediate materialization. Further, when C is highly sparse, Galley iterates over the non-zeros of C and looks them up in A and B, rather than doing a symmetric intersection algorithm. In the third experiment, Fig. 7.7c, Galley's logical optimizer pushes down the outer summation to A and C first which avoids doing any expensive matrix multiplications. In almost all cases, Galley's optimizations even overcome the benefits of parallelism when PyTorch is provided with 24 cores.

7.6.3 Subgraph Counting

In this section, we stress test Galley's ability to optimize programs with a large number of highly sparse inputs by implementing several sub-graph counting benchmarks. These workloads represent the far end of the complexity and sparsity spectrum for sparse tensor compilers. Suppose you are counting the occurrences of $H(V, E)$ in a data graph G with adjacency matrix M; we can represent the count as,

$$c = \sum_{v_i \in V} \prod_{(v_i, v_j) \in E} M_{v_i v_j}$$

We add sparse binary vector factors for each labeled vertex. We use subgraph workloads from the G-Care benchmark and the paper "In-Memory Subgraph: an In-Depth Study" [137, 116]. We restrict them to query graphs with up to 8 vertices and 24 edges. Because this is a relational workload, we compare it with DuckDB and Umbra, two state-of-the-art modern OLAP databases [122, 110]. The latter is known to be one of the fastest databases for complex joins and aggregations, and we include both serial and parallel exe-

cution [34]. We did not include these systems in our other experiments due to the difficulty of framing the problems in SQL and because prior work has already demonstrated their challenges on pure LA workloads [130]. To separately discern the impact of logical vs physical optimization and our use of Finch, we provide a version of Galley that executes each logical query with a SQL query run on DuckDB. We also provide results for the greedy logical optimizer.

Logical Optimization. Fig. 7.8a shows the execution time for each method and benchmark. The comparison between 'DuckDB' and 'Galley + DuckDB Backend' demonstrates the benefits of Galley's logical optimizer. Galley's logical optimizer breaks down the program into a series of aggregations which minimizes the necessary computation and materialization. This has the largest impact on graphs with high skew like the social network graphs, 'dblp_lite' and 'youtube_lite'. In these cases, pushing down aggregates avoids very large intermediate results. DuckDB hits the 300 second timeout on 56 out of 120 queries in the youtube_lite benchmark, as does Umbra on 46 queries. In contrast, Galley never times out across all workloads.

Physical Optimization. The impact of Galley's physical optimizer can be seen by comparing 'Galley' with 'Galley + DuckDB'. Galley's median execution is up to 8x faster than DuckDB even with the same logical plan. This shows that Galley is selecting efficient loop orders and formats, effectively leveraging STCs.

Optimization Time. Fig. 7.8b shows the mean optimization time for each method on each workload. Galley has a mean optimization time of less than .15 seconds across all workloads, faster than Umbra's optimizer for 2 workloads.

Compilation Time. Because it performs compilation at runtime, Galley incurs a compilation overhead when it invokes an STC kernel. These kernels are cached by Finch, reducing this cost when workloads repeatedly use similar kernels. We show the mean compilation time for each subgraph workload in Fig. 7.8d. On the simpler workloads, which often reuse kernels, this cost is lower. More complex workloads reuse kernels less, significantly increasing compilation time.

Comparing Figures 8 and 9, Galley's optimization overhead is minimal (generally less than 1%) compared to the compilation overhead. Reducing this requires performance im-

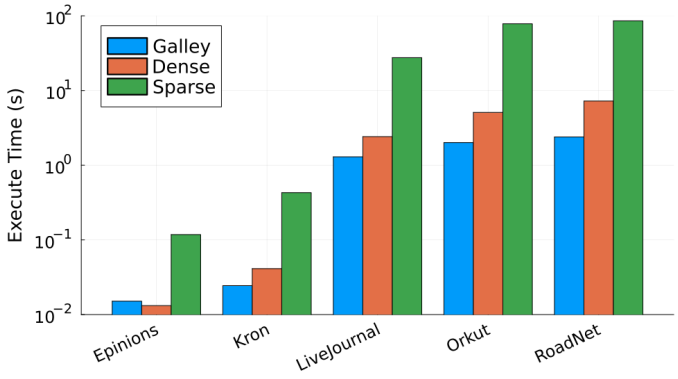


Figure 7.9: BFS Execution Time

provements to the underlying compiler which are out of scope for this work. Fortunately, the Finch project is working to improve this in two ways (1) by caching compilation to disk and (2) by migrating to the MLIR compiler infrastructure. As these improvements are made, Galley will immediately reap the benefits.

Sparsity Estimation. Finally, in Fig. 7.8c, we use the sub-graph matching workloads to compare sparsity estimators and their effect on performance. Across all workloads, we see that the chain bound has significantly better tail performance. This is because it encourages more conservative query plans which better handle correlated and skewed queries/datasets.

7.6.4 Breadth-First Search

To demonstrate the importance of format selection, we implement a breadth-first search algorithm using Galley and hand-coded Finch implementations. Both systems receive a single iteration at a time, and the total execution time is reported. The core computation is a masked sparse matrix times sparse vector to compute the new frontier vector. The main decision is the visited and frontier vectors' formats. The former's sparsity grows monotonically over iterations, while the latter peaks in the middle iterations. We provide two implementations of Finch, using either a sparse or a dense vector for both. Fig. 7.9 shows that Galley's mixture of sparse and dense formats is significantly fastest for 4 of the

5 graphs and is competitive for all graphs. For 4/5 graphs, the total optimization time (not depicted in the figure) is less than .25 seconds. This experiment demonstrates the utility of sparsity-aware format selection, and future work should consider ways to amortize optimization time for iterative workloads.

7.7 Related Work

Galley differs from other work on cost-based optimization for tensor processing due to its targeting of STCs and its expressive input language. SystemDS, formerly SystemML, focuses on end-to-end ML over matrices and tabular data [25, 24, 16]; it takes as input linear algebra (LA) programs and targets a combination of LA libraries and distributed computing via Spark. Later work, SPORES, extended its logical optimizer to leverage relational algebra when optimizing sum-product expressions[147]; their core insight was that LA rewrites, which always match and produce 0-2D expressions, are not sufficient and that optimal rewrites must pass through higher order intermediate expressions. Other related work translated sum-product expressions to SQL to leverage highly efficient database execution engines [23]. These systems can perform well for highly sparse inputs but struggle on mixed dense-sparse workloads. Tensor relational algebra proposes a relational layer on top of dense tensor algebra that provides a strong foundation for automatically optimizing distributed dense tensor computations [157, 27]. The compiler community has made attempts to automatically optimize sparse tensor sum-product kernels based on asymptotic performance analyses[5, 46]. These systems each target a different execution context and focus on different aspects of optimization. Galley expands on this line of work by targeting a new execution engine, proposing novel optimization techniques, and handling a wider range of tensor programs.

7.8 Limitations

We are excited to enrich Galley with new optimizations in the future. Currently, Galley lacks support for complex loop structures (e.g., a single outer FOR loop that wraps multiple inner FOR loops), higher order functions (e.g. matrix inversion) and parallelism. However, we believe that these areas could benefit from cost-based optimization. Similarly, Galley

does not consider hard memory constraints during optimization, but our use of cardinality bound methods provides an avenue for addressing this in future work.

Chapter 8

CONCLUSION & FUTURE WORK

This dissertation has presented a series of new techniques for data-aware complexity analysis and program optimization. In Ch. 3 and Ch. 4, we presented new theoretical techniques for analyzing data-dependent programs. Each of these chapters proposed a statistic that can be easily computed from the data and used to produce tight bounds on the output size of queries which imply bounds on the runtime of programs. Ch. 3 showed that it was possible to produce tight bounds on the size of queries by leveraging the degree sequence of every join column. Ch. 4 dug deeper into the structure of relations and showed that they could be decomposed into multiple pieces with distinct distributions and that this could be used to produce an asymptotically tighter bound. In this work, we went further and developed a new class of asymptotically faster join algorithms by optimizing with full knowledge of these statistics.

Beyond theory, Ch. 5 and Ch. 6 presented systems that computed these bounds for real queries and data. Systems for cardinality bounding need to cope with a stringent set of requirements on the accuracy, memory footprint, and latency, and they need to produce bounds for more complex queries that include selection predicates. To do this, Ch. 5 introduced novel generalizations of histograms and MCV lists to produce degree sequences, and it proposed a new algorithm for efficiently compressing degree sequences using piece-wise functions. Ch. 6 showed that we could leverage stable colorings to produce bounds on the output of queries in the subgraph matching setting. To make this approach space and time efficient, it showed that you could drastically reduce the number of colors while only modestly loosening the bound.

Lastly, in Ch. 7, we presented Galley, a system that leveraged data-aware analysis to perform data-aware program optimization for sparse tensor programs. In this chapter, we showed how to model the cost of these programs as a function of the sparsity of sub-

expressions, and we proved the equivalence between this and the cardinality of database queries. By applying these analyses, Galley is able to optimize complex sparse tensor programs and produce speedups of multiple orders of magnitude over SOTA systems.

Going forward, there is still much work to be done in both data-aware analysis and program optimization.

Space Complexity In this dissertation, our contributions to complexity analysis have been implicitly focused on measuring the time complexity of data-dependent programs. However, when working with big data or on edge devices, the amount of memory that a program requires is a major concern and often results in out-of-memory errors [12, 12]. In future work, we can leverage the techniques that we have established here to analyze the space complexity of data-dependent programs. The goal of this work would be to identify optimal or near-optimal space-time tradeoffs and provide tools for computing the optimal time complexity given a fixed space budget. With these tools in hand, we could build systems for optimizing tensor programs that treat memory constraints as a first-class concern and provide strong guarantees of avoiding out-of-memory errors. Similarly, we could develop database algorithms which avoid the slow-down incurred when large queries need to spill to disk.

Parallelism Modern hardware offers a massive degree of parallelism, whether its multi-core CPUs or GPUs, and the complexity of data skew poses a major challenge to fully utilizing these resources. Depending on the data distribution, different partitioning and load balancing schemes may be more effective. For example, hash-partitioning is extremely fast and results in even worker loads on uniform data, but it struggles if the hashed attributes are highly skewed[150]. There are algorithms which ameliorate this, but with a data-aware analysis we could determine whether these additional overheads are necessary[150, 151, 104, 163]. Building on the work in this dissertation, we could develop data-aware techniques for bounding the tail latency of parallel algorithms, significantly improving the automatic optimization of parallel programs.

New Areas of Data-Dependent Computing This dissertation touched on only a few areas of data-dependent computing, but there are many more that could benefit from this approach. While it began in the database setting, there are many fields of data-

dependent programs that lack the automatic program optimization afforded to SQL users, and providing this optimization would not only improve the productivity of programmers but also result in programs that are more efficient. In particular, text processing via regular expressions and other complex manipulation languages is more important than ever, and its optimization could benefit significantly from knowledge about the character frequency distribution [159]. Similarly, simulations, both in the sciences and in video game engines, have performance characteristics that depend greatly on the distribution and interaction of objects within a scene [105]. Lastly, data processing via spreadsheets is a subtly different model than traditional databases, and this has excluded them from the benefits of relational query optimization [124]. Bringing data-aware optimization to bear on these systems could potentially allow them to scale to far larger datasets.

BIBLIOGRAPHY

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhang. Tensorflow: A system for large-scale machine learning. *CoRR*, abs/1605.08695, 2016.

[2] Hameer Abbasi. Sparse: A more modern sparse array library. In *SciPy*, pages 65–68, 2018.

[3] Willow Ahrens, Teodoro Fields Collin, Radha Patel, Kyle Deeds, Changwan Hong, and Saman P. Amarasinghe. Finch: Sparse and structured tensor programming with control flow. *Proc. ACM Program. Lang.*, 9(OOPSLA1):1042–1072, 2025.

[4] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. Looplets: A Language for Structured Coiteration. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2023, pages 41–54, New York, NY, USA, February 2023. Association for Computing Machinery.

[5] Willow Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 269–285, New York, NY, USA, June 2022. Association for Computing Machinery.

[6] Einstein Albert, W Perrett, and G Jeffery. The foundation of the general theory of relativity. *Annalen der Physik*, 354(7):769, 1916.

[7] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking join and self-join sizes in limited storage. In *PODS*, pages 10–20, 1999.

[8] Edward C. Anderson, Zhaojun Bai, Jack J. Dongarra, Anne Greenbaum, A. McKenney, Jeremy Du Croz, Sven Hammarling, James Demmel, Christian H. Bischof, and Danny C. Sorensen. LAPACK: a portable linear algebra library for high-performance computers. In Joanne L. Martin, Daniel V. Pryor, and Gary R. Montry, editors, *Proceedings Supercomputing '90, New York, NY, USA, November 12-16, 1990*, pages 2–11. IEEE Computer Society, 1990.

[9] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, 50(5):1–40, 2017.

[10] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 739–748. IEEE Computer Society, 2008.

[11] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

[12] Jonathan Bader, Fabian Skalski, Fabian Lehmann, Dominik Scheinert, Jonathan Will, Lauritz Thamsen, and Odej Kao. Sizey: Memory-efficient execution of scientific workflow tasks. In *IEEE International Conference on Cluster Computing, CLUSTER 2024, Kobe, Japan, September 24-27, 2024*, pages 370–381. IEEE, 2024.

[13] Henrik Barthels, Christos Psarras, and Paolo Bientinesi. Linnea: Automatic generation of efficient linear algebra programs. *ACM Trans. Math. Softw.*, 47(3):22:1–22:26, 2021.

[14] Hannah Bast and Björn Buchhold. Qlever: A query engine for efficient sparql+text search. In Ee-Peng Lim, Marianne Winslett, Mark Sanderson, Ada Wai-Chee Fu, Jimeng Sun, J. Shane Culpepper, Eric Lo, Joyce C. Ho, Debora Donato, Rakesh Agrawal, Yu Zheng, Carlos Castillo, Aixin Sun, Vincent S. Tseng, and Chenliang Li, editors, *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, pages 647–656. ACM, 2017.

[15] Douglas Bauer, Haitze J Broersma, Jan van den Heuvel, Nathan Kahl, A Nevo, E Schmeichel, Douglas R Woodall, and Michael Yatauro. Best monotone degree conditions for graph properties: a survey. *Graphs and combinatorics*, 31(1):1–22, 2015.

[16] Sebastian Baunsgaard and Matthias Boehm. AWARE: workload-aware, redundancy-exploiting linear algebra. *Proc. ACM Manag. Data*, 1(1):2:1–2:28, 2023.

[17] Bradley R. Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, Simone Rondelli, Alexander Ryazanov, Michael Schmidt, Kunal Sengupta, Bryan B. Thompson, Divij Vaidya, and Shawn Wang. Amazon neptune: Graph data management in the cloud. In Marieke van Erp, Medha Atre, Vanessa López, Kavitha Srinivas, and Carolina Fortuna, editors, *Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic*

Web Conference (ISWC 2018), Monterey, USA, October 8th - to - 12th, 2018, volume 2180 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[18] Suman K. Bera, Lior Gishboliner, Yevgeny Levanzov, C. Seshadhri, and Asaf Shapira. Counting subgraphs in degenerate graphs. *J. ACM*, 69(3):23:1–23:21, 2022.

[19] Suman K. Bera, Noujan Pashanasangi, and C. Seshadhri. Linear time subgraph counting, graph degeneracy, and the chasm at size six. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, volume 151 of *LIPIcs*, pages 38:1–38:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[20] Suman K. Bera, Noujan Pashanasangi, and C. Seshadhri. Near-linear time homomorphism counting in bounded degeneracy graphs: The barrier of long induced cycles. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2315–2332. SIAM, 2021.

[21] Suman K. Bera and C. Seshadhri. How the degeneracy helps for triangle counting in graph streams. In Dan Suciu, Yufei Tao, and Zhewei Wei, editors, *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*, pages 457–467. ACM, 2020.

[22] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Transactions on Architecture and Code Optimization*, 19(4):50:1–50:25, September 2022.

[23] Mark Blacher, Julien Klaus, Christoph Staudt, Sören Laue, Viktor Leis, and Joachim Giesen. Efficient and portable einstein summation in sql. *Proceedings of the ACM on Management of Data*, 1(2):1–19, 2023.

[24] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. Systemds: A declarative machine learning system for the end-to-end data science lifecycle. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[25] Matthias Boehm, Michael Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, and Shirish Tatikonda. Systemml: Declarative machine learning on spark. *Proc. VLDB Endow.*, 9(13):1425–1436, 2016.

[26] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. On optimizing operator fusion plans for large-scale machine learning in systemml. *Proc. VLDB Endow.*, 11(12):1755–1768, 2018.

[27] Daniel Bourgeois, Zhimin Ding, Dimitrije Jankov, Jiehui Li, Mahmoud Sleem, Yuxin Tang, Jiawen Yao, Xinyu Yao, and Chris Jermaine. Eindecomp: Decomposition of declaratively-specified machine learning and numerical computations for parallel execution. *arXiv preprint arXiv:2410.02682*, 2024.

[28] Marco Bressan and Marc Roth. Exact and approximate pattern counting in degenerate graphs: New algorithms, hardness results, and complexity dichotomies. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 276–285. IEEE, 2021.

[29] Benjamin Brock, Aydin Buluç, Timothy G. Mattson, Scott McMillan, and José E. Moreira. Introduction to graphblas 2.0. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2021, Portland, OR, USA, June 17-21, 2021*, pages 253–262. IEEE, 2021.

[30] Walter Cai, Magdalena Balazinska, and Dan Suciu. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 18–35. ACM, 2019.

[31] Jeremy Chen, Yuqing Huang, Mushi Wang, Semih Salihoglu, and Kenneth Salem. Accurate summary-based cardinality estimation through the lens of cardinality estimation graphs. *SIGMOD Rec.*, 52(1):94–102, 2023.

[32] Lingjiao Chen, Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. Towards linear algebra over normalized data. *arXiv preprint arXiv:1612.07448*, 2016.

[33] Xiaowei Chen and John C. S. Lui. Mining graphlet counts in online social networks. *ACM Trans. Knowl. Discov. Data*, 12(4):41:1–41:38, 2018.

[34] ClickHouse.

[35] Rina Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.

[36] Kyle Deeds, Willow Ahrens, Magdalena Balazinska, and Dan Suciu. Galley: Modern query optimization for sparse tensor programs. *Proc. ACM Manag. Data*, 3(3):164:1–164:24, 2025.

[37] Kyle Deeds and Timo Camillo Merkl. Partition constraints for conjunctive queries: Bounds and worst-case optimal joins. In Sudeepa Roy and Ahmet Kara, editors, *28th International Conference on Database Theory, ICDT 2025, March 25-28, 2025, Barcelona, Spain*, volume 328 of *LIPIcs*, pages 17:1–17:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

[38] Kyle Deeds, Dan Suciu, Magda Balazinska, and Walter Cai. Degree sequence bound for join cardinality estimation. In Floris Geerts and Brecht Vandevoort, editors, *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece*, volume 255 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[39] Kyle Deeds, Dan Suciu, Magdalena Balazinska, and Walter Cai. Degree sequence bounds. *ACM Transactions on Database Systems*, 2025.

[40] Kyle B. Deeds. Safebound: A practical system for generating cardinality bounds, 2023. Software implementation.

[41] Kyle B. Deeds, Diandre Sabale, Moe Kayali, and Dan Suciu. COLOR: A framework for applying graph coloring to subgraph cardinality estimation. *Proc. VLDB Endow.*, 18(2):130–143, 2024.

[42] Kyle B. Deeds, Diandre Sabale, Moe Kayali, and Dan Suciu. COLOR: A framework for applying graph coloring to subgraph cardinality estimation. `https://github.com/uwdb/color`, 2024. Julia implementation.

[43] Kyle B. Deeds, Dan Suciu, and Magdalena Balazinska. Safebound: A practical system for generating cardinality bounds. *Proc. ACM Manag. Data*, 1(1):53:1–53:26, 2023.

[44] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. Graph pattern matching in GQL and SQL/PGQ. In Zachary G. Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 2246–2258. ACM, 2022.

[45] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. Tigergraph: A native MPP graph database. *CoRR*, abs/1901.08248, 2019.

[46] Adhitha Dias, Logan Anderson, Kirshanthan Sundararajah, Artem Pelenitsyn, and Milind Kulkarni. Sparseauto: An auto-scheduler for sparse tensor computations using recursive loop nest restructuring. *CoRR*, abs/2311.09549, 2023.

[47] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Trans. Math. Softw.*, 28(2):239–267, 2002.

[48] Jack Edmonds. Minimum partition of a matroid into independent subsets. *J. Res. Nat. Bur. Standards Sect. B*, 69:67–72, 1965.

[49] Tarek Elgamal, Shangyu Luo, Matthias Boehm, Alexandre V. Evfimievski, Shirish Tatikonda, Berthold Reinwald, and Prithviraj Sen. SPOOF: sum-product optimization and operator fusion for large-scale machine learning. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[50] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. Compressed linear algebra for large-scale machine learning. *Proc. VLDB Endow.*, 9(12):960–971, 2016.

[51] Orri Erling and Ivan Mikhailov. Virtuoso: RDF support in a native RDBMS. In Roberto De Virgilio, Fausto Giunchiglia, and Letizia Tanca, editors, *Semantic Web Information Management - A Model-Based Perspective*, pages 501–519. Springer, 2009.

[52] Ronald Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3):514–550, 1983.

[53] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(8):1162–1175, 2020.

[54] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[55] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1433–1445. ACM, 2018.

[56] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*, pages 1433–1445, 2018.

[57] Amol Ghoting, Rajasekar Krishnamurthy, Edwin P. D. Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 231–242. IEEE Computer Society, 2011.

[58] Lior Gishboliner, Yevgeny Levanzov, Asaf Shapira, and Raphael Yuster. Counting homomorphic cycles in degenerate graphs. *ACM Trans. Algorithms*, 19(1):2:1–2:22, 2023.

[59] Georg Gottlob, Stephanie Tien Lee, and Gregory Valiant. Size and treewidth bounds for conjunctive queries. In Jan Paredaens and Jianwen Su, editors, *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2009, June 19 - July 1, 2009, Providence, Rhode Island, USA*, pages 45–54. ACM, 2009.

[60] Georg Gottlob, Stephanie Tien Lee, Gregory Valiant, and Paul Valiant. Size and treewidth bounds for conjunctive queries. *J. ACM*, 59(3):16:1–16:35, 2012.

[61] Yordan Grigorov, Haralampos Gavriilidis, Sergey Redyuk, Kaustubh Beedkar, and Volker Markl. P2d: A transpiler framework for optimizing data science pipelines. In *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning*, pages 1–4, 2023.

[62] Martin Grohe. *Descriptive Complexity, Canonisation, and Definable Graph Structure Theory*, volume 47 of *Lecture Notes in Logic*. Cambridge University Press, 2017.

[63] Martin Grohe, Kristian Kersting, Martin Mladenov, and Pascal Schweitzer. Color refinement and its applications. MIT Press, 2021.

[64] Martin Grohe and Daniel Neuen. Recent advances on the graph isomorphism problem. *CoRR*, abs/2011.01366, 2020.

[65] Martin Grohe, Gaurav Rattan, and Gerhard J. Woeginger. Graph Similarity and Approximate Isomorphism. In Igor Potapov, Paul Spirakis, and James Worrell, editors, *43rd International Symposium on Mathematical Foundations of Computer Science (MFCS 2018)*, volume 117 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[66] Martin Grohe and Pascal Schweitzer. The graph isomorphism problem. *Commun. ACM*, 63(11):128–134, 2020.

[67] Dimitrios Gunopulos, George Kollios, Vassilis J. Tsotras, and Carlotta Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 463–474. ACM, 2000.

[68] Laura M. Haas. Review - access path selection in a relational database management system. *ACM SIGMOD Digit. Rev.*, 1, 1999.

[69] László Hajdu and Miklós Krész. Temporal network analytics for fraud detection in the banking sector. In Ladjel Bellatreche, Mária Bieliková, Omar Boussaïd, Barbara Catania, Jérôme Darmont, Elena Demidova, Fabien Duchateau, Mark M. Hall, Tanja Mercun, Boris Novikov, Christos Papatheodorou, Thomas Risse, Oscar Romero, Lucile Sautot, Guilaine Talens, Robert Wrembel, and Maja Zumer, editors, *ADBIS, TPDL and EDA 2020 Common Workshops and Doctoral Consortium - International Workshops: DOING, MADEISD, SKG, BBIGAP, SIMPDA, AIMinScience 2020 and Doctoral Consortium, Lyon, France, August 25-27, 2020, Proceedings*, volume 1260 of *Communications in Computer and Information Science*, pages 145–157. Springer, 2020.

[70] S Louis Hakimi and Edward F Schmeichel. Graphs and their degree sequences: A survey. In *Theory and applications of graphs*, pages 225–235. Springer, 1978.

[71] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. Cardinality estimation in dbms: A comprehensive benchmark evaluation. *arXiv preprint arXiv:2109.05877*, 2021.

[72] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. Cardinality estimation in DBMS: A comprehensive benchmark evaluation. *Proc. VLDB Endow.*, 15(4):752–765, 2021.

[73] Charles R. Harris, K. Jarrod Millman, Stéfan van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with numpy. *Nat.*, 585:357–362, 2020.

[74] Olaf Hartig and Jorge Pérez. Semantics and complexity of graphql. In *Proceedings of the 2018 World Wide Web Conference*, pages 1155–1164, 2018.

[75] Mike Heddes, Igor Nunes, Tony Givargis, and Alex Nicolau. Convolution and cross-correlation of count sketches enables fast cardinality estimation of multi-join queries. *Proc. ACM Manag. Data*, 2(3):129, 2024.

[76] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. Simplicity done right for join ordering. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.

[77] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. Deepdb: Learn from data, not from queries! *Proc. VLDB Endow.*, 13(7):992–1005, 2020.

[78] T. C. Hu and M. T. Shing. Computation of matrix chain products. part I. *SIAM J. Comput.*, 11(2):362–373, 1982.

[79] T. C. Hu and M. T. Shing. Computation of matrix chain products. part II. *SIAM J. Comput.*, 13(2):228–251, 1984.

[80] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics*, 38(6):201:1–201:16, November 2019.

[81] Yannis E. Ioannidis and Stavros Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Trans. Database Syst.*, 18(4):709–748, 1993.

[82] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Saleh Ashkboos, and Torsten Hoefler. Sten: Productive and efficient sparsity in pytorch. *arXiv preprint arXiv:2304.07613*, 2023.

[83] Yesdaulet Izenov, Asoke Datta, Florin Rusu, and Jun Hyung Shin. COMPASS: online sketch-based query optimization for in-memory databases. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 804–816. ACM, 2021.

[84] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 631–646. ACM, 2016.

[85] Moe Kayali and Dan Suciu. Quasi-stable coloring for graph compression: Approximating max-flow, linear programs, and centrality. *Proc. VLDB Endow.*, 16(4):803–815, 2022.

[86] Mahmoud Abo Khamis, Phokion G. Kolaitis, Hung Q. Ngo, and Dan Suciu. Bag query containment and information theory. In Dan Suciu, Yufei Tao, and Zhewei Wei, editors, *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*, pages 95–112. ACM, 2020.

[87] Mahmoud Abo Khamis, Vasileios Nakos, Dan Olteanu, and Dan Suciu. Join size bounds using lp-norms on degree sequences. *Proc. ACM Manag. Data*, 2(2):96, 2024.

[88] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: questions asked frequently. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 13–28. ACM, 2016.

[89] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. Computing join queries with functional dependencies. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 327–342. ACM, 2016.

[90] 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 Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 429–444. ACM, 2017.

[91] 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 Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 429–444. ACM, 2017.

[92] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. Learned cardinality estimation: An in-depth study. In Zachary Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1214–1227. ACM, 2022.

[93] Kyoungmin Kim, Hyeonji Kim, George Fletcher, and Wook-Shin Han. Combining sampling and synopses with worst-case optimal runtime and quality guarantees for graph pattern cardinality estimation. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 964–976. ACM, 2021.

[94] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[95] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, 2017.

[96] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.

[97] Robert Kooi. *The Optimization of Queries in Relational Databases*. PhD thesis, Case Western Reserve University, September 1980.

[98] Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1969–1984, 2015.

[99] 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.

[100] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *VLDB J.*, 27(5):643–668, 2018.

[101] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join and XDB: online aggregation via random walks. *ACM Trans. Database Syst.*, 44(1):2:1–2:41, 2019.

[102] David Lion, Adrian Chiu, Michael Stumm, and Ding Yuan. Investigating managed language runtime performance: Why {JavaScript} and python are 8x and 29x slower than c++, yet java and go can be faster? In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 835–852, 2022.

[103] Guy M. Lohman. Is query optimization a "solved" problem? *ACM SIGMOD Blog*, April 2014. Blog post.

[104] Puya Memarzia, Suprio Ray, and Virendra C. Bhavsar. On improving data skew resilience in main-memory hash joins. In Bipin C. Desai, Sergio Flesca, Ester Zumpano, Elio Masciari, and Luciano Caroprese, editors, *Proceedings of the 22nd International Database Engineering & Applications Symposium, IDEAS 2018, Villa San Giovanni, Italy, June 18-20, 2018*, pages 226–235. ACM, 2018.

[105] Farouk Messaoudi, Adlen Ksentini, Gwendal Simon, and Philippe Bertin. Performance analysis of game engines on mobile and fixed devices. *ACM Trans. Multim. Comput. Commun. Appl.*, 13(4):57:1–57:28, 2017.

[106] Christopher Morris, Yaron Lipman, Haggai Maron, Bastian Rieck, Nils M. Kriege, Martin Grohe, Matthias Fey, and Karsten M. Borgwardt. Weisfeiler and leman go machine learning: The story so far. *CoRR*, abs/2112.09992, 2021.

[107] Daniel Müllner. fastcluster: Fast hierarchical, agglomerative clustering routines for r and python. *Journal of Statistical Software*, 53:1–18, 2013.

[108] M. Muralikrishna and David J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In Haran Boral and Per-Åke Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3, 1988*, pages 28–36. ACM Press, 1988.

[109] Inc. Neo4j, Mar 2007.

[110] Thomas Neumann and Michael J. Freitag. Umbra: A disk-based system with in-memory performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[111] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 984–994. IEEE Computer Society, 2011.

[112] Hung Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In Jan Van den Bussche and Marcelo Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 111–124. ACM, 2018.

[113] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In Michael Benedikt, Markus Krötzsch, and Maurizio

Lenzerini, editors, *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 37–48. ACM, 2012.

[114] 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.

[115] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. End-to-end optimization of machine learning prediction queries. In *Proceedings of the 2022 International Conference on Management of Data*, pages 587–601, 2022.

[116] Yeonsu Park, Seongyun Ko, Sourav S. Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. G-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1099–1114. ACM, 2020.

[117] Yeonsu Park, Seongyun Ko, Sourav S Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. G-care: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1099–1114, 2020.

[118] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[119] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, pages 256–276. ACM Press, 1984.

[120] Pydata. Pydata/sparse: Sparse multi-dimensional arrays for the pydata ecosystem.

[121] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proc. VLDB Endow.*, 11(12):1876–1888, 2018.

[122] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1981–1984. ACM, 2019.

[123] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 519–530. ACM, 2013.

[124] Sajjadur Rahman, Kelly Mack, Mangesh Bendre, Ruilin Zhang, Karrie Karahalios, and Aditya G. Parameswaran. Benchmarking spreadsheet systems. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1589–1599. ACM, 2020.

[125] Silvan Reiner and Michael Grossniklaus. Sample-efficient cardinality estimation using geometric deep learning. *Proc. VLDB Endow.*, 17(4):740–752, 2023.

[126] Emma Rollon and Javier Larrosa. On mini-buckets and the min-fill elimination ordering. In Jimmy Ho-Man Lee, editor, *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 759–773. Springer, 2011.

[127] Florin Rusu and Alin Dobra. Sketches for size of join estimation. *ACM Trans. Database Syst.*, 33(3):15:1–15:46, 2008.

[128] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.*, 29(2-3):595–618, 2020.

[129] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*, pages 3–18, 2016.

[130] Maximilian E. Schüle, Thomas Neumann, and Alfons Kemper. The duck's brain: Training and inference of neural networks in modern database engines. *CoRR*, abs/2312.17355, 2023.

[131] 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 Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, pages 23–34. ACM, 1979.

[132] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. Functional collection programming with semi-ring dictionaries. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):89:1–89:33, April 2022.

[133] Wonseok Shin, Siwoo Song, Kunsoo Park, and Wook-Shin Han. Cardinality estimation of subgraph matching: A filtering-sampling approach. *Proc. VLDB Endow.*, 17(7):1697–1709, 2024.

[134] Stack Overflow. 2024 stack overflow developer survey, May 2024. Survey of over 65,000 developers conducted in May 2024.

[135] Giorgio Stefanoni, Boris Motik, and Egor V. Kostylev. Estimating the cardinality of conjunctive queries over RDF data using graph summarisation. In Pierre-Antoine Champin, Fabien Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis, editors, *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, pages 1043–1052. ACM, 2018.

[136] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. Learned cardinality estimation: A design space exploration and A comparative evaluation. *Proc. VLDB Endow.*, 15(1):85–97, 2021.

[137] Shixuan Sun and Qiong Luo. In-memory subgraph matching: An in-depth study. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1083–1098. ACM, 2020.

[138] Shixuan Sun and Qiong Luo. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1083–1098, 2020.

[139] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[140] Brian Tsan, Asoke Datta, Yesdaulet Izenov, and Florin Rusu. Approximate sketches. *Proc. ACM Manag. Data*, 2(1):66:1–66:24, 2024.

[141] Todd L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR*, abs/1210.0481, 2012.

[142] Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, pages 96–106. OpenProceedings.org, 2014.

[143] David Vengerov, Andre Cavalheiro Menck, Mohamed Zaït, and Sunil Chakkappen. Join size estimation subject to filter conditions. *Proc. VLDB Endow.*, 8(12):1530–1541, 2015.

[144] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, Ilhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy. Scipy 1.0-fundamental algorithms for scientific computing in python. *CoRR*, abs/1907.10121, 2019.

[145] Hanchen Wang, Rong Hu, Ying Zhang, Lu Qin, Wei Wang, and Wenjie Zhang. Neural subgraph counting with wasserstein estimator. In Zachary G. Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 160–175. ACM, 2022.

[146] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. Are we ready for learned cardinality estimation? *Proc. VLDB Endow.*, 14(9):1640–1654, 2021.

[147] Yisu Remy Wang, Shana Hutchison, Dan Suciu, Bill Howe, and Jonathan Leang. SPORES: sum-product optimization via relational equality saturation for large scale linear algebra. *Proc. VLDB Endow.*, 13(11):1919–1932, 2020.

[148] Yisu Remy Wang, Max Willsey, and Dan Suciu. From binary join to free join. *SIGMOD Rec.*, 53(1):25–31, 2024.

[149] Max Willsey, Yisu Remy Wang, Oliver Flatt, Chandrakana Nandi, Pavel Panchekha, and Zachary Tatlock. egg: Fast and extensible e-graphs, 2020.

[150] Joel L. Wolf, Daniel M. Dias, Philip S. Yu, and John Turek. An effective algorithm for parallelizing hash joins in the presence of data skew. In *Proceedings of the Seventh International Conference on Data Engineering, April 8-12, 1991, Kobe, Japan*, pages 200–209. IEEE Computer Society, 1991.

[151] Joel L. Wolf, Philip S. Yu, John Turek, and Daniel M. Dias. A parallel hash join algorithm for managing data skew. *IEEE Trans. Parallel Distributed Syst.*, 4(12):1355–1371, 1993.

[152] Jaeyeon Won, Charith Mendis, Joel S. Emer, and Saman P. Amarasinghe. WACO: learning workload-aware co-optimization of the format and schedule of a sparse tensor program. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 920–934. ACM, 2023.

[153] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. Factorjoin: A new cardinality estimation framework for join queries. *Proc. ACM Manag. Data*, 1(1):41:1–41:27, 2023.

[154] Ziniu Wu, Amir Shaikhha, Rong Zhu, Kai Zeng, Yuxing Han, and Jingren Zhou. Bayescard: Revitilizing bayesian frameworks for cardinality estimation. *arXiv e-prints*, pages arXiv–2012, 2020.

[155] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. Neurocard: One cardinality estimator for all tables. *Proc. VLDB Endow.*, 14(1):61–73, 2020.

[156] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. Neurocard: one cardinality estimator for all tables. *arXiv preprint arXiv:2006.08109*, 2020.

[157] Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. Tensor relational algebra for distributed machine learning system design. *Proc. VLDB Endow.*, 14(8):1338–1350, 2021.

[158] Haozhe Zhang, Christoph Mayer, Mahmoud Abo Khamis, Dan Olteanu, and Dan Suciu. Lpbound: Pessimistic cardinality estimation using lp-norms of degree sequences. *Proc. ACM Manag. Data*, 3(3):184:1–184:27, 2025.

[159] Ling Zhang, Shaleen Deep, Avrilia Floratou, Anja Gruenheid, Jignesh M. Patel, and Yiwen Zhu. Exploiting structure in regular expression queries. *Proc. ACM Manag. Data*, 1(2):152:1–152:28, 2023.

[160] Kangfei Zhao, Jeffrey Xu Yu, Qiyan Li, Hao Zhang, and Yu Rong. Learned sketch for subgraph counting: a holistic approach. *VLDB J.*, 32(5):937–962, 2023.

[161] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. Random sampling over joins revisited. In Gautam Das, Christopher M. Jermaine, and Philip A.

Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1525–1539. ACM, 2018.

[162] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. Flat: fast, lightweight and accurate method for cardinality estimation. *arXiv preprint arXiv:2011.09022*, 2020.

[163] Zichen Zhu, Xiao Hu, and Manos Athanassoulis. NOCAP: near-optimal correlation-aware partitioning joins. *Proc. ACM Manag. Data*, 1(4):252:1–252:27, 2023.

Appendix A

DEGREE SEQUENCE BOUNDS APPENDIX

A.1 Discussion of the Framework

Throughout this paper, we restricted the discussion to Berge-acyclic queries. In this section, we briefly review their standard definition and comment on their utility in practice.

First, we review the definition of a Berge-cycle, following [52]. Fix an arbitrary conjunctive query Q, which we view as a hypergraph, and consider its incidence graph, defined as the following bipartite graph: $T \stackrel{\text{def}}{=} (\boldsymbol{R} \cup \boldsymbol{X}, E \stackrel{\text{def}}{=} \{(R, Z) \mid Z \in \boldsymbol{X}_R\})$. A *Berge-Cycle* is a sequence $(R_1, X_1, R_2, X_2, \ldots, R_m, X_m, R_{m+1})$ such that,

1. $R_i \; \forall \; 1 \leq i \leq m$ is a unique relation in $\boldsymbol{R}$.
2. $R_1 = R_{m+1}$
3. $X_i \; \forall \; 1 \leq i \leq m$ is a unique variable in $\boldsymbol{X}$.
4. $m \geq 2$
5. $X_i \in \boldsymbol{X}_{R_i}$ and $X_i \in \boldsymbol{X}_{R_{i+1}} \; \forall \; 1 \leq i \leq m$

It follows immediately that a Berge-cycle is a cycle in the incidence graph, and vice versa. A *Berge-Acyclic* query is one that does not contain a Berge-Cycle, or, equivalently, one whose incidence graph is a tree. This coincides with our definition in Sec. 3.1.

Berge-acyclic queries capture many traditional flavors of queries found in practice, such as chain, star, and snowflake queries. As an illustration of this, we note that many modern cardinality estimation benchmarks include exclusively Berge-Acyclic queries, for example the JOB benchmark [99] and the STATS-CEB benchmark [71]. Additionally, many state of the art approaches to cardinality estimation have this limitation as well. For example,

NeuroCard, FLAT, and BayesCard cannot produce estimates for cyclic queries [156, 162, 154].

While rare, cyclic queries do occur in practice, and, in that case, the degree sequence bound is still useful, as follows. Consider all possible spanning trees of the incidence graph. The query defined by each such spanning tree must produce a larger output than the original query. This can be seen by viewing joins as a set of filters on the Cartesian product of the input tables. A spanning tree operates on the same Cartesian product, but it applies a subset of the cyclic query's filters, and, therefore, it must produce at least as many rows. This means that we can produce a bound on cyclic queries by taking the minimum of the degree sequence bound for each spanning tree. In general, this bound is not tight, but is still an upper bound, and still useful. In contrast, cardinality estimation frameworks like NeuroCard, FLAT, BayesCard aim at *estimating* the cardinality, but an estimate on the cardinality of a spanning tree is useless for estimating the cardinality of the cyclic query. In other words, while our degree sequence bound makes similar restrictions on the queries as other state-of-the-art cardinality estimators, the latter cannot be used on cyclic queries, while ours is still useful.

A.2 Completing Proofs from Sec. 3.2

A.2.1 Proof of Lemma 3.2.6

We prove here Lemma 3.2.6. For that we show the following:

Lemma A.2.1. *Let $\boldsymbol{M} \in \mathbb{R}_+^{[\boldsymbol{n}]}$ be an $\boldsymbol{X}$-tensor s.t. $\boldsymbol{M} \circ \boldsymbol{\sigma}^{-1} \in \mathcal{M}^+_{\boldsymbol{f}(\boldsymbol{X}),B}$ for some $\boldsymbol{\sigma}$. Then there exists an $\boldsymbol{X}$-tensor $\boldsymbol{N} \in \mathbb{R}_+^{[\boldsymbol{n}]}$ satisfying:*

- $||\boldsymbol{M}||_\infty = ||\boldsymbol{N}||_\infty$.
- *For every variable X_p, $\texttt{SUM}_{\boldsymbol{X}-\{X_p\}}(\boldsymbol{N})$ is non-increasing, and is equal, up to a permutation, to $\texttt{SUM}_{\boldsymbol{X}-\{X_p\}}(\boldsymbol{M})$. In particular, $\boldsymbol{N} \in \mathcal{M}^+_{\boldsymbol{f}(\boldsymbol{X}),B}$.*
- *For any non-increasing vectors $\boldsymbol{a}^{(X_p)} \in \mathbb{R}_+^{[n_p]}$:*

$$\boldsymbol{M} \cdot \boldsymbol{a}^{(X_1)} \cdots \boldsymbol{a}^{(X_d)} \leq \boldsymbol{N} \cdot \boldsymbol{a}^{(X_1)} \cdots \boldsymbol{a}^{(X_d)} \tag{A.1}$$

We first show that Lemma A.2.1 implies Lemma 3.2.6. Let $\boldsymbol{M}, \boldsymbol{\sigma}$ be given as in Lemma 3.2.6. We apply Lemma A.2.1 to $\boldsymbol{M} \circ \boldsymbol{\sigma}$, and obtain some tensor $\boldsymbol{N} \in \mathcal{M}^{+}_{\boldsymbol{f}(\boldsymbol{X}),B}$ such that $(\boldsymbol{M} \circ \boldsymbol{\sigma}) \cdot \boldsymbol{a}^{(X_1)} \cdots \boldsymbol{a}^{(X_d)} \leq \boldsymbol{N} \cdot \boldsymbol{a}^{(X_1)} \cdots \boldsymbol{a}^{(X_d)}$. Thus, it remains to prove Lemma A.2.1.

The proof of Lemma A.2.1 consists of modifying $\boldsymbol{M}$ by sorting each coordinate X_p separately. To sort the coordinate X_p, we cannot simply switch some hyperplane $X_p = i$ with another hyperplane $X_p = j$, because that doesn't guarantee (A.1). Instead, we move mass "up" in the tensor along the coordinate X_p. We first illustrate the proof idea with an example.

Example A.2.2. *We want to maximize the product* $\boldsymbol{a}^T \cdot \boldsymbol{M} \cdot \boldsymbol{b}$*, for some* 2×3 *matrix* $\boldsymbol{M}$*, constraint by the degree sequences* $\boldsymbol{f} = (16, 12)$ *and* $\boldsymbol{g} = (14, 11, 3)$*, and element bound* $B = 10$*. Our current candidate is*

$$\boldsymbol{M} = \begin{array}{c} \\ 12 \\ 16 \end{array} \begin{array}{c} \begin{array}{ccc} 14 & 3 & 11 \end{array} \\ \begin{pmatrix} 10 & 1 & 1 \\ 4 & 2 & 10 \end{pmatrix} \end{array}$$

We will modify $\boldsymbol{M}$ *to a new matrix* $\boldsymbol{N}$*, whose row-sums are* $16, 12$*, while* $\boldsymbol{a}^T \cdot \boldsymbol{M} \cdot \boldsymbol{b} \leq \boldsymbol{a}^T \cdot \boldsymbol{N} \cdot \boldsymbol{b}$*, for all* $\boldsymbol{ab}$*. Simply switching rows 1 and 2 doesn't work: indeed, if we set* $\boldsymbol{a}^T \stackrel{def}{=} (1, 0)^T$*,* $\boldsymbol{b}^T \stackrel{def}{=} (1, 0, 0)$ *then* $\boldsymbol{a}^T \cdot \boldsymbol{M} \cdot \boldsymbol{b} = 10$*, but after we switch the two rows,* $\boldsymbol{a}^T \cdot \boldsymbol{N} \cdot \boldsymbol{b} = 4$*. Instead, we move positive mass up, separately in each column. The maximum we can move in each column* k *is* $v_k \stackrel{def}{=} (M_{k2} - M_{k1})^+$*, where* $x^+ \stackrel{def}{=} \max(x, 0)$*. Denoting by* $\boldsymbol{v}$ *the vector* (v_1, v_2, v_3)*, we have:*

$$\boldsymbol{v} = \begin{pmatrix} 0 & 1 & 9 \end{pmatrix}$$

The row-sum in $\boldsymbol{v}$ *is 10, but we only need to move a total mass of 4 (the difference* $16 - 12$*), hence we will move only a* $\theta = 4/10$*-fraction of* $\boldsymbol{v}$*, namely* $\theta \cdot \boldsymbol{v} = \begin{pmatrix} 0 & 0.4 & 3.6 \end{pmatrix}$*, and obtain:*

$$\boldsymbol{N} \stackrel{def}{=} \begin{array}{c} \\ 16 \\ 12 \end{array} \begin{array}{c} \begin{array}{ccc} 14 & 3 & 11 \end{array} \\ \begin{pmatrix} 10 & 1.4 & 4.6 \\ 4 & 1.6 & 7.4 \end{pmatrix} \end{array}$$

Finally, we check that the quantity $\boldsymbol{a}^T \cdot \boldsymbol{M} \cdot \boldsymbol{b}$ *has not decreased:*

$$\begin{aligned}\boldsymbol{a}^T \cdot \boldsymbol{N} \cdot \boldsymbol{b} - \boldsymbol{a}^T \cdot \boldsymbol{M} \cdot \boldsymbol{b} &= \left(\boldsymbol{a}^T \cdot (\boldsymbol{N} - \boldsymbol{M})\right) \cdot \boldsymbol{b} = \\ &= \theta \cdot \begin{pmatrix}(a_1 - a_2)v_1 & (a_1 - a_2)v_2 & (a_1 - a_2)v_3\end{pmatrix} \cdot \boldsymbol{b}\end{aligned}$$

This quantity is ≥ 0 *because* $a_1 \geq a_2$ *and* $\boldsymbol{b} \geq 0$*. Observe that* $\boldsymbol{N}$ *has exactly the same column-sums as* $\boldsymbol{M}$*, and* $||\boldsymbol{M}||_\infty = ||\boldsymbol{N}||_\infty$*, while its rows are sorted in non-increasing order of their sums. If we repeat the same argument for the columns 2 and 3, then we obtain a matrix whose row-sums and column-sums are* $(16, 12)$*, and* $(14, 11, 3)$*, and which upper bounds the quantity* $\boldsymbol{a}^T \cdot \boldsymbol{M} \cdot \boldsymbol{b}$ *for all non-increasing* $\boldsymbol{a}, \boldsymbol{b}$*.*

Before we prove Lemma A.2.1 we need some notations. An *inversion* in a vector $\boldsymbol{f} \in \mathbb{R}^{[n]}$ is a pair $i < j$ such that $f_i \geq f_j$. The vector is non-increasing iff it has the maximum number of inversions, $n(n-1)/2$. If (i, j) is not an inversion, then $\boldsymbol{f} \circ \tau_{ij}$ has strictly more inversions, where τ_{ij} is the permutation swapping i and j. The proof of Lemma A.2.1 follows from the following Proposition:

Proposition A.2.3. *Let* $\boldsymbol{M} \in \mathbb{R}_+^{[\boldsymbol{n}]}$ *be* $\boldsymbol{X}$*-tensor,* $q \in [d]$ *be one of its coordinates, and* $\boldsymbol{f} \stackrel{def}{=} \mathtt{SUM}_{\boldsymbol{X}-\{X_q\}}(\boldsymbol{M})$*. If* $\boldsymbol{f}$ *as no inversion at* (i, j)*, then there exists a* $\boldsymbol{X}$*-tensor* $\boldsymbol{N} \in \mathbb{R}_+^{\boldsymbol{n}}$ *with the following properties:*

- $||\boldsymbol{M}||_\infty = ||\boldsymbol{N}||_\infty$.
- $\mathtt{SUM}_{\boldsymbol{X}-\{X_p\}}(\boldsymbol{M}) = \mathtt{SUM}_{\boldsymbol{X}-\{X_p\}}(\boldsymbol{N})$ *for all* $p \neq q$.
- $\mathtt{SUM}_{\boldsymbol{X}-\{X_q\}}(\boldsymbol{N}) = \boldsymbol{f} \circ \tau_{ij}$.
- *Inequality* (A.1) *holds for any vectors* $\boldsymbol{a}^{(X_p)} \in \mathbb{R}_+^{[n_p]}$ *such that* $\boldsymbol{a}^{(X_q)}$ *is non-increasing,*

By repeatedly applying the proposition to coordinate q we can ensure that $\mathtt{SUM}_{\boldsymbol{X}-\{X_q\}}(\boldsymbol{M})$ is non-increasing; the proof of Lemma A.2.1 follows by repeating this for each coordinate q.

Proof. (of Proposition A.2.3) Define:

$$\delta \stackrel{\text{def}}{=} f_j - f_i$$

The idea of the proof is to modify $\boldsymbol{M}$ by pushing a total mass of δ from the hyperplane j to the hyperplane i. When doing so we must ensure that the new values in hyperplane i don't exceed B, those in hyperplane j don't become < 0, and we don't push more mass than δ.

Denote by $x^+ \stackrel{\text{def}}{=} \max(x, 0)$ and define the following quantities, where $\boldsymbol{k} \in \prod_{p=1,d;p\neq q}[n_p]$, and $\boldsymbol{k}i$ denotes the tuple $(k_1, \ldots, k_{q-1}, i, k_{q+1}, \ldots k_d)$:

$$\delta_{\boldsymbol{k}} \stackrel{\text{def}}{=} M_{\boldsymbol{k}j} - M_{\boldsymbol{k}i} \qquad \theta \stackrel{\text{def}}{=} \frac{\delta}{\sum_{\boldsymbol{k}} \delta^+_{\boldsymbol{k}}} \qquad \varepsilon_{\boldsymbol{k}} \stackrel{\text{def}}{=} \theta \cdot \delta^+_{\boldsymbol{k}}$$

Intuitively, $\delta^+_{\boldsymbol{k}}$ is the maximum amount of mass we could move from $M_{\boldsymbol{k}j}$ to $M_{\boldsymbol{k}i}$ without causing $M_{\boldsymbol{k}j} < 0$ or $M_{\boldsymbol{k}i} > B$ (when $\delta_{\boldsymbol{k}} < 0$ then we cannot move any mass in coordinate $\boldsymbol{k}$), while θ represents the fraction of maximum mass that equals to the desired δ.

To define $\boldsymbol{N}$ formally, we introduce some notations. For each variable $X_p \in \boldsymbol{X}$ and $t \in [n_p]$, let $\boldsymbol{e}^{(X,t)} \stackrel{\text{def}}{=} (0, \ldots, 0, 1, 0, \ldots, 0)$ where the sole 1 is on position t. Given $\boldsymbol{t} = (t_1, \ldots, t_d) \in [\boldsymbol{n}]$ denote by $\boldsymbol{D}^{(\boldsymbol{t})} \stackrel{\text{def}}{=} \bigotimes_{p=1,d} \boldsymbol{e}^{(X_p,t_p)}$. This tensor has value 1 on position $(t_1, \ldots, t_d)$ and 0 everywhere else. Then, we define:

$$\boldsymbol{N} \stackrel{\text{def}}{=} \boldsymbol{M} + \sum_{\boldsymbol{k}} \varepsilon_{\boldsymbol{k}} (\boldsymbol{D}^{(\boldsymbol{k}i)} - \boldsymbol{D}^{(\boldsymbol{k}j)})$$

Notice that $\delta = \sum_{\boldsymbol{k}} \delta_{\boldsymbol{k}} > 0$, by the assumption that $\boldsymbol{f}$ has no inversion at i, j, and $0 < \theta \leq 1$. We start by checking that all entries in $\boldsymbol{N}$ are ≥ 0 and $\leq B$. The only entries that have decreased are those in the hyperplane $X_q = j$:

$$\begin{aligned} N_{\boldsymbol{k}j} =& M_{\boldsymbol{k}j} - \theta\delta^+_{\boldsymbol{k}} \geq M_{\boldsymbol{k}j} - \delta^+_{\boldsymbol{k}} \\ =& M_{\boldsymbol{k}j} - \max(M_{\boldsymbol{k}j} - M_{\boldsymbol{k}i}, 0) = \min(M_{\boldsymbol{k}i}, M_{\boldsymbol{k},j}) \geq 0 \end{aligned}$$

The only entries that have increased are in hyperplane $X_p = i$:

$$\begin{aligned} \boldsymbol{N}_{\boldsymbol{k}i} =& \boldsymbol{M}_{\boldsymbol{k}i} + \theta\delta^+_{\boldsymbol{k}} \leq \boldsymbol{M}_{\boldsymbol{k}i} + \delta^+_{\boldsymbol{k}} \\ =& M_{\boldsymbol{k}i} + \max(\boldsymbol{M}_{\boldsymbol{k}j} - \boldsymbol{M}_{\boldsymbol{k}i}) = \max(\boldsymbol{M}_{\boldsymbol{k}j}, \boldsymbol{M}_{\boldsymbol{k}i}) \leq B \end{aligned}$$

By Eq. (3.6) we have:

$$\mathtt{SUM}_{\boldsymbol{X}-\{X_p\}}\left(\boldsymbol{D}^{\boldsymbol{t}}\right) = \mathtt{SUM}_{\boldsymbol{X}-\{X_p\}}\left(\bigotimes_{\ell=1,d} \boldsymbol{e}^{(X_\ell,t_\ell)}\right) = \boldsymbol{e}^{(X_p,t_p)} \otimes \bigotimes_{\ell=1,d;\ell\neq p} \mathtt{SUM}_{X_\ell}(\boldsymbol{e}^{(X_\ell,t_\ell)}) = \boldsymbol{e}^{(X_p,t_p)}$$

because $\mathtt{SUM}_{\boldsymbol{X}}\left(\boldsymbol{e}^{(X_\ell,t_\ell)}\right) = 1$. Therefore, $\mathtt{SUM}_{\boldsymbol{X}-\{X_p\}}\left(\boldsymbol{D}^{(\boldsymbol{k}i)}\right)$ is $\boldsymbol{e}^{X_p,k_p}$ when $p \neq q$, and is $\boldsymbol{e}^{X_q,i}$ when $p = q$, and we obtain:

$$\begin{aligned}
p \neq q: \qquad & \mathtt{SUM}_{\boldsymbol{X}-\{X_p\}}(\boldsymbol{N}-\boldsymbol{N}) = \sum_{\boldsymbol{k}} \varepsilon_{\boldsymbol{k}} \mathtt{SUM}_{\boldsymbol{X}-\{X_p\}}\left(\boldsymbol{D}^{(\boldsymbol{k}i)} - \boldsymbol{D}^{(\boldsymbol{k}j)}\right) = 0 \\
p = q: \qquad & \mathtt{SUM}_{\boldsymbol{X}-\{X_q\}}(\boldsymbol{N}-\boldsymbol{N}) = \sum_{\boldsymbol{k}} \varepsilon_{\boldsymbol{k}} \mathtt{SUM}_{\boldsymbol{X}-\{X_q\}}\left(\boldsymbol{D}^{(\boldsymbol{k}i)} - \boldsymbol{D}^{(\boldsymbol{k}j)}\right) \\
& \qquad = \left(\sum_{\boldsymbol{k}} \varepsilon_{\boldsymbol{k}}\right)(\boldsymbol{e}^{X_q,i} - \boldsymbol{e}^{X_q,j}) = \delta(\boldsymbol{e}^{X_q,i} - \boldsymbol{e}^{X_q,j})
\end{aligned}$$

proving that $\mathtt{SUM}_{\boldsymbol{X}-\{X_q\}}(\boldsymbol{N}) = \boldsymbol{f} \circ \tau_{ij}$.

Finally, we check Inequality (A.1). First, we use the definition of dot product in Def. 3.1.1 and Eq. (3.6) to derive:

$$\begin{aligned}
\boldsymbol{D}^{(\boldsymbol{t})} \cdot \boldsymbol{a}^{(X_1)} \cdots \boldsymbol{a}^{(X_d)} &= \mathtt{SUM}_{\boldsymbol{X}}\left(\boldsymbol{D}^{(\boldsymbol{t})} \otimes \boldsymbol{a}^{(X_1)} \otimes \cdots \otimes a^{(X_d)}\right) \\
&= \mathtt{SUM}_{\boldsymbol{X}}\left((\boldsymbol{e}^{(X_1,t_1)} \otimes \boldsymbol{a}^{(X_1)}) \otimes \cdots \otimes (\boldsymbol{e}^{(X_d,t_d)} \otimes \boldsymbol{a}^{(X_d)})\right) \\
&= \mathtt{SUM}_{X_1}(\boldsymbol{e}^{(X_1,t_1)} \otimes \boldsymbol{a}^{(X_1)}) \cdots \mathtt{SUM}_{X_d}(\boldsymbol{e}^{(X_d,t_d)} \otimes \boldsymbol{a}^{(X)}) \\
&= a_{t_1}^{(X_1)} \times \cdots \times a_{t_d}^{(X_d)} = \prod_{p=1,d} a_{t_p}^{(X_p)}
\end{aligned}$$

This implies:

$$\begin{aligned}
(\boldsymbol{N}-\boldsymbol{M}) \cdot \boldsymbol{a}^{(X_1)} \cdots \boldsymbol{a}^{(X_d)} &= \sum_{\boldsymbol{k}} \varepsilon_{\boldsymbol{k}} (\boldsymbol{D}^{(\boldsymbol{k}i)} - \boldsymbol{D}^{(\boldsymbol{k}j)}) \cdot \boldsymbol{a}^{(X_1)} \cdots \boldsymbol{a}^{(X_d)} \\
&= \sum_{\boldsymbol{k}} \left(\varepsilon_{\boldsymbol{k}} a_i^{(X_q)} \prod_{p=1,d;p\neq q} a_{k_p}^{(X_p)}\right) - \sum_{\boldsymbol{k}} \left(\varepsilon_{\boldsymbol{k}} a_j^{(X_q)} \prod_{p=1,d;p\neq q} a_{k_p}^{(X_p)}\right) \\
&= \left(a_i^{(X_q)} - a_j^{(X_q)}\right) \sum_{\boldsymbol{k}} \varepsilon_{\boldsymbol{k}} \prod_{p=1,d;p\neq q} a_{k_q}^{(X_p)}
\end{aligned}$$

This quantity is ≥ 0 because $i < j$ and $a^{(X_q)}$ is non-increasing by assumption. □

A.2.2 Completing the Proof of Item 1

To complete the Proof of Item 1 we need to show that the tensor $\boldsymbol{C}$ defined by (3.21) satisfies the degree constraints $\boldsymbol{f}^{(\boldsymbol{X})}$. For that purpose we will the strong duality theorem from linear optimization.

Recall that a the primal and dual linear programs have the following forms:

$$\begin{aligned} &\text{Maximize: } \boldsymbol{c}^T\boldsymbol{X} \\ &\qquad\qquad\qquad\qquad (A.2) \\ &\text{Where: } \boldsymbol{A}\cdot\boldsymbol{X} \le \boldsymbol{b} \\ &\qquad\qquad\quad \boldsymbol{X} \ge 0 \end{aligned}$$

$$\begin{aligned} &\text{Minimize: } \boldsymbol{Y}^T\boldsymbol{b} \\ &\text{Where: } \boldsymbol{Y}^T\boldsymbol{A} \ge \boldsymbol{c}^T \\ &\qquad\qquad\quad \boldsymbol{Y}^T \ge 0 \end{aligned}$$

The *Strong Duality Theorem* states:

Theorem A.2.4. *Let X^* and $\boldsymbol{Y}^*$ be optimal solutions to the primal and dual program, respectively. Then, the following holds,*

$$c^T X^* = Y^{*T} b$$

Moreover, if $Y_i^* > 0$ then the i'th constraint in the primal is tight, $(\boldsymbol{A}\cdot\boldsymbol{X})_i = b_i$.

The tensor $\boldsymbol{C}$ is defined from $\boldsymbol{V}$, whose entries $V_{\boldsymbol{m}}$ are optimal values of a primal LP shown in Eq. (3.20). We consider the dual LP. To avoid clutter, we show the primal/dual assuming $d = 3$ coordinates, and will use the same simplification when we prove items 3, 5, and 6; the general case follows straightforwardly. Thus, assuming $d = 3$, we will rename the variables X_1, X_2, X_3 used in the primal LP (3.20) to I, J, K, and rename the degree sequences to $\boldsymbol{f} \in \mathbb{R}_+^{n_1}, \boldsymbol{g} \in \mathbb{R}_+^{n_2}, \boldsymbol{h} \in \mathbb{R}_+^{n_3}$. Then each entry $V_{m_1 m_2 m_3}$ is the solution to the following primal or dual linear programs (all indices i, j, k range over $[m_1]$, $[m_2]$, $[m_3]$ respectively):

$$V_{m_1m_2m_3} = \max \sum_{(i,j,k)\in[m_1]\times[m_2]\times[m_3]} M_{ijk},$$

Where $M_{ijk} \geq 0$ and

$$\forall i: \sum_{jk} M_{ijk} \leq f_i$$

$$\forall j: \sum_{ik} M_{ijk} \leq g_j,$$

$$\forall k: \sum_{ij} M_{ijk} \leq h_k,$$

$$\forall ijk: M_{ijk} \leq B$$

$$V_{m_1m_2m_3} = \min \sum_i \alpha_i f_i + \sum_j \beta_j g_j + \sum_k \gamma_k h_k + B \sum_{ijk} \mu_{ijk} \tag{A.3}$$

Where: $\alpha_i, \beta_j, \gamma_k, \mu_{ijk} \geq 0$ and:

$$\forall i,j,k: \ \alpha_i + \beta_j + \gamma_k + \mu_{ijk} \geq 1$$

In order to prove $\boldsymbol{C} \in \mathcal{M}_{\boldsymbol{f}^{(\boldsymbol{X})},\infty}$, we show a stronger result: any sub-tensor of $\boldsymbol{C}$ is consistent with the degree sequences $\boldsymbol{f}^{(\boldsymbol{X})}$.

Lemma A.2.5. *For any $(m_1, m_2, m_3) \leq (n_1, n_2, n_3)$, let $\boldsymbol{C}'$ be the $[m_1] \times [m_2] \times [m_3]$-subtensor of $\boldsymbol{C}$. Then $\boldsymbol{C}'$ is consistent with the degree sequences $\boldsymbol{f}, \boldsymbol{g}, \boldsymbol{h}$.*

Proof. We will show that $\mathtt{SUM}_{JK}(\boldsymbol{C}') \leq \boldsymbol{f}$, which proves that $\boldsymbol{C}$ is consistent with $\boldsymbol{f}$; consistency with $\boldsymbol{g}, \boldsymbol{h}$ is proven similarly. Let $\boldsymbol{V}'$ be the $[m_1] \times [m_2] \times [m_3]$-subtensor of $\boldsymbol{V}$:

$$\begin{aligned}\mathtt{SUM}_{JK}(\boldsymbol{C}') &\overset{(3.21)}{=} \mathtt{SUM}_{JK}(\Delta_I\Delta_J\Delta_K\boldsymbol{V}') = \Delta_I\left(\mathtt{SUM}_{JK}\left(\Delta_J\Delta_k\boldsymbol{V}'\right)\right) \\ &\overset{(3.19)}{=} \Delta_I\left((V_{im_2m_3})_{i=1,m_1}\right) = (V_{im_2m_3} - V_{(i-1)m_2m_3})_{i=1,m_1}\end{aligned}$$

Consider an optimal dual solution $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\mu}$ to the $(i-1) \times m_2 \times m_3$ problem (A.3), which defines the value of $V_{(i-1)m_2m_3}$. We extended it to a solution to the $i \times m_2 \times m_3$ problem, by setting $\alpha_i \overset{\text{def}}{=} 1$, $\gamma_{ijk} \overset{\text{def}}{=} 0$ for all $j \in [m_2], k \in [m_3]$. The expression (A.3) has increased by f_i. At minimality, the value of $V_{im_2m_3}$ is therefore $\leq V_{(i-1)m_2m_3} + f_i$, proving that $(\mathtt{SUM}_{JK}(\boldsymbol{C}'))_i \leq f_i$. □

A.2.3 Completing the Proof of Item 2

To complete the Proof of Item 2, we consider $d = 2$ and prove that $\boldsymbol{C} \geq 0$ and, furthermore, if $\boldsymbol{f}^{(X_1)}, \boldsymbol{f}^{(X_2)}, B$ are integral, then $\boldsymbol{C}$ is integral. For the integral proof we use the following result from linear optimization. A matrix $\boldsymbol{A}$ is called *totally unimodular* iff every square

submatrix has a determinant equal to $0, +1$ or -1. A sufficient condition for $\boldsymbol{A}$ to be totally unimodular is the following:

Theorem A.2.6. *If the rows of a matrix $\boldsymbol{A}$ can be separated into sets S_1, S_2 and satisfy the following conditions, then $\boldsymbol{A}$ is totally unimodular,*

1. *Every entry in $\boldsymbol{A}$ is either 0 or 1*
2. *Every column in $\boldsymbol{A}$ has at most two non-zero entries.*
3. *If there are two non-zero entries in a column of $\boldsymbol{A}$, then one entry is in a row in S_1 and the other is in a row in S_2.*

This unimodularity property is useful because of the following fact,

Theorem A.2.7. *Consider the linear program given in Eq.* (A.2)*. If the constraint matrix $\boldsymbol{A}$ is totally unimodular and $\boldsymbol{b}$ is integral, then there exists an optimal solution $\boldsymbol{X}^*$ that is integral. In particular, if $\boldsymbol{c}^T$ is also integral, then the optimal value is integral.*

We will apply total unimodularity to the linear program (3.20) defining the entries of $\boldsymbol{V}$, in the case when $d = 2$. We denote the two degree sequences by $\boldsymbol{f} \in \mathbb{R}_+^{[n_1]}$ and $\boldsymbol{g} \in \mathbb{R}_+^{[n_2]}$ (rather than $\boldsymbol{f}^{(X_1)}, \boldsymbol{f}^{(X_2)}$). We show the LP here again, and also show the dual LP, which we need later to prove that $\boldsymbol{C} \geq 0$.

$$
\begin{aligned}
V_{pq} \stackrel{\text{def}}{=} \text{Maximize: } & \sum_{(i,j)\leq(p,q)} M_{ij} \qquad \text{(A.4)} \\
\text{Where: } & \forall i \leq p : \sum_{j=1,q} M_{ij} \leq f_i \\
& \forall j \leq q : \sum_{i=1,p} M_{ij} \leq g_j \\
& \forall (i,j) \leq (p,q) : M_{ij} \leq B
\end{aligned}
$$

$$V_{pq} \stackrel{\text{def}}{=} \text{Minimize: } \sum_{i \le p} \alpha_i f_i + \sum_{j \le q} \beta_j g_j + B \sum_{(i,j) \le (p,q)} \mu_{ij} \tag{A.5}$$

$$\text{Where: } \forall (i,j) \le (p,q) : \alpha_i + \beta_j + \mu_{ij} \ge 1$$

Lemma A.2.8. *The matrix $\boldsymbol{A}$ of the LP defining $\boldsymbol{V}$ in* (A.4) *is totally unimodular. It follows that there exists an integral optimal solution $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$ of the dual LP. Furthermore, if the two degree sequences $\boldsymbol{f}, \boldsymbol{g}$ are also integral, then $\boldsymbol{V}$ and $\boldsymbol{C} \stackrel{def}{=} \Delta_I \Delta_J \boldsymbol{V}$ are integral as well.*

Proof. Let $\boldsymbol{A}$ be the matrix of the primal LP (A.4). Notice that we have a different LP and different matrix $\boldsymbol{A}$ for every $p \le n_1$ and $q \le n_2$. We show that this matrix is unimodular by showing that it satisfies the conditions of Thm. A.2.6. We describe the three important sets of rows in $\boldsymbol{A}$: (1) Rows that arise from row-sum constraints of the form $\sum_j M_{ij} \le f_i$; let S_1 denote this set of rows: $|S_1| = p$. (2) Rows that arise from column-sum constraints of the form $\sum_i M_{ij} \le g_j$; let S_2 denote this set of rows, $|S_2| = q$. (3) Rows that arise from pairwise bounds of the form $M_{ij} \le B$: there are pq such rows. This can be visualized as

follows where the groups appear in order,

$$A = \left(\begin{array}{cccccccccc}
\ldots & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & \ldots \\
\ldots & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & \ldots \\
 & & & & \vdots & & & & & \\
\hline
 & & & & & & & & & \\
\ldots & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & \ldots \\
\ldots & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \ldots \\
 & & & & \vdots & & & & & \\
\hline
 & & & & & & & & & \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 & 0 \\
 & & & & \vdots & & & & & \\
0 & 0 & \ldots & 0 & 1 & 0 & 0 & \ldots & 0 & 0 \\
 & & & & \vdots & & & & & \\
0 & 0 & \ldots & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}\right)$$

The third group of rows form the $pq \times pq$ identity matrix, which has 1 on the diagonal and 0 everywhere else, because there is one constraint $M_{ij} \leq B$ for every variable M_{ij}. Consider any square submatrix $\boldsymbol{A}_0$ of $\boldsymbol{A}$. If it contains any row from the third group of rows, then that row is either entirely 0 (hence $\det \boldsymbol{A}_0 = 0$) or it contains a single value 1, hence the determinant is equal to that obtained by removing the row and column of that value 1. Therefore, it suffices to prove that the submatrix of $\boldsymbol{A}$ consisting of the first two groups of rows is totally unimodular.

So consider the matrix consisting of the rows in the sets S_1 and S_2 above. Looking at our conditions from Theorem A.2.6, the first condition holds immediately (all entries in $\boldsymbol{A}$ are 0 or 1). The second condition holds because every M_{ij} occurs exactly in one row and in one column, hence it occurs in exactly two constraints. Lastly, the last condition holds because the set of rows S_1 contain all row-wise constraints for M_{ij} and the rows S_2 contain all column-wise constraints for M_{ij}, hence the two 1-entries will occur one in a row in S_1 and the other in a row in S_2. This proves that the conditions Theorem A.2.6 are satisfied, hence A is totally unimodular. $\square$

Next, we will prove that $\boldsymbol{C} \geq 0$, and for that we need a deeper look at the dual LP (A.5). Before we do that, we build some intuition by considering a simple example:

Example A.2.9. *Consider the* 2×2 *problem given by the degree sequences* $\boldsymbol{f} = \boldsymbol{g} = (2B, 1)$*, where* B *is some large number, representing the max tuple multiplicity. It is easy to see that* $V_{11} = B$ *(the "best"* 1×1 *matrix that is consistent with* (f_1, g_1, B) *is* (B)*), and* $V_{12} = V_{21} = B + 1$*. To compute* V_{22} *we need to solve the LP* (A.4) *or dual LP* (A.5) *for* $p = q = 2$*. The optimal solutions are:*

$$\boldsymbol{M}^* = \begin{pmatrix} B & 1 \\ 1 & 0 \end{pmatrix} \qquad \boldsymbol{\mu}^* = \begin{array}{c} \\ \\ \boldsymbol{\alpha}^* \end{array} \begin{array}{c} \begin{array}{cc} & \end{array} \boldsymbol{\beta}^* \\ \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{matrix} 0 \\ 1 \end{matrix} \\ \begin{array}{cc} 0 & 1 \end{array} \end{array}$$

To check that these are optimal solutions, it suffices to observe that the objective function of the primal and dual are equal: indeed, $\sum_{ij} M^*_{ij} = B + 2$ *and* $\sum_i \alpha^*_i f_i + \sum_j \beta^*_j g_j + B \sum_{ij} \mu^*_{ij} = 1 + 1 + B = B + 2$*. Therefore the matrices* $\boldsymbol{V}, \boldsymbol{C}$ *are:*

$$\boldsymbol{V} = \begin{pmatrix} B & B+1 \\ B+1 & B+2 \end{pmatrix} \qquad \boldsymbol{C} = \begin{array}{c} \\ 2B \\ 1 \end{array} \begin{array}{c} \begin{array}{cc} 2B & 1 \end{array} \\ \begin{pmatrix} B & 1 \\ 1 & 0 \end{pmatrix} \end{array}$$

Both degree sequences of $\boldsymbol{C}$ *are* $(B+1, 1)$*, and are strictly lower than the allowed sequences* $(2B, 1)$*, in other words we have slack at the degrees* f_1 *and* g_1*: this is witnessed by the fact that the dual has* $\alpha^*_1 = \beta^*_1 = 0$*.*

The proof of $\boldsymbol{C} \geq 0$ follows from several lemmas. First:

Lemma A.2.10. *There exists an optimal solution,* $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\mu}$*, to the dual LP* (A.5) *satisfying:*

$$\alpha_i \leq \alpha_{i+1} \quad \beta_j \leq \beta_{j+1} \; \forall \; i, j$$

Proof. Suppose that there exists an optimal solution such that $\alpha_i > \alpha_{i+1}$. Let $\boldsymbol{\alpha}', \boldsymbol{\mu}'$ be an alternative solution where we swap the values at i and $i+1$, i.e. $\alpha'_i = \alpha_{i+1}, \alpha'_{i+1} = \alpha_i, \mu'_{ij} = \mu_{i+1,j}, \mu'_{i+1,j} = \mu_{ij}$. We can quickly check that this is a valid solution, i.e. it satisfies the

constraints for all j,

$$\alpha_{i+1} + \beta_j + \mu_{i+1,j} \geq 1 \;\rightarrow\; \alpha'_i + \beta_j + \mu'_{i,j} \geq 1$$
$$\alpha_i + \beta_j + \mu_{i,j} \geq 1 \;\rightarrow\; \alpha'_{i+1} + \beta_j + \mu'_{i+1,j} \geq 1$$

Further, we argue that the alternative solution produces at least as large an objective value. Because the values of μ have simply been swapped at i and $i+1$, the sum of the μ values is the same between the two solutions. Therefore, the difference in their objective values is the following,

$$\begin{aligned}\alpha_i f_i + \alpha_{i+1} f_{i+1} - (\alpha'_i f(i) + \alpha_{i+1} f_{i+1}) &= (\alpha_i - \alpha_{i+1}) f_i + (\alpha_{i+1} - \alpha_i) f_{i+1} \\ &= (\alpha_i - \alpha_{i+1})(f_i - f_{i+1}) \leq 0\end{aligned}$$

The final inequality comes from the fact that f is non-increasing and $\alpha_i > \alpha_{i+1}$. Therefore, from any unsorted solution, you can generate an equal or better solution from sorting the α's (the same logic holds identically for the β's). This implies that there exists an optimal solution with sorted values, proving the lemma. □

The combination of Lemmas A.2.8 and A.2.10 implies that, for every p, q, the dual LP has an optimal solution of the following form:

$$\boldsymbol{\alpha}^* = (\underbrace{0, \ldots, 0}_{s_{p,q} \text{ values} = 0}, 1, \ldots, 1) \qquad \boldsymbol{\beta}^* = (\underbrace{0, \ldots, 0}_{t_{p,q} \text{ values} = 0}, 1, \ldots, 1)$$

$$\mu^*_{ij} = 1, \;\; \forall (i,j) \leq (s_{p,q}, t_{p,q}) \qquad \mu^*_{ij} = 0, \;\; \forall (i,j) \not\leq (s_{p,q}, t_{p,q})$$

This optimal solution is uniquely defined by the two indices $s_{p,q} \leq p$ and $t_{p,q} \leq q$. To prove that $\boldsymbol{C} \geq 0$ we will examine how these indices change as we increase p, q to either $p+1, q$ or $p, q+1$.

Recall that $\boldsymbol{F} = \Sigma \boldsymbol{f}$ and $\boldsymbol{G} = \Sigma \boldsymbol{g}$ denote the cumulative degree sequences, i.e. $F_k = \sum_{i \leq k} f_i$ and $G_\ell = \sum_{j \leq \ell} g_j$. From the objective function of the dual LP (A.5) we derive the following expression for V_{pq}:

$$V_{p,q} = \sum_{i \leq p} \alpha^*_i f_i + \sum_{j \leq q} \beta^*_j g_j + B \sum_{(i,j) \leq (p,q)} \mu_{ij} = F_p - F_{s_{p,q}} + G_q - G_{t_{p,q}} + s_{p,q} t_{p,q} B$$

Lemma A.2.11. *If we fix a value of $s_{p,q}$, the optimal value of $t_{p,q}$ is the smallest t such that $g_{t+1} \leq s_{p,q} * B \leq g_t$. Similarly, if we fix $t_{p,q}$, then the optimal value of $s_{p,q}$ is such that $f_{s_{p,q}+1} \leq t_{p,q} * B \leq f_{s_{p,q}}$.*

Proof. This can be seen by taking the discrete derivative of the above expression with respect to $t_{p,q}$,

$$\Delta_{t_{p,q}} V_{p,q} = s_{p,q} B - g_{t_{p,q}} \tag{A.6}$$

This demonstrates that if we set t as before, it is the smallest value of t such that increasing it by 1 would increase $V_{p,q}$. This directly implies that it is the optimal integral value of t. The same logic applies directly to $s_{p,q}$ when $t_{p,q}$ is fixed. □

Lemma A.2.12. *Using the notation of $s_{p,q}, t_{p,q}$ for the optimal solution to the dual for $V_{p,q}$,*

$$s_{p,q} \in \{s_{p-1,q}, p\} \quad t_{p,q} \in \{t_{p,q-1}, q\} \tag{A.7}$$

Proof. We first note the following. If, $s, s' \leq p-1$, and $t, t' \leq q-1$, then,

$$V_{p-1,q}(s,t) \leq V_{p-1,q}(s',t') \leftrightarrow V_{p,q}(s,t) \leq V_{p,q}(s',t')$$

This is easily checked by noticing $V_{p,q}(s,t) = V_{p-1,q}(s,t) + f_p$ and similarly $V_{p,q}(s',t') = V_{p-1,q}(s',t') + f_p$. This implies that if some $s < p-1$ is optimal for the (p,q) sub-problem, then it must be optimal for the $(p-1,q)$ sub-problem. Therefore, a new optimal value can only come from the new □

At this point, we list a few useful facts about this structure which are easily checked.

Lemma A.2.13. *The following are facts about $s_{p,q}, t_{p,q}, V_{p,q}$,*

1. $s_{p,q} \geq s_{p-1,q}, t_{p,q} \geq t_{p,q-1}$
2. $t_{p,q} \leq t_{p-1,q}, s_{p,q} \leq s_{p,q-1}$
3. $V_{p,q} - V_{p-1,q} \leq f_p, V_{p,q} - V_{p,q-1} \leq g_q$

Proof. The first item comes directly from applying the previous lemma. The second comes from applying first fact and Lmm. A.2.11. If $s_{p,q} \geq s_{p-1,q}$ and g is non-increasing, then the description of t for a fixed s immediately implies $t_{p,q} \leq t_{p-1,q}$. The third comes from the fact that you could always transition from $p-1, q$ to p, q by setting $\alpha_p = 1$, similarly for q. □

Okay, we are now prepared to prove that $C \geq 0$ when $d = 2$.

Theorem A.2.14. *When $d = 2$, then C will have non-negative entries. This is equivalent to the following two statements,*

$$V_{p,q} - V_{p-1,q} \geq V_{p,q-1} - V_{p-1,q-1} \tag{A.8}$$

$$V_{p,q} - V_{p,q-1} \geq V_{p-1,q} - V_{p-1,q-1} \tag{A.9}$$

Proof. We will do this by breaking the problem down into a series of cases based on the values of $s_{p,q}$,$s_{p,q-1}$, and $t_{p,q}$.

CASE 1: $s_{p,q-1} = s_{p-1,q-1}$

This directly implies $V_{p,q-1} - V_{p-1,q-1} = f_p$. Further, because $s_{p,q} \leq s_{p,q-1} \leq p-1$, this also implies $V_{p,q} - V_{p-1,q} = f_p$, proving the first inequality.

CASE 2: $s_{p,q-1} = p$ and $s_{p,q} = s_{p-1,q}$

From fact 3, we know $V_{p,q-1} - V_{p-1,q-1} \leq f_p$, and from $s_{p,q} = s_{p-1,q}$ we know $V_{p,q} - V_{p-1,q} = f_p$, proving the first inequality.

CASE 3: $s_{p,q-1} = s_{p,q} = p$ and $t_{p,q} = t_{p,q-1}$

From fact 3, we know $V_{p-1,q} - V_{p-1,q-1} \leq g_q$, and from $t_{p,q} = t_{p,q-1}$ we know $V_{p,q} - V_{p,q-1} = g_q$, proving the second inequality.

CASE 4: $s_{p,q-1} = s_{p,q} = p$ and $t_{p,q} = q$

Because $V_{p,q} = p * q * B$ and $V_{p,q-1} \leq p * (q-1) * B$, we can lower bound the first difference as $V_{p,q} - V_{p,q-1} \geq p \cdot B$.

Looking at $V_{p-1,q} - V_{p-1,q-1}$, setting $mu_{i,q} = 1, \beta_q = 0$ and keeping all other variables consistent with the optimal solution to $V_{p-1,q-1}$ is a valid solution to $V_{p-1,q}$. This solution has value $V_{p-1,q-1} + (p-1)B$, so $V_{p-1,q} \leq V_{p-1,q-1} + (p-1)B$. Therefore, we can upper bound the difference $V_{p-1,q} - V_{p-1,q-1} \leq (p-1)B$. Therefore, the second inequality is true.

Lmm A.2.12 directly implies that our cases are complete. □

A.2.4 Proof of Item 5

Item 5: *For any non-increasing vectors $\boldsymbol{a}^{(X_p)} \in \mathbb{R}_+^{[n_p]}$, $p = 2, d$, the vector $\boldsymbol{C} \cdot \boldsymbol{a}^{(X_2)} \cdots \boldsymbol{a}^{(X_d)}$ is in $\mathbb{R}_+^{[n_1]}$ and non-increasing.*

Recall that we continue to restrict the discussion to $d = 3$, in order to simplify the notation and reduce clutter. The general case is similar, and omitted. The proof of this item requires two lemmas. First:

Lemma A.2.15. *For any vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$, the following "summation-by-parts" holds:*

$$\sum_{i=1,n} (\Delta \boldsymbol{x})_i \cdot y_i = x_n y_n - \sum_{i=1,n-1} x_i \cdot (\Delta \boldsymbol{y})_{i+1} \tag{A.10}$$

The formula is analogous with integration by parts: $\int_0^n f'g dx = fg\,|_0^n - \int_0^n fg' dx$. The proof follows immediately by expanding and rearranging the LHS as follows:

$$\begin{aligned}
\sum_{i=1,n} (\Delta \boldsymbol{x})_i \cdot y_i &= \sum_{i=1,n} x_i y_i - \sum_{i=1,n} x_{i-1} y_i \\
&= \sum_{i=1,n} x_i y_i - \sum_{i=1,n-1} x_i y_{i+1} \\
&= x_n y_n - \left(\sum_{i=1,n-1} x_i y_{i+1} - \sum_{i=1,n-1} x_i y_i \right) \\
&= x_n y_n - \sum_{i=1,n-1} x_i (\Delta \boldsymbol{y})_{i+1}
\end{aligned}$$

The second lemma assets that the second discrete derivative of $\boldsymbol{V}$ on any variable is always ≤ 0. Recall that $\boldsymbol{V}$ is the *value tensor*, introduced in Def. 3.2.1, and we have described it in terms of a primal linear program, and a dual linear program (A.3).

Lemma A.2.16. $\Delta_I^2 \boldsymbol{V} \leq 0$, $\Delta_J^2 \boldsymbol{V} \leq 0$, *and* $\Delta_K^2 \boldsymbol{V} \leq 0$.

Proof. $\Delta_K^2 \boldsymbol{V} \leq 0$ is equivalent to the statement:

$$\forall (m_1, m_2, k) \leq (n_1, n_2, n_3) \qquad 2V_{m_1 m_2 (k-1)} \geq V_{m_1 m_2 (k-2)} + V_{m_1 m_2 k} \tag{A.11}$$

Consider an optimal dual solution $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\mu}$ to the $m_1 \times m_2 \times (k-1)$ problem. Then $V_{m_1 m_2 (k-1)}$ is given by the objective function of the dual linear program, i.e. by expression (A.3). Denote by $w \stackrel{\text{def}}{=} \alpha_{k-1} f_{k-1} + B \sum_{ij} \mu_{ij(k-1)}$, i.e. the contribution to (A.3) of all terms in the plane $K = k-1$. On one hand, we derive from this solution a solution to the $m_1 \times m_2 \times (k-2)$ problem by simply removing the variables in the plane $K = k-1$: expression (A.3) decreases by the quantity w, which proves $V_{m_1 m_2 (k-2)} \leq V_{m_1 m_2 (k-1)} - w$. On the other hand we can extend the solution to a solution of the $m_1 \times m_2 \times k$ problem, by setting the variables in the plane $K = k$ to those in the plane $K = k-1$, more precisely $\alpha_k \stackrel{\text{def}}{=} \alpha_{k-1}$, $\mu_{ijk} \stackrel{\text{def}}{=} \mu_{ij(k-1)}$: it is obvious that all new constraints are satisfied, and the expression (A.3) has increased by w. This implies $V_{m_1 m_2 k} \leq V_{m_1 m_2 (k-1)} + w$. By adding these two inequalities we prove (A.11), as required. □

To prove item 5, let $\boldsymbol{a} \in \mathbb{R}_+^{n_1}, \boldsymbol{b} \in \mathbb{R}_+^{n_2}$ be non-increasing, and let $\boldsymbol{v} \stackrel{\text{def}}{=} \boldsymbol{C} \cdot \boldsymbol{a} \cdot \boldsymbol{b}$. We need to prove that $\boldsymbol{v} \geq 0$ and that $\boldsymbol{v}$ is non-increasing. We start by proving that $\boldsymbol{v} \geq 0$ and for that it suffices to assume that both $\boldsymbol{a}$ and $\boldsymbol{b}$ are one-zero vectors, because any non-decreasing vector is a non-negative linear combination of one-zero vectors (see Sec. 3.2). Suppose the first m_1 values in $\boldsymbol{a}$ are 1 and the others are 0, and similarly the first m_2 values in $\boldsymbol{b}$ are 1 and the others are 0. Then, using an argument similar to that in Eq. (3.24) we have $(\boldsymbol{C} \cdot \boldsymbol{a} \cdot \boldsymbol{b})_k = \sum_{(i,j) \leq (m_1, m_2)} C_{ijk}$ and therefore:

$$v_k = (\boldsymbol{C} \cdot \boldsymbol{a} \cdot \boldsymbol{b})_k = (\Sigma_I \Sigma_J \boldsymbol{C})_{m_1 m_2 k} = (\Sigma_I \Sigma_J \Delta_I \Delta_J \Delta_K \boldsymbol{V})_{m_1 m_2 k} = (\Delta_K \boldsymbol{V})_{m_1 m_2 k} = V_{m_1 m_2 k} - V_{m_1 m_2 (k-1)}$$

The value $V_{m_1 m_2 (k-1)}$ is defined by a $m_1 \times m_2 \times (k-1)$ linear program in Eq. (3.20). Let $\boldsymbol{M}$ be the optimal solution, such that $V_{m_1 m_2 (k-1)} = \sum_{(i,j,p) \leq (m_1, m_2, k-1)} M_{ijp}$. We define a new tensor $\boldsymbol{M'}$ obtained from $\boldsymbol{M}$ by setting $M'_{ijk} \stackrel{\text{def}}{=} 0$ and setting $M'_{ijp} = M_{ijp}$ when $p \neq k$. Then $\boldsymbol{M'}$ is a feasible solution to the $m_1 \times m_2 \times k$-linear program Eq. (3.20), proving that, at optimality, $V_{m_1 m_2 k} \geq \sum_{(i,j,p) \leq (m_1, m_2, k)} M'_{ijp} = V_{m_1 m_2 (k-1)}$, proving that $v_k \geq 0$.

Next we prove that $\boldsymbol{v}$ is non-increasing, or, equivalently, that $\Delta_K \boldsymbol{v}$ is ≤ 0. By definition, $v_k \stackrel{\text{def}}{=} (\boldsymbol{C} \cdot \boldsymbol{a} \cdot \boldsymbol{b})_k = \sum_{ij} C_{ijk} a_i b_j$. Since $\boldsymbol{C} = \Delta_I \Delta_J \Delta_K \boldsymbol{V}$, the summation-by parts formula (A.10) allows us to move the derivatives $\Delta_I \Delta_J$ from $\boldsymbol{V}$ to $\boldsymbol{a}, \boldsymbol{b}$ and obtain the following:

$$\Delta_K \left(\sum_{ij} C_{ijk} a_i b_j \right) \overset{(3.21)}{=} \Delta_K \left(\sum_{ij} \left(\Delta_I \Delta_J \Delta_K V_{ijk} \right) a_i b_j \right) = \Delta_K^2 \left(\sum_j \left(\Delta_J \sum_i (\Delta_I V_{ijk}) a_i \right) b_j \right)$$

$$\sum_j \left(\Delta_J \sum_i (\Delta_I V_{ijk}) a_i \right) b_j \overset{(A.10)}{=} \left(\sum_j \left(\Delta_J \left(V_{n_1 jk} a_{n_1} - \sum_i V_{ijk} (\Delta_I \boldsymbol{a})_{i+1} \right) \right) b_j \right)$$

$$= \left(\sum_j \left(\Delta_J \left(V_{n_1 jk} a_{n_1} \right) \right) b_j \right) - \left(\sum_j \left(\Delta_J \left(\sum_i V_{ijk} (\Delta_I \boldsymbol{a})_{i+1} \right) \right) b_j \right)$$

$$\overset{(A.10)}{=} \left(V_{n_1 n_2 k} a_{n_1} b_{n_2} \right) - \left(\sum_j V_{n_1 jk} a_{n_1} (\Delta_J \boldsymbol{b})_{j+1} \right) - \left(\sum_i V_{in_2 k} (\Delta_I \boldsymbol{a})_{i+1} b_{n_2} \right) + \tag{A.12}$$

$$\left(\sum_{ij} V_{ijk} (\Delta_I \boldsymbol{a})_{i+1} (\Delta_J \boldsymbol{b})_{j+1} \right) \tag{A.13}$$

(In general, for $d \geq 3$, this expression is a sum of 2^{d-1} terms.) Therefore:

$$\Delta_K \left(\sum_{ij} C_{ijk} a_i b_j \right) = \left(\Delta_K^2 (V_{n_1 n_2 k}) a_{n_1} b_{n_2} \right) - \left(\sum_j \Delta_K^2 (V_{n_1 jk}) a_{n_1} (\Delta_J \boldsymbol{b})_{j+1} \right) \tag{A.14}$$

$$- \left(\sum_i \Delta_K^2 (V_{in_2 k}) (\Delta_I \boldsymbol{a})_{i+1} b_{n_2} \right) + \left(\sum_{ij} \Delta_K^2 (V_{ijk}) (\Delta_I \boldsymbol{a})_{i+1} (\Delta_J \boldsymbol{b})_{j+1} \right) \tag{A.15}$$

Since $\boldsymbol{a}, \boldsymbol{b}$ are non-increasing, we have $\Delta_I \boldsymbol{a} \leq 0$ and $\Delta_J \boldsymbol{b} \leq 0$. By Lemma A.2.16 each term $\Delta_K^2 V$ is ≤ 0, which proves that the expression (A.15) is ≤ 0.

A.2.5 Proof of Item 6

Finally, we prove item 6, and continue to restrict the discussion to $d = 3$.

Item 6: *Assume $B = \infty$. Then the following holds:*

$$\forall \boldsymbol{m} \in [\boldsymbol{n}]: \qquad V_{\boldsymbol{m}} = \min\left(F_{m_1}^{(X_1)}, \ldots, F_{m_d}^{(X_d)}\right) \tag{A.16}$$

where $F_r^{(X_p)} \stackrel{def}{=} \sum_{j \le r} f_j^{(X_p)}$ is the CDF associated to the PDF $\boldsymbol{f}^{(X_p)}$, for $p = 1, d$. Moreover, $\boldsymbol{C}$ can be computed by Algorithm 1, which runs in time $\mathbf{O}(\sum_p n_p)$. This further implies that $\boldsymbol{C} \ge 0$, in other words $\boldsymbol{C} \in \mathcal{M}^+_{\boldsymbol{f}^{(\boldsymbol{X})},\infty}$.

We start by showing Eq. (3.22): for all $(m_1, m_2, m_3) \le (n_1, n_2, n_3)$, $V_{m_1 m_2 m_3} = \min(F_{m_1}, G_{m_2}, H_{m_3})$. Assume w.l.o.g. that $F_{m_1} = \min(F_{m_1}, G_{m_2}, H_{m_3})$. We claim that the following is an optimal solution to the dual program (A.3): $\boldsymbol{\alpha} = (1, 1, \cdots, 1)$, $\boldsymbol{\beta} = \mathbf{0}$, $\boldsymbol{\gamma} = \mathbf{0}$, $\boldsymbol{\mu} = \mathbf{0}$: the claim implies that the value of $V_{m_1 m_2 m_3}$ given by Eq. (A.3) is $V_{m_1 m_2 m_3} = \sum_{i=1,m_1} \alpha_i f_i = F_{m_1}$, as required. To prove the claim, let $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$ be any optimal solution to the dual; obviously, $\boldsymbol{\mu} = \mathbf{0}$, because $B = \infty$. We first find another optimal solution where $\boldsymbol{\gamma} = \mathbf{0}$. Denoting $\varepsilon \stackrel{\text{def}}{=} \min_{ij}(\alpha_i + \beta_j)$, we have $\gamma_k \ge 1 - \varepsilon$ for all k. Define $\alpha'_i \stackrel{\text{def}}{=} \alpha_i + (1 - \varepsilon)$ for all i, and $\gamma'_k \stackrel{\text{def}}{=} 0$. Then $\boldsymbol{\alpha}', \boldsymbol{\beta}, \boldsymbol{\gamma}'$ is also a feasible solution, and the value of (A.3) is no larger, because: $\sum_i \alpha'_i f_i = \sum_i \alpha_i f_i + (1 - \varepsilon) \sum_i f_i \le \sum_i f_i \alpha_i + (1 - \varepsilon) \sum_k h_k \le \sum_i f_i \alpha_i + \sum_k \gamma_k h_k$. We repeat this for β (and omit the details, for lack of space) and obtain a new feasible solution $\boldsymbol{\alpha}'', \boldsymbol{\beta}'(= 0), \boldsymbol{\gamma}'(= 0)$ where the value of Eq. (A.3) is still optimal, proving the claim.

It remains to prove that Algorithm 1 is correct. We show that if $\boldsymbol{f}, \boldsymbol{g}, \boldsymbol{h}$ are any 3 vectors (not necessarily decreasing) then the tensor $\boldsymbol{C}$ computed by the algorithm satisfies $\Sigma_I \Sigma_J \Sigma_K \boldsymbol{C} = \min(\boldsymbol{F}, \boldsymbol{G}, \boldsymbol{H})$, i.e. $\sum_{i=1,m_1} \sum_{j=1,m_2} \sum_{k=1,m_3} C_{ijk} = \min(F_{m_1}, G_{m_2}, H_{m_3})$ (correctness follows from (3.25)). Assume w.l.o.g. that $f_1 = \min(f_1, g_1, h_1)$; then the algorithm sets $C_{111} = f_1$ and leaves $C_{1 m_2 m_3} = 0$ for all $(m_2, m_3) \ne (1, 1)$ (since $\boldsymbol{C}$ is initially 0). Therefore the claim holds for $m_1 = 1$: $\sum_{j=1,m_2} \sum_{k=1,m_3} C_{1ij} = f_1$. For $m_1 > 1$, denote by $\boldsymbol{C}'$ the rest of the tensor produced by the algorithm, i.e. using $\boldsymbol{f}' \stackrel{\text{def}}{=} (f_2, f_3, \ldots, f_{n_1})$,

$\boldsymbol{g}' \stackrel{\text{def}}{=} (g_1 - f_1, g_2, \ldots, g_{n_2})$ and $\boldsymbol{h}' \stackrel{\text{def}}{=} (h_1 - f_1, h_2, \ldots, h_{n_3})$. Then:

$$\begin{aligned}\sum_{i=1,m_1}\sum_{j=1,m_2}\sum_{k=1,m_3} C_{ijk} &= f_1 + \sum_{i=2,m_1}\sum_{j=1,m_2}\sum_{k=1,m_3} C'_{ijk} \\ &= f_1 + \min(F'_{m_1}, G'_{m_2}, H'_{m_3}) \qquad /^* \text{ by induction on } \boldsymbol{C}' \ */ \\ &= \min(f_1 + F'_{m_1}, f_1 + G'_{m_2}, f_1 + H'_{m_3}) = \min(F_{m_1}, G_{m_2}, H_{m_3})\end{aligned}$$

because $f_1 + F'_{m_1} = f_1 + \sum_{i=2,m_1} f_i = F_{m_1}$, $f_1 + G'_{m_2} = f_1 + (g_1 - f_1) + \sum_{j=2,m_2} g_j = G_{m_2}$ and similarly $f_1 + H'_{m_3} = H_{m_3}$.

A.3 Completing Proofs from Sec. 3.3

A.3.1 Chain Bound for Berge Acyclic Queries (Theorem 3.4.1)

In this section we review the polymatroid bound, then prove its closed form expression for Berge-acyclic queries stated in Theorem 3.4.1. Throughout this section, we will assume set semantics before extending the discussion to bag semantics at the end.

Background on Information Theory We need to review some notions from information theory. Fix a set of variables $\boldsymbol{X}$. A *polymatroid* is a function $h : 2^{\boldsymbol{X}} \to \mathbb{R}_+$ satisfying the following three conditions, where X, Y are sets of variables.

$$\begin{aligned} h(\emptyset) &= 0 \\ h(X \cup Y) &\geq h(X) \\ h(X) + h(Y) &\geq h(X \cup Y) + h(X \cap Y)\end{aligned}$$

The last two conditions are called *monotonicity* and *submodularity*. We follow standard convention and write XY for the union $X \cup Y$. A *conditional expression* is defined

$$h(Y|X) \stackrel{\text{def}}{=} h(XY) - h(X)$$

Given a joint distribution over random variables $\boldsymbol{X}$, we denote by H its entropy. Then the mapping $Z(\subseteq \boldsymbol{X}) \mapsto H(Z)$ is a polymatroid. When $\boldsymbol{X}$ has 4 or more variables, then there exists polymatroids h that are not entropic functions.

The polymatroid Bound The polymatroid bound was introduced in [90, 91]. A good overview can be found in [112]. We review it here briefly.

Fix a general conjunctive query Q, not necessarily Berge-acyclic. Recall that N_R and $f_1^{(R,Z)}$ denote the upper bound on the cardinality of R and of the maximum degree of its variable Z. The original definition also considers max degrees for sets of variables, but in this paper we restrict to a single variable. Denote by HDC ("degree constraints applied to h") the following set of constraints on a polymatroid h:

$$HDC: \quad \forall R \in \boldsymbol{R}(Q): \quad h(\boldsymbol{X}_R) \leq \log N_R \quad \forall Z \in \boldsymbol{X}_R: \quad h(\boldsymbol{X}_R|Z) \leq \log f_1^{(R,Z)} \tag{A.17}$$

The polymatroid bound of a query Q, which we denote here by $PB(Q)$, is defined in [91, Theorem 1.5] by specifying its logarithm:

$$\log(PB(Q)) \stackrel{\text{def}}{=} \max\{h(\boldsymbol{X}) \mid h \text{ is a polymatroid that satisfies } HDC\} \tag{A.18}$$

More precisely, $PB(Q)$ is the exponent of the RHS of the expression above. Since HDC, the monontonicity, and the submodularity constraints are all linear inequalities, it follows that $\log(PB_{SET}(Q))$ can be computed by solving a linear optimization problem. When the query has 4 or more variables, then the polymatroid bound is not tight in general.

Linear Inequalities An alternative way to describe the polymatroid bound uses *linear inequalities*, which we describe next.

Fix a vector $\boldsymbol{c} \in \mathbb{R}^{2^{\boldsymbol{X}}}$. We view the vector as defining a linear expression E over polymatroids, namely:

$$E(h) \stackrel{\text{def}}{=} \sum_{V \subseteq \boldsymbol{X}} c_V h(V) \tag{A.19}$$

We will assume that $\boldsymbol{c}$ has integer coefficients, and consider linear inequalities of the form:

$$E(h) \geq 0 \tag{A.20}$$

We say that the inequality (A.20) is *valid for all polymatroids*, or *valid for all entropic functions*, if it holds for all polymatroids h, or for all entropic functions H respectively.

The Polymatroid Bound through Inequalities Given a conjunctive query Q, we say that a linear expression $E(h)$ is *associated* to the query Q if it has the form:

$$E(h) \stackrel{\text{def}}{=} \sum_{R \in \boldsymbol{R}(Q)} a_R h(\boldsymbol{X}_R) + \sum_{R \in \boldsymbol{R}(Q), Z \in \boldsymbol{X}_R} b_{R,Z} h(\boldsymbol{X}_R | Z) \tag{A.21}$$

where all coefficients $a_R, b_{R,Z}, k$ are natural numbers. In other words the linear expression contains only the terms $h(\boldsymbol{X}_R)$ and $h(\boldsymbol{X}_R|Z)$ that are also used in the HDC constraints. If the inequality $E(h) \geq k \cdot h(\boldsymbol{X})$ is valid for all polymatroids h, then the following holds for the polymatroid bound $PB_{SET}(Q)$:

$$k \cdot \log PB(Q) \leq \sum_R a_R \log N_R + \sum_{R, Z \in \boldsymbol{X}_R} b_{R,Z} \log(f_1^{(R,Z)}) \tag{A.22}$$

By applying Farkas' lemma one can prove that there exists a valid inequality for which (A.22) becomes an inequality. In other words, one could compute $PB(Q)$ by trying out all (infinitely many) linear expressions $E(h)$ associated to Q, trying all numbers k, then take the minimum of the expression (A.22) where the inequality $E(h) \geq k \cdot h(\boldsymbol{X})$ is valid.

Example A.3.1. *We continue with the query $Q = R(X,Y), S(Y,Z), T(Z,U), K(U,V)$ in Example 3.4.2. The following are valid inequalities associated to Q:*

$$h(YZ) + h(X|Y) + h(U|Z) + h(V|U) \geq h(XYZU)$$
$$h(XY) + h(ZU) + h(V|U) \geq h(XYZU)$$
$$h(XY) + h(UV) + h(Z|U) \geq h(XYZU)$$
$$h(XY) + h(ZU) + h(UV) \geq h(XYZU)$$
$$3h(XY) + h(YZ) + h(ZU) + h(UV) + h(X|Y) + h(U|Z) + h(Z|U) + 2h(V|U) \geq 4h(XYZU)$$

All five are valid inequalities for polymatroids. The first four are easy to check directly, and the fifth is simply the sum of the first four (and thus is also valid). These inequalities lead

to the following upper bounds on $PB(Q)$:

$$\begin{aligned}
PB(Q) \leq & f_1^{(R,Y)} N_S f_1^{(T,Z)} f_1^{(K,U)} \\
PB(Q) \leq & N_R \cdot N_T f^{(K,U)} \\
PB(Q) \leq & N_R \cdot f^{(T,U)} N_k \\
PB(Q) \leq & N_R \cdot N_T \cdot N_K \\
PB(Q) \leq & \left(N_R^3 N_S N_T N_U f_1^{(R,Y)} f_1^{(T,Z)} f_1^{(T,U)} \left(f_1^{(K,U)} \right)^2 \right)^{1/4}
\end{aligned}$$

Notice that the last expression is the geometric mean of the first four.

Term Cover We describe a simple necessary (but not sufficient) condition for an inequality to be valid for all entropic functions. We say that a term $h(V)$ in $E(h)$ *covers* a set of variables $U \subseteq \boldsymbol{X}$, if $U \cap V \neq \emptyset$. For any linear expression $E(h) = \sum_V c_V h(V)$ define the coverage of U by E as:

$$\texttt{cover}(U, E) \stackrel{\text{def}}{=} \sum_{V : V \cap U \neq \emptyset} c_V$$

Once can check that, if $E(h) \geq 0$ holds for all entropic functions, then $\forall U \subseteq \boldsymbol{X}$, if $U \neq \emptyset$, then $\texttt{cover}(U, E) \geq 0$, by observing that $\texttt{cover}(U, E)$ is the value of $E(h)$ computed on the following polymatroid, called a *step function*: $h(V) \stackrel{\text{def}}{=} 0$ if $V \cap U = \emptyset$ and $h(V) \stackrel{\text{def}}{=} 1$ otherwise. In fact, h is an entropy, namely it is the entropy of the following probability space:

U	$\boldsymbol{X} - U$	
$0 \cdots 0$	$0 \cdots 0$	$1/2$
$1 \cdots 1$	$0 \cdots 0$	$1/2$

Simple Inequalities While a characterization of general information-theoretic inequalities is still open, a complete characterization was described in [86] for a restricted class. We present here a slightly more general version than stated in Theorem 3.9 (ii) [86], which follows immediately by using the same Lemma 3.10 as done in [86].

A *simple expression* is a linear expression of the form:

$$E(h) \stackrel{\text{def}}{=} \sum_{V \subseteq \boldsymbol{X}} a_V h(V) - \sum_{Z \in \boldsymbol{X}} b_Z h(Z) \tag{A.23}$$

where all coefficients are natural numbers. In other words, E must have only non-negative coefficients, except for singleton terms $h(Z)$, where Z is a single variable, which may be negative. A *simple inequality* is an inequality of the form:

$$E(h) \geq k \cdot h(\boldsymbol{X}) \tag{A.24}$$

where E is a simple expression. The following was proven in [86].

Theorem A.3.2. *Consider a simple expression $E(h)$, as in Eq.* (A.23)*. Then the following are equivalent:*

1. *The inequality $E(h) \geq kh(\boldsymbol{X})$ is valid for all polymatroids.*
2. *The inequality $E(h) \geq kh(\boldsymbol{X})$ is valid for all entropic functions.*
3. *For every set of variables $U \neq \emptyset$, $\texttt{cover(U,E)} \geq k$.*

In other words, the sufficient condition mentioned earlier becomes a necessary condition, in the case of simple inequalities.

We are interested in simple expression of a special form. A *conditional simple expression* is:

$$E(h) \stackrel{\text{def}}{=} \sum_{V,Y \subseteq \boldsymbol{X}} c_{V|Y} h(V|Y) \tag{A.25}$$

where $c_{V|Y} \geq 0$ and whenever $c_{V|Y} > 0$ then Y is either $\emptyset$ or a single variable. We prove:

Lemma A.3.3. *Consider a simple expression E as in* (A.23)*, and suppose that for every set of variables U, $\texttt{cover}(U, E) \geq 0$. Then E can be written (not uniquely) as a conditional simple expression, i.e. as in* (A.25)*.*

Every linear expression E associated to a Berge-acyclic query, as in Equation (A.21) is a conditional simple expression and, therefore, it is also a simple expression, once we expand the conditionals. The lemma provides a simple test for the converse to hold: take a simple expression and regroup it so it becomes associated to a query. For example, $h(XY) + h(YZ) + h(ZU) - h(Y) - h(Z)$ can be rearranged as $h(X|Y) + h(YZ) + h(U|Z)$,

and is associated to a Berge-acyclic query $R(X,Y), S(Y,Z), T(Z,U)$, but the expression $E = h(XY)+h(YZ)+h(UV)-h(X)-h(Y)-h(Z)$ cannot, because $\texttt{cover}(XYZ, E) = -1$.

Proof. (of Lemma A.3.3) Recall that the coefficients a_V, b_Z of E are natural numbers. Consider the following bipartite graph $G = (\text{Nodes}_1, \text{Nodes}_2, \text{Edges})$. For the set Nodes_1 we create, for $Z \in \boldsymbol{X}$, b_Z nodes labeled $h(Z)$. For the set Nodes_2 we create, for each $V \subseteq \boldsymbol{X}$, a_V nodes labeled $h(V)$. Finally, we connect two nodes $h(Z)$, $h(V)$ if $Z \in V$. Next, we use Hall's theorem to prove that this bipartite graph has a maximal matching, and, for that, it suffices to check that, for each subset $S \subseteq \text{Nodes}_1$, $|N(S)| \geq |S|$, where $N(S)$ are the neighbors of the nodes in S. If S includes some node labeled $h(Z)$, then we can assume w.l.o.g. that it includes all b_Z copies, otherwise we simply add them, which only increases S but does not affect $N(S)$. Therefore, S is uniquely defined by a set of variables $U \subseteq \boldsymbol{X}$, and $|S| = \sum_{Z \in U} b_Z$, while $|N(S)| = \sum_{V \subseteq \boldsymbol{X}, U \cap V \neq \emptyset} a_V$. It follows that $|N(S)| - |S| = \texttt{cover}(U, E) \geq 0$. By Hall's theorem, the graph admits a maximal matching, in other words every node $h(Z)$ is uniquely matched with some $h(V)$ where $Z \in V$, hence we combine the two terms $h(V) - h(Z) = h(V|Z)$. The lemma follows from here. □

Chain Expressions We define a *connected, simple chain expression* an expression of the form:

$$E(h) \stackrel{\text{def}}{=} h(U_0) + h(U_1|Z_1) + \cdots + h(U_k|Z_k) \tag{A.26}$$

where $U_0, U_1, \ldots, U_k$ form a partition of the variables $\boldsymbol{X}$, for all $i = 1, k$, Z_i is a single variable, and $Z_i \in U_0 \cup U_1 \cup \ldots \cup U_{i-1}$.

A chain expression is the simplest linear expression we can associate to a Berge-acyclic query Q. Namely, choose arbitrarily a root relation $\texttt{ROOT} \in \boldsymbol{R}(Q)$, orient its tree and, as in Section 3.4, denote by $Z_R \in \boldsymbol{X}_R$ the parent variable of the relation R, i.e. connecting R to $\texttt{ROOT}$. The *connected, simple chain expression associated to* Q is:

$$E_Q(h) \stackrel{\text{def}}{=} h(\boldsymbol{X}_{\texttt{ROOT}}) + \sum_{R \in \boldsymbol{R}(Q), R \neq \texttt{ROOT}} h(\boldsymbol{X}_R - \{Z_R\}|Z_R) \tag{A.27}$$

The inequality $E_Q(h) \geq h(\boldsymbol{X}_R)$ is valid, and defines the following bound on $PB(Q)$ (see Equation (A.22)):

$$PB(Q) \leq N_{\texttt{ROOT}} \cdot \prod_{R \neq \texttt{ROOT}} f_1^{(R,Z_R)} = PB(Q, \texttt{ROOT})$$

where is precisely the expression $PB(Q, \texttt{ROOT})$ defined in Sec. 3.3, Eq. (3.31). We also observe that E_Q does not depend on the choice of the root relation R (but the bound $PB_{SET}(Q, \texttt{ROOT})$ does depend!). To see this, denote by $\text{atoms}_Q(Z)$ the set of atoms $R \in \boldsymbol{R}(Q)$ that contain the variable Z, and observe that:

$$E_Q(h) = \sum_{R \in \boldsymbol{R}(Q)} h(\boldsymbol{X}_R) - \sum_{Z \in \boldsymbol{X}} (|\text{atoms}_Q(Z)| - 1) \cdot h(Z)$$

Recall from Sec. 3.3 that a *vertex cover* of Q is a set $\boldsymbol{W}_V = \{Q_1, \ldots, Q_m\}$ where each Q_i is a connected subquery of Q, and each variable of Q occurs in at least one Q_i. The *simple chain expression associated to* $\boldsymbol{W}$ is:

$$E_{\boldsymbol{W}_V}(h) = E_{Q_1}(h) + \cdots + E_{Q_m}(h) \tag{A.28}$$

One can check that $E_{\boldsymbol{W}_V}(h) \geq h(\boldsymbol{X})$ is a valid inequality, and that $E_{\boldsymbol{W}_V}$ is an expression associated to Q, in the sense that it is of the form (A.21). If we write each E_{Q_i} as a conditional simple expression, i.e. as in (A.27), by choosing some root relation $\texttt{ROOT}_i$, then it defines the following bound on $PB(Q)$ (see Equation (A.22)):

$$PB(Q) \leq \prod_{i=1,m} PB(Q_i, \texttt{ROOT}_i) \tag{A.29}$$

where $PB(Q_i, \texttt{ROOT}_i)$ was defined in Equation (3.31), and the form $|\texttt{ROOT}_i| \prod_R f_1^{(R,Z_R)}$, i.e. the cardinality of $\texttt{ROOT}_i$ times the max degrees of the other relations in the i'th component. We prove that *every* valid inequality associated to the query is some combination of inequalities of this form:

Theorem A.3.4 (Chain Decomposition)**.** *Let Q be a Berge-acyclic query, and let $E(h)$ be an expression associated to Q, as shown in Equation* (A.21)*, where all coefficients $a_R, b_{R,Z}$ are natural numbers. If $E(h) \geq k \cdot h(\boldsymbol{X})$, then there exists k vertex covers $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_k$, not*

necessarily distinct, such that:

$$E(h) = \sum_i E_{\boldsymbol{W}_i}(h) + (\cdots)$$

where $(\cdots)$ *represents a sum of positive terms (i.e. some* $h(V)$*'s or some* $h(V|Z)$*'s).*

In other words, the inequality $E(h) \geq k \cdot h(\boldsymbol{X})$ is (implied by) the sum of k chain inequalities $E_{\boldsymbol{W}_i}(h) \geq h(\boldsymbol{X})$. Theorem 3.4.1 follows immediately from here. Indeed, each chain inequality $E_{\boldsymbol{W}_i}(h) \geq h(\boldsymbol{X})$ defines a bound on $PB_{SET}(Q)$ of the form (A.29), and this is precisely the form used in Theorem 3.4.1. The original inequality $E(h) \geq k \cdot h(\boldsymbol{X})$ defines a bound that is no better than their geometric means, which, in turn, is no better than the smallest of these bounds. It follows that the minimum bound over everything we can derive from a valid inequality $E(h) \geq k \cdot h(\boldsymbol{X})$, can be already derived from a chain inequality, and that leads to a bound of the form (A.29). It remains to prove Theorem A.3.4.

Proof of Theorem A.3.4 Consider any valid inequality $E(h) \geq kh(\boldsymbol{X})$ associated to the query Q (Equation (A.21)), i.e. all terms in $E(h)$ correspond to atoms in the query, possibly conditioned by one of their variables. After expanding the conditional terms, $h(\boldsymbol{X}_R|Z) = h(\boldsymbol{X}_R) - h(Z)$, we can write the expression as:

$$E(h) = \sum_{R \in \boldsymbol{R}(Q)} a_R h(\boldsymbol{X}_R) - \sum_{Z \in \boldsymbol{X}} b_Z h(Z)$$

where a_R, b_Z are natural numbers. Since $E(h)$ is a simple expression, the inequality $E(h) \geq k \cdot h(\boldsymbol{X})$ is valid iff $\texttt{cover}(U, E) \geq k$ for all sets of variables U.

Let Z be some private variable, in other words it occurs in only one relation R, and suppose $b_Z > 0$. In that case we must also have $a_{\boldsymbol{X}_R} \geq b_Z$, otherwise $\texttt{cover}(Z, E) < 0$. Write $a_R \cdot h(\boldsymbol{X}_R) - b_Z \cdot h(Z) = (a_R - b_Z) \cdot h(\boldsymbol{X}_R) + b_Z h(\boldsymbol{X}_R) - b_Z \cdot h(Z) = (a_R - b_Z) \cdot h(\boldsymbol{X}_R) + b_Z \cdot h(\boldsymbol{X}_R|Z)$. We claim that we can remove $b_Z \cdot h(\boldsymbol{X}_R|Z)$ from E, and the resulting expression E_0 still satisfies the inequality $E_0(h) \geq k \cdot h(\boldsymbol{X})$. To prove the claim it suffices to check $\texttt{cover}(U, E_0) \geq k$ for all $U \subseteq \boldsymbol{X}$. If $Z \in U$ or $U \cap \boldsymbol{X}_R = \emptyset$, then $\texttt{cover}(U, E_0) = \texttt{cover}(U, E) \geq k$, hence it suffices to assume $Z \notin U$, and $U \cap \boldsymbol{X}_R \neq \emptyset$. In that case we prove that $\texttt{cover}(W, E_0) = \texttt{cover}(WZ, E_0)$, and the latter is $\geq k$ as we saw. The only term in E_0 that covers Z is $(a_{\boldsymbol{X}_R} - b_Z) \cdot h(\boldsymbol{X}_R)$, and this also covers W, which

proves $\texttt{cover}(W, E_0) = \texttt{cover}(WZ, E_0)$. Therefore, we can assume w.l.o.g. that no isolated variables Z occur in the expression E.

We prove now the theorem by induction on the number of relations in Q. If Q has a single relation $R(\boldsymbol{X})$, then all variables are isolated, hence E must be $a_{\boldsymbol{X}} h(\boldsymbol{X})$, and since every variable is covered $\geq k$ times, we have $a_{\boldsymbol{X}} \geq k$.

Assume Q has > 1 atoms, let R be a leaf relation, and let Q' be the query without the relation R; we assume that the theorem holds for Q'. Since Q is Berge-acyclic, there exists a single variable $Z \in \boldsymbol{X}_R$ that also occurs in Q'. Therefore, $E(h)$ can be written as:

$$E(h) = a_{\boldsymbol{X}_R} \cdot h(\boldsymbol{X}_R) - b_Z \cdot h(Z) + E'(h)$$

where $E'(h)$ contains the remaining terms of the expression $E(h)$, and thus, is an expression associated to the query Q', i.e. it only contains terms that correspond to atoms and variables in Q'. Let $\boldsymbol{X}'$ be the variables of Q'. Our plan is to apply induction hypothesis to Q', and for that we check the coverage in E' of sets of variables $U' \subseteq \boldsymbol{X}'$. If $Z \notin U'$, then $\texttt{cover}(U', E') = \texttt{cover}(U', E) \geq k$. If $Z \in U'$, then we may have $\texttt{cover}(U', E') < k$. Define:

$$k' \stackrel{\text{def}}{=} \min_{U' \subseteq \boldsymbol{X}'} \texttt{cover}(U', E')$$

and let $U' \subseteq \boldsymbol{X}'$ a set s.t. $\texttt{cover}(U', E') = k'$ (i.e. argmin of the expression above). Assuming $k' < k$, we must have $Z \in U'$ (the case when $k' \geq k$ is simple, because in that case E' already satisfies $E'(h) \geq k \cdot h(\boldsymbol{X}')$, and we omit it).

We use the fact that $\texttt{cover}(U', E) = a_{\boldsymbol{X}_R} - b_Z + \texttt{cover}(U', E') = a_{\boldsymbol{X}_R} - b_Z + k' \geq k$ and derive that $a_{\boldsymbol{X}_R} = \delta + (k - k') + b_Z$, for some $\delta \geq 0$. This implies:

$$E(h) = a_{\boldsymbol{X}_R} \cdot h(\boldsymbol{X}_R) - b_Z \cdot h(Z) - (k - k') \cdot h(Z) + \big((k - k') \cdot h(Z) + E'(h)\big) \tag{A.30}$$

$$= \delta \cdot h(\boldsymbol{X}_R) + \big((k - k') + b_Z\big) \cdot h(\boldsymbol{X}_R | Z) + \left(\underbrace{(k - k') \cdot h(Z) + E'(h)}_{\stackrel{\text{def}}{=} E''(h)} \right) \tag{A.31}$$

Let $\boldsymbol{X}'$ denote the variables of the query Q', including Z. Once can check that, for all $U \subseteq \boldsymbol{X}'$, $\texttt{cover}(U, E'') \geq k$. Indeed, if $Z \notin U$ then $\texttt{cover}(U, E'') = \texttt{cover}(U, E') \geq k$ as we

argued earlier, and if $Z \in U$ then:

$$\texttt{cover}(U, E'') = (k - k') + \texttt{cover}(U, E') \geq (k - k') + k' = k$$

Therefore, $E''(h) \geq kh(\boldsymbol{X}')$ is a valid simple inequality, and, by induction on the query Q', we can write E'' as:

$$E''(h) = \sum_{i=1,k} E_{\boldsymbol{W}_i'}(h) + (\cdots) \tag{A.32}$$

where each W_i' is a cover of Q' and $(\cdots)$ hides some positive terms. In other words, $E''(h)$ is written as a sum of simple chain expressions associated to covers $\boldsymbol{W}_i'$ of Q'. We prove now that $E(h)$ can also be written as a sum of simple chain expressions associated to covers of Q. For that, we take the first terms in (A.31), argue that there are at least k of them, and combine each with one of the covers $\boldsymbol{W}_i'$.

More precisely, we consider two cases. First, if the relation we have eliminated, R, has no private variables. In other words $\boldsymbol{X}_R = \{Z\}$. In that case, $E''(h)$ is already of the required form by the theorem, since it contains all variables of Q. Second, suppose R has at least one private variable, call it Y. Then, from Eq. (A.31) we obtain $\texttt{cover}(Y, E) = \delta + ((k - k') + b_Z) \geq k$. This means that we can produce k terms, either $h(\boldsymbol{X}_R)$ or $h(\boldsymbol{X}_R|Z)$, and write $E(h)$ as a sum of k terms, where each term has one of these two forms:

$$h(\boldsymbol{X}_R) + E_{\boldsymbol{W}_i'}(h) \qquad\qquad h(\boldsymbol{X}_R|Z) + E_{\boldsymbol{W}_i'}(h)$$

In the first case we extend the cover $\boldsymbol{W}_i'$ with one new set $\{R\}$, and denote the resulting cover of Q by $\boldsymbol{W}_i$. In the second case we extended the cover by including R in the set containing some other relation that contains Z (if there are multiple such sets, we choose one arbitrarily) and denote $\boldsymbol{W}_i$ the resulting cover of Q. In both bases, the expression above becomes $E_{\boldsymbol{W}_i}(h)$, and therefore $E(h) = \sum_i E_{\boldsymbol{W}_i}(h) + (\cdots)$ as required.

Extension to Bag Semantics We now leverage this proof to show prove the second half of Thm. 3.4.1for queries under bag semantics. Bag semantics can be simulated by adding a tuple id attribute T_R to each relation $R \in \boldsymbol{R}(Q)$ then evaluating under set semantics. We denote the query over these amended relations as Q', and we can then define PB_{BAG}

directly as,

$$PB_{BAG}(Q) = PB_{SET}(Q') = \min_{\boldsymbol{W}_V} \prod_{Q'_i \in W_V} \min_{\texttt{ROOT}_i} PB(Q'_i, \texttt{ROOT}'_i)$$

Note that any variable cover of Q' must be an edge cover because each edge has a unique variable.

$$PB_{BAG}(Q) = \min_{\boldsymbol{W}_E} \prod_{Q'_i \in W_E} \min_{\texttt{ROOT}_i} PB(Q'_i, \texttt{ROOT}'_i)$$

Further, the addition of the tuple id attribute does not affect the degree sequence of the other attributes, and this attribute's degree sequence does not appear in chain bounds because it does not participate in any joins. This implies that $PB(Q, ROOT) = PB(Q', ROOT)$. Therefore, we can immediately prove our claim about the polymatroid bound over bags,

$$PB_{BAG}(Q) = \min_{\boldsymbol{W}_E} \prod_{Q_i \in W_E} \min_{\texttt{ROOT}_i} PB(Q_i, \texttt{ROOT}_i)$$

$$PB_{BAG}(Q) = \min_{\boldsymbol{W}_E} PB(\boldsymbol{W}_E)$$

A.4 Completing Proofs from Sec. 3.5

A.4.1 Proof of Lemma 3.5.2

We start by proving:

Claim 2. For all $p = 2, d$, the following hold:

- If $F_1(i_1 - 1) \geq F_p(i_p)$ then $C_{i_1 \cdots i_p} = 0$.
- If $F_p(i_p - 1) \geq F_1(i_1)$ then $C_{i_1 \cdots i_p} = 0$.

Proof. We prove the first implication only; the second implication is similar and omitted.

Assume that $F_1(i_1 - 1) \geq F_p(i_p)$. We also have $F_1(i_1) \geq F_p(i_p)$ (because F_1 is non-decreasing), and therefore the following hold:

$$\min(F_1(i_1 - 1), F_2(i_2), \ldots, F_d(i_d)) = \min(F_2(i_2), \ldots, F_d(i_d))$$

$$\min(F_1(i_1), F_2(i_2), \ldots, F_d(i_d)) = \min(F_2(i_2), \ldots, F_d(i_d))$$

This implies:

$$\Delta_1 \min(F_1(i_1), \ldots, F_d(i_d)) = 0 \tag{A.33}$$

On the other hand, $F_1(i_1) \geq F_p(i_p - 1)$ (because F_p is non-decreasing), and therefore we can repeat the argument above for $i_p - 1$ instead of i_p, and obtain:

$$\Delta_1 \min(F_1(i_1), \ldots, F_p(i_p - 1), \ldots F_d(i_d)) = 0 \tag{A.34}$$

Combining (A.33) and (A.34) we obtain:

$$\Delta_p \Delta_1 \min(F_1(i_1), \ldots, F_d(i_d)) = 0$$

Since this holds for all indices i_q, $q \neq 1, q \neq p$, it follows that:

$$C_{i_1 \cdots i_d} = \Delta_1 \cdots \Delta_d \min(F_1(i_1), \ldots, F_d(i_d)) = 0$$

The second implication of the claim is proven similarly. □

We now prove Lemma 3.5.2. Assume that $C_{i_1 \cdots i_d} \neq 0$. The claim implies the following two conditions:

$$F_p(i_p) > F_1(i_1 - 1)$$
$$F_p(i_p - 1) < F_1(i_1)$$

By our definition of F^{-1} in Sec. 3.5, this implies:

$$i_p > F_p^{-1}(F_1(i_1 - 1)) \stackrel{\text{def}}{=} j$$
$$i_p < F_p^{-1}(F_1(i_1)) + 1 \stackrel{\text{def}}{=} k$$

(Notice that j, k may be real values.) This means that in the sum below we can restrict i_p to range between j and k only:

$$\sum_{i_p = 1, n_p} C_{i_1 \cdots i_d} \cdot a_p(i_p) = \sum_{i_p = \lceil j \rceil, \lfloor k \rfloor} C_{i_1 \cdots i_d} \cdot a_p(i_p) \tag{A.35}$$

Since $\boldsymbol{a}_p$ is non-increasing we have

$$\sum_{i_p = \lceil j \rceil, \lfloor k \rfloor} C_{i_1 \cdots i_d} \cdot a_p(i_p) \geq \left(\sum_{i_p = \lceil j \rceil, \lfloor k \rfloor} C_{i_1 \cdots i_d} \right) \cdot a_p(\lfloor k \rfloor) = \left(\sum_{i_p = 1, n_p} C_{i_1 \cdots i_d} \right) \cdot a_p(\lfloor F_p^{-1}(F_1(i_1)) \rfloor + 1)$$

and similarly:

$$\sum_{i_p=\lceil j\rceil,\lfloor k\rfloor} C_{i_1\cdots i_d}\cdot a_p(i_p) \le \left(\sum_{i_p=\lceil j\rceil,\lfloor k\rfloor} C_{i_1\cdots i_d}\right)\cdot a_p(\lceil j\rceil) = \left(\sum_{i_p=1,n_p} C_{i_1\cdots i_d}\right)\cdot a_p(\lceil F_p^{-1}(F_1(i_1-1))\rceil)$$

Inequalities (3.40) and (3.41) follow by repeating the argument above for all $p = 2, d$, then observing that

$$\sum_{i_2=1,n_2}\cdots\sum_{i_d=1,n_d} C_{i_1 i_2\cdots i_d} = \Delta_1 \min(F_1(i_1), F_2(n_2),\ldots,F_d(n_d)) = \Delta_1 F_1(i_1)$$

by Eq. (3.18) and $F_1(i_1) \le F_1(n_1) \le F_p(n_p)$.

A.4.2 Proof of Theorem 3.5.3

We begin by proving item (1) of the theorem: $FDSB(Q,\boldsymbol{f}) \ge DSB(Q,\boldsymbol{f})$. To do this, we first prove the following lemma.

Lemma A.4.1.

$$DSB(Q,\boldsymbol{f}) \le FDSB(Q,\boldsymbol{f},\texttt{ROOT}) \tag{A.36}$$

Proof. Let Q^* be an alteration of the query Q where everything is identical except for the addition of a variable X_0 which is only present in relation `ROOT`. Additionally, we set the degree sequence equal to $\mathbf{1}$, i.e. $f^{ROOT,X_0}(i) = 1\ \forall\ i \in [1,|ROOT|]$. We claim that $DSB(Q^*,\boldsymbol{f},\infty) = DSB(Q,\boldsymbol{f},\infty)$. To see this, consider the final summation for $DSB(Q^*,\boldsymbol{f},\infty)$, $DSB(Q,\boldsymbol{f},\infty)$, and $FDSB(Q,\boldsymbol{f},\texttt{ROOT})$ in algorithm 2,

$$DSB(Q^*,\boldsymbol{f}) = \sum_{i_1=1,D_1}\cdots\sum_{i_k=1,D_k}\sum_{i_0=1,|ROOT|} C_{i_1,\ldots,i_k,i_0}\cdot a^{(X_0)}(i_0)\cdot\prod_{p=1,k} a_{i_p}^{(X_p)}$$

$$DSB(Q,\boldsymbol{f}) = \sum_{i_1=1,D_1}\cdots\sum_{i_k=1,D_k}\sum_{i_0=1,|ROOT|} C_{i_1,\ldots,i_k,i_0}\cdot a^{(X_0)}(i_0)\cdot\prod_{p=1,k} a_{i_p}^{(X_p)}$$

$$FDSB(Q,\boldsymbol{f},\texttt{ROOT}) = \sum_{i=1,|\texttt{ROOT}|}\prod_{p=1,k} a_p(max(1,F_p^{-1}(i)))$$

Because X_0 is not shared by any other relations, we have $a^{(X_0)}(i_0) = 1$. Further, we can break $\sum_{i_0=1,|ROOT|} C_{i_1,\ldots,i_k,i_0}$ into $\sum_{i_0=1,|ROOT|}\Delta_{i_0}\Delta_{i_1}\ldots\Delta_{i_k}V_{i_1,\ldots,i_k,i_0}$. Here,

the summation and delta cancel out to leave us with $\Delta_{i_1} \ldots \Delta_{i_k} V_{i_1,\ldots,i_k,|ROOT|}$. Because $B = \infty$, we know from Thm. 3.2.2 item 6 that V is a minimum of the form $\Delta_{i_1} \ldots \Delta_{i_k} \min F_1(i_1), \ldots, F_k(i_k), |ROOT|$. Therefore, we can remove this final term and we get back to our expression for $DSB(Q, \boldsymbol{f})$.

We now prove that $DSB(Q^*, \boldsymbol{f}) \leq FDSB(Q, \boldsymbol{f}, \texttt{ROOT})$. To do this, we rewrite the expression for $DSB(Q^*, \boldsymbol{f})$ as follows,

$$DSB(Q^*, \boldsymbol{f}) = \sum_{i_0=1,|ROOT|} \boldsymbol{w}^{(\texttt{ROOT})}(i)$$

This is possible because the root relation now has a parent variable X_0. Further, according to Lemma 3.5.2, $\boldsymbol{w} \leq \hat{\boldsymbol{w}}$, so replacing $\boldsymbol{w}$ for $\hat{\boldsymbol{w}}$ throughout the algorithm will only increase the result. Doing this replacement, we get,

$$DSB(Q^*, \boldsymbol{f}) \leq \sum_{i_0=1,|ROOT|} \hat{\boldsymbol{w}}^{(\texttt{ROOT})}(i)$$

$$DSB(Q^*, \boldsymbol{f}) \leq \sum_{i_0=1,|ROOT|} f_0(i) \cdot \prod_{i=1,p} a_p(\max(1, F_p^{-1}(F_0(i))))$$

Because the degree sequence of X_0 is equal to $\mathbf{1}$, we know $f_0(i) = 1$ and $F_0(i) = i$.

$$DSB(Q^*, \boldsymbol{f}) \leq \sum_{i_0=1,|ROOT|} 1 \cdot \prod_{i=1,p} a_p(\max(1, F_p^{-1}(i)))$$

Recognizing the RHS as the summation for $FDSB$, we get the result.

$$DSB(Q^*, \boldsymbol{f}) \leq FDSB(Q, \boldsymbol{f}, \texttt{ROOT})$$

□

Therefore, for any cover $Q_1, \ldots, Q_m$, we have $\prod_{i=1,m} DSB(Q_i, \boldsymbol{f}) \leq \prod_{i=1,m} \min_{ROOT \in \boldsymbol{R}(Q_i)} FDSB(Q_i, \boldsymbol{f}, \texttt{ROOT})$. To finish the proof of part 1, we simply note the result from 3.4.3 that $DSB(Q, \boldsymbol{f}) \leq DSB(Q_1) \cdots DSB(Q_m)$ for any cover $Q_1, \ldots, Q_m$.

We prove the second part of the theorem in three steps. First, we prove some basic facts about the complexity of computing the composition and multiplication of piece-wise functions. Second, we bound the complexity of computing $\hat{\boldsymbol{w}}^{(R)}$ given its inputs are piece-wise

functions. Lastly, we extend this bound to fully computing Alg. 3 which proves the theorem.

We start by proving the tractability of piece-wise composition,

Lemma A.4.2. *Let $F(x), G(x)$ be non-decreasing piece-wise linear functions with the set of separators $S_F = [m_1, \ldots, m_{s_F}], S_G = [n_1, \ldots, n_{s_G}]$ with cardinalities s_F, s_G. Then, their composition, $F(G(x))$, is a piece-wise linear function with separators $S' = G^{-1}(S_F) \cup S_G = [l_1, \ldots, l_{|S'|}]$ and has the following complexity:*

$$C_{comp} = O((s_F + s_G)(\log(s_F) + \log(s_G)))$$

Proof. Because S' refines the separators of S_G, $G(x)$ is a simple linear function when restricted to one interval of S'. Further, consider a interval defined by this new set of separators $(l_i, l_{i+1}]$, then this interval is contained within an interval of S_G because otherwise there would be a separator of S_G which is not within S'. Further, $(G(l_i), G(l_{i+1})]$ is within an interval of S_F. Suppose the latter were not true, then there would be a separator m_j such that $l_i < G^{-1}(m_j) < l_{i+1}$. Because $G^{-1}(m_j)$ is in the set S', this is impossible. Therefore, the function $F(G(x))$ can be fully defined by simply looking up the linear function at $G(m)$ and at $F(G(m))$ and composing them for every interval $(l_i, l_{i+1}]$ in S'.

Looking at the computational time piece-by-piece, we have to compute the result of G^{-1} s_F times and lookup the linear functions of F and G at a particular point $s_F + s_G$ times. Each of these computations/lookups takes $\log(s_G), \log(s_F)$ operations, respectively, so the overall computational complexity is $O((s_F + s_G)(\log(s_G) + \log(s_F)))$. □

The lemma for multiplication proceeds very similarly,

Lemma A.4.3. *Let $f_1(x), ..., f_d(x)$ be non-decreasing s_i-staircase functions with the set of separators denoted S_{F_i}. Then, their multiplication, $\prod_{p\in[1,d]} F_p(x)$, is an $\sum_{i=1}^{d} s_i$-staircase function with divisors $S' = \bigcup_{p\in[1,d]} S_{F_p} = [l_1, \ldots, l_{|S'|}]$ and has the following complexity:*

$$C_{mult} = O((\sum_{p=1}^{d} s_{F_p})(\sum_{p=1}^{d} \log(s_{F_p})))$$

Proof. Because the joint set of separators S' refines each factor's separators S_{F_p}, within a particular interval $(l_i, l_{i+1}]$ each function is a simple polynomial. Therefore, the function

$\prod_{p=1}^{d} F_p(x)$ can be fully defined by simply looking up each function's constant at l_i and multiplying them for each interval $(l_i, l_{i+1}]$ in S'.

Considering the computational time, computing the product requires looking up each factor's constant and computing these constant's product within each interval of S'. Each of these lookups takes $\log(s_{F_p})$ operations. So, the overall computational complexity is $O((\sum_{p=1}^{d} s_{F_p})(\sum_{p=1}^{d} \log(s_{F_p}))$. □

Given these lemmas, we can bound the computational time of computing $\hat{\boldsymbol{w}}^{(R)}$ and describe the resulting function,

Lemma A.4.4. *Let each vector $f_p \stackrel{def}{=} f^{(R,X_p)}$ be an s_{R,X_p}-staircase and $a_p \stackrel{def}{=} a^{(X_p)}$ be t_{X_P}-staircases with dividers S_{F_p} and S_{a_p}, respectively. Then, the vector $\hat{\boldsymbol{w}}^{(R)}$ as defined in Alg. 3 is a $(\sum_{p=1}^{d} s_{R,X_p} + \sum_{p=2}^{d} t_{X_p})$-staircase and can be computed in time.*

$$C_{total,R} = O((ds_{R,X_1} + \sum_{p=2}^{d} s_{R,X_p} + \sum_{p=2}^{d} t_{X_p}) \cdot (d\log(s_{R,X_1}) + \sum_{p=2}^{d} \log(s_{R,X_p}) + \sum_{p=2}^{d} \log(t_{X_p}) + (\sum_{p=2}^{d} k_{X_p})^2))$$

Proof. We start by restating the expression for $\hat{\boldsymbol{w}}^{(R)}$,

$$\hat{\boldsymbol{w}}^{(R)} \stackrel{\text{def}}{=} (\Delta_{i_1} F_1(i_1)) \prod_{p \in [2,d]} a_p\left(F_p^{-1}(F_1(i_1 - 1))\right)$$

We can now calculate the set of separators of $\hat{\boldsymbol{w}}^{(R)}$ using the previous two lemmas'. First, we consider the separators for each factor of the product. Based on Lemma A.4.2 and the fact that $F_p^{-1} \circ F_1 = F_1^{-1} \circ F_p$, we get the following expression for their separators,

$$S_{a_p \circ F_p^{-1} \circ F_1} = F_1^{-1}(F_p(S_{a_p})) \cup F_1^{-1}(S_{F_p^{-1}}) \cup S_{F_1}$$

At this point, we take the union of the factors' separators to get the full set of separators for $\hat{\boldsymbol{w}}^{(R)}$,

$$S_{\hat{\boldsymbol{w}}^{(R)}} = S_{F_1} \cup \left(\bigcup_{p=2}^{d} F_1^{-1}(F_p(S_{a_p})) \cup F_1^{-1}(S_{F_p^{-1}}) \cup S_{F_1} \right) = S_{F_1} \cup \left(\bigcup_{p=2}^{d} F_1^{-1}(F_p(S_{a_p})) \right) \cup \left(\bigcup_{p=2}^{d} F_1^{-1}(S_{F_p^{-1}}) \right)$$

Therefore, $\hat{\boldsymbol{w}}^{(R)}$ has at most $\sum_{p=1}^{d} s_{R,X_p} + \sum_{p=1}^{d} t_{X_p}$ separators.

Next, we consider the computational complexity and degree in two steps: 1) $C_{comp,R}$, defined as the time to calculate all of the $a_p(F_p^{-1}(F_1(i_1)))$ expressions and 2) $C_{mult,R}$, defined

as the time to calculate the product of these expressions. Starting with the compositions, calculating a single expression of the form $a_p(F_p^{-1}(F_1(i_1)))$ requires calculating two compositions $F_p^{-1} \circ F_1$ and $a_p \circ (F_p^{-1} \circ F_1)$. This first composition takes the following operations by Lmm. A.4.2,

$$O((s_{R,X_p} + s_{R,X_1})(\log(s_{R,X_p}) + \log(s_{R,X_1})))$$

The second composition takes,

$$O((t_{X_p} + s_{R,X_p} + s_{R,X_1})(\log(t_{X_p}) + \log(s_{R,X_p} + s_{R,X_1})))$$

Because $\log(s_{R,X_p} + s_1) \leq \log(s_{R,X_p}) + \log(s_1)$, the sum of these two times is upper bounded by,

$$O((t_{X_p} + s_{R,X_p} + s_{R,X_1})(\log(t_{X_p}) + \log(s_{R,X_p}) + \log(s_{R,X_1})))$$

So, the first step has the following complexity and results in a staircase function because composition with a constant function results in a constant function,

$$C_{comp,R} = O\left(\sum_{p=2}^{d}(t_{X_p} + s_{R,X_p} + s_{R,X_1})(\log(s_{R,X_p}) + \log(s_{R,X_1}) + \log(t_{X_p}))\right)$$

The product of these functions then takes the following operations by Lmm. A.4.3,

$$C_{mult,R} = O\left(\left(s_{R,X_1} + (\sum_{p=2}^{d} s_{R,X_p} + t_{X_p} + s_{R,X_1})\right)\left(\log(s_{R,X_1}) + \sum_{p=2}^{d}\log(s_{R,X_p} + t_{X_p} + s_{R,X_1})\right)\right)$$

Splitting the logarithms results in the following cleaner expression,

$$C_{mult,R} \leq O\left(\left(ds_{R,X_1} + \sum_{p=2}^{d} s_{R,X_p} + \sum_{p=2}^{d} t_{X_p}\right)\left(d\log(s_{R,X_1}) + \sum_{p=2}^{d}\log(s_{R,X_p}) + \sum_{p=2}^{d}\log(t_{X_p})\right)\right)$$

Because $C_{comp,R} \leq C_{mult,R}$ (the sum of a product is less than the product of sums), we can then say that the whole computation has complexity:

$$C_{total,R} = O\left(\left(ds_{R,X_1} + \sum_{p=2}^{d} s_{R,X_p} + \sum_{p=2}^{d} t_{X_p}\right)\left(d\log(s_{R,X_1}) + \sum_{p=2}^{d}\log(s_{R,X_p}) + \sum_{p=2}^{d}\log(t_{X_p})\right)\right)$$

□

Next, we extend this analysis to the full bound computation for a given root,

Theorem A.4.5. *Let Q be a Berge-acyclic query with M relations (as in Eq. (3.7)), whose degree sequences are represented by $s_{R,Z}$-staircase functions. Then, (1) $FDSB(Q, \texttt{ROOT}) \geq DSB(Q)$, and (2) it can be computed in the following polynomial time combined complexity,*

$$T_{FDSB} = O(M(s + \max_{R \in \boldsymbol{R}, Z \in \boldsymbol{X}_R}(Arity(R) \cdot s_{R,Z}))(s_{log} + \max_{R \in \boldsymbol{R}, Z \in \boldsymbol{X}_R}(Arity(R) \cdot \log(s_{R,Z}))))$$

where $s = \sum_{R,Z} s_{R,Z}$ and $s_{log} = \sum_{R,Z} \log(s_{R,Z})$.

Proof. To produce this bound, we bound the maximum computation necessary to compute any $\hat{\boldsymbol{w}}^{(R)}$ then we multiply that bound by $|R|$, i.e. the number of $w^{(R)}$ expressions that we have to compute. To produce this first bound, we start with the complexity of computing an arbitrary $\hat{\boldsymbol{w}}^{(R)}$,

$$C_{total,R} = O\left(\left(ds_{R,X_1} + \sum_{p=2}^{d} s_{R,X_p} + \sum_{p=2}^{d} t_{X_p}\right)\left(d\log(s_{R,X_1}) + \sum_{p=2}^{d}\log(s_{R,X_p}) + \sum_{p=2}^{d}\log(t_{X_p})\right)\right)$$

First, we replace $\sum_p s_{R,X_p} + \sum_p t_{X_p}$ with the sum of all segments in the database, similarly for the log terms. To see why this is valid, we note that t_{X_p} is equal to the number of segments in relations in the sub-tree of the incidence graph rooted at X_p. By the same logic, $\sum_p s_{R,X_p} + \sum_p t_{X_p}$ is equivalent to the sum of all segments in relations in the subtree of the incidence graph rooted at R. Therefore, this expression must be upper bounded by the number of segments in the entire incidence graph, i.e. $\sum_{R \in \boldsymbol{R}} \sum_{Z \in \boldsymbol{X}_R} s_{R,Z}$). This gives us the following expression,

$$C_{total,R} \;=\; O((d \cdot s_{R,X_1} + \sum_{R \in \boldsymbol{R}} \sum_{Z \in \boldsymbol{X}_R} s_{R,Z}) \cdot (d\log(s_{R,X_1}) + \sum_{R \in \boldsymbol{R}} \sum_{Z \in \boldsymbol{X}_R} \log(s_{R,Z})))$$

Next, we need to remove the reliance on dS_{R,X_1} as it refers to the particular table R. We do so by replacing it with its maximum over all relations,

$$C_{total,R} = O((\max_{\boldsymbol{R}, Z \in \boldsymbol{X}_R}(Arity(R) \cdot s_{R,Z}) + \sum_{R \in \boldsymbol{R}} \sum_{Z \in \boldsymbol{X}_R} s_{R,Z}) \\ (\max_{\boldsymbol{R}, Z \in \boldsymbol{X}_R}(Arity(R) \cdot \log(s_{R,Z})) + \sum_{R \in \boldsymbol{R}} \sum_{Z \in \boldsymbol{X}_R} \log(s_{R,Z})))$$

Therefore, we can bound the overall computation by this maximum per-table-computation multiplied by the number of tables,

$$C_{total} = O(\sum_{R \in \boldsymbol{R}} C_{total,R})$$

$$C_{total} = O(|\boldsymbol{R}| \cdot (\max_{\boldsymbol{R}, Z \in \boldsymbol{X}_R} (Arity(R) \cdot s_{R,Z}) + \sum_{R \in \boldsymbol{R}} \sum_{Z \in \boldsymbol{X}_R} s_{R,Z}) \cdot (\max_{\boldsymbol{R}, Z \in \boldsymbol{X}_R} (Arity(R) \cdot \log(s_{R,Z})) + \sum_{R \in \boldsymbol{R}} \sum_{Z \in \boldsymbol{X}_R} \log(s_{R,Z})))$$

□

A.4.3 Proof of Lemma 3.5.4

Lemma A.4.6. *Suppose Q is a Berge-acyclic query with degree sequences $\boldsymbol{f}$, then the following is true,*

$$FDSB(Q, \boldsymbol{f}, ROOT) \leq PB(Q, ROOT) \tag{A.37}$$

Proof. We prove by induction on the tree that, for all $R \neq ROOT$, and every value $i \geq 1$, $\hat{w}^{(R)}(i) \leq \prod_{S \in tree(X_p)} f^{(S,Z_S)}(1)$. Assuming this holds for all children of R, we consider the definition of $\hat{\boldsymbol{w}}^{(R)}$. Because the functions a_p are non-increasing and the maximum restricts their input to ≥ 1, we know that $\hat{w}(i) \leq f_1(i) \cdot \prod_{i=2,p} a_p(1)$. By induction hypothesis, for each function a_p, we know $a_p(1) \leq \prod_{S \in tree(X_p)} f_1^{(S,Z_S)}$. Therefore, we immediately get,

$$\hat{w}(i) \leq f_1(i) \cdot \prod_{i=2,p} \prod_{S \in tree(X_P)} f^{(S,Z_s)}(1) \leq \prod_{S \in tree(R)} f^{(S,Z_S)}(1)$$

As before, this completes the proof of our inductive step. The base case of a leaf relation is trivial as in that case $w(i) = f_1(i) \leq f_1(1)$. Lastly, we consider the expression for the final sum and see that the lemma holds.

$$\sum_{i=1,|\texttt{ROOT}|} \prod_{p=1,k} a_p(\max(1, F_p^{-1}(i))) \leq \sum_{i=1,|\texttt{ROOT}|} \prod_{p=1,k} \prod_{S \in tree(X_p)} f^{(S,Z_S)}(1)$$

$$\leq |\texttt{ROOT}| \prod_{S \neq \texttt{ROOT} \in tree(\texttt{ROOT})} f^{(S,Z_S)}(1) = PB(Q, \texttt{ROOT})$$

□

Proof of Theorem 3.5.6

Theorem A.4.7. *Let Q be a Berge-acyclic query with M relations (as in Eq. (3.7)), whose degree sequences are represented by $s_{R,Z}$-staircase functions. Then, $FDSB_{SET}(Q)$ and $FDSB_{BAG}(Q)$ can be computed in combined complexity $O(2^m \cdot (2^m + m \cdot T_{FDSB}))$.*

Proof. We achieve this complexity result via the dynamic programming algorithm 10 which uses $FDSB(Q, \texttt{ROOT})$ as a sub-routine. Throughout, we will drop the distinction between $FDSB_{SET}$ and $FDSB_{BAG}$ because the only change required is in the definition of a cover which we don't rely on in the analysis.

Algorithm 10 Computing $FDSB(Q)$

$FDSB(Q) = \infty$
for $i \in 1, |\boldsymbol{R}|$ **do**
  for subset S in all size i subsets of $\boldsymbol{R}$ **do**
    $S_1, \ldots, S_l = ConnectedComponents(S)$
    $COST(S) = \prod_{p=1,l} \min_{\texttt{ROOT} \in S_p} FDSB(S_p, \texttt{ROOT})$
    for Q_1, Q_2 in Partitions of S **do**
      $COST(S) = min(COST(S), COST(Q_1) * COST(Q_2))$
    end for
    if S is cover **then**
      $FDSB(Q) = min(FDSB(Q), COST(S))$
    end if
  end for
end for
return $FDSB(Q)$

This algorithm runs in time $O(2^m \cdot (2^m + m \cdot T_{FDSB}))$. Because the number of subsets of $\boldsymbol{R}$ is 2^m, the outer two for loops will only result in 2^m iterations. Within each of those iterations, computing the minimum $FDSB$ for each root within connected component requires checking at most m roots, resulting in $O(mT_{FDSB})$. Further, you need to consider

every partition of S into two sub-queries which can occur at most 2^m ways. However, constant work is done for each of those sub-queries because $COST(Q_1)$ and $COST(Q_2)$ have already been calculated in previous iterations.

We now show that this algorithm is correct, i.e. it returns $FDSB(Q)$. Because it returns a multiplication of FDSB's calculated over partitions in coverings of Q, it is immediate to see that the return value is greater than or equal to $FDSB(Q)$. Therefore, we just need to show that its return value is less than or equal to $FDSB(Q)$. Let $Q_1, \ldots, Q_k$ be the optimal cover of Q, $\texttt{ROOT}_1, \ldots, \texttt{ROOT}_k$ be the optimal roots, and $S^* = R_1, \ldots, R_r$ be the relations in the optimal covering. Lastly, let $\boldsymbol{P}(S)$ represent the set of disjoint partitions of S.

We proceed inductively on the iterations of the outer loop with the following hypothesis,

$$COST(S) \leq \min_{S_1,\ldots,S_k \in \boldsymbol{P}(S')} \prod_{i=1,k} \min_{\texttt{ROOT}} FDSB(S_i, \texttt{ROOT}) \forall\ S \subseteq \boldsymbol{R}$$

Therefore, the following is our inductive assumption,

$$COST(S') \leq \min_{S_1,\ldots,S_k \in \boldsymbol{P}(S')} \prod_{i=1,k} \min_{\texttt{ROOT}} FDSB(S_i, \texttt{ROOT}) \forall\ |S'| < i, S' \subset \boldsymbol{R}$$

Let $S \subseteq \boldsymbol{R}$ be a subset of size i. We note that any partition of S can be described as a series of binary partitions. From this, we can see immediately that $COST(S)$ satisfies our inductive hypothesis because it is the minimum over all binary partitions as well as the trivial partition, $Q_1 = S, Q_2 = \{\}$. Lastly, our base case of $i = 1$ is immediate as the trivial partition is the only available partition. This inductive proof immediately gives us that $COST(S)$ is less than or equal to the product of minimum $FDSB$ over all roots for each partition which is equivalent to the definition of $FDSB(Q)$. □

A.4.4 Proof of Theorem 3.5.7

We start by proving that it holds for star queries before expanding to all Berge-acyclic queries. Consider the following query,

$$Q_{STAR} = R(V_1, \ldots, V_d) S_1(V_1) \ldots S_d(V_d)$$

If $\boldsymbol{M}$ is the count tensor of the relation R and $\boldsymbol{a}^{(V_i)}$ is the count tensor of S_i, in this case a simple non-increasing vector, then we can express the query size as follows,

$$|Q_{STAR}(D)| = \boldsymbol{M} \cdot \boldsymbol{a}^{(V_1)} \cdots \boldsymbol{a}^{(V_d)}$$

Given this notation, we consider the following subset of Theorem 3.2.2,

Theorem A.4.8. *Let $\boldsymbol{f}_R$ be the set of degree sequences as above, and let $\boldsymbol{V}, \boldsymbol{C}$ defined by* (3.20)-(3.21)*. Then:*

1. *We can define the value tensor as follows,*

$$\forall \boldsymbol{m} \in [\boldsymbol{n}]: \qquad E^{\boldsymbol{F}_R}_{\boldsymbol{m}} = \min\left(F_{R.V_1}(m_1), \ldots, F_{R.V_d}(m_d)\right) \tag{A.38}$$

2. *For any non-increasing vectors $\boldsymbol{a}^{(V_p)} \in \mathbb{R}_+^{[n_p]}$, $p = 2, d$, the vector $\boldsymbol{C}^{\boldsymbol{f}_R} \cdot \boldsymbol{a}^{(V_2)} \cdots \boldsymbol{a}^{(V_d)}$ is in $\mathbb{R}_+^{[n_1]}$ and non-increasing.*

3. *For all count tensors $\boldsymbol{M}_R$, and all non-increasing vectors $\boldsymbol{a}^{(X_1)} \in \mathbb{R}_+^{[n_1]}, \ldots, \boldsymbol{a}^{(X_d)} \in \mathbb{R}_+^{[n_d]}$:*

$$\boldsymbol{M}_R \cdot \boldsymbol{a}^{(V_1)} \cdots \boldsymbol{a}^{(V_d)} \leq \boldsymbol{C}^{\boldsymbol{f}_R} \cdot \boldsymbol{a}^{(V_1)} \cdots \boldsymbol{a}^{(V_d)} \tag{A.39}$$

Let $\hat{\boldsymbol{F}}_R$ be an upper bound of $\boldsymbol{F}_R$, i.e. $\hat{\boldsymbol{F}}_{R.V}(i) \geq \boldsymbol{F}_{R.V}(i)\ \forall\ V \in \boldsymbol{V}_R, i$, and define $\hat{f}_{R.V} = \Delta_V \hat{\boldsymbol{F}}_{R.V}$ and $\hat{\boldsymbol{f}}_R$ as the set of these degree sequences which, as specified in Theorem 3.5.7, must be non-increasing. Further, note that item 1 and item 2 relies only on the properties of the worst-case instance's inherent structure, so it immediately applies to $\boldsymbol{C}^{\hat{\boldsymbol{F}}_R}$.

Based on the above, we can prove the following lemma,

Lemma A.4.9. *For all non-increasing vectors $\boldsymbol{a}^{(V_1)} \in \mathbb{R}_+^{[n_1]}, \ldots, \boldsymbol{a}^{(V_d)} \in \mathbb{R}_+^{[n_d]}$:*

$$\boldsymbol{C}^{\boldsymbol{f}_R} \cdot \boldsymbol{a}^{(V_1)} \cdots \boldsymbol{a}^{(V_d)} \leq \boldsymbol{C}^{\hat{\boldsymbol{F}}_R} \cdot \boldsymbol{a}^{(V_1)} \cdots \boldsymbol{a}^{(V_d)} \tag{A.40}$$

Proof. Following the original proof of item 1, we begin by simplifying the problem using 1-0 vectors. In particular, let $\boldsymbol{b}^{(m)} \in \mathcal{R}^n$ be the vector with m 1's followed by $n - m$ 0's. Because the $\boldsymbol{a}^{V_i}$ are non-increasing integral vectors, they can be represented as a sum of 1-0

vectors, so it suffices to consider the case where each of them is a 1-0 vector. In this case, the problem description becomes,

$$\boldsymbol{C}^{\boldsymbol{f}_R} \cdot \boldsymbol{b}^{(m_1)} \dots \boldsymbol{b}^{(m_d)} \leq \boldsymbol{C}^{\hat{\boldsymbol{F}}_R} \cdot \boldsymbol{b}^{(m_1)} \dots \boldsymbol{b}^{(m_d)}$$

Multiplying against $\boldsymbol{b}^{(m)}$ is the same as summing over the first m indices, so this can be alternatively expressed as,

$$\Sigma_{m_1} \dots \Sigma_{m_d} \boldsymbol{C}^{\boldsymbol{f}_R} \leq \boldsymbol{\Sigma}_{m_1} \dots \Sigma_{m_d} C^{\hat{\boldsymbol{f}}_R}$$

Considering the definition of the value tensor $V_{\boldsymbol{m}}^{\boldsymbol{f}_R}$, we can rephrase this as follows where $\boldsymbol{m} = (m_1, \dots, m_d)$,

$$V_{\boldsymbol{m}}^{\boldsymbol{f}_R} \ \leq \ V_{\boldsymbol{m}}^{\hat{\boldsymbol{f}}_R}$$

Lastly, we insert the alternative definition of $V^{\boldsymbol{f}_R}$ provided in item 6 and the fact that each $\hat{\boldsymbol{F}}_{R.V_i}$ is an upper bound of $\boldsymbol{F}_{R.V_i}$ to prove the lemma,

$$\min(F_{R.V_1}(m_1), \dots, F_{R.V_d}(m_d)) \ \leq \ \min(\hat{F}_{R.V_1}(m_1), \dots, \hat{F}_{R.V_d}(m_d))$$

□

To prove that this can be extended to general queries, we simply note that our proof of Thm. 3.3.1 only relied on two properties: 1) the vector $\boldsymbol{C}^{\boldsymbol{f}_R} \cdot \boldsymbol{a}^{(V_2)} \dots \boldsymbol{a}^{(V_d)}$ is non-increasing and 2) $\boldsymbol{M}_R \cdot \boldsymbol{a}^{(V_1)} \dots \boldsymbol{a}^{(V_d)} \leq \boldsymbol{C}^{\boldsymbol{f}_R} \cdot \boldsymbol{a}^{(V_1)} \dots \boldsymbol{a}^{(V_d)}$. The former is derived from the structure of the worst-case instance and therefore immediately applies to $C^{(\hat{\mathbf{f}}^{(R)}}$ while the latter is proven by Lemma A.4.9. Therefore, the proof of Thm. 3.3.1 immediately applies to our new, looser worst-case instance.

Appendix B

PARTITION CONSTRAINTS APPENDIX

B.1 Further Details for Section 4.3

Lemma B.1.1. *The combinatorics bound of $Q_{\hexagon}$ based on* **DC** *is in $\Omega(n^{\frac{4}{3}})$.*

Proof. It suffices to provide a collection of databases $\mathcal{I}$ such that the $I \vDash \mathbf{DC}$ and $|Q^I_{\hexagon}| = \Omega(n^{\frac{4}{3}})$ for $I \in \mathcal{D}$. To accomplish this, we introduce a new relation R defined by (the subscripts X, Y, Z are only used for clarity as to what constant belongs to the domain of which attribute)

$$R_{X,Y,Z} = \{(i_X, j_Y, (i + j - \lfloor \frac{1}{2} n^{\frac{1}{3}} \rfloor \mod n^{\frac{2}{3}})_Z) \mid i = 0, \ldots, n^{\frac{2}{3}} - 1, j = 0, \ldots, n^{\frac{1}{3}} - 1\}.$$

For simplicity, we assume n to be the cube of an odd number.

Intuitively, think of R as a bipartite graph from the domain of X to the domain of Z where Y identifies the edge for a given $x \in dom(X)$ or $z \in dom(Z)$. The domain of X and Z is of size $n^{\frac{2}{3}}$ while the domain of Y is of size $n^{\frac{1}{3}}$. Thus, every $x \in dom(X)$ has $n^{\frac{1}{3}}$ neighbors in $dom(Z)$ and, due to symmetry, also the other way around. $R_{X,Y,Z}$ is depicted in Figure B.1 for $n = 27$.

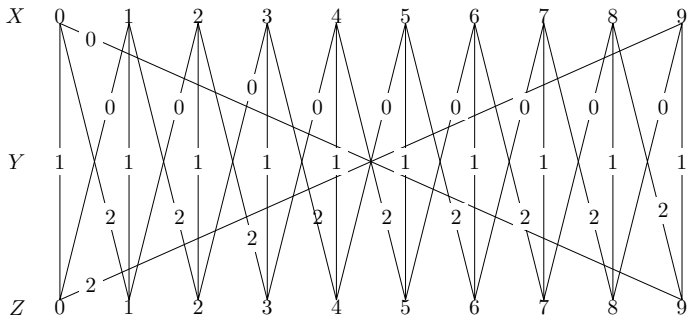


Figure B.1: Depiction of $R_{X,Y,Z}$ for $n = 27$

Notice that, $DC_{R_{X,Y,Z}}(XY, XYZ, 1)$ and $DC_{R_{X,Y,Z}}(YZ, XYZ, 1)$. We set

$$R_1 = R_{A,W,B}, \qquad R_2 = R_{B,U,C}, \qquad R_3 = R_{C,V,A}.$$

Lastly, we set

$$R_4 = \{(i_U, j_V, k_W) \mid i, j, k = 1, \ldots, n^{\frac{1}{3}}\}.$$

Clearly, the database constructed in this way satisfies **DC** (but not **PC**). Now let us compute the number of answers in $Q^I_{\hexagon}$. First notice that $R_4 = dom(U) \times dom(V) \times dom(W)$. Thus, we can simply ignore R_4 and it suffices to look at $\pi_{AB}R_1 \bowtie \pi_{BC}R_2, \bowtie \pi_{AC}R_3$.

Let $i_A \in dom(A)$ be arbitrary. It is connected to $\{(i - \lfloor \frac{1}{2}n^{\frac{1}{3}} \rfloor)_B, \ldots, (i + \lfloor \frac{1}{2}n^{\frac{1}{3}} \rfloor)_B\}$ via $\pi_{AB}R_1$ and to $\{(i - \lfloor \frac{1}{2}n^{\frac{1}{3}} \rfloor)_C, \ldots, (i + \lfloor \frac{1}{2}n^{\frac{1}{3}} \rfloor)_C\}$ via $\pi_{AC}R_3$. Furthermore, $N(i, B) := \{(i - \lfloor \frac{1}{4}n^{\frac{1}{3}} \rfloor)_B, \ldots, (i + \lfloor \frac{1}{4}n^{\frac{1}{3}} \rfloor)_B\}$ are all connected to $N(i, C) := \{(i - \lfloor \frac{1}{4}n^{\frac{1}{3}} \rfloor)_C, \ldots, (i + \lfloor \frac{1}{4}n^{\frac{1}{3}} \rfloor)_C\}$ in $\pi_{BC}R_2$. Thus,

$$\{\{i_A\} \times N(i, B) \times N(i, C) \mid i = 0, \ldots, n^{\frac{2}{3}} - 1\} \subseteq \pi_{AB}R_1 \bowtie \pi_{BC}R_2, \bowtie \pi_{AC}R_3.$$

Consequently, $|Q^I_{\hexagon}| = |\pi_{AB}R_1 \bowtie \pi_{BC}R_2, \bowtie \pi_{AC}R_3| = \Omega(n^{\frac{4}{3}})$. □

Lemma B.1.2. *Algorithm 4 enumerates $Q^I_{\hexagon}$ in time $O(n)$ for databases $I \vDash$ **PC**.*

Proof. We start by proving that a partitioning $R_4 = R_4^U \cup R_4^V \cup R_4^W$ with $DC_{R_4^U}(U, UVW, 3)$, $DC_{R_4^V}(V, UVW, 3)$, $DC_{R_4^W}(W, UVW, 3)$ can be computed in linear time when $PC_{R_4}(\{U, V, W\}, UVW, 1)$. Let us assume, w.l.o.g, that the domains of U, V, W are disjoint and let $\mathcal{V}(d)$ be the variable associated with the constant $d \in \mathcal{D} := dom(U) \cup dom(V) \cup dom(W)$. We claim that there exists a $d \in \mathcal{D}$ such that $|\sigma_{\mathcal{V}(d)=d}R_4| \leq 3$, i.e., d appears at most 3 times.

Assume towards a contradiction that $|\sigma_{\mathcal{V}(d)=d}R_4| > 3$ for all $d \in \mathcal{D}$. Then, $|R_4| > 3|dom(U)|, |R_4| > 3|dom(V)|, |R_4| > 3|dom(W)|$. However, $PC_{R_4}(\{U, V, W\}, UVW, 1)$ ensures that there is a three-way partition $R_4 = R_4^U \cup R_4^V \cup R_4^W$ satisfying $DC_{R_4^U}(U, UVW, 1)$, $DC_{R_4^V}(V, UVW, 1)$, $DC_{R_4^W}(W, UVW, 1)$. One of the parts, w.l.o.g., say R_4^U, then has to contain at least $\frac{1}{3}|R_4|$ tuples. Thus, $|R_4^U| \geq \frac{1}{3}|R_4| > |dom(U)|$. But, at the same time, $|R_4^U| = \sum_{d \in dom(U)} |\sigma_{U=d}R_4^U| \leq \sum_{d \in dom(U)} 1 = |dom(U)|$ leads to a contradiction.

Therefore, we can construct R_4^U, R_4^V, R_4^W as follows: Initialize R_4^U, R_4^V, R_4^W to be empty. Then, select a $d_1 \in \mathcal{D}$ such that $\sigma_{\mathcal{V}(d_1)=d_1} R_4 \leq 3$, add $\sigma_{\mathcal{V}(d_1)=d_1} R_4$ to $R_4^{\mathcal{V}(d_1)}$, and remove $\sigma_{\mathcal{V}(d_1)=d_1} R_4$ from R_4. We can repeat this and select a $d_2 \in \mathcal{D}$ such that $\sigma_{\mathcal{V}(d_2)=d_2} R_4 \leq 3$, add $\sigma_{\mathcal{V}(d_2)=d_2} R_4$ to $R_4^{\mathcal{V}(d_2)}$, and remove $\sigma_{\mathcal{V}(d_2)=d_2} R_4$ from R_4, and so on.

Since each d_i is unique and $PC_{R_4'}(\{U, V, W\}, UVW, 1)$ also holds for any $R_4' \subseteq R_4$, we can be sure that $DC_{R_4^X}(X, UVW, 3)$ holds for every $X = U, V, W$ along the way and in the end. This process can be implemented to run in linear time by maintaining three priority queues, one for each variable U, V, W. To that end, we start by bucket sorting all tuples three times to get the values of $|\sigma_{\mathcal{V}(d)=d} R_4|$ for each $d \in \mathcal{D}$. These collections $\sigma_{\mathcal{V}(d)=d} R_4$ are then added to the queue for the variable $\mathcal{V}(d)$ with the key $|\sigma_{\mathcal{V}(d)=d} R_4|$. Lookups in these queue only happen up to a key value of 3 and thus only take constant time. For updates, when $\sigma_{\mathcal{V}(d_i)=d_i} R_4$ are added to a partition R_4^U, R_4^V or R_4^W, we simply have to update the values for the other constants in $\sigma_{\mathcal{V}(d_i)=d_i} R_4$. Concretely, lets assume $\mathcal{V}(d_i) = U$. Then, for each tuple (u, v, w), we have to remove (u, v, w) from $\sigma_{V=v} R_4$ and $\sigma_{W=w} R_4$ from the queue for V and W, respectively, as well as updating the corresponding key values $|\sigma_{V=v} R_4|$ and $\sigma_{W=w} R_4$. The order in the queue is only important for key values up to a value of 3. Thus, updates can implemented in constant time.

Now let us proceed to the for loops. Notice that the sizes of the projections are constant due to the DCs. For example, consider the first set of nested loops. Given a (a, u, b) from R_1, there are at most 3 matching (v, s) from $R_{4,U}$ due to $DC_{R_{4,U}}(U, UVS, 3)$. Furthermore, there is at most one matching c from R_2 and R_3, independanty due to $DC_{R_2}(BV, BVC, 3)$ and $DC_{R_3}(AS, ASC, 1)$. Thus, for the loops to only take linear time, the relations used in the nested loops simply need to be sorted such that the selection part only requires constant time. This is possible by hashing.

The correctness of the algorithm follows since the sets of nested loops respectively compute $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4^U$, $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4^V$, and $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4^S$. Together they equal $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 = Q_{\hexagon}^I$ as required. □

Lemma B.1.3. *VAAT algorithms require time $\Omega(n^{1.5})$ to enumerate $Q_{\hexagon}^I$ for databases $I \vDash \mathbf{PC}$.*

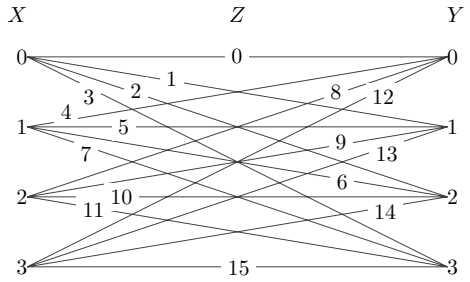


Figure B.2: Depiction of $C_{X,Y,Z}$ for $\frac{n}{7} = 16$

Proof. It suffices to provide a collection of databases $\mathcal{I}$ such that $I \vDash \mathbf{PC}$ and $\max_i |Q^I_{\bigcirc_i}| = \Omega(n^{1.5})$ for every $I \in \mathcal{I}$ and ordering of of the variables. To that end, we introduce two relations, a relation $C_{X,Y,Z}(X,Y,Z)$ and a set of disjoint paths $P_{X,Y,Z}(X,Y,Z)$:

$$C_{X,Y,Z} = \{(i_X, j_Y, (i\sqrt{\frac{n}{7}} + j)_Z) \mid i,j = 0,\ldots,\sqrt{\frac{n}{7}} - 1\},$$
$$P_{XYZ} = \{(i_X, i_Y, i_Z) \mid i = 0,\ldots,\frac{n}{7} - 1\}.$$

For $P_{X,Y,Z}(X,Y,Z)$, the domains of X,Y,Z are of size $\Theta(n)$ and $P_{X,Y,Z}$ simply matches X to Y and Z such that each $d \in dom(X) \cup dom(Y) \cup dom(Z)$ appears in exactly one tuple of $P_{X,Y,Z}$. On the other hand, think of $C_{X,Y,Z}$ as a complete bipartite graph from the domain of X to the domain of Y and Z uniquely identifies the edges. Thus, $|dom(X)| = |dom(Y)| = \Theta(\sqrt{n})$ while $|dom(Z)| = \Theta(n)$. Notice that for both relations, $DC(XY, XYZ, 1), DC(Z, XYZ, 1)$ hold. I.e., any pair of variables determine the last variable and there is a variable that determines the whole tuple on its own. $C_{X,Y,Z}$ is depicted in Figure B.2 for $\frac{n}{7} = 16$.

The database instance will be the disjoint union (by renaming constants) of 7 databases. 4 are of the form $D^R = (R_1^R, R_2^R, R_3^R, R_4^R), R \in \{R_1, R_2, R_3, R_4\}$, and 3 are of the form $D^{XY} = (R_1^{XY}, R_2^{XY}, R_3^{XY}, R_4^{XY}), XY \in \{AU, BV, CW\}$. For D^{R_i}, We define $R_i^{R_i} = P_{var(R_i)}$ and for $j \neq i$ we set $R_j^{R_i} = C_{var(R_j)\setminus var(R_i), var(R_j)\cap var(R_i)}$. Thus, e.g., $R_1^{R_1} = R_{A,W,B}$ and $R_2^{R_1} = C_{C,U,B}$

Furthermore, for D^{XY}, we define $R_i^{XY} = C_{var(R_i)\setminus\{X,Y\}, var(R_i)\cap\{XY\}}$. Thus, e.g., $R_1^{AU} = C_{B,W,A}$ Now, by renaming of constants, we can assume the constants of all databases

D^R, D^{XY} to be pairwise disjoint. In total, D is the disjoint union

$$D = \bigcup_{R\in\{R_1,R_2,R_3,R_4\}} D^R \cup \bigcup_{XY\in\{AV,BS,CU\}} D^{XY}.$$

Clearly, $D \vDash \mathbf{PC}$. Now, let $X_1, \ldots, X_6$ be an arbitrary variable order for $Q_{\bigcirc}$ and the database D as described before. Then a VAAT algorithm based on this variable order at least needs to compute the sets $Q^D_{\bigcirc 1} = \bowtie_i \pi_{X_1} R_i, \ldots, Q^D_{\bigcirc 6} = \bowtie_i \pi_{X_1\cdots X_6} R_i$. Importantly, let us consider the set $Q^D_{\bigcirc 4} = \bowtie_i \pi_{X_1\cdots X_4} R_i$. There are two cases:

Case 1: X_5 and X_6 appear conjointly in a relation. Due to the symmetry of the query and the database we can assume,w.l.o.g, $X_5X_6 = UV$ and

$$\bowtie_i \pi_{X_1\cdots X_4} R_i = \bowtie_i \pi_{ABCW} R_i.$$

Furthermore,

$$\begin{aligned}\bowtie_i \pi_{ABCU} R_i &\supseteq \bowtie_i \pi_{ABCU} R_i^{R_4} \\ &= R_1^{R_4} \bowtie \pi_{BC} R_2^{R_4} \bowtie \pi_{AC} R_3^{R_4} \bowtie \pi_U R_4^{R_4} \\ &= C_{A,B,W} \bowtie \pi_{BC} C_{C,B,U} \bowtie \pi_{AC} C_{A,C,V} \bowtie \pi_W P_{U,V,W}.\end{aligned}$$

Notice that

$$\begin{aligned}C_{A,B,W} &= \{(i_A, j_B, (i\sqrt{\frac{n}{7}} + j)_W) \mid i,j = 0,\ldots,\sqrt{\frac{n}{7}} - 1\} \\ \pi_{BC} C_{C,B,U} &= \{(i_C, j_B) \mid i,j = 0,\ldots,\sqrt{\frac{n}{7}} - 1\} \\ \pi_{AC} C_{A,C,V} &= \{(i_A, j_C) \mid i,j = 0,\ldots,\sqrt{\frac{n}{7}} - 1\} \\ \pi_W P_{U,V,W} &= \{(i\sqrt{\frac{n}{7}} + j)_W) \mid i,j = 0,\ldots,\sqrt{\frac{n}{7}} - 1\}.\end{aligned}$$

Thus,

$$\bowtie_i \pi_{ABCU} R_i \supseteq \{i_A, j_B, k_C, (i\sqrt{n} + j)_U) \mid i,j,k = 0,\ldots,\sqrt{\frac{n}{7}} - 1\}$$

and

$$\max_j |Q^D_{\bigcirc j}| \geq |\bowtie_i \pi_{ABCU} R_i| = \Omega(n^{1.5}).$$

Case 2: X_5 and X_6 do not appear conjointly in a relation. Due to the symmetry of the query and the database we can assume, w.l.o.g, $X_5X_6 = AU$ and

$$\bowtie_i \pi_{X_1 \cdots X_4} R_i = \bowtie_i \pi_{BCUV} R_i.$$

Furthermore,

$$\begin{aligned}\bowtie_i \pi_{BCUS} R_i \supseteq & \pi_{WB} R_1^{AU} \bowtie \pi_{BC} R_2^{AU} \bowtie \pi_{CV} R_3^{AU} \bowtie \pi_{VW} R_4^{AU} \\ = & \pi_{WB} C_{W,B,A} \bowtie \pi_{BC} C_{B,C,U} \bowtie \pi_{CV} C_{C,V,A} \bowtie \pi_{VW} C_{V,W,U}.\end{aligned}$$

Notice that

$$\begin{aligned}\pi_{WB} C_{W,B,A} &= \{(i_W, j_B) \mid i,j = 0, \ldots, \sqrt{\frac{n}{7}} - 1\} \\ \pi_{BC} C_{B,C,U} &= \{(i_B, j_C) \mid i,j = 0, \ldots, \sqrt{\frac{n}{7}} - 1\} \\ \pi_{CV} C_{C,V,A} &= \{(i_C, j_V) \mid i,j = 0, \ldots, \sqrt{\frac{n}{7}} - 1\} \\ \pi_{VW} C_{V,W,U} &= \{(i_V, j_W) \mid i,j = 0, \ldots, \sqrt{\frac{n}{7}} - 1\}.\end{aligned}$$

Thus,

$$\bowtie_i \pi_{ABCU} R_i \supseteq \{(i_B, j_C, k_U, l_S) \mid i,j,k,l = 0, \ldots, \sqrt{\frac{n}{7}} - 1\}$$

and

$$\max_j |Q^D_{\bigcirc j}| \geq | \bowtie_i \pi_{BCUS} R_i| = \Omega(n^2).$$

This completes the proof. □

B.2 Further Details for Section 4.4

Theorem B.2.1. *For a relation $R(\mathbf{Z})$ and subsets $\mathbf{Y} \subseteq \mathbf{Z}, \mathcal{X} \subseteq 2^{\mathbf{Y}}$, Algorithm 5 computes a partitioning $\bigcup_{\mathbf{X} \in \mathcal{X}} R^{\mathbf{X}} = R$ in time $O(|R|)$ (data complexity) such that*

$$DC_{R^{\mathbf{X}}}(\mathbf{X}, \mathbf{Y}, |\mathcal{X}|d)$$

holds for every $\mathbf{X} \in \mathcal{X}$ where $d = PC(\mathcal{X}, \mathbf{Y})$.

Proof. On the one hand, we need to verify that Algorithm 5 can be implemented to run in linear time, and, on the other hand, that this process indeed leads to the stated approximation guarantee.

We start with discussing the runtime. To that end, we have to ensure that the while-loop only requires linear time and thus, we must poll the minimum in constant time. This is achievable by storing pairs $\mathbf{X} \in \mathcal{X}, \mathbf{x} \in \pi_{\mathbf{X}} R$ in a queue with the value determining the position in the queue being $|\pi_{\mathbf{Y}} \sigma_{\mathbf{X}=\mathbf{x}} R| \neq 0$. Ties are broken arbitrarily. Using bucket sort, creating the priority queue only requires linear time as $|\pi_{\mathbf{Y}} \sigma_{\mathbf{X}=\mathbf{x}} R| \leq |R|$. We assume sufficient pointers are saved while creating the queue, in particular between neighbors in the queue. Thus, note that decreasing the key value of an element by 1 is possible in constant time. Then, given a pair $\mathbf{X}, \mathbf{x}$, computing $\sigma_{\mathbf{X}=\mathbf{x}} R$ is possible in time $O(|\sigma_{\mathbf{X}=\mathbf{x}} R|)$. Furthermore, we claim that in time $O(|\sigma_{\mathbf{X}=\mathbf{x}} R|)$ we can remove $\sigma_{\mathbf{X}=\mathbf{x}} R$ from R and maintain the queue. Maintaining the queue means

- removing pairs $\mathbf{X}', \mathbf{x}'$ from the queue where $\mathbf{x}'$ is no longer in $\pi_{\mathbf{X}'}(R \setminus \sigma_{\mathbf{X}=\mathbf{x}} R)$, and
- updating the value of pairs $\mathbf{X}', \mathbf{x}'$ where $|\pi_{\mathbf{Y}} \sigma_{\mathbf{X}'=\mathbf{x}'} R| \neq |\pi_{\mathbf{Y}} \sigma_{\mathbf{X}'=\mathbf{x}'}(R \setminus \sigma_{\mathbf{X}=\mathbf{x}} R)|$.

To accomplish this, for each $\mathbf{X}' \in \mathcal{X} - \mathbf{X}$ and $\mathbf{x}' \in \pi_{\mathbf{X}'} \sigma_{\mathbf{X}=\mathbf{x}} R$ we decrease the value of $\mathbf{X}', \mathbf{x}'$ by $|\sigma_{\mathbf{X}'=\mathbf{x}'} \sigma_{\mathbf{X}=\mathbf{x}} R|$ in the queue (in total, this requires at most $|\mathcal{X}||\sigma_{\mathbf{X}=\mathbf{x}} R|$ decreases of a key value by 1). Notice that, $\mathbf{x}'$ no longer being in $\pi_{\mathbf{X}'}(R \setminus \sigma_{\mathbf{X}=\mathbf{x}} R)$ happens exactly when $|\pi_{\mathbf{Y}} \sigma_{\mathbf{X}'=\mathbf{x}'} R| \neq |\pi_{\mathbf{Y}} \sigma_{\mathbf{X}'=\mathbf{x}'}(R \setminus \sigma_{\mathbf{X}=\mathbf{x}} R)| = 0$. Hence, if the value of a tuple drops to 0, the corresponding tuple can be removed from the queue. Hence, as we are considering data complexity, Algorithm 5 requires linear time in total.

For the correctness of the approximation assume towards a contradiction that this is not the case. To that end, let $\bigcup_{\mathbf{X} \in \mathcal{X}} R^{\mathbf{X}} = R$ be the partitioning created by Algorithm 5 and $\mathbf{X}' \in \mathcal{X}, \mathbf{x}' \in \pi_{\mathbf{X}'} R$ be the first pair $(\mathbf{X}', \mathbf{x}')$ such that $|\pi_{\mathbf{Y}} \sigma_{\mathbf{X}'=\mathbf{x}'} R^{\mathbf{X}'}| > |\mathcal{X}| d$. Note that the tuples $|\pi_{\mathbf{Y}} \sigma_{\mathbf{X}'=\mathbf{x}'} R^{\mathbf{X}'}|$ all have to be added to $R^{\mathbf{X}'}$ at the same time. Let us consider the state of R in the algorithm right before they are removed from R, which we will denote by $R_{\mathcal{A}}$. Hence, $\mathbf{X}', \mathbf{x}' = \operatorname{argmin}_{\mathbf{X} \in \mathcal{X}, \mathbf{x} \in \pi_{\mathbf{X}} R_{\mathcal{A}}} |\pi_{\mathbf{Y}} \sigma_{\mathbf{X}=\mathbf{x}} R_{\mathcal{A}}|$ and, consequently, $|\pi_{\mathbf{Y}} \sigma_{\mathbf{X}=\mathbf{x}} R_{\mathcal{A}}| > |\mathcal{X}| d$ for all $\mathbf{X} \in \mathcal{X}, \mathbf{x} \in \pi_{\mathbf{X}} R_{\mathcal{A}}$. However, since $R_{\mathcal{A}} \subseteq R$, there exists an optimal partition

$\bigcup_{\mathbf{X}\in\mathcal{X}} R^{\mathbf{X}}_{\mathcal{A}} = R_{\mathcal{A}}$ such that $|\pi_{\mathbf{Y}}\sigma_{\mathbf{X}=\mathbf{x}}R^{\mathbf{X}}_{\mathcal{A}}| \leq d$ holds for all $\mathbf{X} \in \mathcal{X}, \mathbf{x} \in \pi_{\mathbf{X}}R_{\mathcal{A}}$. Therefore, on the one hand, $|\mathcal{X}| \cdot |\pi_{\mathbf{Y}}R_{\mathcal{A}}| = \sum_{\mathbf{X}\in\mathcal{X}} \sum_{\mathbf{x}\in\pi_{\mathbf{X}}R_{\mathcal{A}}} |\pi_{\mathbf{Y}}\sigma_{\mathbf{X}=\mathbf{x}}R_{\mathcal{A}}| > \sum_{\mathbf{X}\in\mathcal{X}} \sum_{\mathbf{x}\in\pi_{\mathbf{X}}R_{\mathcal{A}}} |\mathcal{X}|d$ and, on the other hand, $|\pi_{\mathbf{Y}}R_{\mathcal{A}}| \leq \sum_{\mathbf{X}\in\mathcal{X}} \sum_{\mathbf{x}\in\pi_{\mathbf{X}}R_{\mathcal{A}}} |\pi_{\mathbf{Y}}\sigma_{\mathbf{X}=\mathbf{x}}R^{\mathbf{X}}_{\mathcal{A}}| \leq \sum_{\mathbf{X}\in\mathcal{X}} \sum_{\mathbf{x}\in\pi_{\mathbf{X}}R_{\mathcal{A}}} d$. Both cannot be true at the same time. □

Theorem B.2.2. *For a relation $R(\mathbf{Z})$ and subsets $\mathbf{Y} \subseteq \mathbf{Z}, \mathcal{X} \subseteq 2^{\mathbf{Y}}$, Algorithm 6 computes a partitioning $\bigcup_{\mathbf{X}\in\mathcal{X}} R^{\mathbf{X}} = R$ in $O(|R|^2)$ time (data complexity) such that*

$$DC_{R^{\mathbf{X}}}(\mathbf{X}, \mathbf{Y}, d)$$

holds for every $\mathbf{X} \in \mathcal{X}$ where $d = PC(\mathcal{X}, \mathbf{Y})$.

Proof. We start by arguing the correctness of the algorithm. To that end, Algorithm 6 maintains that $(R^{\mathbf{X}})_{\mathbf{X}\in\mathcal{X}}$ is an optimal decomposition of $R_{\text{dec}} := \bigcup_{\mathbf{X}\in\mathcal{X}} R^{\mathbf{X}}$ with $PC_{R_{\text{dec}}}(\mathcal{X}, \mathbf{Y}) = d = \max_{\mathbf{X}\in\mathcal{X}} DC_{R^{\mathbf{X}}}(\mathbf{X}, \mathbf{Y})$ as a loop invariant (for Lines 5-14). Additionally, $R_{\text{dec}} \cup R_{\text{cur}} = R_{\text{ori}}$, where R_{cur} and R_{ori} respectively are the current and original value of R.

Of course, the loop invariant holds before the first loop iteration. Then, the algorithm iterative checks the existence of an augmenting path to allocate the next $\mathbf{y}_0 \in \pi_{\mathbf{Y}}R$ and, if it exists, computes a shortest one $(\mathbf{y}_1, \ldots, \mathbf{y}_m, \mathbf{X}_1, \ldots, \mathbf{X}_{m+1})$. In that case, it cascadingly reallocates $\mathbf{y}_1, \ldots, \mathbf{y}_m$ to $\mathbf{X}_2, \ldots, \mathbf{X}_{m+1}$ and newly allocates $\mathbf{y}_0$ to $\mathbf{X}_1$. As alluded to before, this can only result in an increase of the DC for $R^{\mathbf{X}_{m+1}}$. However, Property 4 of augmenting paths ensures that this does not increase the overall maximum $\max_{\mathbf{X}\in\mathcal{X}} DC_{R^{\mathbf{X}}}(\mathbf{X}, \mathbf{Y})$.

Thus, it only remains to argue that there always exists an augmenting path if the PC of $R_{\text{dec}} \cup \sigma_{\mathbf{Y}=\mathbf{y}_0}R$ is the same as the PC of R_{dec}. Hence, if no augmenting path can be found, it is justified to increase d by 1 and it does not matter where $\sigma_{\mathbf{Y}=\mathbf{y}_0}R$ is allocated to.

To that end, let $(R^{\mathbf{X}}_{\text{opt}})_{\mathbf{X}\in\mathcal{X}}$ be an optimal partitioning of $R_{\text{dec}} \cup \sigma_{\mathbf{Y}=\mathbf{y}_0}R$. Moreover, let $\max_{\mathbf{X}} DC_{R^{\mathbf{X}}_{\text{opt}}}(\mathbf{X}, \mathbf{Y}) = \max_{\mathbf{X}} DC_{R^{\mathbf{X}}}(\mathbf{X}, \mathbf{Y})$. Intuitively, the optimal partitioning $(R^{\mathbf{X}}_{\text{opt}})_{\mathbf{X}\in\mathcal{X}}$ tells us where the tuples belong and leads us to an augmenting path. We start by setting $\mathbf{X}_1$ such that $\mathbf{y}_0 \in \pi_{\mathbf{Y}}R^{\mathbf{X}_1}_{\text{opt}}$. Then, either $(\mathbf{X}_1)$ is an augmenting path (w.r.t. $(R^{\mathbf{X}})_{\mathbf{X}\in\mathcal{X}}$) or Property 4 is not satisfied. In the latter case, there must be at least as many elements in $|\pi_{\mathbf{Y}}\sigma_{\mathbf{X}_1=\mathbf{y}_0}R^{\mathbf{X}_1}|$ as in $|\pi_{\mathbf{Y}}\sigma_{\mathbf{X}_1=\mathbf{y}_0}R^{\mathbf{X}_1}_{\text{opt}}|$. However, $\pi_{\mathbf{Y}}R^{\mathbf{X}_1}_{\text{opt}}$ contains $\mathbf{y}_0$ and thus there is a

"misplaced" element $\mathbf{y}_1 \in \pi_{\mathbf{Y}}\sigma_{\mathbf{X}_1=\mathbf{y}_0}R^{\mathbf{X}_1} \setminus \pi_{\mathbf{Y}}\sigma_{\mathbf{X}_1=\mathbf{y}_0}R^{\mathbf{X}_1}_{\text{opt}}$. We can then set $\mathbf{X}_2$ such that $\mathbf{y}_1 \in \pi_{\mathbf{Y}}R^{\mathbf{X}_2}_{\text{opt}}$.

Due to a similar argumentation as before, either $(\mathbf{y}_1, \mathbf{X}_1, \mathbf{X}_2)$ is an augmenting path or there must exist a misplaced $\mathbf{y}_2 \in \pi_{\mathbf{Y}}\sigma_{\mathbf{X}_2=\mathbf{y}_1}R^{\mathbf{X}_2} \setminus \pi_{\mathbf{Y}}\sigma_{\mathbf{X}_2=\mathbf{y}_1}R^{\mathbf{X}_2}_{\text{opt}}$ and we can set $\mathbf{X}_3$ such that $\mathbf{y}_2 \in \pi_{\mathbf{Y}}R^{\mathbf{X}_3}_{\text{opt}}$.

We proceed inductively: Let $(\mathbf{y}_1, \ldots, \mathbf{y}_i, \mathbf{X}_1, \ldots, \mathbf{X}_{i+1})$ be such that

1. For all $j \in \{0, \ldots, i\}$ we have $\mathbf{y}_j \in \pi_{\mathbf{Y}}R^{\mathbf{X}_{j+1}}_{\text{opt}}$.

2. For all $j \in \{1, \ldots, i\}$ we have $\mathbf{y}_j \in \pi_{\mathbf{Y}}\sigma_{\mathbf{X}_j=\mathbf{y}_{j-1}}R^{\mathbf{X}_j} \setminus (\pi_{\mathbf{Y}}\sigma_{\mathbf{X}_j=\mathbf{y}_{j-1}}R^{\mathbf{X}_j}_{\text{opt}} \cup \{\mathbf{y}_1, \ldots, \mathbf{y}_{j-1}\})$.

Then, there are three possibilities: Either $(\mathbf{y}_1, \ldots, \mathbf{y}_i, \mathbf{X}_1, \ldots, \mathbf{X}_{i+1})$ is an augmenting path. In that case we are done. Or there is a further misplaced element $\mathbf{y}_{j+1} \in \pi_{\mathbf{Y}}\sigma_{\mathbf{X}_{j+1}=\mathbf{y}_j}R^{\mathbf{X}_{j+1}} \setminus (\pi_{\mathbf{Y}}\sigma_{\mathbf{X}_{j+1}=\mathbf{y}_j}R^{\mathbf{X}_{j+1}}_{\text{opt}} \cup \{\mathbf{y}_1, \ldots, \mathbf{y}_j\})$. In that case, simply consider $(\mathbf{y}_1, \ldots, \mathbf{y}_{i+1}, \mathbf{X}_1, \ldots, \mathbf{X}_{i+2})$ with $\mathbf{y}_{j+1} \in \pi_{\mathbf{Y}}R^{\mathbf{X}_{j+2}}_{\text{opt}}$. Note that this case cannot lead to an infinite induction as the $\mathbf{y}_j$ are all different. In the last case, $\pi_{\mathbf{Y}}\sigma_{\mathbf{X}_{j+1}=\mathbf{y}_j}R^{\mathbf{X}_{j+1}} \setminus (\pi_{\mathbf{Y}}\sigma_{\mathbf{X}_{j+1}=\mathbf{y}_j}R^{\mathbf{X}_{j+1}}_{\text{opt}} \cup \{\mathbf{y}_1, \ldots, \mathbf{y}_j\})$ is empty. However, this is not possible as $(\mathbf{y}_1, \ldots, \mathbf{y}_i, \mathbf{X}_1, \ldots, \mathbf{X}_{i+1})$ would then be an augmenting path. This is since reallocating along $(\mathbf{y}_1, \ldots, \mathbf{y}_i, \mathbf{X}_1, \ldots, \mathbf{X}_{i+1})$ can only increase the DC of $R^{\mathbf{X}_{i+1}}$ due to the size of $\pi_{\mathbf{Y}}\sigma_{\mathbf{X}_{j+1}=\mathbf{y}_j}R^{\mathbf{X}_{i+1}}$ and, furthermore, there being no more misplaced element implies that after reallocation, $\pi_{\mathbf{Y}}\sigma_{\mathbf{X}_{j+1}=\mathbf{y}_j}R^{\mathbf{X}_{i+1}}$ is a subset of $\pi_{\mathbf{Y}}\sigma_{\mathbf{X}_{j+1}=\mathbf{y}_j}R^{\mathbf{X}_{j+1}}_{\text{opt}}$.

With regard to the time complexity, the main part that merits discussion is the computation of augmenting paths. For this, however, simply keep pointers from every $\mathbf{y} \in \pi_{\mathbf{Y}}R$ to $\sigma_{\mathbf{X}=\mathbf{y}}R^{\mathbf{X}}$ for all $\mathbf{X} \in \mathcal{X}$. Then, when searching for an augmenting path for $\mathbf{y}_0 \in \pi_{\mathbf{Y}}R$, we simply have to do a breadth-first search, i.e., start with $\mathbf{y}_0$, then move on to all $\bigcup_{\mathbf{X}\in\mathcal{X}}\sigma_{\mathbf{X}=\mathbf{y}_0}R^{\mathbf{X}}$ and follow their pointers if necessary. This only requires linear time (per $\mathbf{y}_0 \in \pi_{\mathbf{Y}}R$) as we need to consider each pointer at most once and each tuple only has a constant number of pointers. Furthermore, the creation of the pointers can be done once in a preprocessing step. □

Appendix C

COLOR APPENDIX

C.1 Proof of Thm. 6.3.5

In this section, we prove the following theorem for simple graphs, but the extension to property graphs is straightforward.

Theorem C.1.1. *Let $\mathcal{G}$ be a lifted graph defined by a stable coloring σ. Then $\tau_{min} = \tau_{avg} = \tau_{max}$, and, for any acyclic query Q, the lifted graph estimator is exact:*

$$|\hom(Q, G)| = \Phi(Q, \mathcal{G}) \tag{C.1}$$

Proof. We start by noting that each match in the data graph, $\pi \in Hom(Q, G)$, is associated with precisely one coloring, $\pi' \in Hom(Q, F)$, based on the matched vertices' colors, i.e. $\pi' = \sigma_S \circ \pi$. We denote the set of matches with this coloring as $Hom(Q, G|\pi')$. If we calculate $|Hom(Q, G|\pi')|$ for each coloring, π', then we can compute the total as $\sum_{\pi' \in Hom(Q,F)} |Hom(Q, G|\pi')|$. By this logic and equation 6.4, we simply need to show that $W(\pi') = |Hom(Q, G|\pi')|$.

We note that Q is a acyclic and proceed inductively on the topological ordering $v_1, \ldots, v_{|Q|}$. Let Q_i be the sub-tree restricted to the vertices $v_1, \ldots, v_i$, and let π'_i be the coloring restricted to these vertices. We begin with the base case,

$$W(\pi'_1) = \psi(\pi'_1(v_1)) = |Hom(Q_1, G|\pi'_1)|$$

The number of matches to a vertex of a color is equal to the number of vertices in that color, so this is immediately true. Now, suppose that $W(\pi'_i) = |Hom(Q_i, G|\pi'_i)|$, and we will show that $W(\pi'_{i+1}) = |Hom(Q_{i+1}, G|\pi'_{i+1})|$. We restate Eq. 6.4 as follows,

$$W(\pi'_{i+1}) = \psi(\pi'_{i+1}(v_1)) \prod_{(v_j, v_k) \in E_{Q_{i+1}}} \tau((\pi'_{i+1}(v_j), \pi'_{i+1}(v_k)))$$

Let v_j be the parent of v_{i+1}, and we can express $W_{\psi,\tau}(\pi'_{i+1})$ inductively,

$$W(\pi'_{i+1}) = W(\pi'_i) \cdot \tau((\pi_{i+1}(v_j), \pi_{i+1}(v_{i+1})))$$

By our inductive assumption, this means,

$$W(\pi'_{i+1}) = |Hom(Q_i, G|\pi'_i)| \cdot \tau((\pi_{i+1}(v_j), \pi_{i+1}(v_{i+1})))$$

Denote the number of edges that a vertex $v \in V_G$ has to vertices with color C as $deg_G(v|C)$. Because σ_S is a stable coloring, we know,

$$deg_G(v|C) = deg_G(v'|C) \;\; \forall v, v' \in V_G \; s.t. \; \sigma_S(v) = \sigma_S(v')$$

By the definition of τ, this degree is precisely,

$$deg_G(v|C) = \tau(\sigma_S(v_j), C)$$

We denote this degree with its color as $deg_G(C|C')$. Returning to our expression for $W_{\psi,\tau}(\pi'_{i+1})$, we can now plug in our degree expression,

$$W(\pi'_{i+1}) = |Hom(Q_i, G|\pi'_i)| \cdot deg_G(\pi'_{i+1}(v_j)|\pi'_{i+1}(v_{i+1}))$$

By simply expanding $|Hom(Q_i, G|\pi'_i)|$ into a sum, we get,

$$W^{std}(\pi'_{i+1}) = \sum_{\pi_i \in Hom(Q_i, G|\pi'_i)} deg_G(\pi'_{i+1}(v_j)|\pi'_{i+1}(v_{i+1})) \tag{C.2}$$

At this point, we use the fact that $deg(\pi'_i(v_j)|\pi'_{i+1}(v_{i+1}))$ $= deg(\pi_i(v_j)|\pi'_{i+1}(v_{i+1}))$ when $\sigma_S(\pi_i(v_j)) = \pi'_i(v_j)$, which is assured by $\pi \in Hom(Q_i, G|\pi'_i)$.

$$W^{std}(\pi'_{i+1}) = \sum_{\pi_i \in Hom(Q_i, G|\pi'_i)} deg_G(\pi_i(v_j)|\pi'_{i+1}(v_{i+1})) \tag{C.3}$$

Because Q is a tree, the RHS is the target of the induction,

$$W^{std}(\pi'_{i+1}) = |Hom(Q_{i+1}, G|\pi'_{i+1})|$$

To prove the corollary, we reconsider Eq. (C.2) in the case of quasi-stable or approximate colorings. We know that the maximum degree of a node in the color $\pi'_{i+1}(v_j)$ to another color $\pi'_{i+1}(v_{i+1})$ is at most ϵ times greater than the minimum degree. Therefore, the relative error between the true degree, $deg(\pi_i(v_j)|\pi'_{i+1}(v_{i+1}))$ and the average degree, $\tau(\pi'_{i+1}(v_j)|\pi'_{i+1}(v_{i+1}))$ must also be less than ϵ. Because each term in the sum on the RHS of Eq. (C.2) is within ϵ of the true degree for each node, the total sum must be only an ϵ multiplicative factor from the true in the RHS of (C.3). Lastly, we note that each step of our inductive proof only incurs an ϵ relative error, so the entire result must be within the desired $\epsilon^{(|Q|-1)}$ relative error.

□

C.2 Proof of Theorem 6.6.1

We begin by restating the theorem,

Theorem C.2.1. *Given a query graph, Q, a lifted graph, F, and a decomposable estimator W, $\Phi(Q, F, W_{\psi,\tau})$ can be computed in time $O(|C|^{tw(Q)})$ where $tw(Q)$ is the treewidth of Q where $\Phi(Q, F, W_{\psi,\tau})$ is defined,*

$$\Phi(Q, F, W_{\psi,\tau}) = \sum_{c_{v_1},...,c_{v_{|Q|}} \in C} \psi(c_0) \prod_{i=1}^{|E_Q|} \omega(\pi_{v_k \to c_{v_k}}, e_i | e_1, ..., e_{i-1})$$

We now define tree decompositions,

Definition C.2.2. Given a graph H, a tree decomposition $T(E_T, V_T, \chi, \gamma)$ is composed of four pieces,

1. E_T, V_T are the edges and vertices of a tree
2. $\chi : V_T \to 2^{V_H}$ is a function which maps vertices in the tree to sets of vertices in H
3. $\gamma : V_T \to 2^{E_H}$ is a function which maps vertices in the tree to sets of edges in H

Lastly, it has two requirements,

1. For all $v \in V_H$, the set of vertices in V_T that contain v form a connected sub-tree

2. Each edge in E_H is mapped to exactly one vertex of V_T by γ and the vertex of V_T which it is mapped to includes both the endpoints

The treewidth is then defined as follows,

Definition C.2.3. Given a graph H, let the set of valid tree decompositions be $\mathcal{T}_H$. The treewidth is then defined as,

$$\min_{T \in \mathcal{T}_H} \max_{v \in V_T} |\chi(v)| - 1$$

Intuitively, graphs that are "more acyclic" will have a lower treewidth and graphs that are "more cyclic" have a higher treewidth.

Proof. For this problem, we can take advantage of tree decompositions by using them to structure the summation in (6.10). Suppose that the treewidth of Q is k and let $T(E_T, V_T, \chi)$ be a tree decomposition which matches this width. To avoid confusion, we will denote vertices of T as v'_i and vertices of Q as v_i. Further, let $v'_1, \cdots, v'_{|V_T|}$ be a topological ordering of T where v'_1 is the root of the tree, and let $v_1, \cdots, v_{|V_Q|}$ be an ordering of Q such that the sub-tree corresponding to v_i is not a sub-tree of $v_{>i}$. Let $Par(v')$ denote the parent of v' in T and let $Ch(v')$ be the set of child vertices of v' in T. Lastly, we denote the set of query vertices which associated with $v'_{|V_T|}$ and not its parent as $\mathbf{X}_i = \chi(v'_i) \setminus \chi(Par(v'_i))$. We denote the set of query vertices which are associated with $v'_{|V_T|}$ AND its parent as $\mathbf{Y}_{|V_T|} = \chi(v'_{|V_T|}) \cap \chi(Par(v'_{|V_T|}))$.

We will proceed iteratively on these vertices starting with $v'_{|V_T|}$ and proceeding backwards towards the root.

Base Case: Our base case is $v'_{|V_T|}$ which is necessarily a leaf of the true due to the topological ordering. We denote the output of the base case as the function $S_{|V_T|}(c_{y_1}, \ldots, c_{y_m})$ and define it as follows,

$$S_{|V_T|}(c_{y_1}, \ldots, c_{y_m}) = \sum_{c_{x_1}, \ldots, c_{x_n} \in C} \prod_{e_i \in \gamma(v'_{|V_T|})} \omega(\pi_{x_j \to c_{x_j}}, e_i | e_1, \ldots, e_{i-1})$$

At this point of the computation, we will fully materialize the values of the function $S_{|V_T|}$ which has a domain of size $|C|^{|Y_{|V_T|}|}$ and each point requires computing a summation of $|C|^{|X_{|V_T|}|}$ terms. Therefore, the entire computation requires $|C|^{|\chi(v'_{|V_T|})|} \leq |C|^k$ time.

Inductive Case: Suppose that all vertices v'_j where $j > i$ have already been processed, we now consider handling v'_i. The output of this step is defined as,

$$S_i(c_{Y_{v'_i,1}}, \ldots, c_{Y_{v'_i,m}}) = \sum_{c_{X_{v'_i,1}}, \ldots, c_{X_{v'_i,n}} \in C} \prod_{e_i \in \gamma(v'_i)} \omega(\pi_{X_{v'_i,j} \to c_{X_{v'_i,j}}}, e_i | e_1, \ldots, e_{i-1}) \cdot \prod_{v'_j \in Ch(v'_i)} S_{v'_j}(c_{Y_{v'_j,1}}, \ldots, c_{Y_{v'_j,m}})$$

As in the base case, the computation required to fully materialize the values of S_i is bounded by $|C|^{|\chi(v'_{|V_T|})|} \leq |C|^k$.

At this point, we just need to confirm that S_1 is equal to $\Phi(Q, F, W_{\psi,\tau})$ by showing that the sequence of computations represents a valid rearrangement of the sums of the original formula. To this end, we note that every query vertex appears in the summation of precisely one inductive step. This is due to the fact that each query vertex forms a sub-tree in the tree decomposition, so the root of that sub-tree is the only tree vertex which contains it and whose parent does not. Further, each instance of $\omega(\pi_{X_{v'_i,j} \to c_{X_{v'_i,j}}}, e_i | e_1, \ldots, e_{i-1})$ occurs precisely once because of the requirement that each edge of the query graph occurs in one vertex of the tree decomposition.

□

VITA